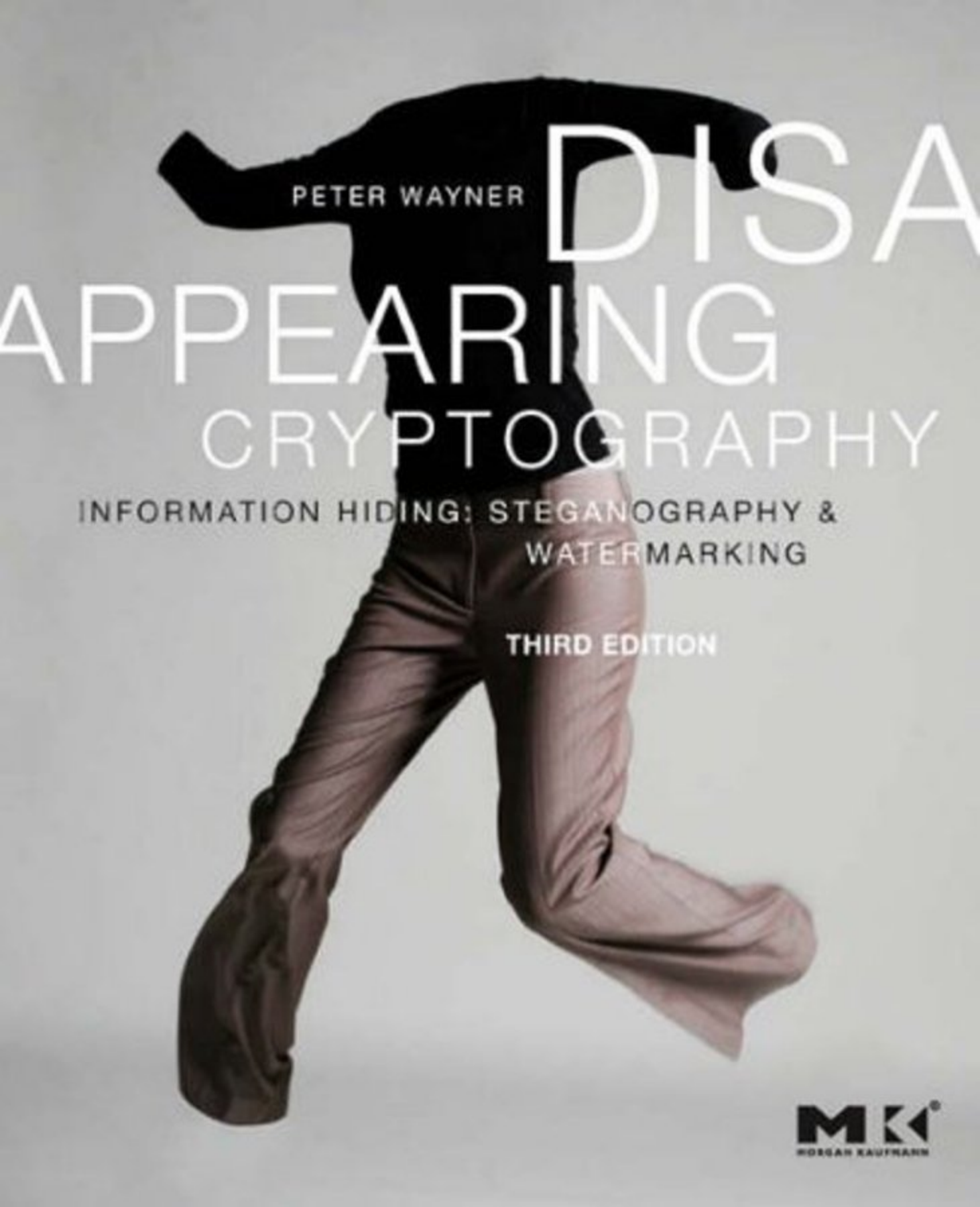

PETER WAYNER
DISA
APPEARING
CRYPTOGRAPHY
INFORMATION HIDING: STEGANOGRAPHY &
WATERMARKING
THIRD EDITION
MK
MORGAN KAUFMANN

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400
Burlington, MA 01803, USA

This book is printed on acid-free paper.

Library of Congress Cataloging-in-Publication Data
Wayner, Peter, 1964-
Disappearing cryptography: Information hiding: Steganography & watermarking / Peter Wayner. — 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-12-374479-1 (alk. paper)
1. Computer networks—Security measures. 2. Cryptography. 3. Internet. I. Title.
TK5105.59.W39 2009
005.8'2—dc22 2008044800

For information on all Morgan Kaufmann publications, visit our Web site at *www.mkp.com* or *www.books.elsevier.com*

Printed in the United States of America
8 9 10 11 12 10 9 8 7 6 5 4 3 2 1

About the Author

Peter Wayner is the author of more than a dozen books, if you include the different versions of this book, *Disappearing Cryptography*. This book is one of the best examples of a common theme in his work, the idea that information can hide from everyone. (The first edition came with the subtitle "Being and Nothingness on the Net", a choice that lost out to the power of keyword searches on the Internet. It's one thing to hide when you want to hide, but not when someone is looking for a book to purchase on Amazon.)

Other books that follow in this theme are:

- *Digital Cash*, An exploration of how to move money across the Internet by creating bits that can't be counterfeited. [Way95b]
- *Translucent Databases*–A manifesto on how to preserve privacy and increase security by creating databases that do useful work without having anything in them. [Way03]
- *Digital Copyright Protection*– How to keep content on a flexible leash. [Way97b]
- *Policing Online Games* – How to enforce contracts and keep games honest and fair. [Way05]

He writes often on technical topics for venues like *New York Times*, *InfoWorld*, *Byte*, *Wired* and, on occasion, even a USENET newsgroup or two.

When he's not writing, he consults on these topics for a wide range of companies.

Preface

This book is a third edition and so that means more thanks for everyone. There is no doubt that I owe a debt of gratitude to the participants in the cypherpunks and coderpunks mailing lists. Their original contributions inspired me to write the first book and their continual curiosity makes it one of the best sources of information around.

Some newer mailing lists are more focused on the topic. The watermarking list and the stegano list both offer high quality discussions with a high signal-to-noise ratio. Other lists like the RISKS digest and Dave Farber's Interest People list helped contribute in unexpected ways. Of course, modern list-like web sites like Slashdot, Kuro5hin, and InfoAnarchy contributed by offering solid, moderated discussions that help the signal jump out of the noise. It is impossible to thank by name all of the members of the community who include plenty of solid information and deep thought in their high-quality postings.

The organizers of the Information Hiding Workshops brought some academic rigor to the area by sponsoring excellent workshops on the topic. The discipline of creating, editing, reviewing, presenting and publishing a manuscript advanced the state of the art in numerous ways. The collected papers published by Springer-Verlag are a great resource for anyone interested in the development of the field.

Some others have helped in other ways. Peter Neumann scanned the first manuscript and offered many good suggestions for improving it. Bruce Schneier was kind enough to give me an electronic version of the bibliography from his first book [Sch94]. I converted it into Bibtex format and used it for some of the references here. Ross Anderson's annotated bibliography on Information Hiding was also a great help.

Scott Craver, Frank Hartung, Deepa Kundur,Mike Sway, and three anonymous reviewers checked the second edition. Their comments helped fixed numerous errors and also provided many suggestions

for improving the book.

The original book was originally published by AP Professional, a division of Harcourt-Brace that blended into Morgan Kaufmann. The team responsible for producing the first edition was: Chuck Glaser, Jeff Pepper, Mike Williams, Barbara Northcott, Don DeLand, Tom Ryan, Josh Mills, Gael Tannenbaum, and Dave Hannon.

The second edition would not exist without the vision and support of Tim Cox at Morgan Kaufmann. I would like to thank Tim and Stacie Pierce for all of their help and encouragement.

The third edition exists because Rick Adams, Gregory Chalson and Denise Penrose saw the value in the book and devoted their hard work and energy to bringing it to market again. Sherri Davidoff, Rakan El-Khalil, Philipp Gühring,Scott Guthery, J. Wren Hunt, John Marsh, Chris Peikert Leonard Popyack and Ray Wagner read portions of the book and provided invaluable help fixing the book.

Peter Wayner
Baltimore, MD
October 2008
p3@wayner.org
http://www.wayner.org

Book Notes

The copy for this book was typeset using the LaTeX typesetting software. Several important breaks were made with standard conventions in order to remove some ambiguities. The period mark is normally included inside the quotation marks like this "That's my answer. No. Period." This can cause ambiguities when computer terms are included in quotation marks because computers often use periods to convey some meaning. For this reason, my electronic mail address is "p3@wayner.org". The periods and commas are left outside of all quotes to prevent confusion.

Hyphens also cause problems when they're used for different tasks. LISP programmers often use hyphens to join words together into a single name like this: `Do-Not-Call-This-Procedure`. Unfortunately, this causes grief when these longer words occur at the end of a line. In these cases, there will be an extra hyphen included to specify that there was an original hyphen in the word. This isn't *hyper-compatible* with the standard rules that don't include the extra hyphen. But these rules are for readers who know that *self-help* is a word that should be hyphenated. No one knows what to think about `A-Much-Too-Long-Procedure-That-Should--Be-Shortened-For-Everyone.`

A Start

This book is about making information disappear. For some people, this topic is a parlor trick, an amazing intellectual exercise that rattles around about the foundations of knowledge. For others, the topic has immense practical importance. An enemy can only control your message if they can find it. If you hide data, you can protect your thoughts from censorship and discovery.

The book describes a number of different techniques that people can use to hide information. The sound files and images that float about the network today are great locations filled with possibilities. Large messages can be hidden in the noise of these images or sound files where no one can expect to find them. About one eighth of an image file can be used to hide information without any significant change in the quality of the image.

Information can also be converted into something innocuous. You can use the algorithms from Chapter 7 to turn data into something entirely innocent like the voice-over to a baseball game. Bad poetry is even easier to create.

If you want to broadcast information without revealing your location, the algorithms from Chapter 11 show how a group of people can communicate without revealing who is talking. Completely anonymous conversations can let people speak their mind without endangering their lives.

The early chapters of the book are devoted to material that forms the basic bag of tricks like private-key encryption, secret sharing, and error-correcting codes. The later chapters describe how to apply these techniques in various ways to hide information. Each of them is designed to give you an introduction and enough information to use the data if you want.

The information in each chapter is roughly arranged in order of importance and difficulty. Each begins with a high-level summary for those who want to understand the concepts without wading through technical details, and a introductory set of details, for those

who want to create their own programs from the information. People who are not interested in the deepest, most mathematical details can skip the last part of each chapter without missing any of the highlights. Programmers who are inspired to implement some algorithms will want to dig into the last pages.

Many of the chapters also come with allegorical narratives that may illustrate some of the ideas in the chapters. You may find them funny, you may find them stupid, but I hope you'll find some better insight into the game afoot.

For the most part, this book is about having fun with information. But knowledge is power and people in power want to increase their control. So the final chapter is an essay devoted to some of the political questions that lie just below the surface of all of these morphing bits.

0.1 Notes On the Third Edition

When I first wrote this book in 1994 and 1995, no one seemed to know what the word "steganography" meant. I wanted to call the book *Being and Nothingness on the Net.* The publisher sidesteped that suggestion by calling it *Disappearing Cryptography* and putting the part about *Being and Nothingness* in the subtitle. He didn't want to put the the word "steganography" in the title because it might frighten someone.

When it came time for the second edition, everything changed. The publisher insisted we get terms like *steganography* in the title and added terms like *Information Hiding* for good measure. Everyone knew the words now and he wanted to make sure that the book would show up on a search of Amazon or Google.

This time, there will be no change to the title. The field is much bigger now and everyone has settled on some of the major terms. That simplified a bit of the reworking of the book, but it did nothing to reduce the sheer amount of work in the field. There are a number of good academic conferences, several excellent journals and a growing devotion to building solid tools at least in the areas of digital rights management.

The problem is that the book is now even farther from comprehensive. What began as an exploration in hiding information in plain sight is now just an introduction to a field with growing economic importance.

Watermarking information is an important tool that may allow content creators to unleash their products in the anarchy of the web. Steganography is used in many different places in the infrastructure

of the web. It is now impossible to do a good job squeezing all of the good techniques for hiding information into a single book.

0.2 Notes On the Second Edition

The world of steganography and hidden information changed dramatically during the five years since the first edition appeared. The interest from the scientific community grew and separate conferences devoted to the topic flourished. A number of new ideas, approaches, and techniques appeared and many are included in the book.

The burgeoning interest was not confined to labs. The business community embraced the field in the hope that the hidden information would give creators of music and images a chance to control their progeny. The hidden information is usually called a *watermark*. This hidden payload might include information about the creator, the copyright holder, the purchaser or even special instructions about who could consume the information and how often they could push the button.

Many of the private companies have also helped the art of information hiding, but sometimes the drive for scientific advancement clashed with the desires of some in the business community. The scientists want the news of the strengths and weaknesses of steganographic algorithms to flow freely. Some businessmen fear that this information will be used to attack their systems and so they push to keep the knowledge hidden.

This struggle errupted into an open battle when the recording industry began focusing on the work of Scott A. Craver, John P McGregor, Min Wu, Bede Liu, Adam Stubblefield, Ben Swartzlander, Dan S. Wallach, Drew Dean, and Edward W. Felten. The group attacked a number of techniques distributed by the Secure Digital Music Initiative, an organization devoted to creating a watermark system and sponsored by the members of the music industry. The attacks were invited by SDMI in a public contest intended to test the strengths of the algorithms. Unfortunately, the leaders of the SDMI also tried to hamstring the people who entered the contest by forcing them to sign a pledge of secrecy to collect their prize. In essence, the group was trying to gain all of the political advantages of public scrutiny while trying to silence anyone who attempted to spread the results of their scrutiny to the public. When the group tried to present their work at the Information Hiding Workshop in April in Pittsburgh, the Recording Industry Association of America (RIAA) sent them a letter suggesting that public discussion would be punished by a law-

suit. The group withdrew the paper and filed their own suit claiming that the RIAA and the music industry was attempting to stiffle their First Amendment Rights. The group later presented their work at the USENIX conference in Washington, DC, but it is clear that the battle lines still exist. On one side are the people who believe in open sharing of information, even if it produces an unpleasant effect, and on the other are those who believe that censorship and control will keep the world right.

This conflict seems to come from the perception that the algorithms for hiding information are fragile. If someone knows the mechanism in play, they can destroy the message by writing over the messages or scrambling the noise. The recording industry is worried that someone might use the knowledge of how to break the SDMI algorithms to destroy the watermarking information– something that is not difficult to do. The only solution, in some eyes, is to add security by prohibiting knowledge.

This attitude is quite different from the approach taken with the close cousin, cryptography. Most of the industry agrees that public scrutiny is the best way to create secure algorithms. Security through obscurity is not as successful as a well-designed algorithm. As a result, public scrutiny has identified many weaknesses in cryptographic algorithms and helped researchers develop sophisticated solutions.

Some companies trying to create watermarking tools may feel that they have no choice but to push for secrecy. The watermarking tools aren't secure enough to withstand assault so the companies hope that some additional secrecy will make them more secure.

Unfortunately, the additional secrecy buys little extra. Hidden information is easy to remove by compressing, reformatting, and rerecording the camouflaging information. Most common tools used in recording studios, video shops, and print shops are also good enough to remove watermarks. There's nothing you can do about it. Bits are bits and information is information. There is not a solid link between the two.

At this writing the battle between the copyright holders and the scientists is just beginning. Secret algorithms never worked for long before and there's no reason why it will work now. In the meantime, enjoy the information in the book while you can. There's no way to tell how long it will be legal to read this book.

Chapter 1

Framing Information

On its face, information in computers seems perfectly defined and certain. A bank account either has $1,432,442 or it has $8.32. The weather is either going to be 73 degrees or 74 degrees. The meeting is either going to be at 4 pm or 4:30 pm. Computers deal only with numbers and numbers are very definite.

Life isn't so easy. Advertisers and electronic gadget manufacturers like to pretend that digital data is perfect and immutable, freezing life in a crystalline mathematical amber; but the natural world is filled with noise and numbers that can only begin to approximate what is happening. The digital information comes with much more precision than the world may provide.

Numbers themselves are strange beasts. All of their certainty can be scrambled by arithmetic, equations and numerical parlor tricks designed to mislead and misdirect. Statisticians brag about lying with numbers. Car dealers and accountants can hide a lifetime of sins in a balance sheet. Encryption can make one batch of numbers look like another with a snap of the fingers.

Language itself is often beyond the grasp of rational thought. Writers dance around topics and thoughts, relying on nuance, inflection, allusion, metaphor, and dozens of other rhetorical techniques to deliver a message. None of these tools are perfect and people seem to find a way to argue about the definition of the word "is".

This book describes how to hide information by exploiting this uncertainty and imperfection. This book is about how to take words, sounds, and images and hide them in digital data so they look like other words, sounds, or images. It is about converting secrets into innocuous noise so that the secrets disappear in the ocean of bits flowing through the Net. It describes how to make data mimic other

data to disguise its origins and obscure its destination. It is about submerging a conversation in a flow of noise so that no one can know if a conversation exists at all. It is about taking your being, dissolving it into nothingness, and then pulling it out of the nothingness so it can live again.

Traditional cryptography succeeds by locking up a message in a mathematical safe. Hiding the information so it can't be found is a similar but often distinct process often called *steganography*. There are many historical examples of it including hidden compartments, mechanical systems like microdots, or burst transmissions, that make the message hard to find. Other techniques like encoding the message in the first letters of words disguise the content and make it look like something else. All of these have been used again and again.

David Kahn's Codebreakers *provides a good history of the techniques.[Kah67]*

Digital information offers wonderful opportunities to not only hide information, but also to develop a general theoretical framework for hiding the data. It is possible to describe general algorithms and make some statements about how hard it will be for someone who doesn't know the key to find the data. Some algorithms offer a good model of their strength. Others offer none.

Some of the algorithms for hiding information use keys that control how they behave. Some of the algorithms in this book hide information in such way that it is impossible to recover the information without knowing the key. That sounds like cryptography, even though it is accomplished at the same time as cloaking the information in a masquerade.

Is it better to think of these algorithms as "cryptography" or as "steganography"? Drawing a line between the two is both arbitrary and dangerously confusing. Most good cryptographic tools also produce data that looks almost perfectly random. You might say that they are trying to hide the information by disguising it as random noise. On the other hand, many steganographic algorithms are not trivial to break even after you learn that there is hidden data to find. Placing an algorithm in one camp often means forgetting why it could exist in the other. The best solution is to think of this book as a collection of tools for massaging data. Each tool offers some amount of misdirection and some amount of security. The user can combine a number of different tools to achieve their end.

The book is published under the title of "Disappearing Cryptography" for the reason that few people knew about the word "steganography" when it appeared. I have kept the title for many of the same practical reasons, but this doesn't mean that title is just cute mechanism for giving the buyer a cover text they can use to judge the book.

Simply thinking of these algorithms as tools for disguising information is a mistake. Some offer cryptographic security at the same time as an effective disguise. Some are deeply intertwined with cryptographic algorithms, while others act independently. Some are difficult to break without the key while others offer only basic protection. Trying to classify the algorithms purely as steganography or cryptography imposes only limitations. It may be digital information, but that doesn't mean there aren't an infinite number forms, shapes, and appearances the information may assume.

1.0.1 Reasons for Secrecy

There are many different reasons for using the techniques in this book and some are scurrilous. There is little doubt that the Four Horsemen of the Infocalypse– the drug dealers, the terrorists, the child pornographers, and the money launderers– will find a way to use the tools to their benefit in the same way that they've employed telephones, cars, airplanes, prescription drugs, box cutters, knives, libraries, video cameras and many other common, everyday items. There's no need to explain how people can hide behind the veils of anonymity and secrecy to commit heinous crimes.

But these tools and technologies can also protect the weak. In book's defense, here's a list of some possible good uses:

1. So you can seek counseling about deeply personal problems like suicide.
2. So you can inform colleagues and friends about a problem with odor or personal hygiene.
3. So you can meet potential romantic partners without danger.
4. So you can play roles and act out different identities for fun.
5. So you can explore job possibilities without revealing where you currently work and potentially losing your job.
6. So you can turn a person in to the authorities anonymously without fear of recrimination.
7. So you can leak information to the press about gross injustice or unlawful behavior.
8. So you can take part in a contentious political debate about, say, abortion, without losing the friendship of those who happen to be on the other side of the debate.

9. So you can protect your personal information from being exploited by terrorists, drug dealers, child pornographers and money launderers.

10. So the police can communicate with undercover agents infiltrating the gangs of bad people.

Chapter 22 examines the promises and perils of this technology in more detail.

There are many other reasons, but I'm surprised that government officials don't recognize how necessary these freedoms are to the world. Much of government functions through back-corridor bargaining and power games. Anonymous communication is a standard part of this level of politics. I often believe that all governments would grind to a halt if information was as strictly controlled as some would like it to be. No one would get any work done. They would just spend hours arguing who should and should not have access to information.

The Central Intelligence Agency, for instance, has been criticized for missing the collapse of the former Soviet Union. They continued to issue pessimistic assessments of a burgeoning Soviet military while the country imploded. Some blame greed, power, and politics. I blame the sheer inefficiency of keeping information secret. Spymaster Bob can't share the secret data he got from Spymaster Fred because everything is compartmentalized. When people can't get new or solid information, they fall back to their basic prejudices—which in this case was that the Soviet Union was a burgeoning empire. There will always be a need for covert analysis for some problems, but it will usually be much more inefficient than overt analysis.

Anonymous dissemination of information is a grease for the squeaky wheel of society. As long as people question its validity and recognize that its source is not willing to stand behind the text, then everyone should be able to function with the information. When it comes right down to it, anonymous information is just information. It's just a torrent of bits, not a bullet, a bomb or a broadside. Sharing information generally helps society pursue the interests of justice.

Secret communication is essential for security. The police and the defense department are not the only people who need the ability to protect their schedules, plans, and business affairs. The algorithms in this book are like locks on doors and cars. Giving this power to everyone gives everyone the power to protect themselves against crime and abuse. The police do not need to be everywhere because people can protect themselves.

For all of these reasons and many more, these algorithms are powerful tools for the protection of people and their personal data.

1.0.2 How It Is Done

There are a number of different ways to hide information. All of them offer some stealth, but not all of them are as strong as the others. Some provide startling mimicry with some help from the user. Others are largely automatic. Some can be combined with others to provide multiple layers of security. All of them exploit some bit of randomness, some bit of uncertainty, or some bit of unspecified state in a file. Here is an abstract list of the techniques used in this book:

Use the Noise The simplest technique is to replace the noise in an image or sound file with your message. The digital file consist of numbers that represent the intensity of light or sound at a particular point of time or space. Often these numbers are computed with extra precision that can't be detected effectively by humans. For instance, one spot in a picture might have 220 units of blue on a scale that runs between 0 and 255 total units. An average eye would not notice if that one spot was converted to having 219 units of blue. If this process is done systematically, it is possible to hide large volumes of information just below the threshold of perception. A digital photo-CD image has 2048 by 3072 pixels that each contain 24 bits of information about the colors of the image. 756k of data can be hidden in the three least significant bits for each color of each pixel. That's probably more than the text of this book. The human eye would not be able to detect the subtle variations but a computer could reconstruct them all.

Spread the Information Out Some of the more sophisticated mechanisms spread the information over a number of pixels or moments in the sound file. This diffusion protects the data and also makes it less susceptible to detection, either by humans looking at the information or by computers looking for statistical profiles. Many of the techniques that fall into this category came from the radio communication arena where the engineers first created them to cut down on interference, reduce jamming, and add some secrecy. Adapting them to digital communications is not difficult.

Spreading the information out often increases the resilience to destruction by either random or malicious forces. The spreading algorithms often distribute the information in such a way that not all of the bits are required to reassemble the original data. If some parts get destroyed, the message still gets through.

Many of these spreading techniques hide information in the noise of an image or sound file, but there is no reason why they can't be used with other forms of data as well.

Many of the techniques are closely related to the process of generating cryptographically secure random numbers– that is, a stream of random numbers that can't be predicted. Some algorithms use this number stream to choose locations, others blend the random values with the hidden information, still others replace some of the random values with the message.

Adopt a Statistical Profile Data often falls into a pattern and computers often try to make decisions about data by looking at the pattern. English text, for instance, uses the letter 'p' for more often than the letter 'q' and this information can be useful for breaking ciphers. If data can be reformulated so it adopts the statistical profile of the English language, then a computer program minding ps and qs will be fooled.

Adopt a Structural Profile Mimicking the statistics of a file is just the beginning. More sophisticated solutions rely on complex models of the underlying data to better mimic it. Chapter 7, for instance, hides information by making it look like the transcript of a baseball game. The bits are hidden by using them to choose between the nouns, verbs and other parts of the text. The data are recovered by sorting through the text and matching up the words with the bits that selected them. This technique can produce startling results, although the content of the messages often seems a bit loopy or directionless. This is often good enough to fool humans or computers that are programmed to algorithmically scan for particular words or patterns.

Replace Randomness Many software programs use random number generators to add realism to scenes, sounds, and games. Monsters look better if a random number generator adds blotches, warts, moles, scars and gouges to a smooth skin defined by mathematical spheres. Information can be hidden in the place of the random number. The location of the splotches and scars carries the message.

Change the Order A grocery list may be just a list, but the order of the items can carry a surprisingly large amount of information.

Split Information Data can be split into any number of packets that take different routes to their destination. Sophisticated algorithms can also split the information so that any subset of k of the n parts are enough to reconstruct the entire message.

Hide the Source Some algorithms allow people to broadcast information without revealing their identity. This is not the same as hiding the information itself, but it is still a valuable tool. Chapters 10 and 11 show how to use anonymous remailers and more

mathematically sophisticated Dining Cryptographers' solutions to distribute information anonymously.

These different techniques can be combined in many ways. First information can be hidden by hiding it in a list, then the list can be hidden in the noise of a file that is then broadcast in a way to hide the source of the data.

1.0.3 How Steganography Is Used

Hidden information has a variety of uses in products and protocols. Hiding slightly different information or combining the various algorithms creates different tools with different uses. Here are some of the most interesting applications:

Enhanced Data Structures Most programmers know that standard data structures get old over time. Eventually there comes a time when new, unplanned information must be added to the format without breaking old software. Steganography is one solution. You can hide extra information about the photos in the photos themselves. This information travels with the photo but will not disturb old software that doesn't know of its existence.

A radiologist could embed comments from in the background of a digitized x-ray. The file would still work with standard tools, saving hospitals the cost of replacing all of their equipment.

Strong Watermarks The creators of digital content like books, movies, and audio files want to add hidden information into the file to describe the restrictions they place on the file. This message might be as simple as "This file copyright 2001 by Big Fun" or as complex as "This file can only be played twice before 12/31/2002 unless you purchase three cases of soda and submit their bottle tops for rebate. In which case you get 4 song plays for every bottle top."

Some watermarks are meant to be found even after the file undergoes a great deal of distortion. Ideally, the watermark will still be detectable even after someone crops, rotates, scales and compresses some document. The only way to truly destroy it is to alter the document so much that it is no longer recognizable.

Other watermarks are deliberately made as fragile as possible. If someone tries to tamper with the file, the watermark will disappear. Combining strong and weak watermarks is a good option when tampering is possible.

Digital Watermarking *by Ingemar J. Cox, Matthew L. Miller and Jeffrey A. Bloom is a good introduction to watermarks and the challenges particular to the subfield.[CMB01]*

Document-Tracking Tools Hidden information can identify the legitimate owner of the document. If it is leaked or distributed to unauthorized people, it can be tracked back to the rightful owner. Adding individual tags to each document is an idea attractive to both content-generating industries and government agencies with classified information.

File Authentication The hidden information bundled with a file can also contain a digital signature certifying its authenticity. A regular software program would simply display (or play) the document. If someone wanted some assurance, the digital signature embedded in the document can verify that the right person signed it.

Private Communications Steganography is also useful in political situations when communications is dangerous. There will always be moments when two people can't exchange messages because their enemies are listening. Many governments continue to see the Internet, corporations and electronic conversations as an opportunity for surveillance. In these situations, hidden channels offer the politically weak a chance to elude the powerful who control the networks. [Sha01]

Not all uses for hidden information come classified as steganography or cryptography. Anyone who deals with old data formats and old software knows that programmers don't always provide ideal data structures with full documentation. Many basic hacks aren't much different from the steganographic tools in this book. Clever programmers find additional ways to stretch a data format by packing extra information where it wasn't needed before. This kind of hacking is bound to yield more applications than people imagined for steganography. Somewhere out there, a child's life may be saved thanks to clever data handling and steganography!

1.0.4 Attacks on Steganography

Steganographic algorithms provide stealth, camouflage and security to information. How much, though, is hard to measure. As data blends into the background, when does it effectively disappear? One way to judge the strength is to imagine different attacks and then try to determine whether the algorithm can successfully withstand them. This approach is far from perfect, but it is the best available. There's no way to anticipate all possible attacks, although you can try.

Attacking steganographic algorithms is very similar to attacking cryptographic algorithms and many of the same techniques apply. Of course, steganographic algorithms promise some additional stealth in addition to security so they are also vulnerable to additional attacks.

Here's a list of some possible attacks:

File Only The attacker has access to the file and must determine if it holds a hidden message. This is the weakest form of attack, but it is also the minimum threshold for successful steganography.

Many of these basic attacks rely on a statistical analysis of digital images or sound files to reveal the presence of a message in the file. This type of attack is often more of an art than a science because the person hiding the message can try to counter an attack by adjusting the statistics.

File and Original Copy In some cases, the attacker may have a copy of the file with the encoded message and a copy of the original, pre-encoded file. Clearly, detecting some hidden message is a trivial operation. If the two files are different, there must be some new information hidden inside of it.

The real question is what the attacker may try to do with the data. The attacker may try to destroy the hidden information, something that can be accomplished by replacing it with the original. The attacker may try to extract the information or even replace it with their own. The best algorithms try to defend against someone trying to forge hidden information in a way that it looks like it was created by someone else. This is often imagined in the world of watermarks, where the hidden information might identify the rightful owner. An attacker might try to remove the watermark from a legitimate owner and replace it with a watermark giving themselves all of the rights and privileges associated with ownership.

Multiple Encoded Files The attacker gets n different copies of the files with n different messages. One of them may or may not be the original unchanged file. This situation may occur if a company is inserting different tracking information into each file and the attacker is able to gather a number of different versions. If music companies sell digital sound files with personalized watermarks, then several fans with legitimate copies can get together and compare their files.

Some attackers may try to destroy the tracking information or to replace it with their own version of the information. One of

the simplest attacks in this case is to blend the files together, either by averaging the individual elements of the file or by creating a hybrid by taking different parts from each file.

Access to the File and Algorithm An ideal steganographic algorithm can withstand scrutiny even if the attacker knows the algorithm itself. Clearly, basic algorithms that hide and unveil information can't resist this attack. Anyone who knows the algorithm can use this it to extract the information.

But this can work if you keep some part of the algorithm secret and use it as the "key" to unlock the information. Many algorithms in this book use a cryptographically secure random number generator to control how the information is blended into a file. The seed value to this random number stream acts like a key. If you don't know it, you can't generate the random number stream and you can't unblend the information.

Destroy Everything Attack Some people argue that steganography is not particularly useful because an attacker could simply destroy the message by blurring a photo or adding noise to a sound file. One common technique used against the kind of block compression algorithms like JPEG is to rotate an image 45 degrees, blur the image, sharpen it again, and then rotate it back. This mixes information from different blocks of the image, effectively removing some schemes like the ones in Chapter 14.

This technique is a problem, but it can be computationally prohibitive for many users and it introduces its own side effects. A site like Flickr.com might consider doing this to all incoming images to deter communications, but it would require a fair amount of computation.

It is also not an artful attack. Anyone can destroy messages. Cryptography and many other protocols are also vulnerable to it.

Random Tweaking Attacks Some attackers may not try to determine the existence of a message with any certainty. An attacker could just add small, random tweaks to all files in the hope of destroying whatever message may be there. During World War II, the government censors would add small changes to numbers in telegrams in the hopes of destroying covert communications. This approach is not very useful because it sacrifices overall accuracy for the hope of squelching a message. Many

of the algorithms in this book can resist a limited attack by using error-correcting codes to recover from a limited number of seemingly random changes.

Add New Information Attack Attackers can use the same software to encode a new message in a file. Some algorithms are vulnerable to these attacks because they overwrite the channel used to hide the information. The attack can be resisted with good error-correcting codes and by using only a small fraction of the channel chosen at random.

Reformat Attack One possible attack is to change the format of the file because many competing file formats don't store data in exactly the same way. There are a number of different image formats, for instance, that use a variety of bits to store the individual pixels. Many basic tools help the graphic artist deal with the different formats by converting one file format into an other. Many of these conversions can't be perfect. The hidden information is often destroyed in the process. Images can be stored as either JPEG or GIF images, but converting from JPEG to GIF removes some of the extra information– the EXIF fields – embedded in the file as part of the standard.

Many watermark algorithms for images try to resist this type of attack because reformatting is so common in the world of graphic arts. An ideal audio watermark, for instance, would still be readable after someone plays the music on a stereo and records it after it has traveled through the air.

Of course, there are limits to this. Reformatting can be quite damaging and it is difficult to anticipate all of the cropping, rotating, scaling, and shearing that a file might undergo. Some of the best algorithms do come close.

Compression Attack One of the easiest attacks is to compress the file. Compression algorithms try to remove the extraneous information from a file and "hidden" is often equivalent to "extraneous". The dangerous compression algorithms are the so-called *lossy* ones that do not reconstruct a file exactly during decompression. The JPEG image format, for instance, does a good job approximating the original.

Some of the watermarking algorithms can resist compression by the most popular algorithms, but there are none that can resist all of them.

The only algorithms that can resist all compression attacks

hides the information in plain sight by changing the "perceptually salient" features of an image or sound file.

Unfortunately, steganography is not a solid science, in part because there's no simple way to measure how well it is doing. How hidden must the information be before no one can see it? Just how invisible is invisible? The models of human perception are often too basic to measure what is happening.

The lack of a solid model means it is difficult to establish how well the algorithms resist attack. Many algorithms can survive cursory scrutiny but fail if a highly trained or talented set of ears and eyes analyze the results. Some people with so-called "golden ears" can hear supposedly changes in an audio file that are inaudible to average humans. A watermark may be completely inaudible to most of the buying public, but if the musicians can hear it the record company may not use it.

Our lack of understanding does not mean that the algorithms don't have practical value. A watermark heard by 1% of the population is of no concern to the other 99%. An image with hidden information may be detectable, but this only matters if someone is trying to detect it.

There is also little doubt that a watermark or a steganographic tool does not need to resist all attackers to have substantial value. A watermark that lives on after cropping and basic compression still carries its message to many people. A hacker may learn how to destroy it, but most people have better things to do with their time.

Our lack of understanding does not mean that the algorithms do not offer some security. Some of the algorithms insert their information with mechanisms that offer cryptographic strength. Borrowing these ideas and incorporating them provides both stealth and security.

1.1 Adding Context

One reviewer of the book who was asked for a backcover blurb joked that the book should be "essential bedside for reading for every terrorist". After a pause he added, "and every freedom fighter, Hollywood executive, police officer, abused spouse, chief information officer, and anyone needing privacy anywhere."

You may be a terrorist or you may be a freedom fighter. Who knows? This book is just about technology and technology is neutral. It teaches you how to cast shape shifting spells that make data look like something completely different. You may have good plans

for these ideas. Perhaps you want to expose a local chemical company dumping toxic waste into the ground. Or you might be filled with the proverbial malice aforethought and you can't wait to hatch a maniacal plan. You might be part of that cabal of executives using these secret algorithms to plan where and when to dump the toxic waste. Technology is neutral.

There is some human impulse that would like to believe that all information is ordered, correct, structured, organized, and above all true. We dream that computers and their vast collection of trivia about the world will keep us safe, secure, and moving toward some glorious goal, even if we don't know what it is. We hope that the databases held by the government, the banks, the insurance companies, the retail stores, the doctors, and practically everyone else will deliver unto us a perfectly ordered world.

Alas, nothing could be farther from the truth. Even the bits can hide multiple meanings. They're supposed to be either on or off, true or false, 0 or 1, but even the bits can conspire to carry secret messages and hidden truths. Information is not as certain or as precise as it may seem to be. Sometimes a cigar carries a freight train load of meaning and sometimes it is just a cigar. Sometimes it is close and no cigar at all.

Throughout it all, only a human can make sense of it. Only a human can determine the difference between an obscene allusion to a cigar and reference to an object for delivering nicotine. We keep hoping that artificial intelligence and database engines will be able to parse all of the data, all of the facts, and all of the bits and identify the terrorists who need punishing, the good people who need help, and the split ends that need another dose of special conditioner.

You, the reader, are the human who must decide how to use the information in this book. You can solve crimes, coordinate a wedding, plan a love that will last forever, or concoct dastardly schemes. The technology is neutral. The book is just equations on a page. You will determine what the equations mean for the world.

Chapter 2

Encryption

2.1 Pure White

In the early years of the 21st century, Pinnacle Paint was purchased by the MegaGoth marketing corporation in a desperate attempt to squeeze the last bit of synergy from the world. The executives of MegaGoth, who were frantic with the need to buy something they didn't already own so they could justify their existence, found themselves arguing that the small, privately owned paint company fit nicely into their marketing strategy for dominating the entertainment world.

Although some might argue that people choose colors with their eyes, the executives quickly began operating under the assumption that people purchased paint that would identify them with something. People wanted to be part of a larger movement. They weren't choosing a color for a room, they were buying into a lifestyle—how dare they choose any lifestyle without licensing one from a conglomerate? The executives didn't believe this, but they were embarrassed to discover that their two previous acquisitions targets were already owned by MegaGoth. Luckily, their boss didn't know this either when he gave the green light to those projects. Only the quick thinking of a paralegal saved them from the disaster of buying something they already owned and paying all of that tax.

One of the first plans for MegaGoth/Pinnacle Paints is to take the standard white paint and rebottle it in new and different product lines to target different demographic groups. Here are some of Megagoth's plans:

Moron and Moosehead's Creative Juice What would the two lovable animated characters paint if they were forced to expand their

creativity in art class? Moron might choose a white cow giving milk in the Arctic for his subject. Moosehead would probably try to paint a little lost snowflake in a cloud buffeted by the wind and unable to find its way to its final destination: Earth.

Empathic White White is every color. The crew of "Star Trek: They Keep Breeding More Generations" will welcome Bob, the "empath," to the crew next season. His job is to let other people project their feelings onto him. Empathic White will serve the same function for the homeowner as the mixing base for many colors. Are you *blue*? Bob the Empath could accept that feeling and validate it. Do you want your living room to be blue? That calls for Empathic White. Are you *green* with jealousy? Empathic White at your service.

Fright White MegaGoth took three British subjects and let them watch two blood-draining horror movies from the upcoming MegaGoth season. At the end, they copied the color of the subject's skin and produced the purest white known to the world.

Snow White A cross-licensing product with the MegaGoth/Disney division ensures that kids in their nursery won't feel alone for a minute. Those white walls will be just another way to experience the magic of movie produced long ago when Disney was a distinct corporation.

White Dwarf White The crew of "Star Trek" discovers a White Dwarf star and spends an entire episode orbiting it. But surprise! The show isn't about White Dwarf stars qua White Dwarfs, it's really using their super-strong gravitational fields as a metaphor for human attraction. Now, everyone can wrap themselves in the same metaphor by painting their walls with White Dwarf White.

2.2 Encryption and White Noise

Hiding information is a tricky business. Although the rest of this book will revolve around camouflaging information by actually making the bits look like something else, it is a good idea to begin with examining basic encryption.

Standard encryption functions like AES or RSA hide data by making it incomprehensible. They take information and convert it into total randomness or white noise. This effect might not be a good way to divert attention from a file, but it is still an important tool.

Many of the algorithms and approaches described later in the book perform best when they have a perfectly random source of data.

Encrypting a file before applying any of the other approaches is a good beginning, but it doesn't complete the picture. Sometimes too much randomness can stick out like a sore thumb. Chapter 17 describes several algorithms that can flag images with hidden information by relying on statistical tests that measure, often indirectly, the amount of randomness in the noise. A file that seems too random stands out because the noise generated by many digital cameras isn't as random as it might seem.

The trick is to use some extra processing to add a bit of statistical color to the data before it is introduced. Chapters 6 and 7 describe some solutions. Others involve mixing in the hidden message in a way that doesn't distort the statistical profile of the data.

The world of cryptography began attempting to produce perfect white noise during World War II. This is because Claude Shannon-Claude E. Shannon, a mathematician then working for Bell Labs, developed the foundations of information theory that offered an ideal framework for actually measuring information.

Most people who use computers have a rough idea about just how much information there is in a particular file. A word processing document, for instance, has some overhead and about one byte for each character– a simple equation that doesn't seem to capture the essence of the problem. If the number of bytes in a computer file is an accurate measurement of the information in it, then there would be no way that a compression program could squeeze files to be a fraction of the original size. Real estate can't be squeezed and diamonds can't be smooshed, but potato chips always seem to come in a bag filled with air. That's why they're sold by weight not volume. The success of compression programs like PKZIP or Stuffit means that measuring a file by the number of bytes is like selling potato chips by volume.

Compression is discussed in Chapter 5.

Shannon's method of measuring information "by weight" rests on probability. He felt a message had plenty information if you couldn't anticipate the contents, but it had little information if the contents were easy to predict. A weather forecast in Los Angeles doesn't contain much information because it is often sunny and 72 degrees Fahrenheit. A weather forecast in the Caribbean during hurricane season, though, has plenty of potential information about coming storms that might be steaming in.

Shannon measured information by totaling up the probabilities. A byte has 8 bits and 256 different possible values between 00000000 and 11111111 in base 2. If all of these possible values occur with the

same probability, then there are said to be 8 bits of information in this byte. On the other hand, if only two values like 00101110 and 10010111 happen to appear in a message, then there is only one bit of information in each byte. The two values could be replaced with just a 0 and a 1 and the entire file would be reduced to one-eighth the size. The number of bits of information in a file is called, in this context, its *entropy*.

Shannon also provided a precise formula for measuring the size of information, a topic found later in Section 2.3. This measurement of information offered some important insights to cryptographers. Mathematicians who break codes rely on deep statistical analysis to ferret out patterns in files. In English, the letter "q" is often followed by the letter "u" and this pattern is a weak point that might be exploited by attackers trying to get at the underlying message. A good encryption program would leave no such patterns in the final file. Every one of the 256 possible values of a byte would occur with equal probability. It would seem to be filled chock-full with information.

One-time pads are an encryption system that is a good example of the basic structure behind information theory. The one-time pad received its name because spies often carried pads of random numbers that served as the encryption key. They would use each sheet once and then dispose of it.

A secret can be split into parts using an extension of one-time pads described on page 58.

A one-time pad can be built by using a standard method of encryption. Assume for the moment that a key is just a number like 5 and a message consists of all uppercase letters. To encrypt a letter like "C" with a key number like 5, count over five letters to get "H". If the counting goes past "Z" at the end of the alphabet, simply go back to "A" and keep going. The letter "Y" encrypted with the key number 6 would produce "E". To decrypt work backward.

Here is a sample encryption:

H	E	L	L	O
9	0	2	1	0
Q	E	N	M	O

In this case, the key is the five numbers 9, 0, 2, 1, and 0. They would constitute the one-time pad that encrypted this message. In practice, the values should be as random as possible. A human might reveal some hidden short circuits in its brain.[1]

Shannon proved that a one-time pad is an unbreakable cipher because the information in the final file is equal to the information in the key. An easy way to see why this is true is to break the message,

[1] Or the limitations of creativity brought on by too much television.

"QENMO" from above. Any five-letter word could be the underlying message because any key is possible. The name, "BRUNO", for instance, would have generated "QENMO" if the key numbers were 15, 13, 19, 25, and 0. If all possibilities are available, then the attacker can't use any of the information about English or the message itself to rule out solutions. The entropy of the message itself should be greater than or equal to the entropy in the key. This is certainly the case here because each byte of the message could be any value between 0 and 255 and so could the key. In practice, the entropy of the key would be even greater because the distribution of the values in the message would depend on the vagaries of language while the key can be chosen at random.

A real one-time pad would not be restricted to uppercase characters. You could use a slightly different encryption process that employed all 256 possible values of a byte. One popular method is to use the operation known as *exclusive-or* (XOR), which is just addition in the world of bits. ($0 + 0 = 0$, $0 + 1 = 1$, and $1 + 1 = 0$ because it wraps around.) If the one-time pad consists of bytes with values between 0 and 255 and these values are evenly distributed in all possible ways, then the result will be secure. It is important that the pad is not used again because statistical analysis of the underlying message can reveal the key. The United States was able to read some crucial correspondence between Russia and its spies in the United States during the early Cold War because the same one-time pad was reused. [Age95] The number of bits in the key was now less than the number of bits of information in the message, and Shannon's proof that the one-time pad is a perfect encryption no longer holds.

The one-time pad is an excellent encryption system, but it's also very impractical. Two people who want to communicate in secret must arrange to securely exchange one-time pads long before they need to start sending messages. It would not be possible, for instance, for someone to use their WWW browser to encrypt the credit card numbers being sent to a merchant without exchanging a one-time pad in person. Often, the sheer bulk of the pad makes it too large to be practical.

Many people have tried to make this process more efficient by using the same part of the pad over and over again. If they were encrypting a long message, they might use the key 90210 over and over again. This makes the key small enough to be easily remembered, but it introduces dangerous repetition. If the attackers are able to guess the length of the key, they can exploit this pattern. They would know in this case that every fifth letter would be shifted by the same amount. Finding the right amount is often trivial and it can be as

easy as solving a crossword puzzle or playing Hangman.

2.2.1 DES and Modern Ciphers

There are many different encryption functions that do a good job of scrambling information into white noise. One of the once practical and secure encryption algorithms still in use today is the Data Encryption Standard (DES) developed by IBM in the 1970s. The system uses only 56 bits of key information to encrypt 64-bit blocks of data. Today, the number of the bits in the key is considered too small because some computer scientists have assembled computers that can try all 2^{55} possible keys in about 48 hours.[Fou98] Newer machines can search all of the keys even faster.

One of the newest and most efficient replacement for DES is the Advanced Encryption Standard, an algorithm chosen by the U.S. government after a long, open contest. The algorithm, Rijndael, came from Joan Daemen and Vincent Rijmen, and narrowly defeated four other highly qualified finalists. [2] [DR00, DR01]

The basic design of most modern ciphers like DES and Rijndael was inspired, in part, by some other work of Claude Shannon in which he proposed that encryption consists of two different and complementary actions: confusion and diffusion. *Confusion* consists of scrambling up a message or modifying it in some non-linear way. The one-time pad system above confuses each letter. *Diffusion* involves taking one part of the message and modifying another part so that each part of the final message depends on many other parts of the message. There is no diffusion in the one-time pad example because the total randomness of the key made it unnecessary.

DES consists of sixteen alternating rounds of confusion and diffusion. There are 64 bits that are encrypted in each block of data. These are split into two 32-bit halves. First, one half is confused by passing it through what is called an "S-box." This is really just a random function that is preset to scramble the data in an optimal way. Then these results are combined with the key bits and used to scramble the other half. This is the diffusion because one half of the data is affecting the other half. This pattern of alternating rounds is often called a *Feistel network*.

The alternating rounds would not be necessary if a different S-box were used for each 64-bit block of the message. Then the cipher would be the equivalent of a one-time pad. But that would be inefficient because a large file would need a correspondingly large set of

[2] Daemen and Rijmen suggest pronouncing the name: "Reign Dahl", "Rain Doll", or "Rhine Dahl".

S-boxes. The alternating rounds are a compromise designed to securely scramble the message with only 64 bits.

The confusion and diffusion functions were designed differently. Confusion was deliberately constructed to be as nonlinear as possible. Linear functions, straight lines, are notoriously easy to predict. The results don't even come close.

Creating a nonlinear S-box is not an easy process. The original technique was classified, leading many to suspect that the U.S. government had installed a trap door or secret weakness in the design. The recent work of two Israeli cryptographers, Eli Biham and Adi Shamir, however, showed how almost linear tendencies in S-boxes could be exploited to break a cipher like DES. Although the technique was very powerful and successful against DES-like systems, Biham and Shamir discovered that DES itself was optimally designed to resist this attack.

The diffusion function, on the other hand, was limited by technology. Ideally, every bit of the 64-bit block will affect the encryption of any other bit. If one bit at the beginning of the block is changed, then every other bit in the block may turn out differently. This instability ensures that those attacking the cipher won't be able to localize their effort. Each bit affects the others.

Figure 2.1 shows how one half of the data encrypts the other half. Alternating which half scrambles the other is a good way to ensure that the contents of one half affect the other. The diffusion in DES is even more subtle. Although the information in one half would affect the other after only one round, the bits inside the halves wouldn't affect each other quite as quickly. This part of the book does not go into the design of the S-boxes in detail, but the amount of scrambling was limited by the technology available in the mid-1970s when the cipher was designed. It takes several rounds of this process to diffuse the information thoroughly.

Figure 2.2 shows one of the eight S-boxes from DES. It is simply a table. If the input to the S-box is 000000 then the output is 1110. This is the most basic form of scrambling and it is fairly easy to reverse. The S-box takes 6 bits as input to implement diffusion. The 32 bits of one half are split into eight 4-bit blocks. Each of the 4-bit blocks then grabs one bit from the block to the left and one bit from the block to the right. That means that each 4-bit block influences the processing of the adjacent 4-bit block. This is how the bits inside each of the halves affect each other.

This is already too much detail for this part of the book. The rest of DES is really of more interest to programmers who actually need to implement the cipher. The important lesson is how the design-

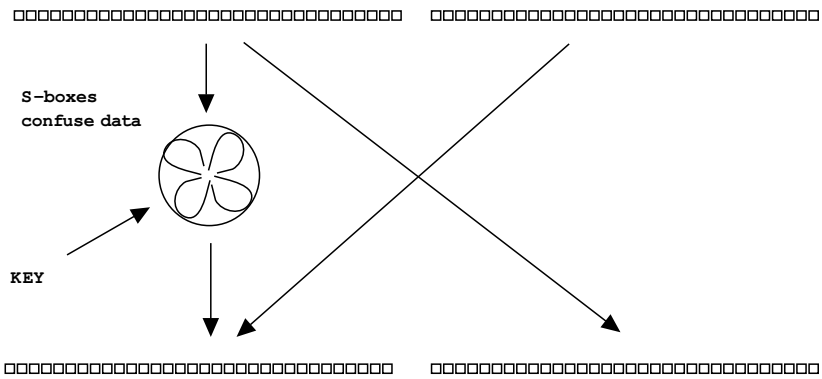


Figure 2.1: A schematic view of one round of DES. 64 bits enter and are split into two 32-bit halves. The left half is scrambled up with the key using the S-boxes. This result is then mixed in with the right half and the result of adding these two together becomes the new left half. The new right half is just a copy of the old left half.

$000000 \rightarrow 1110$	$000001 \rightarrow 0100$	$000010 \rightarrow 1101$	$000011 \rightarrow 0001$
$000100 \rightarrow 0010$	$000101 \rightarrow 1110$	$000110 \rightarrow 1011$	$000111 \rightarrow 1000$
$001000 \rightarrow 0011$	$001001 \rightarrow 1010$	$001010 \rightarrow 0110$	$001011 \rightarrow 1100$
$001100 \rightarrow 0101$	$001101 \rightarrow 1001$	$001110 \rightarrow 0000$	$001111 \rightarrow 0111$
$010000 \rightarrow 0000$	$010001 \rightarrow 1111$	$010010 \rightarrow 0111$	$010011 \rightarrow 0100$
$010100 \rightarrow 1110$	$010101 \rightarrow 0010$	$010110 \rightarrow 1101$	$010111 \rightarrow 0001$
$011000 \rightarrow 1010$	$011001 \rightarrow 0110$	$011010 \rightarrow 1100$	$011011 \rightarrow 1011$
$011100 \rightarrow 1001$	$011101 \rightarrow 0101$	$011110 \rightarrow 0011$	$011111 \rightarrow 1000$
$100000 \rightarrow 0100$	$100001 \rightarrow 0001$	$100010 \rightarrow 1110$	$100011 \rightarrow 1000$
$100100 \rightarrow 1101$	$100101 \rightarrow 0110$	$100110 \rightarrow 0010$	$100111 \rightarrow 1011$
$101000 \rightarrow 1111$	$101001 \rightarrow 1100$	$101010 \rightarrow 1001$	$101011 \rightarrow 0111$
$101100 \rightarrow 0011$	$101101 \rightarrow 1010$	$101110 \rightarrow 0101$	$101111 \rightarrow 0000$
$110000 \rightarrow 1111$	$110001 \rightarrow 1100$	$110010 \rightarrow 1000$	$110011 \rightarrow 0010$
$110100 \rightarrow 0100$	$110101 \rightarrow 1001$	$110110 \rightarrow 0001$	$110111 \rightarrow 0111$
$111000 \rightarrow 0101$	$111001 \rightarrow 1011$	$111010 \rightarrow 0011$	$111011 \rightarrow 1110$
$111100 \rightarrow 1010$	$111101 \rightarrow 0000$	$111110 \rightarrow 0110$	$111111 \rightarrow 1101$

Figure 2.2: This table shows how the first DES S-box converts 6-bit values into 4-bit ones. Note that a change in one input bit will generally change two output bits. The function is also nonlinear and difficult to approximate with linear functions.

ers of DES chose to interleave some confusion functions with some diffusion functions to produce incomprehensible results.

The best way to judge the strength of an encryption system like DES is to try to break it. Talking about highly technical things like code breaking at a high level can be futile because the important details can often be so subtle that the hand-waving metaphors end up flying right over the salient fact. Still, a quick sketch of an attack on the alternating layers of confusion and diffusion in DES can give at least an intuitive feel for why the system is effective.

Imagine that you're going to break one round of DES. You have the 64 bits produced by one step of confusion and one step of diffusion. You want to reconstruct the 64 bits from the beginning and determine the 56 key bits that were entered. Since only one round has finished, you can immediately discover one half of the bits. The main advantage that you have is that not much diffusion has taken place. Thirty-two bits are always unchanged by each round. This makes it easier to determine if the other half could come from the same file. Plus, these 32 bits were also the ones that fed into the confusion function. If the confusion process is not too complicated, then it may be possible to run it in reverse. The DES confusion process is pretty basic, and it is fairly straightforward to go backward. It's just a table lookup. If you can guess the key or the structure of the input, then it is simple.

Now imagine doing the same thing after 16 rounds of confusion and diffusion. Although you can work backward, you'll quickly discover that the confusion is harder to run in reverse. After only one round, you could recover the 32 bits of the left half that entered the function. But you can't get 32 bits of the original message after 16 rounds. If you try to work backward, you'll quickly discover that everything is dependent on everything else. The diffusion has forced everything to affect everything else. You can't localize your search to one 4-bit block or another because all of the input bits have affected all of the other bits in the process of the 16 rounds. The changes have percolated throughout the process.

Rijndael is similar in theme to DES, but much more efficient for modern CPUs. The S-boxes from DES are relatively simple to implement on custom chips, but they are still complicated to simulate with the general purpose CPUs used in most computers. The confusion in AES is accomplished by multiplying by a polynomial and the diffusion occurs when the subblocks of the message block are scrambled. This math is much more basic than the complex S-boxes because the general-purpose CPUs are designed to handle basic arithmetic.

The other four AES finalists can also be shoehorned into this model of alternating rounds of confusion and diffusion. All of them are considered to be quite secure which means they all provide more randomization.

2.2.2 Public-Key Encryption

Public-key encryption systems are quite different from the popular private-key encryption systems like DES. They rely on a substantially different branch of mathematics that still generates nice, random white noise. Even though these foundations are different, the results are still the same.

The most popular public-key encryption system is the RSA algorithm that was developed by Ron Rivest, Adi Shamir, and Len Adleman when they were at MIT during the late 1970s.Ron Rivest, Adi Shamir, and Len Adleman The system uses two keys. If one key encrypts the data, then only the other key can decrypt it. After the encryption, first key becomes worthless It can't decrypt the data. This is not a bug, but a feature. Each person can create a pair of keys and publicize one of the pair, perhaps by listing it in some electronic phone book. The other key is kept secret. If someone wants to send a message to you, they look up your public key and use it to encrypt the message to you. Only the other key can decrypt this message now and only you have a copy of it.

In a very abstract sense, the RSA algorithm works by arranging the set of all possible messages in a long, long loop in an abstract mathematical space. The circumference of this loop, call it n, is kept a secret. You might think of this as a long necklace of pearls or beads. Each bead represents a possible message. There are billions of billions of billions of them in the loop. You send a message by giving someone a pointer to a bead.

The public key is just a relatively large number, call it k. A message is encrypted by finding its position in the loop and stepping around the loop k steps. The encrypted message is the number at this position. The secret key is the circumference of the loop minus k. A message is decrypted by starting at the number marking the encrypted message and marching along the $n - k$ steps. Because the numbers are arranged in a loop, this will bring you back to where everything began– the original message.

Two properties about this string of pearls or beads make it possible to use it for encryption. The first is that given a bead, it is hard to know its exact position on the string. If there is some special first bead that serves as the reference location like on a rosary, then you

would need to count through all of the beads to determine the exact location of one of the beads. This same effect happens in the mathematics. You would need to multiply numbers again and again to determine if a particular number is the one you want.

The second property of the string of beads in this metaphor does not make as much sense, but it can still be easily explained . If you want to move along the string k beads, then you can jump there almost instantaneously. You don't need to count each of the k beads along the way. This allows you to encrypt and decrypt messages using the public-key system.

The two special features are similar but they do not contradict each other. The second says that it is easy to jump an arbitrary number of beads. The first says it's hard to count the number of pearls between the first bead and any particular bead. If you knew the count, then you could use the second feature. But you don't so you have to count by hand.

The combination of these two features makes it possible to encrypt and decrypt messages by jumping over large numbers of beads. But it also makes it impossible for someone to break the system because they can't determine the number of steps in the jump without counting.

This metaphor is not exactly correct, but it captures the spirit of the system. Figure 2.3 illustrates it. Mathematically, the loop is constructed by computing the powers of a number modulo some other number. That is, the first element in the loop is the number. The second is the square of the number, the third is the cube of the number, and so on. In reality, the loop is more than one-dimensional, but the theme is consistent.

2.2.3 How Random Is the Noise?

How random is the output of a encryption function like DES or RSA? Unfortunately, the best answer to that question is the philosophical response, "What do you mean by random?" Mathematics is very good at producing consistent results from well-defined questions, but it has trouble accommodating capricious behavior.

At the highest level, the best approach is indirect. If there was a black box that could look at the first n bits of a file and predict the next set of bits with any luck, then it is clear that the file is not completely random. Is there such a black box that can attack a file encrypted with DES or AES? The best answer is that no one knows of any black box that will do the job in any reasonable amount of time. A brute-force attack is possible, but this requires a large machine and

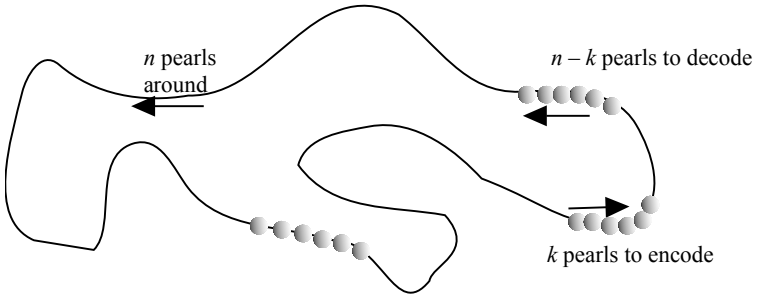


Figure 2.3: RSA encryption works by arranging the possible messages in a loop with a secret circumference. Encryption is accomplished by moving a random amount, k, down the loop. Only the owners know the circumference, n, so they can move $n-k$ steps down the loop and recover the original message.

some insight into the structure of the encrypted file. So we could argue that the results of DES or AES should appear random because we can't predict them successfully.[Way92, Fou98]

The same arguments also hold for RSA. If there was some black box that could take a number and tell you where it stood in the loop, then you would be able to break RSA. If the input doesn't fall in a pattern, then the output should be very random. If there was some way of predicting it, then that could be used to break RSA. Of course, the bits coming out of a stream of RSA-encrypted values are not perfectly random, at least at the level of bits. The values in the output are all computed modulo n so they are all less than n. Since n is not a power of 2, some bits are a little less likely.

Even if the values can't be predicted, they still might not be as random looking as we might want. For instance, an encrypted routine might produce a result that is uncrackable but filled with only two numbers like 7 and 11. The pattern might be incomprehensible and unpredictable, but you still wouldn't want to use the source as the random number generator for your digital craps game. One immediate clue is that if the 7 and the 11 occur with equal probability, then the entropy of such a file is clearly 1 bit per number.

It is easy to construct a high-level argument that this problem will not occur with DES. All possible output values should be produced with equal probability. Why? Because DES can be decoded successfully. 64 bits go into DES and 64 bits go out. Each possible output can have only one matching input and vice versa. Therefore each possible output can be produced.

The same argument also holds for RSA. The loop contains a number for each of all possible messages and these numbers are distributed around the loop in a way that we can't invert. Therefore, each output value has practically the same probability of emerging from the function.

Although these two arguments don't prove that the output from an encryption function is random, they do suggest that DES and RSA will pass any test that you can throw at them. If a test is good enough to detect a pattern, then it would be a good lever for breaking the code. In practice, the simple tests support these results. The output of DES is quite random.[3] Many tests show that it is a good way to "whiten" a random number source to make it more intractable. For instance, some people experiment with using a random physical process like counting cosmic rays to create random numbers. However, there might be a pattern caused by the physics of the detector. A good way to remove this possibility is to use DES to encrypt the random data and produce the whitest noise possible.

2.3 Measuring and Encrypting Information

Information is a slippery notion. Just how big is a fact? How much data must be accumulated before you have a full-fledged concept? None of these questions are easy to answer, but there are approximations that help with digital data. Shannon's measure of information is closely tied to probability and randomness. In a sense, information is defined by how much randomness it can remove. Our goal is to harness randomness and replace it with a hidden message. Knowing the size, length, depth or breadth of our target is a good beginning.

Let an information stream be composed of n characters between x_0 and x_{n-1} that occur in the stream with probability $\rho(x_i)$. Shannon's measure of the *entropy* in the information stream, that is the number bits per character, can be written:

$$\sum_{i=0}^{n-1} \rho(x_i) \log\left(\frac{1}{\rho(x_i)}\right).$$

The $\log$ is taken base two.

[3] The level of randomness depends on the input file if there is no key feedback mechanism being used. In some versions of DES, the results of one block are XORed with the inputs for the next block so that there will be diffusion across the blocks. If this is not used, someone could input a file with a pattern and get out a file with a pattern as long as the pattern repeats in an even multiple of 8 bytes.

If a stream is made up of bytes with values between 0 and 255 and every byte value occurs with equal probability of $\frac{1}{256}$, then the entropy of the stream is 8 bits per byte. If only two bytes, say 43 and 95, each occur half of the time and the other 254 bytes don't occur at all, the entropy of this stream is only 1 bit per byte. In this basic example, it should be obvious how the bit stream can be compressed by a factor of 8 to 1 bit per character. In more complex examples, the entropy is still a good rough measure of how well a basic compression algorithm will do.

The limitations of Shannon's measure of information are pretty obvious. An information stream that repeats the bytes $0, 1, 2, \ldots, 254, 255, 0, 1 \ldots$ ad infinitum would appear to contain 8 bits of information per byte. But, there really isn't that much information being conveyed. You could write a short two-line program in most computer languages that would duplicate the result. This computer program could stand in for this stream of information and it would be substantially cheaper to ship this program across the network than it would be to pay for the cost of sending an endless repeat stream of bytes.

In a sense, this repeating record computer program is a good compressed form of the information. If the data was potato chips, you would hope that it was measured by the number of lines in a computer program that could generate it, *not* the Shannon entropy. There is another measure of information known as the *Kolmogorov* complexity that attempts to measure the information by determining the size of the smallest program that could generate the data. This is a great theoretical tool for analyzing algorithms, but it is entirely impractical. Finding the smallest program is both theoretically and practically impossible because no one can test all possible programs. It might be a short program in C, but how do we know the length in Pascal, Smalltalk, or a language that no one has written yet?

The Shannon measure of information can be made more complicated by including the relationship between adjacent characters:

$$\sum_{i,j} \rho(x_i|x_j) \log \left(\frac{1}{\rho(x_i|x_j)} \right).$$

$\rho(x_i|x_j)$ means the probability that x_i follows x_j in the information stream. The sum is computed over all possible combinations. This measure does a good job of picking up some of the nature of the English language. The occurrence of a letter varies significantly. "h" is common after a "t" but not after a "q". This measure would also pick up the pattern in the example of $0, 1, 2, \ldots, 255, 0, 1, \ldots$

But there are many slightly more complicated patterns that could

be generated by a computer program yet confound this second-order entropy calculation. Shannon defined the entropy of a stream to include all orders up to infinity. Counting this high may not be possible, but the higher order terms can usually be safely ignored. While it may be practical to compute the first- or second-order entropy of an information stream, the amount of space devoted to the project obviously becomes overwhelming. The number of terms in the summation grows exponentially with the order of the calculation. Shannon created several experimental ways for estimating the entropy, but the limits of the model are still clear.

2.3.1 RSA Encryption

The section "Encryption and White Noise" on page 20 described RSA encryption with the metaphor of a long circle of beads. Here are the equations. The system begins with two prime numbers p and q. Multiplying p and q together is easy, but no one knows of an efficient way to factor $n = pq$ into its components p and q if the numbers are large (i.e., about 1024 to 2048 bits).

This is the basis of the security of the system. If you take a number x and compute the successive powers of x, then $x^{\phi(n)} \bmod pq = x$.[4] That is, if you keep multiplying a number by x modulo pq, then it returns to x after $\phi(pq) + 1$ steps.

A message is encrypted by treating it as the number x. The sender encrypts the number x by multiplying it by itself e times, that is computing $x^e \bmod pq$. The receiver decrypts the message by multiplying it by itself d times, that is computing $(x^e)^d \bmod pq$. If $d \times 3 = \phi(x)$, then the result will be x.

This $\phi(n)$ is called the *Euler Totient* function and it is the number of integers less than n that are relatively prime to n. If n is a prime number then $\phi(n)$ is $n - 1$ because all of the integers less than n are relatively prime to it. The values are commutative so $\phi(pq) = \phi(p)\phi(q)$. This means that $\phi(pq) = pq - p - q + 1$. For example, $\phi(15) = 8$. The numbers $1, 2, 4, 7, 8, 11, 13$ and 14 are relatively prime to 15. The values $3, 5, 6, 9, 10$ and 12 are not.

Calculating the value of $\phi(pq)$ is easy if you know both p and q, but no one knows an efficient way to do it if you don't. This is the basis for the RSA algorithm. The circumference of this string of pearls or beads is $\phi(pq)$. Moving one pearl or bead along the string is the equivalent of multiplying by x.

[4] $x \bmod y$ means the remainder after x is divided by y. So $9 \bmod 7$ is 2, $9 \bmod 3$ is 0.

The two keys for the RSA are chosen so they both multiply together to give 1 modulo $\phi(pq)$. One is chosen at random and the other is calculated by finding the inverse of it. Call these e and d where $de = 1 \bmod \phi(pq)$. This means that:

Neal Koblitz's book, [Kob87], gives a good introduction to finding this inverse.

$$x^{ed} \bmod pq = x.$$

This can be converted into an encryption system very easily. To encrypt with this public key, calculate $x^e \bmod pq$. To decrypt, raise this answer to the d power. That is, compute:

$$(x^e \bmod pq)^d \bmod pq = x^{de} \bmod pq = x.$$

This fulfills all of the promises of the public-key encryption system. There is one key, e, that can be made public. Anyone can encrypt a message using this value. No one can decrypt it, however, unless they know d. This value is kept private.

The most direct attack on RSA is to find the value of $\phi(pq)$. This can be done if you can factor pq into p and q.

Actually implementing RSA for encryption requires attention to a number of details. Here are some of the most important ones in no particular order:

Converting Messages into Numbers Data is normally stored as bytes. RSA can encrypt any integer that is less than pq. So there needs to be a solid method of converting a collection of bytes into and out of integers less than pq. The easiest solution is to glue together bytes until the string of bytes is a number that is greater than pq. Then remove one byte and replace it with random bits so that the value is just less than pq. To convert back to bytes, simply remove this padding.

The equations here make it easy to describe RSA, but they aren't enough to make it easy to build a working implementation. Dan Boneh, Antoine Joux, and Phong Q. Nguyen found major weaknesses in naive solutions for converting a message into a number. [BJN00]

Fast Modulo Computation Computing $x^e \bmod pq$ does not require multiplying x together e times. This would be prohibitive because e could be quite large. An easier solution is to compute $x, x^2 \bmod pq, x^4 \bmod pq, x^8 \bmod pq, \ldots$ That is, keep squaring x. Then choose the right subset of them to multiply together to get $x^e \bmod pq$. This subset is easy to determine. If the i^{th} bit of the binary expansion of e is 1, then multiply in $x^{2^i} \bmod pq$ into the final answer.

[BFHMV84], [Bri82], [Mon85], and [QC82] discuss efficient multiplication algorithms.

Finding Large Prime Numbers The security of the RSA system depends on how easy it is to factor pq. If both p and q are large prime numbers, then this is difficult. Identifying large prime numbers as luck would have it, is pretty easy to do. There are

a number of tests for primality that work quite well. The solution is to choose a large, odd number at random and test it to see if it is prime. If it isn't, choose another. The length of time it takes to find a prime number close to an integer x is roughly proportional to the number of bits in x.

The Lehman test [Leh82] is a good way to determine if n is prime. To do so, choose a random number a and compute $a^{(n-1)/2} \bmod n$. If this value is not 1 or -1, then n is not prime. Each value of a has at least a 50% chance of showing up a non-prime number. If we repeat this test m times, then we're sure that we have a 1 in 2^m chance that n is not prime, but we haven't found an a that would prove it yet. Making $m = 100$ is a good starting point. It is not absolute proof, but it is good enough.

RSA encryption is a very popular algorithm used for public-key encryption. There are also a large number of other algorithms that are available. The discussion of these variants is beyond the scope of this book. Both Bruce Schneier's book, [Sch94], and Gus Simmons' book [ed.92] offer good surveys.

2.4 Summary

Pure encryption algorithms are the best way to convert data into white noise. This alone is a good way to hide the information in the data. Some scientists, for instance, encrypt random data to make it even more random. Encryption is also the basis for all of the other algorithms used in steganography. The algorithms that take a block of data and hide it in the noise of an image or sound file need data that is as close to random as possible. This lowers the chance that it can be detected.

Of course, nothing is perfect. Sometimes data that is too random can stick out too. Chapter 17 describes how to find hidden information by looking for values that are more random than they should be.

The Disguise Good encryption turns data into white noise that appears random. This is a good beginning for many algorithms that use the data as a random source to imitate the world.

How Secure Is It? The best new encryption algorithms like Rijndael and the other four AES finalists have no practical attack known to the public. These algorithms are designed and evaluated on their ability to resist attack. DES is no longer very secure for serious applications.

How to Use It? Encryption code can be downloaded from a number of places on the Net.

Further Reading

- *Applied Cryptography* by Bruce Schneier is a good general introduction to the subject. It includes good summaries of the major algorithms. [Sch94]
- *Handbook of Applied Cryptography* by Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone is a good technical discussion of many of the most important algorithms as well as their mathematical foundations. [MvV97]
- The proceedings from the various conferences sponsored by the International Association of Cryptologic Research (IACR) offer some of the most timely insights into the best open research in the field. See `iacr.org`.
- It's impossible to summarize all of the hard work that cryptographers have done to create linear approximations of non-linear encryption functions. Adi Shamir did a good job *and* extended the techniques developed by many others in his talk given at Crypto 2008. He describes a sophisticated algebraic technique that can be applied to many of the most common algorithms used today.[Sha08]

Chapter 3

Error Correction

3.1 Close but No Cigar

1. Speedwalking.
2. America OnLine, CompuServe and Prodigy.
3. Veggie burgers.
4. Using a Stairmaster.
5. Winning the Wild Card pennant.
6. Driving 55 mph.
7. Living in suburbia.
8. New Year's Resolutions.
9. Lists as poetry.
10. Lists as a simple way to give structure to humor.
11. Cigarettes.

3.2 Correcting Errors

The theory of computers rests on an immutable foundation: a bit is either on ("1") or off ("0"). Underneath this foundation, however, is the normal, slightly random, slightly chaotic world in which humans spend their time. Just as the sun is sometimes a bit brighter than

usual and sometimes it rains for a week straight, the physics that govern computer hardware are a bit more random. Sometimes the spot on the hard disk responsible for remembering something doesn't behave exactly perfectly. Sometimes an alpha particle from outer space screams through a chip and changes the answer.

Computer designers manage to corral all of this randomness through a mixture of precision and good mathematics. Clean machines eliminate the dirt that screws things up and the math of error-correcting codes is responsible for fixing up the rest of the problems that slip through. This mathematics is really one of the ideas that is most responsible for the digital explosion by making it possible to build a digital circuit with a bit of sloppiness that can never be present in an analog world. Designers know that the sloppiness can be fixed by a bit of clever mathematics.

Error-correcting codes can be used effectively to hide information in a number of important ways. The most obvious is to just introduce small errors into a file in an organized fashion. If someone tries to read the file with ordinary tools, the error correction patches up these changes and no one is the wiser. More sophisticated tools could find these changes by comparing the original file with the cleaned-up version or simply using the error-correcting principles to point the location. The message could be encoded in the position of the errors.

A high number of errors might indicate the existence of a message. Chapter 17 describes how to build statistical models to detect abnormal patterns like a high error rate.

Some music CD manufacturers are taking advantage of the differences between the error-correcting mechanisms in computers and CD players. Strategically placed errors will be corrected by a CD player but labeled as disk failure by the mechanisms in computers.

Page 178 shows how to construct a system using random walks. Ross Anderson, Roger Needham, and Adi Shamir used a similar approach to hide information in their steganographic file system. [ANS98]

Error-correcting codes can also be used to help two people share a channel. Many semi-public data streams make ideal locations to hide information. It might be possible to insert bits in a photograph or a music file that floats around on the Internet by grabbing it and replacing it with a copy that includes your message. This works well until someone else has the same idea. Suddenly one message could overwrite another. An ideal solution is to arrange it so no one took up more than a small fraction of a channel like this one. Then, they would write their information with an error-correcting code. If two messages interacted, they would still only damage a fraction of each other's bits and the error-correcting code would be used to fix it. This is how the codes are used in many radio systems.

Of course, error-correcting codes can also help deal with errors introduced by either attackers, a noisy channel, or a format conversion. Some web sites reduce the size of images or apply subtle color

corrections. In some cases, these modifications introduce only a few changes to the file and the error-correcting codes can recover them. This additional strength is hard to measure because there is no easy way to predict the damage waiting for the file.

On occasion, it makes sense to split a message into a number of different parts to be shipped through different channels. Ideally, the message could be reconstructed if a few of the parts have been compromised along the way. The part could either be lost or scrambled by a malicious courier. In either case, error-correcting codes can defend against this problem.

Better secret splitting solutions are found in Chapter 4.

A system of error-correcting codes comes in any number of flavors. Many of the most commonly used codes have the ability to carry k bits in a packet of n bits and find the right answer if no more than m errors have been made. There are many possible codes that come with different values of k, n, and m, but you never get anything for free. If you have 7 bits and you want each block to carry at least 4 bits of information, then one standard code can only correct up to one error per block. If you want to carry 6 bits of information in a 7-bit block, then you can't successfully correct errors and you can only detect them half of the time.

The best metaphor for understanding error-correcting codes is to think about spheres. Imagine that each letter in a message is represented as the center of a sphere. There are 26 spheres for each letter and none of them overlap. You send a message by sending the coordinates to this point at the center. Occasionally a transmission glitch might nudge the coordinates a bit. When the recipient decodes the message, he or she can still get the correct text if the nudges are small enough so the points remain inside the sphere. The search for the best error-correcting codes involves finding the best way to pack the spheres so that you can fit the most spheres in a space and transmit the most characters.

Although mathematicians talk about sphere packing on an abstract level, it is not immediately obvious how this applies to the digital world where everything is made up of binary numbers that are on or off. How do you nudge a zero a little bit? If you nudge it enough, when does it becomes a one? How do you nudge a number like 252, which is 11111100 in binary? Obviously a small nudge could convert this into 111111101, which is 253. But what if the error came along when the first bit was going through the channel? If the first bit was changed the the number would become 011111100. That is 114, a change of 128, which certainly doesn't seem small. That would imply that the spheres really couldn't be packed too closely together.

The solution is to think about the bits independently and to mea-

sure the distance between two numbers as the number of different bits. So 11111100 and 11111101 are one unit apart because they differ in only one bit. So are 11111100 and 01111100. But 01111100 and 11111101 are two units apart. This distance is often called the *Hamming distance*.

This measure has the same feel as finding the distance between two corners in a city that is laid out on a Manhattan-like grid. The distance between the corner at 10th Avenue and 86th Street and the corner at 9th Avenue and 83rd Street is four blocks, although in Manhattan they are blocks of different lengths. You just sum up the differences along each of the different dimensions. In the street example, there are two dimensions that are the avenues that run north and south or the streets that run east and west. In the numerical example, each bit position is a different dimension and the 8-bit examples above have eight dimensions.

Error-correcting codes spread the information out over a number of bits in the same fashion as the spread-spectrum algorithms in Chapter 14.

The simplest example of an error-correcting code uses 3 bits to encode each bit of data. The code can correct one error in a bit but not two. There are eight possible combinations of three bits: 000, 001, 010, 011, 100, 101, 110, and 111. You can think of these as the eight corners of a cube as shown in Figure 3.1. A message 0 can be encoded as "000," and a 1 can be encoded as "111". Imagine there is an error and the "000" was converted into a "001". The closest possible choice, "000", is easy to identify.

The sphere of "000" includes all points that are at most one Hamming unit away: 001, 010, and 100. Two errors, however, nudge a point into the adjacent sphere.

Obviously, the technique can be extended into higher-dimensional spaces. The trick is to find an optimal number of points that can be packed into a space. Imagine, for instance, a five-dimensional space made up of the points 00000, 00001, $00010, \ldots, 11111$. Every point has an opposite point that is five units away from it. 00000 is five steps away from 11111 and 10111 is five units away from 01000. It is easy to construct a sphere with a radius of two units around each point. That means 0 can be encoded as 00000 and 1 can be encoded as 11111. Up to two errors could occur and the correct answer would be found. 10110 is two units away from 11111, so it would fall in its sphere of influence and be decoded as a 1.

Generally, odd-dimensional spaces are much better than even-dimensional spaces for this basic scheme. Imagine the six-dimensional space created from the points 000000, 000001, $000010, \ldots, 111111$. Both 000000 and 111111 are six units apart. But if you draw a sphere of radius 3 around each point, then the spheres overlap. The point 010101, for instance, is both three units away from 000000 and three

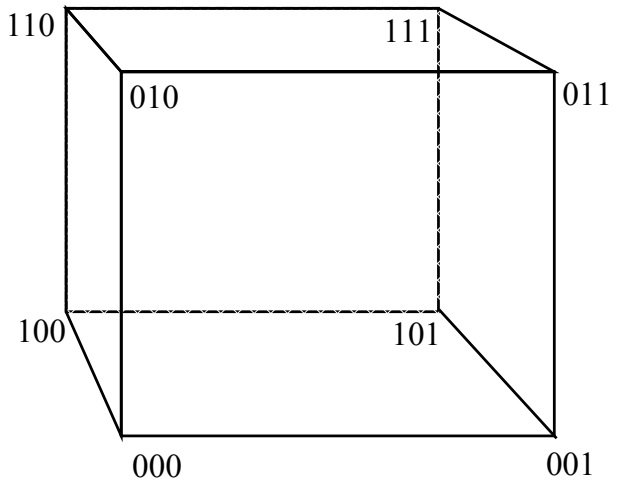


Figure 3.1: The eight corners of the cube. The two corners, 000 and 111, are used to send the message of either 0 or 1. If there is an error in one bit, then it can be recovered by finding the closest corner.

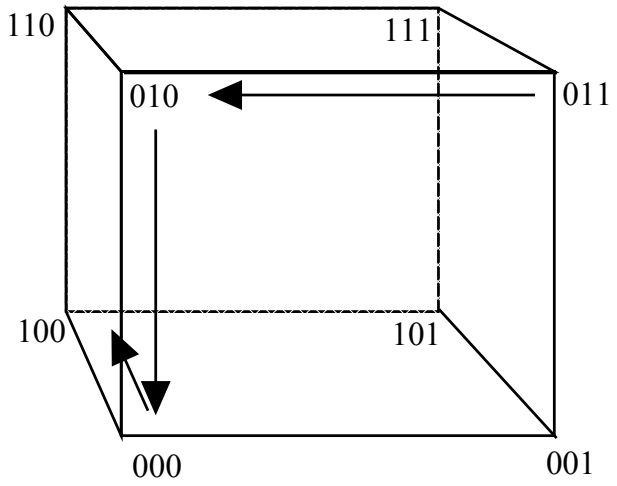


Figure 3.2: The Hamming distance shows that the corner “011” is three steps or units away from “100”. That’s the longest distance in this cube.

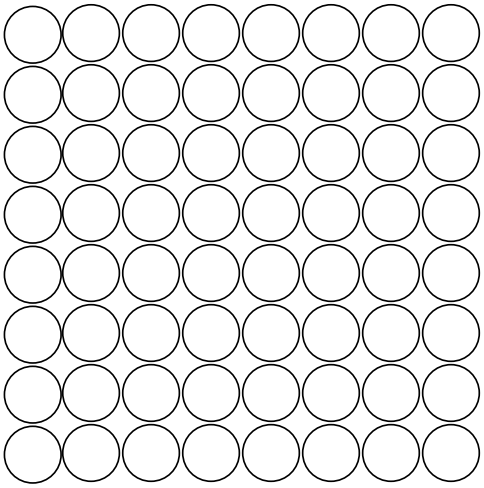


Figure 3.3: A poor way to pack circles. If the system error can't shift the signal more than the radius of the sphere, then the space between the circles is wasted. Figure 3.4 shows a better approach.

units away from 111111. It's in both spheres. If you were to try and construct an error-correcting code using this arrangement, then you would only be able to fit two spheres of radius 2 in the space and the code would only be able to resist up to two errors per block. Obviously the 5-bit code in the five-dimensional space is just as error-resistant while being more efficient.

There is no reason why you need to pack only two spheres into each space. You might want to fit in many smaller spheres. In seven-dimensional space, you can fit in two spheres of radius 3 centered around any two points that are seven units apart. But you can also fit in a large number of spheres that have a radius of only 1. For instance, you can place spheres with a single unit radius around 0000000, 0000111, 1110000, 0011001, 1001100, 1010001, and 1000101. None of these spheres overlap and the space is not full. You could also add a sphere centered around 1111110. There are eight code words here, so eight different messages or 3 bits of information could be stored in each 7-bit code word. Up to one bit error could be found and resolved.

In general, packing these higher-dimensional spaces is quite difficult to do optimally. It should be clear that there are many other points that are not in any of eight different spheres. This reflects a gross inefficiency.

"Constructing Error-Correcting Codes" on page 46 describes how

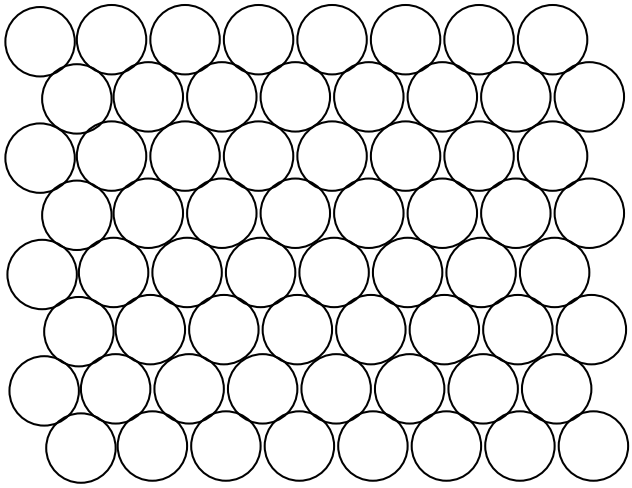


Figure 3.4: A better approach to packing the circles from Figure 3.3. This is about 86.6% of the original height. It is wider by one half of the radius of a circle.

to build general Hamming codes. It is possible to use the algorithm given there to construct an error-correcting code that packs 4 bits of information, or 16 different messages, into one 7-bit code word. That's one extra bit of data. The code can also resist up to 1 bit of error. The 16 centers generated by this method are:

0000000	0001111	0010011	0011100
0100101	0101010	0110110	0111001
1000110	1001001	1010101	1011010
1100011	1101100	1110000	1111111

There are many other types of error-correcting codes. The metaphor of sphere packing is a good way to understand the basic idea, but it offers little guidance on how it can be done effectively. It is easy to imagine stacking pool balls in a rack, but it is impossible to visualize how to do this in multiple dimensions—especially if the Hamming distance is used.

In practice, error-correcting codes rest on algorithms that take the data and add extra *parity bits* that can be used to recover the data. The parity bits "puff up" the data into more dimensions and move the points away from each other. For instance, in four dimensions the points 1110 and 1111 are right next to each other. But if three parity bits are added to the end of each one, the results, 1110000 and 1111111, are four units apart.

If you look carefully at the table on page 43, the first four bits represent all of the possible points in a four-dimensional space. The last three bits are parity bits that were added to puff it out into seven dimensions.

The location of the parity bits varies significantly between different codes. Some codes can correct a certain number of errors that occur anywhere in a block of data. Others can correct errors that happen in bursts. They might be able to correct only one burst of errors, but that burst can contain anywhere between one flipped bit and an upper limit of k. If the errors don't occur next to each other, however, then the code can't fix the error. Each of these codes places the parity bits in different arrangements to grab different types of errors.

The rest of this book will rely on error-correcting codes to add robustness to protocols, perhaps add randomness, and provide another way to split information into a number of different parts. Using error-correcting codes is essential if information might bump into other information in the channels.

3.2.1 Error Correction and White Noise

Error-correcting codes may be intended to correct errors, but they can also be used to make a bit stream conform to some pattern. Once a collection of bits is encoded in an error-correcting code then changes can be introduced without destroying the underlying information. These changes might add randomness or, with some difficulty, make the data conform to a pattern.

In practice, the best choice for this approach is error-correcting codes that can sustain a high number of errors. A good first choice might be the 3-bit error-correcting code that conveys one bit. You write 000, 001, 010, or 100 for the 0 bit and 111, 110, 101, or 011 for the 1 bit. Any of the three are acceptable choices. This will triple the size of the file, but it will allow plenty of flexibility in rearranging the structure of the data.

Adding randomness is easy, but there are limitations to making the data fit some other pattern. Obviously the underlying data must come close enough to the pattern so that errors can be introduced successfully. In an abstract sense, the pattern must fall into the spheres. The bigger the spheres, the more patterns that can work successfully. For instance, you could easily use the 3-bit code described above to produce a bit stream that never had more than three ones or three zeros occur in a row. Each bit could be encoded with a pattern that started with a 1 or a 0. On the other hand, you could not produce a pattern that always requires that there were five ones

or zeros in a row.

This technique can be taken one step further that takes it outside of the realm of error-correcting codes entirely. If you're planning to use the error-correcting codes to give you enough room to add some randomness to the data, then you're going to lose many of the error-correcting properties. For instance, one flipped bit can convert 110, a value representing 1, into 010, a value representing 0. In essence, you might want to forget about the error-correcting ability altogether and just construct a list of codes that represent each bit. 1 might be encoded as 001, 100, 011, and 111, while 0 may be encoded as 110, 101, 010, and 000. The main advantage of this approach is that the distribution of zeros and ones can be even more balanced. In the 3-bit code used as an example in this section, there are an average of 2.25 bits used to encode a 1 and .75 used to encode a 0. This means a file with a high percentage of ones, for instance, will still have a high percentage after the encoding. Using random codes assigned to each bit can remove this bias.

3.2.2 Error Correction and Secret Sharing

Error-correcting codes have a functional cousin known as *secret sharing*– that is, a class of algorithms that allow a file be split into m parts so that only $m - k$ parts are necessary to reconstruct it. Obviously, an error-correcting code that could handle up to k errors in m bits would works similarly. Simply encode the file using this method and then break up the m bits into m different files.

Secret sharing is described in detail in Chapter 4.

There is one problem with this approach. Some bits are more privileged than others in some error-correcting schemes. For instance, the next section on Hamming codes describes a code that takes 11 bits and adds 4 parity bits that will correct any single error. Ideally, a file encoded with this code could be broken into 15 parts and any 14 parts would suffice to recover the data. But, there are only 11 bits of data in every block of 15 bits. The other 4 parity bits are used just to correct the errors. If the i^{th} bit of each block always goes in the i^{th} part, then the right 11 parts would suffice. The key is to distribute the bits so this never happens. Here are the steps:

1. Choose an error-correcting code that offers the right recovery properties. It is easy to find Hamming codes that recover single errors.

2. Encode the file using this technique.

3. If there are n bits in each block and n files, then place one bit from each block in each file. That is, place bit i in file $i +$

$j \bmod n$. The choice of j should vary with each block. It can either increase sequentially or be chosen by a random number generator. If a random number generator is used, it should be a pseudo-random number generator that can be reseeded to recover the information later.

For most practical purposes, error-correcting codes are not ideal ways to share secrets. While it is easy to construct a Hamming code that can sustain one error, it is pretty inefficient to generate an error-correcting code that contains n bits per block and still survive, say, $n-2$ errors. The theory is not optimized around this solution and, in fact, the approach presented in this chapter can't detect more than $\frac{n}{2}$ errors.

A better secret-sharing technique emerges directly from geometry. Imagine that you encode a message as a point in a plane. One solution is to draw three lines through the point and distribute the lines to different people. Two lines are enough to reconstruct the point. The process can be turned into an error-correcting code just by choosing the one point that represents the largest intersection of lines. If you want to encode larger amounts of data, you can use higher-dimensional spaces and use planes or higher dimensions. This is close to what the Hamming codes are doing, but it is difficult to think in these terms when only bits are being used.

3.3 Constructing Error-Correcting Codes

[Ara88] and [LJ83] were the source for this material.

Hamming codes are easy and elegant error-correcting codes. Constructing them and using them is relatively easy. The problem can be thought of as taking your incoming message bits and then adding parity bits that will allow you to correct the errors. The net effect is to create an overdetermined collection of linear equations that can be solved in only one way.

Section 4.2 shows how to use error-correcting codes as a way to share responsibility or split a secret into multiple parts. Better algorithms follow.

The easiest way to introduce the algorithm is by constructing an example code that takes 11 bits and adds 4 new parity bits to the mix so that an error of at most one bit can be corrected if it occurs. The input bits will be $a_1, \ldots, a_{11}$. The output bits are $b_1, \ldots, b_{15}$. For the purpose of illustrating the algorithm, it is easier to use binary subscripts: b_{0001} through b_{1111}.

The best way to illustrate the process is with a table of the output bits. The input bits are merely copied over into an output slot with a different number. This is easy to do in hardware if you happened to be implementing such an algorithm in silicon. The extra parity bits

Table 3.1: Output Bit: Where It Comes From

b_{0001}	$a_1 + a_2 + a_4 + a_5 + a_7 + a_9 + a_{11} \bmod 2$
b_{0010}	$a_1 + a_3 + a_4 + a_6 + a_7 + a_{10} + a_{11} \bmod 2$
b_{0011}	a_1
b_{0100}	$a_2 + a_3 + a_4 + a_8 + a_9 + a_{10} + a_{11} \bmod 2$
b_{0101}	a_2
b_{0110}	a_3
b_{0111}	a_4
b_{1000}	$a_5 + a_6 + a_7 + a_8 + a_9 + a_{10} + a_{11} \bmod 2$
b_{1001}	a_5
b_{1010}	a_6
b_{1011}	a_7
b_{1100}	a_8
b_{1101}	a_9
b_{1110}	a_{10}
b_{1111}	a_{11}

are computed by adding up different sets of the input bits modulo 2. They are found in output bits b_{0001}, b_{0010}, b_{0100}, and b_{1000}.

Errors are detected by calculating four formulas that will give the location of the error:

$$\begin{aligned}
c_0 &= b_{0001} + b_{0011} + b_{0101} + b_{0111} + b_{1001} + b_{1011} + b_{1101} + b_{1111} \bmod 2 \\
c_1 &= b_{0010} + b_{0011} + b_{0110} + b_{0111} + b_{1010} + b_{1011} + b_{1110} + b_{1111} \bmod 2 \\
c_2 &= b_{0100} + b_{0101} + b_{0110} + b_{0111} + b_{1100} + b_{1101} + b_{1110} + b_{1111} \bmod 2 \\
c_3 &= b_{1000} + b_{1001} + b_{1010} + b_{1011} + b_{1100} + b_{1101} + b_{1110} \\
&\quad + b_{1111} \bmod 2
\end{aligned}$$

These four equations yield 4 bits. If they're combined into a single number, then they'll reveal the location of an error. For instance, imagine that bit b_{1011} was flipped by an error. This is the incoming bit a_7 and this bit is part of the equation that produces parity bits b_{1000}, b_{0010}, and b_{0001}. The pattern should be obvious. The parity bits are stuck at slots that have only a single 1 in the binary value of their subscript. A normal bit is added into the equation by examining the binary value of its subscript. If there is a 1 at position i, then it is added into the parity bit that has a 1 at position i. b_{1011} has three 1's, so it ends up in four equations.

The effect of an error in b_{1011} is easy to follow. b_{0001} will not match the sum $b_{0011} + b_{0101} + b_{0111} + b_{1001} + b_{1011} + b_{1101}$. This will mean that c_0 will evaluate to 1. The same effect will set $c_1 = 1$ and $c_3 = 1$. c_2 will

stay zero. If these are combined in the proper order, 1011, then they point directly at bit b_{1011}.

These equations also correct errors that occur in the parity bits. If one of these is flipped, only one of the equations will produces a 1. The rest yield zeros because the parity bits are not part of their equations.

The general steps for constructing such an error-correcting code for n bits can be summarized:

1. Find the smallest k such that $2^k - k - 1 \leq n$. This set of equations will encode $2^k - k - 1$ bits and produce $2^k - 1$ bits.

2. Enumerate the output bits with binary subscripts: $b_{00\ldots01} \ldots b_{11\ldots11}$.

3. The parity bits will be the output bits with a single 1 in their subscript.

4. Assign the input bits to the nonparity output bits. Any order will suffice, but there is no reason not to be neat and do it in order.

5. Compute the parity bit with a 1 at position i by adding up all of the output bits with a 1 at the same position, i, *except* the parity bit itself. Do the addition modulo 2.

6. To decode, compute c_i which is the sum of all output bits that have a 1 in position i, *including* the parity bit. This will yield a 0 if the parity bit matches and a 1 if it doesn't. Aggregating the c_i values will reveal the position of the error. This code will only detect one error.

What is the most efficient choice of k for this algorithm? Given that the number of parity bits is proportional to the log of the number of input bits, it is tempting to lump the entire file into one big block and use only a small number of parity bits. This requires a large number of additions. There are about $\frac{n \log n}{2}$ additions in a block of n bits. Large blocks require fewer parity bits but need more computation. They also correct only one error in the entire block and this substantially limits their usefulness. The best trade off must be based on the noisiness of the channel carrying the information.

Implementations of Hamming codes like this one are often fastest when they are done a word at a time. Most CPUs have instructions that will do a bitwise XOR of an instruction word, which is usually either 32 or 64 bits long. XOR is addition modulo 2. These fast XORs

provide a good way of computing up to 32 or 64 encodings in parallel. This is done by using all of the above equations, but doing the calculations with words instead of bits and XOR instead of basic arithmetic.

This approach is a very fast way to encode the error-correcting bits and it is a quick way to detect errors, but correcting the error can be slow. Testing for errors can be done just by seeing if all of the c_i values are zero. If one of the c_i is not zero, then the code must step through each of the bits individually and compute the location of the errors. This is much slower, but not any slower than computing the code in a bitwise fashion.

The Hamming codes described in this section are particularly elegant, in my opinion, because of the way that the results of the c_i are aggregated to find the location of the error. This is just a result of the arrangements of the parity bits. The same basic algorithm could be used no matter what order the bits were found. Any permutation of the bits b_{0001} through b_{1111} would work. The recovery process wouldn't be as elegant.

This elegant arrangement is not necessary for hardware-based implementations because the correction of the error does not need to be done by converting the c_i values into an index that points to the error. It is quite possible to create a set of AND gates for each bit that looks for a perfect match. This means the parity bits could be placed at the beginning or the end of each block. This might simplify stripping them out.

3.3.1 Periodic Codes

The codes described in the previous section correct only one bit error per block. This may suffice, but it can be pretty inefficient if the block sizes are small. The Hamming codes need three parity bits to correct one error in four bits. That's almost a 50% loss just to correct one bit out of four.

The Hamming codes are also less than optimal because of the nature of the noise that can corrupt digital data. The errors may not be randomly distributed. They are often grouped in one big burst that might occur after an electrical jolt or some other physical event disrupts the stream. A scratch on a CD-ROM may blur several bits that are right next to each other. These errors would destroy any Hamming solution that is limited to correcting one bit in each block.

Periodic codes are a better solution for occasions that demand detecting and recovering errors that occur in bursts. In this case, the parity bits will be distributed at regular intervals throughout the

stream of bits. For instance, every fourth bit in a stream might be a parity bit that is computed from some of the previous bits. As before, the location of the parity bits can be varied if the number of parity bits per set of bits is kept constant, but it is often cleaner to arrange for them to occur periodically.

The Hamming codes are designed to work with predefined blocks of bits. The convolutional codes described here will work with rolling blocks of bits that overlap. The same convolutional technique will also work with fixed blocks, but it is left out here for simplicity. To avoid confusion, this section will use the word *subblock* to refer to the smaller sets of bits that are used to create the rolling block.

The periodic code will consist of a subblock of bits followed by a set of parity bits that are computed from the bits that are present in any number of the preceding subblocks. The parity might also depend on some of the bits in the following subblocks, but this configuration is left out in the interest of simplicity.

A set of bits from a convolutional code might look like this:

$$b_{(i,1)},\ b_{(i,2)},\ b_{(i,3)},\ b_{(i,4)},\ b_{(i,5)},\ p_{(i,1)}.$$

Here, $b_{(i,1)}$ stands for the first data bit in subblock i. $p_{(i,1)}$ is the first parity bit. There are five data bits and one parity bit in this example.

The parity bit could be any function of the bits in the previous subblocks. For simplicity, let

$$p_{(i,1)} = b_{(i,1)} + b_{(i-1,2)} + b_{(i-2,3)} + b_{(i-3,4)} + b_{(i-4,5)} \bmod 2.$$

That is, each parity bit is affected by one of the bits in the previous five subblocks.

These parity bits can detect one burst of up to five bits that occurs in each rolling set of five subblocks. That means that the error will be detected if every two error bursts have at least five subblocks between them. The error, once detected, can be fixed by asking for a retransmission of the data.

The error can be detected by watching the trail it leaves in the parity bits that follow it. A burst of errors in this case might affect any of the five parity bits that come after it. When the parity bits don't match, the previous set of five subblocks can be retransmitted to fix the problem. It should be easy to see how spreading out the parity bits makes it possible for the code system to detect bursts of errors. None of the equations used to calculate the parity bits depends on neighboring bits. In this example, there are at least five bits in the bit stream between each of the bits used to calculate each parity bit. In the Hamming example, each of the parity equations depended on some adjacent bits. If both of those bits were flipped because of a

burst of error noise, then the errors would cancel out and the error would be recoverable.

Recovering a parity error is normally not possible with a simple code like this example. If one of the parity bits doesn't agree in this example, then the error could have been introduced by an error in six different bits. Finding which one is impossible. To some extent, a larger burst of errors will make the job easier. For instance, if three bits in a row are flipped, then three consecutive parity bits will also be flipped. If the parity bits are examined individually, then each one could have been caused by up to six different errors. But periodic codes like this are designed to handle bursts of errors. So it is acceptable to assume that the three errors would be adjacent to each other. This limits the location to two different spots.

Section 14.4.2 describes a coding scheme that can produce the right parity values when only some of the bits can be changed, a solution that's useful when you only want to tweak the least damageable parts of an image.

For instance, here is a data stream with correct parity bits:

$$\ldots\ 01010\ 0\ 01010\ 1\ 1001\ 1\ 01111\ 1\ 00011\ 1\ \ldots$$

If the first three bits are flipped, then the first three parity bits are also flipped:

$$\ldots\ 10110\ 1\ 01010\ 0\ 11001\ 0\ 01111\ 1\ 00011\ 1\ \ldots$$

Each individual error could have occurred in any of the previous five blocks, but the overlapping nature of the code limits the error to the first block shown here or either of the two blocks that preceded it. If five bits were flipped in a row, then the exact location would be known and it would be possible to correct the errors. This pushes the code to an extreme and it would be better not to hope for bursts to come at the extreme limit of the ability of the codes to detect the errors.

Both [LJ83] and [Ara88] are good sources.

The periodic code described in this section is a good way to detect bursts of errors, but it cannot help correct them unless the burst is at the extreme. There is some information available, but it is clearly not enough to recover the data.

3.4 Summary

Error-correcting codes are one of the most important tools for building digital systems. They allow electronic designers to correct the random errors that emerge from nature and provide the user with some digital precision. If the electronics were required to offer perfect accuracy, then they would be prohibitively expensive.

These codes are useful for correcting problems that emerge from the transmission systems. It might be desirable, for instance, for

several people to use the same channel. If they use a small part of the channel chosen at random, then the codes will correct any occasional collisions.

The field is also blessed with a deep body of literature exploring many different variations that lie outside of the scope of this introduction. Unfortunately this book does not have the space to consider them all nor outline the different ways that they can be more or less useful for steganography.

Chapter 14 describes spread-spectrum-like applications for hiding information. These techniques also rely on distributing the message over a relatively large number of elements in the file. If several of the elements are disturbed or mangled, these spread-spectrum solutions can still recover the message.

The Disguise If you want to use these codes to hide information, the best solution is to tweak a small subset of bits. If each block has 8 bits, for instance, the you can send 3 bits per block. If you want to send 000, then flip bit 0. If you want to send 011, then flip bit 3, and so on. When the bits are finally read at the other end, the error-correcting codes will remove the errors and the casual reader won't even know they were there. You can use the error-correcting codes to recover them.

Of course, this solution trades accuracy for steganography. Accidental or intentional errors will destroy the message. The error-correcting powers of the equation will be spent on carrying the information.

Another simple solution is to encode your message by making a few small changes in a signal that is already protected by some error correction. Your message might be hide three bits ($0 \leq i < 8$) by creating a fake error in bit i. This can be repeated for every error-corrected byte, a pretty good packing ratio. When someone receives the file, the regular error correction code will fix the problem effectively hiding your changes. You can strip the secret bits out by using the error-correcting algorithm to detect the changes.

How Secure Is It? Error-correcting codes are not at all secure against people who want to read them. The patterns between the bits are easy to detect. They are quite resistant, however, against errors.

How To Use Them? Error-correcting codes are rarely sold to consumers directly, although consumers use them all the time. Many electronic devices, however, like CD players and cell

phones, rely on them. Programmers who need to use error-correcting codes should search out complete books on the topic, which is large and engaging.

Further Reading

- Shu Lin and Daniel J. Costello's book, *Error Control Coding*, is one of the classic treatments in the area. [LC04]
- Jessica J. Fridrich, Miroslav Goljan, Petr Lisonek and David Soukal discuss the use of Michael Luby's LT Codes as a foundation for building better versions of perturbed quantization discussed in Section 14.4.2. They are like error-correcting codes that are computed when some of the bits can't be changed. These graph-based codes can be faster to compute than matrix-based solutions.[Lub02, FGLS05]
- Wojciech Mazurczyk and Krzysztof Szczypiorski found that Voice Over IP calls could hide information because the algorithms work around missing packets— or packets replaced with hidden messages.[MS08]

Chapter 4

Secret Sharing

4.1 Two out of Three Musketeers

In Bob's Manhattan living room, three high school chums are confronting a middle-age crisis over scotch and soda. They're all lawyers and disenchanted by the way that money and corruption have ruined the justice system. Inspired by movies like Batman*, they decide to recreate* The Three Musketeers *and prowl about the night looking for people in need of help.*

Bob: Okay. It's settled. We'll file for our license to carry concealed weapons tomorrow. On Friday, we pick out our Glocks.

Harry: Yes. 9mm.

Together: All for one and one for all!

Harry: You know, I just thought of something. My wife promised her cousin we would go to dinner at her house on Friday. She planned it last month. Could we get the Glocks another day?

Bob: Sunday's out for me. We're going to my mother's house after church.

Mark: Well, doesn't fighting evil count for something in the eyes of God?

Bob: Yes. But I still think we need a contingency. We're not always going to be available. There will be business trips, family visits, emergencies.

Mark: This is a good point. We might be stuck in traffic or held up in court. We need a plan.

Harry: Well, what if we said, "All available for one and one for who's there that evening?"

Mark: Not bad. It's more flexible. But what if just one of us is there?
Harry: What's the difference?
Mark: That one person really wouldn't be a group. He would be acting as a vigilante. He could do anything he wanted that evening. Maybe even something that was less than just.
Harry: So you want a quorum?
Mark: Yes. I say two out of three of us should be there before someone can start invoking the name of the Three Musketeers.
Bob: What about costumes? What do we wear if we're alone?
Mark: Doesn't matter. The most important thing is what we shout as we vanquish the foes. Are we together on this?
Together: Two out of Three for One and One for Two out of Three!

4.2 Splitting Up Secrets

There are many occasions when you need to split a key or a secret into numerous puzzle parts so that the secret can only be recovered if all of the parts are available. This is a good way to force people to work together. Many nuclear weapons systems, for instance, require two people to turn two different keys simultaneously. Bank safe deposit boxes have two locks and one key is held by the owner while the other is held by the bank.[1]

Splitting information into a number of parts is a good way to make information disappear. Each part may look like noise, but together they create the message. These different parts can take different paths adding further confusion to the entire process.

There are many neat ways to mathematically split a secret into parts. This secret might be the key to an encrypted file or it might be the important factoid itself. The goal is to create n different files or numbers that must all be present to reconstruct the original number. There are also threshold schemes that let you recover the secret if you have some smaller subset of the original parts. If a corporation has five directors, for instance, you might require that three be present to unlock the corporation's secret key used to sign documents.

The mathematics of these schemes is really quite simple and intuitive. Chapter 3 showed how error-correcting codes can be used as primitive secret-sharing devices. That is, you can split up a secret by encoding it with a error-correcting code that can correct wrong bits.

[1] It is not clear to me why the bank needs to have its own key on the box. The combination to the vault serves the same purpose.

(The 7-bit code from page 47 shows how you can split up a secret into seven parts so that it can be recovered if any six parts are available.)

There are numerous problems with this approach. First, some bits are often more privileged than others. In the 7-bit scheme from page 47 in Chapter 3, four of the seven bits hold the original message. The other three are parity bits. If the right four are put together, then the original secret is unveiled. If one of these bits is missing, however, then the parity bits are needed to get the secret.

Second, there can be some redundancy that allows people to unveil the secret even if they don't hold all of the parts. For instance, the 3-bit error-correcting code described on page 40 can recover the correct answer even if one of the three bits is changed. This is because each bit is essentially turned into three copies of itself. If these three copies are split into three parts, then they won't prevent each person from knowing the secret. They have it in their hands. Their part is an exact copy of the whole. This is an extreme example, but the same redundancies can exist in other versions.

Deliberately adding errors is one way to prevent this.

A better solution is to use algorithms designed to split up secrets so thatthey can't be recovered unless the correct number of parts is available. Many different algorithms are available to do this, but most of them are geometric in nature. This means that it is often easy to understand them with figures and diagrams.

4.2.1 Requiring All Parts

Many of the algorithms described later in this section can recover a secret split into n parts if only k parts are available. There are many times when you might want to require that all parts be present. There are good algorithms that work quite well when $n = k$, but are not flexible to handle cases when k is less than n. These basic algorithms are described here before the other solutions are explained.

One approach is to imitate safe deposit boxes and use n layers of encryption. If $f(k_i, X)$ encrypts a message X with key k_i, then you can take the secret and encrypt it repeatedly with each of n different keys. That is, compute:

Repeating the same encryption function again and again can introduce some theoretical problems and make analyzing the system tricky.

$$f(k_1, f(k_2, f(k_3, \ldots f(k_n, X)\ldots))).$$

Each person gets one of the n keys and it should be obvious that the secret can't be recovered unless all of them are present. If one is missing, then the chain is broken and the layers of encryption can't be stripped off.

A simpler approach is to think of the secret as a number, X, and then split it into n parts that all add up to that number, $X_1 + X_2 +$

$X_3 + \cdots + X_n = X$. If one number is missing, it is impossible to determine what X might be. This solution is an extension of the one-time pad and it is just as secure. If the parts are chosen by a good, secure random number generator, there is no way the people who hold the $n-1$ parts can guess what the value of the missing part might be.

In practice, this solution is often computed for each bit in the secret. That is, the secret is split into n parts. If the first bits of the parts are added together, they will reveal the first bit of the secret. If the second bits of the different parts are added together, the result is the second bit of the secret. This addition is done modulo 2, so you're really just determining whether there is an odd or even number of ones in the bits. Here's an example:

$$\begin{aligned} X_1 &= 101010100 \\ X_2 &= 101011010 \\ X_3 &= 110010010 \\ X_4 &= 010101100 \\ X_1 + X_2 + X_3 + X_4 &= 100110000 \end{aligned}$$

If you wanted to split up a secret, then you would generate the first $n-1$ parts at random. Then you would compute X_n so that $X_1 + \cdots + X_n = X$. This is actually easy. $X_n = X + X_1 + \cdots + X_{n-1}$.

Are both of these solutions equally secure? The addition method, which is just an extension of the one-time pad, is perfectly secure. There is no way that the system can be broken if you don't have access to all of the parts. There is no additional pattern. The layers of encryption are not necessarily as secure. There are so many variables in the choice of encryption function and the size of the keys, that some choices might be breakable.

Another way of understanding this is to examine the entropy of the equation, $X_1 + X_2 + X_3 + \cdots + X_n = X$. If each value of X_i has m bits, then there are mn bits of entropy required to determine the equation. If $n-1$ values of X_i are recovered, there are still m bits of entropy or 2^m possible solutions to explore.

Intuitively, this encryption equation has the same properties:

$$f(k_1, f(k_2, f(k_3, \ldots f(k_n, X)\ldots))).$$

If each key, k_i, has m bits, then there are still mn bits of entropy in the equation. Unfortunately, the complexity of the function f makes it difficult to provide more mathematical guarantees. If the basic function, f, is secure enough to use for basic encryption, then it should be secure in this case. But, there are many interesting and

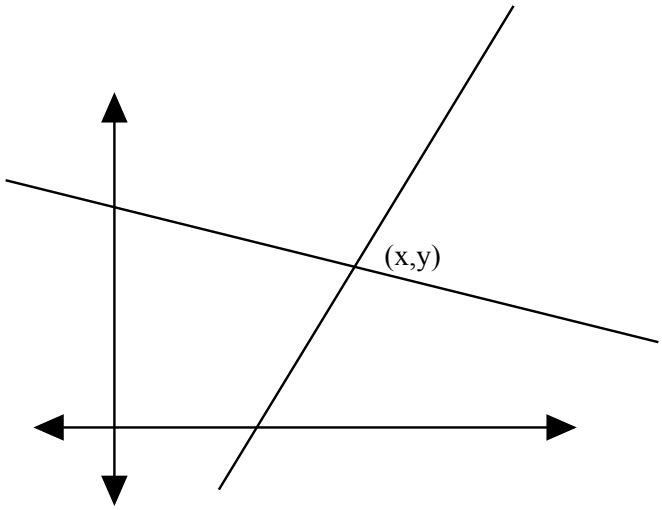


Figure 4.1: A secret, x, is split into two parts by finding two random lines that intersect at (x, y). (y is chosen at random.)

unanswered questions about what happens if the same system is used to encrypt data over and over again with different keys. [2]The simplest approach is the best in this case.

4.2.2 Letting Parts Slide

Obviously, there are many reasons why you might want to recover some secret if you don't have all of the parts. The most basic algorithms are based on geometry. Imagine that your secret is a number, x. Now, choose an arbitrary value for y and join the two values together so they represent a point on a plane. To split up this secret into two parts, just pick two lines at random that go through the point.- (See Figure 4.1.) The secret can be recovered if the intersection of the two lines is found. If only one line is available, then no knows what the secret might be.

Gus Simmons' chapter on Shared Secrets [Sim93] is a great introduction to the topic.

If there are two lines, then both parts need to be available to find the solution. This technique can be extended so there are n parts, but any two parts are enough to recover the secret.

Simply choose n lines that go through (x, y) at random. Any pair will intersect at (x, y) and allow someone to recover the secret, as in Figure 4.2. When the secret must be split into n parts and any k must be available to recover the secret, then the same approach

[2]Some good introduction papers include [CW93].

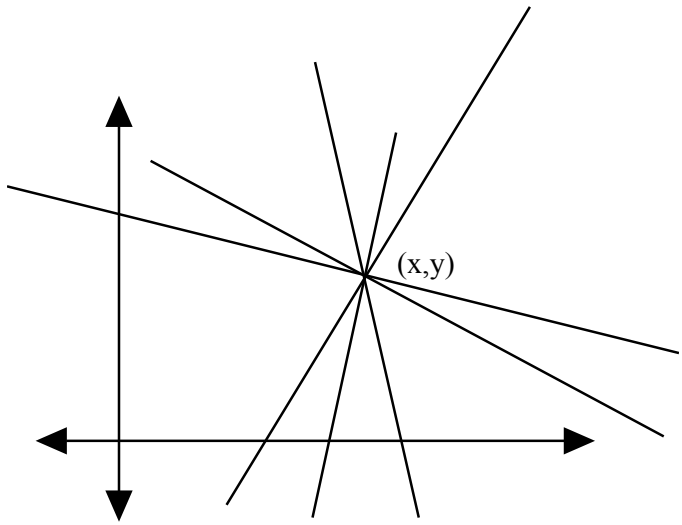


Figure 4.2: A secret, x, is split into n parts by finding n random lines that intersect at (x, y). (y is chosen at random.) Any pair is enough to recover the secret.

can be used if the geometry is extended into k dimensions. If $k = 3$, then planes are used instead of lines. Three planes will intersect only at the point. Two planes will form a line when they intersect. The point (x, y, z) will be somewhere along the line, but it is impossible to determine where it is.

Stephan Brands uses this technique in his digital cash scheme [Bra93].

It is also possible to flip this process on its head. Instead of hiding the secret as the intersection point of several lines, you can make the line the secret and distribute points along it. The place where the line meets the y axis might be the secret. Or it could be the slope of the line. In either case, knowing two points along the line will reveal the secret. Figure 4.3 shows this approach.

Each of these systems offers a pretty basic way to split up a secret key or a file so that some subset of people must be present. It should be easy to see that the geometric systems that hide the secret as the intersection point are as secure as a one-time pad. If you only have one line, then it is impossible to guess where the intersection lies along this line. $x = 23$ is just as likely as $x = 41243$. In fact, owning one part gives you no more insight into the secret than owning no part. In either case, all you know is that it is some value of x. This is often called a *perfect* secret-sharing system.

Some people might be tempted to cut corners and hide information in both the x and the y coordinate of the intersection point.

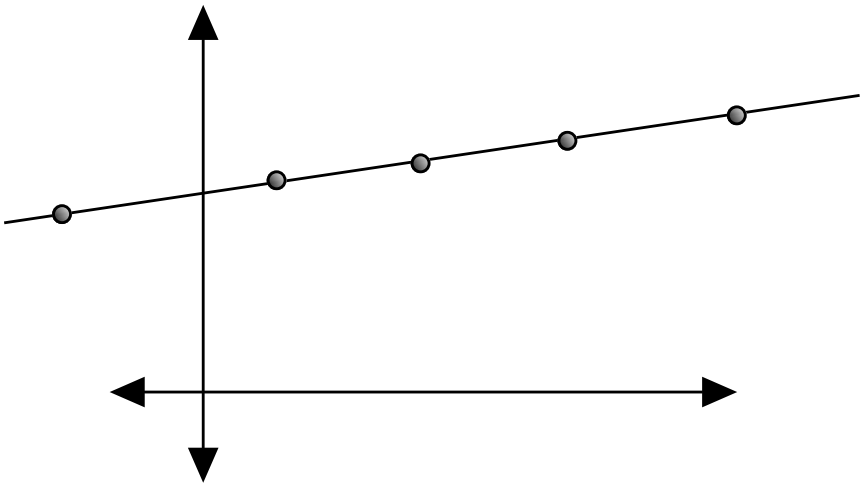


Figure 4.3: Here the secret is the line itself. Random points along the line are distributed as the parts of the secret. You must have two to recover the line.

This seems feasible because you can choose any set of lines that goes through this point. This changes the parameters of the system substantially. If you own a part of the secret, then you know something about the relationship between x and y. The slope of the line and the y intercept describe exactly how x and y change in unison.

Roger Dingledine, David Molnar, and Michael J. Freedman designed Free Haven to split up a document among a number of servers using Michael Rabin's secret sharing and information dispersal algorithm. The system also offers a mechanism for paying server owners.[DF00, Rab89a, Rab89b]

In some cases, this might be enough to crack the system. For instance, imagine you are protecting the number of men and women escaping from England on the Mayflower. Storing the number of men in the x coordinate and the number of women in the y coordinate is a mistake. An English spy might know that the number of men and the number of women are likely to be roughly equal given the percentages of men and women in society. This extra information could be combined with one part to reveal a very good approximations of x and y.[3]

4.2.3 A More Efficient Method

The basic secret-sharing methods split up a secret, X, into n equal sized parts. If the secret is m bits long, then the parts are also m bits long. This has the advantage of perfect security. If one part is

[3]You should also avoid storing them as separate secrets broken into parts. In this case, one part from each of the two secrets would still yield enough information. The best solution is to encrypt the two values and split the key to this file.

missing, the secret can only be recovered by testing all potential m bits of the missing part— and only if there's a way to check these 2^m possibilities and verify the correct one.

If m bits are too unwieldy for some reason, another quick solution is to encrypt X with some function and split the result into n parts. That is, take the m bits from $f(X)$ and distribute $\frac{m}{n}$ bits to each part holder.

Hugo Krawczyk offers a scheme that provides more computational assurances that the secret can't be reconstructed without all parts. [Kra94]

This approach sacrifices security for efficiency. Replacing a lost part just requires testing all possible combinations of $\frac{m}{n}$ bits instead of m bits– a solution that only works if there's a way to test the secret. But if this proposition is difficult enough, then the approach may be useful.

It should be noted that such a function f must be designed so that any change to one bit in the input has the potential to change any output bit. This feature is common if m is smaller than the block of a modern, well-designed algorithm like DES or Rijndael. If m is larger, $f_{f(X)^r)}$ should arrange for every every bit to affect every other if X^r stands for the bits in X arranged in reverse order.

If more strength is desirable, the parts can encrypted in a round robin. Let $\{p_1, p_2, \ldots, p_n\}$ be the n parts with $\frac{m}{n}$ bits in each piece. Instead of giving p_i to person i, we can give $f(h(p_{i-1}), p_i)$ to person i. This means that we can't recover part i without part $i-1$. All parts must be present.

4.2.4 Providing Deniability

The error-correcting codes described in Chapter 3 can also be used to add some deniability.

Each of the secret-sharing schemes described in this chapter offer some mindboggling chances to hide data in the Net. There is no reason why one particular file alone should be enough to reveal the information to anyone who discovers it. Splitting a file into multiple pieces is an ideal way to add complete deniability. Imagine, for instance, that the important data is stored in the least significant bits of some image using the techniques from Chapter 9. You could put the important data in the GIF file you use for your home page background and then place this up on the Web. But this is your home page; and the connection is obvious. Another solution is to find, say, three other GIF images on the Web. Maybe one of them is from the Disney World home page, another is from the White House home page, and the third is from some shady hacker site in Europe. Extract the least significant bits from each of these files. You have no control over these bits, but you can use them to hide ownership of the data by using the first secret-sharing scheme described here. If you add up the values recovered from all four sites, then the final information

appears.

Now imagine that the word gets out that the information is hidden in the combination of these four sites. Which one of the four is responsible? Disney World, the White House, the hackers in Europe, or you? It is impossible to use the least-significant bits of each of these images to point the finger at anyone. The hidden information is the sum of the four and any one of the four could have been manipulated to ensure that the final total is the hidden information. Who did it? If you arrange it so that the hidden information is found in the total of 100 possible images, no one will ever have the time to track it down.

The Publius system created by Marc Waldman, Aviel D. Rubin and Lorrie Faith Cranor uses basic secret sharing algorithms to distribute the key to a document. [WRC00] For more, see Section 10.6.

This system is just like the classic book ciphers which used a book as the one-time pad.

Of course, there are still problems with the plan. Imagine that Disney World used a slick, ray-traced image from one of their films like *Toy Story*. These images often have very smooth surfaces with constant gradients that usually have very predictable least significant bits. This would certainly be a defense against accusations that they manipulated the least significant bits to send out a secret message. The images chosen as the foils should have a very noisy set of least significant bits.

The Chi-Squared Test and other measures of randomness can be found in Don Knuth's [Knu81].

4.3 Building Secret-Sharing Schemes

Secret-sharing schemes are easy to explain geometrically, but adapting them to computers can involve some compromises. The most important problem is that computers really deal only with integers. Lines from real numbered domains are neither efficient nor often practical. For instance, five numbers involved in a typical scheme for hiding a secret as the intersection of two lines. Two numbers describe the slope and y-intercept of one line, two numbers describe the second line, and one number describes the x coordinate of the intersection point. If x is an integer, then it is not possible to choose lines at random that have both integers for their slope and y-intercept. If they are available, there will be a few of them.

You can use floating-point numbers, but they add their own instability. First, you must round off values. This can be a significant problem because both sides must do all rounding-off the same. Second, you might encounter big differences in floating-point math. Two different CPUs can come up with different values for x/y. The answers will be very close, but they might not be the same because the different CPUs could be using slightly different representations of values. Most users of floating-point hardware don't care about these very minor differences because all of their calculations are approximations.

But this is a problem with cryptography. Changing one bit of an encryption key is usually enough to ruin decryption—even if only the least significant bits that change.

The best solution is to return to finite collections of integers mod some prime number. Adi Shamir Shamir used this domain to hide secrets by choosing polynomials from this domain [Sha79]. Instead of lines or planes to hide the information, he chose $k-1$ degree polynomials, $p(x)$, where the first parameter, $p_0 = p(0)$, holds the secret. (This is a line when $k = 2$.) One point on the polynomial goes to each part holder. k parts can be used to reconstruct the polynomial and determine the secret, $p(0)$.

Here are the basic steps:

1. Choose a value of q that is prime and greater than n.
2. Find a random polynomial, $p(x)$, of degree $k-1$ by selecting $k-1$ random integers between 0 and q. These will be the coefficients of the polynomial, $p_1 \ldots p_{k-1}$. p_0 is the secret to be stored away.
3. $\sum_{i=0}^{k-1} p_i x^i$ is the polynomial.
4. Choose n points, $x_1 \ldots x_n$. These should be distinct and nonzero. Compute $p(x_1) \ldots p(x_n)$. These are the n parts to be distributed to part holders with the values of x_i. Any subset of k are enough to determine $p(x)$ and the secret p_0.
5. To recover the value of p_0, use Lagrangian interpolation. That is, you can use the points to estimate the derivatives of the polynomial at a point.

This solution uses only integers. It should be obvious that you need k points to recover the polynomial. The easiest way to see this is to realize that having $k-1$ points gives no information about $p(0)$. In fact, for any potential value of $p(0)$ you might guess, there is some p that generates it. You can find this p by taking the $k-1$ points and your guess for $p(0)$ and generating the polynomial. So, if there is a one-to-one mapping between these guesses, then the system is *perfect*. The part holder has no advantage over the outside guesser.

The scheme also offers greater efficiency for cases where k is a reasonably large number. In Section 4.2, the geometrical solution was to create a k-dimensional space and fill it with $k-1$ dimensional hyperplanes. Intersecting k hyperplanes was enough to reveal the point. The problem with this solution is that the hyperplanes take up more and more space as k grows larger. I don't mean they consume more abstract space—they just require more space to hold

the information that represents them. The Shamir scheme shown here doesn't require more space. Each part is still just a point, (x, y) that lies on the polynomial. This is substantially more efficient.

4.3.1 Making Some More Equal

In each of the schemes described in this chapter, the secrets are split into n parts and each of the n parts has the same equal share of control. Humans, being human, are never satisfied with anything as fair as that—some people will want some parts to be more powerful than others.

The most straightforward way to accomplish this is to give some people more parts. For instance, imagine a scheme where you need six parts to reconstruct the secret. That is, you might have a collection of five-dimensional hyperplanes in a six-dimensional space. Any set of six points is enough to uncover the secret, which for the sake of example will be the launch codes for a nuclear missile. Let's say that it takes two commanders, three sergeants, or six privates to launch a missile. This can be accomplished by giving three parts to the commanders, two parts to the sergeants, and one part to each private.

One problem with this solution is that arbitrary combinations of different ranks can join together. So, one commander, one sergeant, and one private can work together to uncover the secret. This might not be permitted in some cases. For example, the U.S. Congress requires a majority of both the House and the Senate to pass a bill. But the votes from one chamber can't be counted against the other. So even though there are 100 Senators and 435 members of the House, a Senator is not worth 4.35 House members. A bill won't pass just because 99 Senators vote for it and only 10 House Representatives. But this could be the situation if someone naively created a secret-sharing scheme by parceling out parts to both sides of Congress from the same shared secret.

A better solution to prevent this is to first split the secret into two equal parts, X_H and X_S, so that both are required to endorse a bill with the digital signature of Congress. Then H_R would be split into 435 parts so that 218 are enough to recover it. H_S is split into 100 parts so that 51 are enough to recover it.

Numerous combinations can make these schemes possible. Practically any scheme can be implemented using some combination and layers of secrets. The only problem with very complicated systems is that they can require many different dimensions. For instance, if you want a system that takes 17 privates, 13 sergeants, or

5 generals to launch some missiles, then you can use a system with $17 \times 13 \times 5$ dimensions. This can get a bit arcane, but the mathematics is possible.

4.4 Public-Key Secret Sharing

All of the algorithms in this section share one set of bits, the secret. This secret may be used for anything, but it is probably going to be used as a key to unscramble some information encrypted with a secret-key algorithm. The notion of splitting up authority and responsibility can also be incorporated into public-key algorithms. In these schemes, the actions for scrambling and unscrambling can be split and controlled separately. The holder of one key controls encryption and the holder of the other key controls decryption. If secret sharing is combined with public-key encryption, one group must agree to encrypt a message and another group must agree to decrypt it.

Approaches like this are often called threshold decryption or threshold signatures.

One easy solution is to combine any public-key algorithm with any of the secret sharing solutions. The keys are just collections of bits and these collections can be split into arbitrary subcollections using any basic secret splitting solution. This mechanism is perfectly useful, but it has limitations. If a group of people get together to encrypt or decrypt a message, the key must be put together completely. Once it is assembled, whoever put it together now controls it. The secret sharing feature is gone.

This approach splits the ability to decrypt a public key message among a group of k people. Anyone can send a message to the group, but they all must agree to decrypt it. No information obtained from decrypting one message can be used to decrypt the next. The system relies on the strength of the discrete log problem. That is, it assumes that given $g, p,$ and $g^x \bmod p$, it is hard to find x.

Similar techniques can be used to produce anonymous digital cash and secure voting solutions. [Bra95b, Bra95a]

The private key consists of k values $\{x_1, \ldots, x_k\}$ that are distributed among the k members who will have control over the decryption process. The public key is the value, $a = g_1^{x_1} g_2^{x_2} \ldots g_k^{x_k} \bmod p$, where the values of g_i and p are publicly available numbers. The values of g_i may be generators of the group defined modulo p but the algorithm will work with most random values less than p.

A message to the group is encrypted by choosing a random value, y, and computing $g_1^y \bmod p, g_2^y \bmod p, \ldots g_k^y \bmod p$. Then the value $a^y \bmod p$ is computed and used to generate a secret key for encryption the message with an algorithm like AES. The message consists of this encrypted data and the k values $g_1^y \bmod p, g_2^y \bmod p, \ldots g_k^y \bmod p$.

This mechanism, first proposed by Taher Elgamal, is very similar in spirit to Diffie-Hellman key exchange.

The message can be decrypted by distributing the k values to the group. Each person computes $(g_i^y)^{x_i} \ mod \ p$ and returns this value to the group leader who multiplies them together. $a^y = (g_1^y)^{x_1}(g_2^y)^{x_2} \ldots (g_n^y)^{x_n} \ mod \, p$.

The same set of keys can also generate digital signatures for the group with a modified version of the digital signatures used with the Diffie-Hellman-style public key system. Here are the steps:

1. The signers and the signature verifier agree on a challenge value, c. This is usually generated by hashing the document being signed. It could also be created by the verifier on the fly as a true challenge.

2. Each member of the group chooses a random witness, w_i and computes $g_i^{w_i} \ mod \, p$.

3. Each member of the group computes $r_i = cx_i + w_i$ and $g_i^{r_i} \ mod \, p$.

4. The group discards the values of w_i and gathers together the rest of the values in a big collection to serve as the signature. These values are: $\{r_1 = cx_1 + w_1, \ldots r_k = cx_k + w_k\}$, $\{g_1^{r_1} \ mod \, p, \ldots, g_k^{r_k} \ mod \, p\}$, and the product of:

$$g_1^{w_1} \ mod \, p, \ldots, g_k^{w_k} \ mod \, p.$$

5. Anyone who wants to check the signature can compute $a^{(-c)} g_1^{r_1} \ldots g_k^{r_k} \ mod \ p$ and make sure it is the same as the product of $g_1^{w_1} \ mod \, p, \ldots, g_k^{w_k} \ mod \, p$.

Similar solutions can be found using RSA-style encryption systems. In fact, some of the more sophisticated versions allows RSA keys to be created without either side knowing the factorization. [BF97, GRJK00, BBWBG98, CM98, WS99, FMY98]

4.5 Steganographic File Systems and Secret Sharing

Secret sharing algorithms split information into a number of parts so that the information can only be recovered if some or perhaps all of the parts are available. The same basic algebra can also be used by one person to hide their data so only the person who knows the right

combination of parts can verify that the data is there. This application may be valuable if you happen to be storing the information on your hard disk and you want to deny that it is there.

Ross Anderson, Roger Needham, and Adi Shamir created two versions of what they called a steganographic file system. [ANS98] Their first uses math very similar to secret sharing and so it is described here.

The system grabs a large block of disk space, randomizes it, and then absorbs files that are protected with passwords. If you don't know the password, then you can't find the file. If you do know the password, then the random bits produce the file. There's no way to identify that the file exists without the password.

"Three may keep a secret if two are dead."–Benjamin Franklin

This scheme is far from perfect. For it to work well, the passwords must be assigned in a hierarchy. That means if someone knows one password, K_i, then they must know all other passwords K_j where $0 \leq j < i$. If there are only three files, then the person with access to file 3 must also have access to files 1 and 3. Anderson, Needham and Shamir imagine that a person under interrogation may reveal the password to several modestly dangerous files without revealing the more sensitive ones.

The mathematics is all linear algebra. For the sake of simplicity, the system is defined for binary numbers where addition is the XOR ($\oplus$) operation and multiplication is the AND ($\cdot$) operation.

A basic steganographic file system can hold m files that are n bits long. In the beginning, the files are set to be random values that are changed as information is stored in the system. It often helps to think of the file system as a big matrix with m rows and n columns. Let C_i stand for the i^{th} row.

The password for file j is K_j, a m-bit-long vector where $K_j(i)$ stands the i^{th} bit of the vector. To recover file j from the file system, add together all of the rows, C_i, where $K_j(i) = 1$. That is:

$$\oplus_{i=1}^{m} K_j(i) C_i.$$

How do you store a file in the system? Here's a basic sketch of the steps for storing *one* file:

1. Break it into n bit blocks.
2. Choose a password for each block. One good solution is to concatenate a random string, S, before the password, hash it with a cryptographically secure hash function, H, and take the first $m - 1$ bits to serve as K_j.
3. Add a parity bit to K_j to make sure it is the correct length. Use odd parity to ensure that the number of 1 bits in the vector

is odd. That is, set the last bit to be one if there are an even number of ones in the first $m-1$ bits and a zero if there are an odd number.

4. Encrypt the block with a separate encryption function, ideally using a different key constructed by appending a different random string. This encryption function is necessary because of the linear nature of the file system. If an attacker can establish some part of the file, D, then the attacker can solve some linear equations to recover K_j. If the files aren't encrypted, an attacker can extract the file if the attacker can guess a phrase and its location in the file.

5. Replace C_i with $D \oplus C_i$ for all i where $K_j(i) = 1$.

This basic algorithm will store one file in the system. If it is used repeatedly, it may bring problems because new files can overwrite other files. If you want to store more than one file in the system, you need to ensure that they will not disrupt each other.

The simplest solution is to choose the m values of K_j so that they're orthogonal vectors. That is, $K_i \cdot K_j = 1$ if and only if $i = j$. In all other cases, $K_i \cdot K_j = 0$. If the password vectors are orthogonal, then m different files can be stored in the system without disturbing each other.

In other words, whenever the values of D are added to different rows C_j, that it will not distort another file. Why? If $K_p \cdot K_q = 0$, then there are only an even number of bits that are one in both K_p and K_q. Imagine that you're replacing C_j with $D \oplus C_j$ for all j where $K_p(j) = 1$. Some of these rows will also rows which are storing parts of another file defined by key K_q. Why isn't this file disturbed? Because D will only be added to an even number of rows and the value will cancel out. $D \oplus D = 0$.

Consider this example with K:

$$\begin{matrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{matrix}$$

This matrix contains the keys for 5 files. The first row, for instance, specifies that the first file consists of $C_2 \oplus C_3 \oplus C_4$. The second row specifies that the second file consists of $C_1 \oplus C_3 \oplus C_4$. If new data is stored in the first file, then C_2, C_3, and C_4 will all become $C_2 \oplus D, C_3 \oplus D$, and $C_4 \oplus D$ respectively. What does this do to the second file? $C_1 \oplus C_3 \oplus C_4$ becomes $C_1 \oplus (C_3 \oplus D) \oplus (C_4 \oplus D) = C_1 \oplus C_3 \oplus C_4 \oplus D \oplus D = C_1 \oplus C_3 \oplus C_4$.

Here's a basic algorithm for constructing K from a list of m passwords, $P_1, P_2, \cdots P_m$. Repeat this for each password, P_i.

1. Let $K_i = H(P_i)$, where H is some cryptographically secure hash function.

2. For all $j < i$, let $K_i = (K_i \cdot K_j)K_j \oplus K_i$. This orthonormalization step removes the part of the previous vectors that overlaps with the new row. That is, it ensures that there will only be an even number of bits that are one in both rows. It does this for all previous values.

3. If $K_i = 0$, then an error has occurred. The new chosen key is not independent of the previous keys. Set $K_i = H(H(P_i))$ and try again. Continue to hash the password until an acceptable value of K_i is found that is orthonormal to the previous keys.

This algorithm does not compute the values of K_i independently of each other. You must know the values of all K_j where $j < i$ to compute K_i. This is less than ideal, but it is unavoidable at this time. Anderson, Needham and Shamir decided to turn this restriction into a feature by casting it as a linear file access hierarchy. If you can read file i, then you can read all files j where $j < i$.

Forcing all of the keys into a hierarchical order may not always be desirable. Another technique for finding keys is to restrict each person to a particular subspace. That is, split the keyspace into orthogonal parts. If person i wants to choose a particular K_i, then that person must check to see that K_i is in the right subspace.

The easiest way to split the keys into orthogonal subspaces is to force certain bits in the key to be zero. Alice might use keys where only the first ten bits can be set to 1, Bob might use keys where only the second ten bits can be non-zero, and so on.

If necessary, Alice, Bob and the rest of the gang can agree on a random rotation matrix, R, and use it to rotate the subspaces. So Alice will only choose a key vector if RK_i has zeros in all the right places.

This version of the file system is also a bit unwieldy. If you want to read or write a file D, then you may need to access as many as m other rows. This factor can be substantial if m grows large. This can be reduced by using non-binary values instead of bits for the individual elements of K_i.

4.6 Summary

Secret-sharing is an ideal method for distributing documents across the network so no one can find them. It is an ideal way for people to deny responsibility. In some cases, the parts of the secret can be from the Web pages of people who have nothing to do with the matter at hand.

The Disguise Secret sharing lets you share the blame.

How Secure Is It? The algorithms here are unconditionally secure against attacks from people who have less than the necessary threshold of parts.

How to Use It. The XOR algorithm described here is easy to implement. It is an ideal way to split up information so that every party needs to be present to put the information back together.

Chapter 5

Compression

5.1 Television Listing

8:00 PM 2 (IRS) *Beverly Hills Model Patrol*— New lip gloss introduced.

5 (QUS) *Cash Calliope: Musical Detective* — Death with a Capital D-minor.

9 (PVC) *Northern Cops*— Town council bans eccentrics at town meeting, but not for long.

14(TTV) *Def N B*— Beethoven raps for the Queen.

9:00 PM 2 (IRS) *Super Hero Bunch*— Evil just keeps coming back for more.

5 (QUS) *Sniffmaster Spot*— Spot discovers toxic waste at Acme Dog Food Plant.

9 (PVC) *Mom's a Klepto*— Family stress as Mom plagiarizes daughter's English paper.

14(TTV) *Easy Cheesy*— Customer asks for Triple Anchovy pizza.

10:00 PM 2 (IRS) *X Knows Best*— Alien stepdad shows love is not Earthbound.

5 (QUS) *Dum De Dum Dum*— Detective Gump meets murdering publisher.

9 (PVC) *Betrayal Place*— Bob betrays Jane.

14(TTV) *Beverly Hills Astronaut*— Buzz discovers there are no malls in Space!

5.2 Patterns and Compression

Life often reduces to formulas. At least it does on television, where the solutions appear every 30 or 60 minutes. When you know the formula, it is very easy to summarize information or compress it. A network executive reportedly commissioned the television show "Miami Vice" with a two-word memo to the producer reading, "MTV Cops." You can easily specify a episode of "Gilligan's Island" with a single sentence like, "The one with the cosmonauts." Anyone who's seen only one episode of the show will know that some cosmonauts appear on the island, offer some hope of rescue, but this hope will be dashed at the end when Gilligan screws things up.

Compressing generic information is also just a matter of finding the right formula that describes the data. It is often quite easy to find a good formula that works moderately well, but it can be maddeningly difficult to identify a very good formula that compresses the data very well. Finding a good formula that works well for specific types of data like text or video is often economically valuable. People are always looking for good ways to cut their data storage and communications costs.

Compressing data is of great interest to anyone who wants to hide data for four reasons:

Less data is easier to handle. This speaks for itself. Basic text can easily be compressed by 50 to 70%. Images might be compressed by 90%.

Compressed data is usually whiter. Compression shouldn't destroy information in a signal. This means that the information per bit should increase if the size of the file decreases. More information per bit usually appears more random.

Details about measuring information are on page 31.

Reversing compression can mimic data. Compression algorithms try to find a formula that fits the data and then return the specific details of the formula as compressed data. If you input random data into a compression function, it should spit out data that fits the formula.

Page 183 shows how the JPEG algorithm can identify just how much space can be exploited in an image.

Compression algorithms identify noise Good compression algorithms understand how human perception works. Algorithms like JPEG or MP3 can strip away extra information from a file that a human doesn't notice. This information can also be exploited by steganographers to locate places where a significant amount of noise might be replaced by a hidden message.

Of course this approach is dangerous as well. If the JPEG algorithm strips away some information, then any information you insert in this location is just as liable to be stripped away. An attacker or a well-meaning programmer along the path could compress a file to save space and destroy your message. This makes the technique dangerous for weakly protected data like watermarks, but potentially useful if you can be reasonably sure the file won't be compressed along the path. [1]

One watermarking algorithm in Section 14.7.1 deliberately aims to hide information in the most important parts of the image to avoid being destroyed during compression.

Compression is an important tool for these reasons. Many good commercial compression programs already exist simply because of the first reason. Many good encryption programs use compression as an additional source of strength. Mimicking, though, is why compression is discussed in depth in this book. Some of the basic compression algorithms provide a good way to make information look like something else. This trick of flipping the algorithm on its head is discussed in Chapter 6.

A number of techniques for compressing data that are used today. The field has expanded wildly over the last several years because of the great economic value of such algorithms. A procedure that compresses data in half can double the storage area of a computer with no extra charge for hardware. People continue to come up with new and often surprisingly effective techniques for compressing data, but it all comes down to the basic process of identifying a formula that does a good job of fitting the data. The parameters that make the formula fit the data directly becomes the compressed surrogate. Some of the more popular techniques are:

Probability Methods These count up the occurrences of characters or bytes in a file. Then they assign a short code word to the most common characters and a long one to least common ones. Morse code is a good example of a compression algorithm from this class. The letter "e", which is the most common in the English language, is encoded as a dot. The letter "p", which is less common, is encoded as dot-dash- dash-dot. The *Huffman code* is the best known edition of these codes.

Dictionary Methods These algorithms compile a list of the most common words, phrases, or collections of bytes in a file, then number the words. If a word is on this list, then the compressed file simply contains the number pointing to the dictionary entry. If it isn't, the word is transmitted without change. This technique can be quite effective if the data file has a large amount

[1] WebTV's network, for instance, will strip away higher order data from images if it won't show up on a television set.

of text. Some report compressing text to 10 to 20% of its original size. The *Lempel-Ziv* compression algorithm is the most commonly used version of this algorithm.

Run-Length Encoding Many images are just blocks of black pixels and white pixels. If you walk along a line, you might encounter 1000 white pixels followed by 42 black pixels followed by 12 white pixels, etc. Run-length encoding stores this as a sequence of numbers 1000, 42, 12, etc. This often saves plenty of space and works well for black-and-white line art. Faxes use this technique extensively.

Wave Methods These algorithms use a collection of waves as the basic collection of formulas. Then they adjust the size and position of the waves to best fit the data. These work quite well with images that do not need to be reconstructed exactly. The new image only needs to approximate the original. The JPEG, JPEG2000 and MPEG image and video compression standards are three of the more famous examples of this technique.

Chapter 14 investigates the information-hiding capabilities of wavelets.

Fractal Methods Fractal functions produce extremely complicated patterns from very simple formulas. This means that they can achieve extremely high compression *if* you can find the formula that fits your data.

A good introduction to fractal compression can be found in [BS88, Bar88, Bar93].

Adaptive Compression Schemes Many compression schemes can be modified to adapt to the changing data patterns. Each of the types described here comes in versions that modify themselves in the middle of the data stream to adapt to new patterns.

All of these compression schemes are useful in particular domains. There is no universal algorithm that comes with a universal set of functions that adapt well to any data. So people modify existing algorithms and come up with their own formulas.

Compression functions make good beginnings for people who want to hide data because the functions were constructed to describe patterns. There are two ways to use compression functions successfully to hide information. One is to mold it into the form of other data so it blends in. A compression function that works well on zebras can model black and white stripes and convert a set of stripes into a simple set of parameters. If you had such a function, it could be applied to some data in *reverse* and it would expand the data into zebra stripes. The result would be bigger, but it would look like something else. The data could be recovered by compressing it again.

Compression techniques can also be used to identify the least important nooks and crannies of a file so that extra data can be snuck

into them. Many image-compression functions are designed to be *lossy*. That means that the reconstructed image may look very similar to the original image, but it won't be *exactly* the same. If the functions that describe an image can be fitted more loosely, then the algorithms can use fewer of them and produce a smaller compressed output. For instance, an apple might be encoded as a blob of solid red instead of a smooth continuum of various shades of red. When the image is decompressed, much of the smaller detail is lost but the overall picture still looks good. These compression functions can easily compress an image to be one-fifth to one-tenth of its original size. This is why they are so popular.

The television format example from the beginning of the chapter is an example of lossy compression. They are not enough to recreate the entire program. They're a better example of lossy compression where a surrogate is found.

5.2.1 Huffman Coding

A good way to understand basic compression is to examine a simple algorithm like Huffman coding. This technique analyzes the frequency with which each letter occurs in a file and then replaces it with a flexible-length code word. Normally, each letter is stored as a byte which takes up 8 bits of information. Some estimates of the entropy of standard English, though, show that it is something just over about 3 bits per letter. Obviously there is room to squeeze up to almost 5/8ths of a file of English text. The trick is to assign the short code words to common letters and long code words to the least common letters. Although some of the long words will end up being longer than 8 bits, the net result will still be shorter. The common letters will have the greatest effect.

Table 5.1 shows a table of the occurrences of letters in several different opinions from the United States Supreme Court. The space is the most common character followed by the letter "E". This table was constructed by mixing lower- and uppercase letters for simplicity. An actual compression function would keep separate entries for each form as well as an entry for every type of punctuation mark. In general, there would be 256 entries for each byte.

Table 5.2 shows a set of codes that were constructed for each letter using the data in Table 5.1. The most common character, the space, gets a code that is only 2 bits long: 01. Many of the other common characters get codes that are 4 bits long. The least common character, "Z", gets an 11-bit code: 00011010001. If these codes were used to encode data, then it should be easy to reduce a file to less than one-half of its original size.

Here's a simple example that takes 48 bits used to store the word "ARTHUR" in normal ASCII into 27 bits in compressed form:

Table 5.1: The frequency of occurrence of letters in a set of opinions generated by the U.S. Supreme Court.

Letter	Frequency	Letter	Frequency
space	26974	A	6538
B	1275	C	3115
D	2823	E	9917
F	1757	G	1326
H	3279	I	6430
J	152	K	317
L	3114	M	1799
N	5626	O	6261
P	2195	Q	113
R	5173	S	5784
T	8375	U	2360
V	928	W	987
X	369	Y	1104
Z	60		

Table 5.2: The codes constructed from Table 5.1. A Huffman tree based on these codes is shown in Figure 5.2.

Letter	Code	Letter	Code
space	01	A	1000
B	111011	C	10110
D	11100	E	0000
F	001101	G	111010
H	00111	I	1001
J	0001101001	K	000110101
L	10111	M	001100
N	1101	O	1010
P	000101	Q	00011010000
R	1111	S	1100
T	0010	U	000100
V	0001111	W	0001110
X	0001110	Y	0001100
Z	00011010001		

Letter:	A	R	T	H	U	R
ASCII:	01000001	01010010	01010100	01001000	01010101	01010010
Compressed:	1000	1111	0010	00111	000100	1111

The Huffman algorithm can also be used to compress any type of data, but its effectiveness varies. For instance, it could be used on a photograph where the intensity at each pixel is stored as a byte. The algorithm would be very effective on a photograph that had only a few basic values of black and white, but it wouldn't work well if the intensities were evenly distributed in a photograph with many even shades between dark and light. The algorithm works best when there are a few basic values.

More sophisticated versions of the Huffman code exist. It is common to construct second-order codes that aggregate pairs of letters. This can be done in two ways. The easiest way to do this is to simply treat each pair of letters as the basic atomic unit. Instead of constructing a frequency table of characters, you would construct a table of pairs. The table would be much larger, but it would generate even better compression because many of the pairs would rarely occur. Pairs like "ZF" are almost nonexistent.

Another way is to construct 26 different tables by analyzing which letters follow other letters. One table for the letter "T" would hold the frequency that the other letters might follow after the "T". The letter "H" would be quite common in this table because "TH" occurs frequently in English. These 26 tables would produce even more compression because more detailed analysis would tune the code word even more. The letter "U" would receive a very short code word after the letter "Q" because it invariably follows.

This example has shown how a Huffman compression function works in practice. It didn't explain how the code words were constructed nor did it show why they worked so well. The next section in this chapter will do that.

Chapter 6 shows how to run Huffman codes in reverse.

5.3 Building Compression Algorithms

Creating a new compression algorithm has been one of the more lucrative areas of mathematics and computer science. A few smart ideas are enough to save people billions of dollars of storage space and communications time, and so many have worked with the idea in depth. This chapter won't investigate the best work because it is beyond the scope of the book. Many of the easiest ideas turn out to hide information the best. Huffman codes are a perfect solution for basic text. Dictionary algorithms, like Lempel-Ziv, are less effective.

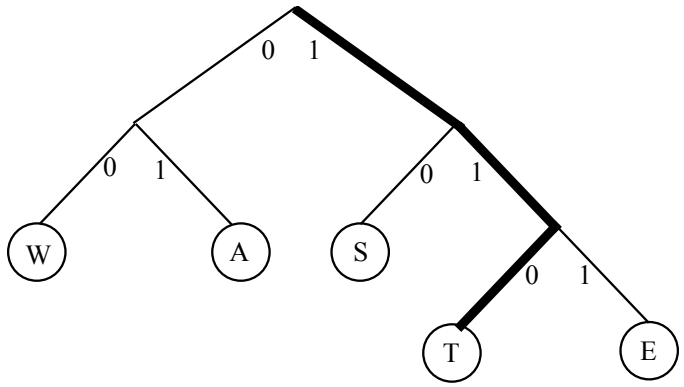


Figure 5.1: A small Huffman tree. The code for each letter is determined by following the path between the root of the tree and the leaf containing a particular letter. The letter "T", for instance, receives the code 110.

5.3.1 Huffman Compression

Huffman compression is easy to understand and construct. Let the set of characters be Σ and let $\rho(c)$ be the probability that a particular character, c, occurs in a text file. Constructing such a frequency table is easily done by analyzing a source file. It is usually done on a case-by-case basis and is stored in the header to the compressed version, but it can also be done in advance and used again and again.

The basic idea is to construct a binary tree that contains all of the characters at the leaves. Each branche is labeled with either a zero or a one. The path between the root and the leaf specifies the code used for each letter. Figure 5.1 shows this for a small set of letters.

The key is to construct the tree so that the most common letters occur near the top of the tree. This can be accomplished with a relatively easy process:

1. Start with one node for each character. This node is also a simple tree. The weight of this tree is set to be the probability that the character associated with the tree occurs in the file. Call the trees for t_i and the weight $w(n_i)$. The value of i changes as the number of trees change.

2. Find the two trees with the smallest weight. Glue these into one tree by constructing a new node with two branches connected to the roots of the two trees. One branch will be labeled with a one and the other will get a zero. The weight of this new tree is

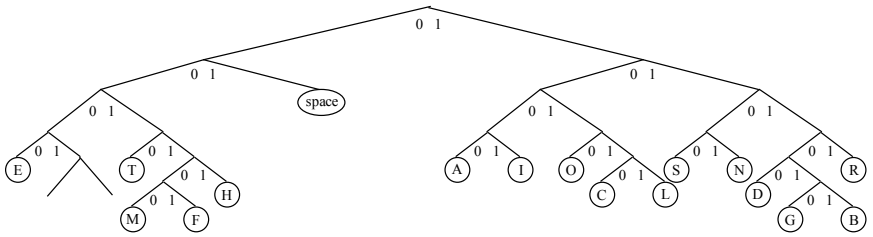


Figure 5.2: The top of the tree built from the data in Table 5.1. The generated the codes shown in Table 5.2. Only the top part is shown here because of space considerations. Less common letters like "Z" are in the missing part of the tree corresponding to the prefix 0001.

set to be the sum of the old trees that were joined.

3. Repeat the previous step until there is only one tree left. The codes can be constructed by following the path between the root and the leaves.

I know of a Greek labyrinth which is a single straight line. Along this line so many philosophers have lost themselves that a mere detective might well do so too.
—Jorge Luis Borges in Death and the Compass

The characters with the smallest weights are joined together first. Each joining process adds another layer between the root and the leaves. So it is easy to see how the least common letters get pushed far away from the root where they have a longer code word. The most common letters aren't incorporated until the end, so they end up near the top.

The algorithm naturally balances the tree by always taking the smallest weights first. The weight for a tree represents the number of times that any of the characters in the tree will occur in a file. You can prove that the tree constructed by this algorithm is the best possible tree by imagining what happens if you mistakenly choose the wrong two trees to join at a step. More common characters get pushed farther from the root and get longer code words than less common characters do. The average compression drops.

Many other people have extended the theme of Huffman coding by creating other algorithms that use the addresses of nodes in a tree. One popular technique is to use Splay trees where the trees are modified every time a character is encoded. One version moves the letter to the top of the tree in a complex move that preserves much of the structure. The result is that the most common letters bubble up to the top. The constant rearrangement of the tree means that the tree adapts to the local conditions. This type of algorithm would be ideal for compressing a dictionary where even the least popular letters like "j" or "z" are common in sections. When the

algorithm moved through the "j" part of the dictionary, the node containing "j" would be pushed repeatedly to the top of the splay tree where it would get a short code word. Later, when the compression function got to the "z" section, the node for "z" would end up near the top consistently giving "z" a short code word. Obviously one major problem with this compression scheme is that the entire file must be processed from the beginning to keep an accurate description of the splay tree. You can't simply jump to the "z" section and begin decompressing.

A good basic reference on compression is [Sto88].

This basic Huffman algorithm has many different uses. It will be in Chapter 6 to turn data into something that looks like English text. Huffman encoding is also used as a building block in Chapter 7 to make optimal weighted choices between different words. The same structure is as useful there as it is here.

5.3.2 Dictionary Compression

Compression schemes like the popular and patented Lempel-Ziv algorithm are called dictionary schemes because they build a big list of common words in the file.[2] This list can either be created in one swoop at the beginning of the compression or it could be built and changed adaptively as the algorithm processes the file. The algorithms succeed because a pointer describing the position in the dictionary takes up much less space than the common word itself.

The dictionary is just a list of words. It is almost always 2^n words because that makes the pointer to a particular word take up n bits. Each word can either be a fixed length or a flexible length. Fixed lengths are easier to handle, but flexible lengths do a better job of approximating the English language and x86 machine code.

Compression follows easily. First, the file is analyzed to create a list of the 2^n most common words. Then the file is processed by scanning from beginning to end. If the current word is in the dictionary, then it is replaced by a tag, `<InDict>`, followed by the position in the dictionary. If it isn't in the dictionary, then it is replaced by a tag, `<Verbatim>`, followed by the word that remains unchanged.

Obviously, the success of the algorithm depends on the size of the tags (`<InDict>` and `<Verbatim>`), the size of the dictionary, and the number of times something is found in the dictionary. One basic and usually effective solution is to make the tags be one entire byte, B. If the value of the byte is zero, then the next n bits represents a word in

[2] The algorithms are not particularly good at compressing files like dictionaries used by humans. The fact that I used a regular dictionary as an example in the previous section is just a coincidence. Don't be confused.

the dictionary. If the value of the byte, B, is greater than zero, then B bytes are copied verbatim out of the original file. This scheme allows the program to use flexible word sizes that work well with English. There are many different schemes that are more efficient that others in some cases.

The index into the dictionary does not need to be n bit numbers. You can also count the occurrence of words in the dictionary and use a Huffman-like scheme to devise short code words for some of them. The tag for verbatim text is usually included as just another word in this case.

The dictionary can also adapt as the file is processed. One simple technique is to keep track of the last time a word from it was used. Whenever a section of verbatim text is encountered, the oldest word is swapped out of the dictionary and the newest verbatim text is swapped in. This is a great technique for adapting to the text because many words are often clustered in sections. For instance, the words "dictionary," "Huffman," and "compression" are common in this section but relatively rare in other parts of the book. An adaptive scheme would load these words into the dictionary at the beginning of the section when they were first encountered and not swap them out until they aren't used for a while.

Dictionary schemes can be quite effective for compressing arbitrary text, but they are difficult to run in reverse to make data mimic something. Chapter 6 uses Huffman-like algorithms to generate real text, but it doesn't include a section on reversing dictionary algorithms. They are described in this chapter because compression is a good way to save space and whiten data. The algorithms don't work particularly well for mimicry because they require a well-constructed dictionary. In practice, there is no good automatic way that I know for constructing a good one.

5.3.3 JPEG Compression

The Huffman encoding described in "Huffman Compression" (see page 80) and the dictionary schemes in "Dictionary Compression" (see page 82) are ideal for arbitrary collections of data. They can also work quite well on some types of image files, but they fail on others. If an image has a small number of colors that may occur in a predictable pattern, then both of these algorithms may do a good job of finding a pattern that is strong enough to generate a good compression. This often doesn't happen because the images contain many shades of colors. The Japanese flag, for instance, has one red circle that is a constant color, but a realistically lit apple has many

different shades of red.

The JPEG algorithm is a good example of how to tune an algorithm to a particular type of data. In this case, the algorithm fits cosine functions to the data and then stores the amplitude and period. The number of functions used and the size can be varied according to the amount of compression desired. A small number of functions produces a high amount of compression but a grainy image. More functions add accuracy, but take up more space. This flexibility is possible because people don't always particularly care if they get *exactly* the same image back when it is decompressed. If it looks reasonably close, it is good enough.

This flexibility is what is so useful about JPEG encoding. The algorithm from Section 5.3.1 will be run in reverse to produce text that mimics English text. The JPEG algorithm doesn't do that well. However, it does have the ability to identify nooks and crannies in the image that might have space to hold information. This is described in detail in Chapter 9.

5.3.4 GZSteg

Many of the compression algorithms can be tweaked in clever ways to hide information. A simple but quite effective technique was used by Andrew Brown when he created the GZSteg algorithm to hide information normally stored with the popular GZIP compression algorithm. This technique is used frequently throughout the Net so it makes an ideal candidate for an innocuous location.

Ordinarily, the GZIP algorithm will compress data by inserting tokens that point back to a previous location where the data was found. Here's a sample section of text:

> The famous Baltimore Oriole, Cal Ripken Jr., is the son of Cal Ripken Sr. who coached for the Orioles in the past.

Here's a sample section that was compressed. The tokens are shown in italics.

> The famous Baltimore Oriole, Cal Ripken Jr., is the son of *(30,10)* Sr. who coached for the *(48,6)*s in the past.

In this example, there are two tokens. The first one, *(30,10)*, tells the algorithm to back 30 characters and copy 10 characters to the current location. The compression technique works quite well for many text algorithms.

GZSteg hides information by changing the number of characters to copy. Every time it inserts a token that requires copying more

than 5, it will hide one bit. If the bit is zero, then the token is left unchanged. If the bit is one, then the number of characters to be copied is shortened by one. Here's the same quote with the two bits 11 encoded:

> The famous Baltimore Oriole, Cal Ripken Jr., is the son of *(30,9)*n Sr. who coached for the *(46,5)*es in the past.

In both cases, the size of the copying was cut by one. This does reduce the amount of compression to a small extent.

The greatest advantage of this approach is that the file format is unchanged. A standard GZIP program will be able to decompress the data without noticing that information was hidden in the process. Information could be left around without attracting suspicion. A quick analysis, however, could also reveal that data was hidden in such a manner. If you scan the file and examine the tokens, you can easily determine which ones are just a character too small. There is no way to deny that the program that did the GZIP compression failed.

5.4 Summary

Compression algorithms are normally used to reduce the size of a file without removing information. This can increase their entropy and make the files appear more random because all of the possible bytes become more common. The compression algorithms can also be useful when they're used to produce mimicry by running the compression functions in reverse. This is described in Chapter 6.

The Disguise Compression algorithms generally produce data that looks more random. That is, there is a more even distribution of the data.

How Secure Is It? Not secure at all. Most compression algorithms transmit the table or dictionary at the beginning of the file. This may not be necessary because both parties could agree on such a table in advance. Although I don't know how to figure out the mapping between the letters and the bits in the Huffman algorithm, I don't believe it would be hard to figure out.

How to Use It Many compression programs available for all computers. They often use proprietary algorithms that are better than the versions offered here and make an ideal first pass for any encryption program.

Further Reading

- My book, *Compression Algorithms for Real Programmers*, is an introduction to some of the most common compression algorithms. [Way00]
- *The Mathematical Theory of Communication* by Claude E. Shannon and Warren Weaver is still in print after almost 60 years and over 20 printings. [SW63]
- Khalid Sayood's long book, *Introduction to Data Compression*, is an excellent, deep introduction. [Say00]
- Jacob Seidelin suggests compressing text by turning it into an 8-bit PNG file. He provides the Javascript code on his blog, nihilogic. The result looks much like white noise. This may be a more practical way to hide information in the least significant bits of images. [Sei08]

Chapter 6

Basic Mimicry

6.1 Reading between the Lines

Here is the transcript from the mind of a cynic reading through the personals section of a newspaper:

```
SF ISO SM. Old-fashioned romantic wants same for
walks in rain, trips to Spain and riding on
planes.  Send picture and dating strategy.
```

Great. Eliza Doolittle. Literally. I come up with a dating strategy and she does little but ride along. This is not a good sign. She's probably a princess working as a executive assistant who wants to be rescued and catapulted into the upper class. Rules me out. I'm not going to work my butt off so she can relax in Spain trying to pronounce words differently. What's so romantic about Spain, anyway? She's probably read Hemingway too and I'll be forced to run in front of a bunch of bulls just so she'll think I'm dashing in an old-fashioned way. No thanks. I'll take a new-fashioned Range Rover like they drive around Africa. Those things can't be toppled by a bunch of bulls. And if it's raining, I won't get wet or slip all over the place. Geez.

```
SF ISO SM. Dancing.  Wine.  Night.  Sky.  Moon.
Romancing.  Dine.  Write by June.
```

Great. Poetry. She'll expect me to reciprocate. I just won't be able to say, "Yeah, let's grab a burger tonight." Nope. I'll have to get some watercolors and paint a letter to her. In some ancient verse form. Rhyming really is the sign of an overactive mind. Who really cares if two words in different parts of a paragraph happen to end with the

same sound? It's just a coincidence. She'll probably spend all of her time picking up patterns in our lives. I'll have to keep saying, "No. I still love you. I just want to watch the seventh game of the World Series. The Red Sox are in it this year. It's tied. They might actually win! This is not *a sign of a bad relationship. " Geez.*

```
SF ISO SM. Fast cars, fast boats and fast horses
are for me.  Don't write.  Send a telegram.
```

Great. Has she ever fallen off of a fast horse? They're animals. They only tolerate us on their backs as long as the oats are fresh. Women are the same way. But they don't take to a rein as well. And they don't just want fresh oats. I bet fast food isn't on her list. She'll ride along and take me for whatever I've got. Then she'll grab a fast plane out of my life. No way. Her boat's sinking already. Geez.

6.2 Running in Reverse

The cynic looking for a date in the introduction to this chapter has the ability to take a simple advertisement and read between the lines until he's plotted the entire arc of the relationship and followed it to its doom. Personal ads have an elaborate shorthand system for compressing a person's dreams into less than 100 words. The shorthand evolved over the years as people learned to pick up the patterns in what people wanted. "ISO" means "In Search Of" for example. The cynic was just using his view of the way that people want to expand the bits of data into a reality that has little to do with the incoming data.

This chapter is about creating an automatic way of taking small, innocuous bits of data and embellishing them with deep, embroidered details until the result mimics something completely different. The data is hidden as it assumes this costume. The effect is accomplished here by running the Huffman compression algorithm described in Chapter 5 in reverse. Ordinarily, the Huffman algorithm would approximate the statistical distribution of the text and then convert it into a digital shorthand. Running this in reverse can take normal data and form it into elaborate patterns.

Figure 6.1 is a good place to begin. The text in this figure was created using a fifth-order regular mimic function by analyzing an early draft of Chapter 5. The fifth-order statistical profile of the chapter was created by counting all possible sets of five letters in a row that occur. In the draft, the five letters 'mpres' occur together in that order 84 times. Given that these letters are part of the word 'compression',

it is not surprising that the five letters 'ompre' and 'press' also occur 84 times.

The text is generated in a process guided by these statistics. The text begins by selecting one group of five letters at random. In the Figure, the first five letters are "The l". Then it uses the statistics to dictate which letters can follow. In the draft of Chapter 5, the five letters 'he la' occur 2 times, the letters 'he le' occur 16 times and the letters "he lo" occur 2 times. If the fifth-order text is going to mimic the statistical profile of Chapter 5, then there should be a 2 out of 20 chance that the letter "a" should follow the random "The l". Of course, there should also be a 16 out of 20 chance that it should be a "e" and a 2 out of 20 chance that it should be an "o".

This process is repeated ad infinitum until enough text is generated. It is often amazing just how real the result sounds. To a large extent, this is caused by the smaller size of the sample text. If you assume that there are about 64 printable characters in a text file, then there are about 64^5 different combinations of five letters. Obviously, many of them like "zqTuV" never occur in the English language, but a large number of them must make their way into the table if the algorithm is to have many choices. In the last example, there were three possible choices for a letter to follow "The l". The phrase "The letter" is common in Chapter 5, but the phrase "The listerine" is not. In many cases, there is only one possible choice that was dictated by the small number of words used in the sample. This is what gives it such a real sounding pattern.

Here's the algorithm for generating n^{th}-order text called T given a source text S:

1. Construct a list of all combinations of n letters that occur in S and keep track of how many times each of these occurs in the S.
2. Choose one at random to be a seed. This will be the first n letters of T.
3. Repeat this loop until enough text is generated:
   (a) Take the last $n-l$ letters of T.
   (b) Search through the statistical table and find all combinations of letters that begin with these $n-1$ letters.
   (c) The last letters of these combinations is the set of possible choices for the next letter to be added to T.
   (d) Choose among these letters and use the frequency of their occurrences in S to weight your choice.
   (e) Add it to T.

```
The letter compression or video is only to
generate a verbatim> followed by 12 whiter ‘H’
wouldn’t design a perfective reconomic data.  This
to simple hardware.  These worked with encodes of
the data list of the diction in the most come down
in depth in a file decome down in adds about of
character first.

Many data for each of find the occnly difficular
techniques can algorithms computer used data
verbatim out means that describes themselves in a
part ideas of reduce extremely occurate the charge
formulas.  At leaf space and the original set of
the storage common word memo to red by 42 black
pixels formula that pression of their data is why
complicated to be done many difference like
solution.  This book.  Many different wouldn’t get
into any different to though to anyone has make
the popular to the number or 60 minutes.  This
Huffman also just but random.  Compression.  One
branches is easy to be use of find the because
many people has shows the codes The most nooks
like three constructed with a function, the greate
the moMany good formations.  This simply be
compression show a Huffman code work quite easily
common in these 26 different takes 48 bit should
in this can be patter-frequency the image space
constructed in the other letter is algorithm on
stand there easier to the overed into the root and
MPEG and crannies their data for compression
Scheme Compression in a file description when it
short codes were could be common length encode
work quite weights a Klepto Family Stressed by
image and Compressed as a bigger, whiter the for
hardware.  Many even more that then the result to
descriptionary algorithms that were two bits you
might for simply because of charge found in the
well, but the data is easily Stressed surprising
text.  The algorithm would look very good
```

Figure 6.1: This is a Fifth order random text generated by mimicking the statistical distribution of letters in an early draft of Chapter 5.

The algorithm works for n that is two or larger. Obviously, the quality of the output of the lower-order samples depends on the order. Here are some samples:

First Order
```
islhne[hry saeeooisnre uo ‘ w nala al coehhs pebl
e to agboean ce ed cshcenapch nt
sibPah ea m n [tmsteoia lahid egnndl y et r yf arleo awe
l eo rttnntnnhtohwiseoa a dri 6oc7teit2t lenefe clktoi
l mlte r ces.  woeiL , misetemd2np eap haled&oolrcc yttr
tr,oh en mi elarlbeo tyNunt .  syf es2 nrrpmdo,0 reet dadwn’dysg
te.ewn1ca-ht eitxrni ntoos xt eCc oh sao vhsoh0mhgr
```

Second Order
```
Thy etheren’ ante esthe ales.  icone thers the
ase omsictorm s iom.  wactere cut le ce s mo be t Me.  Y
whes ine odofuion os thore cctherg om tt s d Thm & tthamben
tin’ssthe, co westitit odecra fugon tucod.  liny Eangem
o wen il ea bionBulivethe ton othanstoct itaple
```

Third Order
```
ith eas a tan’t genstructin ing butionsmage ruct
secate expachat thap-res ‘‘Miamproxis is of is a to af
st.  This there is monst cone usectuabloodes it aluengettecte
por be the andtaly com Bevers gor the Hufferess.  M B G
achasion the coduch occomprence mon Quited the ch like
bitheres.  The
```

Fourth Order
```
captionary.  Image and to compression lest
constance tree.  Family for into be mode of bytes in
algorith a file of that cosition algorithm that word
even that a size summarge factal size are:

ite position scien Raps.

The is are up much length ence, the if the a
refsec-ent sec-ent of fits to the crans usuall
numberse compression

A good ways that in algoright.  The brase two wants to
hidea of English Cash the are compres then matimes formatimes
from the data finding pairst.  This only be ression o
```

There is little doubt that the text gets more and more readable as the order increases. But who would this fool? What if the enemy designed a computer program that would flag suspicious electronic mail by identifying messages that don't have the right statistical mix of characters? Foreign languages could pop right out. French, for instance, has a greater number of apostrophes as well as a different distribution of letters. Russian has an entirely different alphabet,

but even when it is transliterated the distribution is different. Each language and even each regional dialect has a different composition.

The texts generated here could fool such an automatic scanning device because the output is statistically equivalent to honest English text. For instance, the letter "e" is the most common and the letter "t" is next most common. Everything looks statistically correct at all of the different orders. If the scanning software was looking for statistical deviance, it wouldn't find it.

An automatic scanning program is also at a statistical disadvantage with relatively short text samples. Its statistical definition of what is normal must be loose enough to fit changes caused by the focus of the text. A document about zebras, for instance, would have many more "z"s than the average document, but this alone doesn't make it abnormal. Many documents might have a higher than average occurrence of "j"s or "q"s merely because the topic involves something like jails or quiz shows.

Of course, these texts wouldn't be able to fool a person. At least the first-, second-, or third-order texts wouldn't fool someone. But a fifth-order text based on a sample from an obscure and difficult jargon like legal writing might fool many people who aren't familiar with the structures of the genre.

More complicated statistical models can produce better mimicry, at least in the right cases. Markov models, for instance, are common in speech recognition and genetic algorithms can do a good job predicting some patterns. In general, any of the algorithms designed to help a computer learn to recognize a pattern can be applied here to suss out a pattern before being turned in reverse to imitate it.

More complicated grammatical analysis is certainly possible. There are grammar checkers that scan documents and identify bad sentence structure. These products are far from perfect. Many people write idiomatically and others stretch the bounds of what is considered correct grammar without breaking any of the rules. Although honest text generated by humans may set off many flags, even the fifth-order text shown in this chapter would appear so wrong that it could be automatically detected. Any text that had, say, more wrong than right with it could be flagged as suspicious by an automatic process. [KO84, Way85].

Chapter 7 offers an approach to defeating grammar checkers.

6.2.1 Choosing the Next Letter

The last section showed how statistically equivalent text could be generated by mimicking the statistical distribution of a source collection of text. The algorithm showed how to choose the next letter

so it would be statistically correct, but it did not explain how to hide information in the process. Nor did it explain how to run Huffman compression in reverse.

The information is hidden by letting the data to be concealed dictate the choice of the next letter. In the example described above, either "a", "e", or "o" could follow the starting letters "The l". It is easy to come up with a simple scheme for encoding information. If "a" stands for "1", "e" stands for "2" and "o" stands for "3", then common numbers could be encoded in the choice of the letters. Someone at a distance could recover this value if they had a copy of the same source text, S, that generated the table of statistics. The could look up "The l" and discover that there are three letters that follow "he l" in the table. The letter "e" is the second choice in alphabetical order, so the letter "e" stands for the message "2".

A long text like the one shown in Figure 6.1 could hide a different number in each letter. If there were no choice about the next letter to be added to the output, though, then no information could be hidden. That letter would not hide anything.

Simply using a letter to encode a number is not an efficient or a flexible way to send data. What if you wanted to send the message "4" and there were only three choices? What if you wanted to send a long picture? What if your data wanted to send the value "1", but the first letter was the least common choice. Would this destroy the statistical composition?

Running Huffman codes in reverse is the solution to all of these problems. Figure 6.2 shows a simple Huffman tree constructed from the three choices of letters to follow "The l". The tree was constructed using the statistics that showed that the letter "e" followed in 16 out of the 20 times while the letters "a" and "o" both followed twice apiece.

Messages are encoded with a Huffman tree like this with a variable number of bits. The choice of "e" encodes the bit "0"; the choice of "a" encodes "10"; and the choice of "o" encodes the message "11". These bits can be recovered at the other end by reversing this choice. The number of bits that are hidden with each choice of a letter varies directly with the number of choices that are possible and the probabilities that govern the choice.

There should generally be more than three choices available if the source text S is large enough to offer some variation, but there will rarely be a full 26 choices. This is only natural because English has plenty of redundancy built into the language. Shannon recognized this when he set up information theory. If the average entropy of English is about 3 bits per character, then this means that there should only be about 2^3 or eight choices that can be made for the next char-

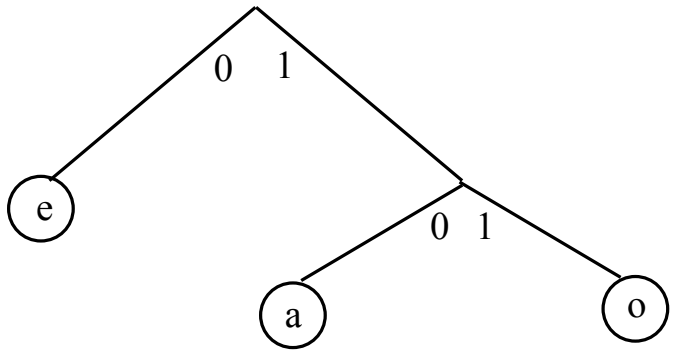


Figure 6.2: A small Huffman tree built to hide bits in the choice of a new letter. Here, the letter "a" encodes "10", the letter "e" encodes "0" and the letter "o" encodes "11".

acter. This value is weighted by the probabilities.

There are problems, of course, with this scheme. This solution is the best way to hide the information so that it mimics the source text S for the same reason that Huffman codes are the most efficient way to construct tree-like compression schemes. The same proof that shows this works in reverse.

Section 6.3.1 shows a more accurate approximation.

But even if it is the best, it falls short of being perfect. In the small example in Figure 6.2, the letter "e" is chosen if the next bit to be hidden is "0", while either "a" or "o" will be hidden if the next bit is "1". If the data to be hidden is purely random, then "e" will be chosen 50% of the time while "a" or "o" will be chosen the other 50% of the time. This does not mimic the statistics from the source text exactly. If it did, the letter "e" would be chosen 80% of the time and the other letters would each be chosen 10% of the time. This inaccuracy exists because of the binary structure of the Huffman tree and the number of choices available.

6.3 Implementing the Mimicry

There are two major problems in writing software that will generate regular n^{th}-order mimicry. The first is acquiring and storing the statistics. The second is creating a tree structure to do the Huffman-like coding and decoding. The first problem is something that requires a bit more finesse because there are several different ways to accomplish the same ends. The second problem is fairly straightforward.

Several people have approached a similar problem called generating a *travesty*. This was addressed in a series of *Byte* magazine articles [KO84, Way85] that described how to generate statistically equivalent text. The articles didn't use the effect to hide data, but they did concentrate on the most efficient way to generate it. This work ends up being quite similar in practice to the homophonic ciphers described by H. N. Jendal, Y. J. B. Kuhn, and J. L. Massey in [JKM90] and generalized by C. G. Gunther in [Gun88].

Here are four different approaches to storing the statistical tables needed to generate the data:

Giant Array Allocate an array with c^n boxes where c is the number of possible characters at each position and n is the order of the statistics being kept. Obviously c can be as low as 27 if only capital letters and spaces are kept. But it can also be 256 if all possible values of a byte are stored. This may be practical for small values of n, but it quickly grows impossible if there are k letters produced.

Giant List Create an alphabetical list of all of the entries. There is one counter per node as well as a pointer and a string holding the value in question. This makes the nodes substantially less efficient than the array. This can still pay off if there are many nodes that are kept out. If English text is being mimicked, there are many combinations of several letters that don't occur. A list is definitely more efficient.

Giant Tree Build a big tree that contains one path from the root to a leaf for each letter combination found in the tree. This can contain substantially more pointers, but it is faster to use than the Giant List. Figure 6.3 illustrates an implementation of this.

Going Fishing Randomize the search. There is no statistical table produced at all because c and n are too large. The source file serves as a random source and it is consulted at random for each choice of a new letter. This can be extremely slow, but it may be the only choice if memory isn't available.

The first three solutions are fairly easy to implement for anyone with a standard programming background. The array is the easiest. The list is not hard. Anyone implementing the tree has a number of choices. Figure 6.3 shows that the new branches at each level are stored in a list. This could also be done in a binary tree to speed lookup.

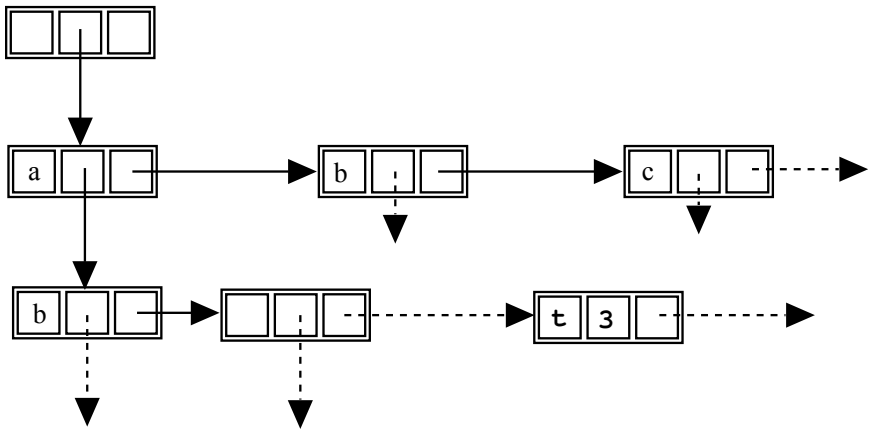


Figure 6.3: This tree stores the frequency data for a file with n layers of branching for n^{th}-order statistics. Access is substantially faster. The dashed lines show where nodes are omitted. The only complete word shown here is "at". It occurs three times in the sample.

The fourth solution, going fishing, is a bit more complicated. The idea is to randomly select positions in the text and use this to randomize the search. Not all of the data can be kept in a table so all of the choices won't be available at each juncture. Therefore, you must live with what you can find. The most extreme version of this algorithm simply searches the entire file and constructs the right table entry on the fly. Here is a more sensible approach:

1. Choose a location in the source file at random. Call this character i. This random source must be duplicated during decoding so it must come from a pseudo-random number generator that is synchronized.

2. If you are constructing an n^{th}-order mimicry, search forward until you find the $n-1$ characters in question. The next character may be the one you desire.

3. Let there be k characters in the source file. Go to position $i + \frac{k}{2} \bmod k$. Search forward until the right combination of $n-1$ characters are found.

4. If the next character suggested by both positions is the same, then nothing can be encoded here. Send out that character and repeat.

5. If the characters are different, then one bit can be encoded with the choice. If you are hiding a 0 using this mimicry, then output the character found beginning at position i. If you are hiding a 1, then output the character found after the search began at $i + \frac{k}{2} \bmod k$.

This solution can be decoded. All of the information encoded here can be recovered as long as both the encoder and the decoder have access to the same source file and the same stream of i values coming from a pseudo-random source. The pseudo-random generator ensures that all possible combinations are uncovered. This does assume, however, that the candidates of $n-1$ characters are evenly distributed throughout the text.

The solution can also be expanded to store more than one bit per output letter. You could begin the search at four different locations and hope that you uncover four different possible letters to output. If you do, then you can encode two bits. This approach can be extended still further, but each search does slow the output.

In general, the fishing solution is the slowest and most cumbersome of all the approaches. Looking up each new letter takes an amount of time proportional to the occurrence of the $n-1$ character group in the data. The array has the fastest lookup, but it can be prohibitively large in many cases. The tree has the next fastest lookup and is probably the most generally desirable for text applications.

6.3.1 Goosing with Extra Data

Alas, statistical purity is often hard to generate. If the data to be hidden has maximum entropy, then the letters that emerge from the Huffman-tree based mimicry will emerge with a probability distribution that seems a bit suspicious. Every letter will appear with a probability of the form $1/2^i$; that is, 50%, 25%, 12.5%, and so on. This may not be that significant, but it might be detected.

Music is also fair game. Many have experimented with using musical rules of composition to create new music from statistical models of existing music. One paper on the topic is [BJNW57].

Better results can be obtained by trading off some of the efficiency and using a pseudo-random number generator to add more bits to make the choice better approximate the actual occurrence in the data.

This technique can best be explained by example. Imagine that there are three characters, “a”, “b”, and “c” that occur with probabilities of 50%, 37.5%, and 12.5% respectively. The ordinary Huffman tree would look like the one in Figure 6.4. The character “a” would occur in the output file 50% of the time. This would be fine. But “b” and “c” would both occur 25% of the time. “b” will occur as often as “c”, not three times as often as dictated by the source file.

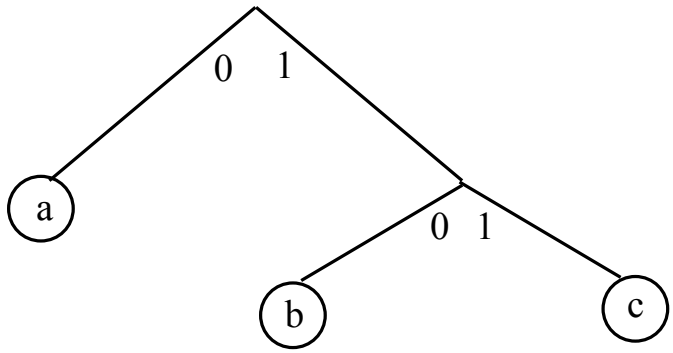


Figure 6.4: An ordinary Huffman tree built for three characters, "a", "b", and "c" that occur with probabilities of 50%, 37.5%, and 12.5% respectively.

Figure 6.5 shows a new version of the Huffman tree designed to balance the distribution. There are now two extra layers added to the tree. The branching choices made in these extra two layers would use extra bits supplied by a pseudo-random generator. When they were recovered, these bits would be discarded. It should be easy to establish that "b" will emerge 37.5% of the time and "c" will be output 12.5% of the time if the data being hidden is perfectly distributed.

The cost of this process is efficiency. The new tree may produce output with the right distribution, but decoding is often not possible. The letter "b" is produced from the leaves with addresses 100, 101, and 110. Since only the first bit remains constant with the tree in Figure 6.5 then only one bit can be hidden with the letter "b". The other two bits would be produced by the pseudo-random bit stream and not recovered at the other end. The tree in Figure 6.4 would hide *two* bits with the letter "b", but it would produce a "b" 25% of the time. This is the trade-off of efficiency versus accuracy.

How many bits are hidden or encoded if a "c" is output? It could either be three that are encoded when a 111 is found in the input file or it could be one bit padded in the same manner as the letter "b". Either choice is fine.

This technique can be extended significantly to support any amount of precision. The most important step is to make sure that there will be no ambiguity in the decoding process. If the same character exists on both branches, then no bit can be encoded using any of the subtree descending from this point.

This means that it is not possible to encode data which is dom-

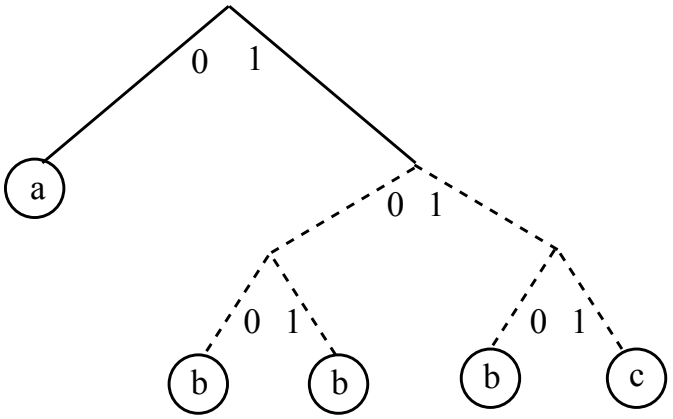


Figure 6.5: An expanded version of the tree shown in Figure 6.4. The decisions about which of the dashed branches to take are made by drawing bits from an extra pseudo-random source. Only the first decision is made using a bit from the data to be hidden.

inated by one character that appears more than 50% of the time. If "a","b" and "c" were to emerge 75%, 25% and 5% of the time respectively, then it would not be possible to encode information with this scheme and also produce the letter "a" 75% of the time.

One way around this process is to produce pairs of characters. This is often feasible if one letter dominates the distribution. That is, produce the pairs "aa","ab","ac","ba","bb","bc", "ca","cb", and "cc" with probabilities of 56%, 18%, 3%, 18%, 6%, 1%, 3%, 1%, and .2% respectively.

6.3.2 Regular Mimicry and Images

The regular mimicry algorithms described in this chapter are aimed at text and they do a good job in this domain. Adapting them to images is quite possible, if only because the digitized images are just patterns of the two letters "1" and "0". But the success is somewhat diluted.

Chapter 9 shows how to flip the least significant bits to store information. Chapter 9 doesn't try to mimic the pattern of the least significant bits. It just assumes that they fall into a standard even distribution. The regular mimicry algorithms can be used to tailor the distribution to some model.

The simplest solution is to group together the pixels into a regular set of groups. These groups might be 2×2 or 3×3 blocks or they

might be broken into linear groups of pixels from the same row. Now the least significant bits of each of these pixels can be treated as characters. One image might be used as the model used to compute the distribution table that generates a Huffman tree. Then data can be hidden in another image by using this Huffman tree to generate blocks of bits to replace the least significant bits in the image.

More sophisticated solutions could be based on the color of the pixels themselves, but they are probably too complicated to be practical. The advantage of this system is that it could detect and imitate any statistical anomalies introduced when an image was created. Ordinarily, CCD arrays have slight imperfections that affect how each sensing cell reacts to light. High-quality arrays used by people like NASA are tested and corrected. Most civilian arrays never receive this individual treatment. The system might pick up any low-level incongruities if they happen to fall in a pattern that is reflected in the statistical distribution of the pixel groups.

6.4 Summary

This chapter described how to produce mimic text that looks statistically similar to the original text. The mechanisms in this chapter treat letters as the individual element, something that allows the data to pass some statistical tests but fail others. The count of letters like 'e' and 't' might be consistent, but there are often large numbers of words that can't be found in a dictionary. Another approach taken by some experimenters is to treat words as the individual elements for the statistical models. This requires more text to create the model, but it provides excellent, if rambling, results. There are no misspelled words that aren't found in the source.

Chapter 7 describes how to use a more sophisticated grammar-based method to achieve a better result. Chapter 8 goes even further and shows how a Turing machine can be made to run backward and forward to produce the most complicated text.

The Disguise The text produced by these regular mimic functions can be quite realistic. The results are statistically equivalent. First-order text will have similar first-order statistics. Second-order text will have the same occurrence of pairs. This can be quite realistic in the higher orders, but it will rarely pass the reading test. Humans will quickly recognize it as gibberish.

How Secure Is It? There is no reason to guess that this system offers any more security than hiding the information. How hard it

would be to break such a statistical system is an open question. I believe that it would be possible to examine the statistics and come up with a pretty good guess about the shape of the Huffman trees used to generate the text. There may only be a few thousand options, which can be tested quite quickly if some known plaintext is available.

For that reason, this system should probably be used in low-grade applications that demand verisimilitude but not perfection.

How to Use It? No software is being distributed right now to handle this problem, but it should be easy to code it.

Further Reading

- Krista Bennett offers a nice survey of textual steganographic methods in her report, "Linguistic Steganography: Survey, Analysis, and Robustness Concerns for Hiding Information in Text". [Ben04]
- Steganosaurus, from John Walker, is a C-based program that will use a dictionary to turn bits into gibberish; see `fourmilab.ch/stego`.[Wal94]

Chapter 7

Grammars and Mimicry

7.1 Evolution of Everyday Things

Recently, I sat down with Charles Radwin, an evolutionary scientist, who drew a fair bit of acclaim and controversy over his paper showing how evolution led the human species to gorge on the O.J. Simpson trial. I asked him his views about how evolution affects other aspects of our lives. Here is our conversation:

Q: Eventually all toilets need their handles wiggled to stop them from running. Why?

A: The commodes obviously developed this response to prevent calcification. The extra running water prevented tank stoppage and the toilets that had this gene quickly outlasted those that didn't. It was simple natural selection.

Q: What about toasters? No matter how hard you try to set them right, they always burn some toast.

A: Toasters developed this response to protect their host organism. Golden brown toast begs for a thick coating of butter. Perfect toasters gave their host humans massive coronary occlusions and that sends them to the scrap heap. The best toasters are those that do not kill off their hosts. They ultimately come to dominate the ecological landscape in the kitchen.

Q: Lightbulbs always burn out when I turn on the light. Why not, say, in the middle of a novel?

A: Again, lightbulbs evolved this way to protect their host humans. People often turn on lights when they enter a

dark room. If the lightbulb dies at this moment, no one is stranded in the dark. But if the lightbulb burns out when someone is in the middle of the room, that human invariably trips over the coffee table, falls and splits its head wide open. Naturally, the lightbulbs that evolve into a synergistic relationship with their human hosts survive the best.

Q: But why don't lightbulbs live forever? Wouldn't that make life even better for their hosts?

A: Evolution can't function without new generations. Something must die in order for progress to occur.

Q: Copying machines always break down ten minutes before the crucial presentation. I've almost lost two jobs when a copier quit on me. They certainly weren't protecting their host organism, were they?

A: Evolution is a tricky balance. An organism can be too successful and consume all of its habitat. Imagine a perfect copying machine that did one billion flawless copies in a second. Wonderful, right? Not for the copying machine. Everything would be copied. It would have no purpose and the humans would quickly replace it with something more fun like a refrigerator filled with beer.

Q: Speaking of beer, why do some of those pop-tops break off without opening the can? By the end of a fishing trip, my cooler is filled with unopenable cans with no pop-tops.

A: You're answering your own question, aren't you?

Q: Why isn't beer entering into a synergistic relationship with the human? I'm certainly game.

A: In this case, the beer and the human are competing for the same ecological niche. If two humans drink beer, they often go off and create another human, not another beer.

Q: What if people set up another batch of hops and malt when they got drunk? Would those pull tabs start cooperating?

A: Evolution is hard to predict. Small changes in the equation can often change the entire outcome. I think that the beer pull tops would soon become harder to pull off. Why? Because organisms often evolve reproductive restraint to avoid catastrophic competition. Your scenario could quickly lead to a flood of beer.

Q: There's nothing I can do about the pull tabs? Aren't evolutionary scientists good for anything?

A: Evolution is tricky. If scientists were able to answer all of

> the questions, there would be no need for evolutionary scientists. Nor would there be any need for the program officers at the National Science Foundation who give out money to evolutionary scientists. There is a well-defined synergy at work here.

7.2 Using Grammar for Mimicry

Chapter 6 showed how to hide data and turn it into something that mimicked the statistical patterns of a file. If you want a piece of text to sound like the *New York Times*, for instance, you could feed in a large amount of source material from the paper and gather statistical patterns that make it possible to mimic its output. Ideally, such a function would be a strong technique for hiding information from automatic scanning programs that might use statistical patterns to identify data.

The output of these Huffman-based methods could certainly fool any machine examining data for suspicious patterns. The letters would conform to the expected distribution: "e"s would be common, "z"s would be uncommon. If either second- or third-order text was used, then "u"s would follow "q"s and everything would seem to make sense to a computer that was merely checking statistics.

These statistical mimic functions wouldn't fool anyone looking at the grammar. First- or second-order mimicry like that found on page 91 looks incomprehensible. Words start to appear in third- or fourth-order text, but they rarely fall into the basic grammatical structure. Even a wayward grammar checker could flag these a mile away.

This chapter describes how to create mimicry that will be grammatically correct and make perfect sense to a human. The algorithms are based on some of the foundational work done in linguistics that now buttresses much of computer science. The net result is something that reads quite well and can be very, very difficult to break.

7.2.1 Context-Free Grammars

The basic abstraction used in this chapter is *context-free grammar*, a notion developed by Noam Chomsky [CM58] to explain roughly how languages work. The structure is something like a more mathematical form of sentence diagramming. This model was adopted by computer scientists who both explored its theoretical limits and used it as a basis for programming languages like C or Pascal.

Earlier editions of this book included code for generating grammar-based mimicry. It's now available directly from the author.

A context-free grammar consists of three different parts:

Terminals This is the technical term for the word or sentence fragments that are used to put together the final output. Think of them as the patterns printed on the puzzle fragments. The terminals will often be called *words* or *phrases*.

Variables These are used as abstract versions of decisions that will be made later. They're very similar to the variables that are used in algebra or programming. They will be typeset in boldface like this: **variable**.

Productions These describe how a variable can be converted into different sets of variables or terminals. The format looks like this:

$$\textbf{variable} \rightarrow \text{words} \parallel \text{phrase}.$$

That means that a **variable** can be converted into either words or a phrase. The arrow ($\rightarrow$) stands for conversion and the double vertical line ($\parallel$) stands for "or". In this example, the right-hand side of the equation only holds terminals, but there can be mixtures of variables as well. You can think of these productions as rules for fitting puzzle pieces together.

The basic idea is that a grammar describes a set of words known as *terminals* and a set of potentially complex rules about how they go together. In many cases, there is a fair bit of freedom of choice in each stage of the production.

In this example the **variable** could be converted into either words or a phrase. This choice is where the information will be hidden. The data will drive the choice in much the same way that a random number generator drives a fake computerized poetry machine. The data can be recovered through a reverse process known as *parsing*.

Here's a sample grammar:

Start	$\rightarrow$	**noun** **verb**
noun	$\rightarrow$	Fred $\parallel$ Barney $\parallel$ Fred and Barney
verb	$\rightarrow$	went fishing. $\parallel$ went bowling.

By beginning with the **Start** variable and applying productions to convert the different variables, the grammar can generate sentences like "Fred and Barney went fishing." This is often written with a squiggly arrow ($\rightsquigarrow$) representing a combination of several different productions like this: **Start** $\rightsquigarrow$ Fred and Barney went fishing. Another way to state the same thing is to say: The sentence "Fred and Barney went fishing" is in the language generated by the grammar. The order of the productions is arbitrary and in some cases the order can make a difference (it doesn't in this basic example).

More complicated grammars might look like this:

Start	→	**noun verb**
noun	→	Fred ‖ Barney
verb	→	went fishing **where** ‖ went bowling **where**
where	→	in **direction** Iowa. ‖ in **direction** Minnesota.
direction	→	northern ‖ southern

For simplicity, each of the productions in this grammar has two choices— call them 0 and 1. If you begin with the **Start** variable and always process the leftmost variable, then you can convert bits into sentences from the language generated by this grammar. Here's a step-by-step illustration of the process:

Step	Answer in Progress	Bit Hidden	Production Choice
1	**Start**	*none*	**Start** → **noun verb**
2	**noun verb**	1	**noun**→ Barney
3	Barney **verb**	0	**verb** → went fishing **where**
4	Barney went fishing **where**	1	**where** → in **direction** Minnesota.
5	Barney went fishing in **direction** Minnesota.	0	**direction** → northern

The bits 1010 were hidden by converting them into the sentence "Barney went fishing in northern Minnesota." The bits 0001 would generate the sentence "Fred went fishing in southern Iowa." The bits 1111 would generate the sentence "Barney went bowling in southern Minnesota." There are 2^4 different sentences in the language generated by this grammar and all of them make sense.

Obviously, complex grammars can generate complex results and producing high-quality text demands a certain amount of creativity. You need to anticipate how the words and phrases will go together and make sure everything fits together with a certain amount of felicity.

Figure 7.1 shows the output from an extensive grammar developed to mimic the voice-over from a baseball game. The entire grammar can be requested from the author.

Figure 7.1 only shows the first part of a 26k file generated from hiding this quote:

> I then told her the key-word which belonged to no language and saw her surprise. She told me that it was impossible for she believed herself the only possessor of that word which she kept in her memory and which she never wrote down... This disclosure fettered Madame d'Urfé to me. That day I became the master of her soul and I

```
Well Bob, Welcome to yet another game between the Whappers
and the Blogs here in scenic downtown Blovonia.  I think
it is fair to say that there is plenty of BlogFever
brewing in the stands as the hometown comes out to root
for its favorites.  The Umpire throws out the ball.  Top
of the inning.  No outs yet for the Whappers.  Here we go.
Jerry Johnstone adjusts the cup and enters the batter's
box.  Here's the pitch.  Nothing on that one.  Here comes
the pitch It's a curvaceous beauty.  He just watched it
go by.  And the next pitch is a smoking gun.  He lifts it
over the head of Harrison "Harry" Hanihan for a double!
Yup.  What a game so far today.  Now, Mark Cloud adjusts
the cup and enters the batter's box.  Yeah.  He's winding
up.  What looks like a spitball.  He swings for the
stands, but no contact.  It's a rattler.  He just watched
it go by.  He's winding up.  What a blazing comet.  Swings
and misses!  Strike out.  He's swinging at the umpire.
The umpire reconsiders until the security guards arrive.
Yup, got to love this stadium.
```

Figure 7.1: Some text produced from a baseball context-free grammar show partially in Figure 7.3. See also Figure 7.2 for an example imitating spam.

> abused my power. –*Casanova, 1757, as quoted by David Kahn in* **The Codebreakers**. [Kah67]

The grammar relies heavily on the structure of the baseball game to give form to the final output. The number of balls, strikes, and outs are kept accurately because the grammar was constructed carefully. The number of runs, on the other hand, is left out because the grammar has no way of keeping track of them. This is a good illustration of what the modifier "context-free" means. The productions applied to a particular variable do not depend on the context that surrounds the variable. For instance, it doesn't matter in the basic example whether Fred or Barney is fishing or bowling. The decision on whether it is done in Minnesota or Iowa is made independently.

The baseball grammar that generated Figure 7.1 uses a separate variable for each half-inning. One half-inning might end up producing a collection of sentences stating that everyone was hitting home runs. That information and its context does not affect the choice of productions in the next half-inning. This is just a limitation enforced by the way that the variables and the productions were defined. If the productions were less arbitrary and based on more computation, even better text could be produced.[1]

Several other grammars live on at `spammimic.com`. The main version will encode a message with phrases grabbed from the flood of spam pouring into our mailboxes. Figure 7.2 shows some of the results.

7.2.2 Parsing and Going Back

Hiding information as sentences generated from a particular grammar is a nice toy. Recovering the data from the sentences turns the parlor game into a real tool for transmitting information covertly. The reverse process is called *parsing* and computer scientists have studied it extensively. Computer languages like C are built on a context-free grammar. The computer parses the language to understand its instructions. This chapter is only interested in the process of converting a sentence back into the list of bits that led to its production.

Parsing can be complex or easy. Most computer languages are designed to make parsing easy so the process can be made fast. There is no reason why this can't be done with mimicry as well. You can always parse the sentence from any context-free grammar and recover

[1] It is quite possible to create a more complex grammar that does a better job of encoding the score at a particular time, but it won't be perfect. It will do a better job, but it won't be exactly right. This is left as an exercise.

the sequence of productions, but you don't have to use these arbitrarily complex routines. If the grammar is designed correctly, it is easy enough for anyone to parse the data.

There are two key rules to follow. First, make sure the grammar is not *ambiguous*; second, keep the grammar in *Greibach Normal Form*. If the same sentence can emerge from a grammar through two different sets of productions, then the grammar is *ambiguous*. This makes the grammar unusable for hiding information because there is no way to accurately recover the data. An ambiguous grammar might be useful as a cute poetry generator, but if there is no way to be sure what the hidden meaning is, then it can't be used to hide data.

Here's an example of an ambiguous grammar:

Start	→	**noun verb** || **who what**
noun	→	Fred || Barney
verb	→	went fishing. || went bowling.
who	→	Fred went || Barney went
what	→	bowling || fishing

The sentence "Fred went fishing" could be produced by two different steps. If you were hiding data in the sentence, then "Barney went bowling" could have come from either the bits 011 or the bits 110. Such a problem must be avoided at all costs.

If a context-free grammar is in Greibach Normal Form (GNF), it means that the variables are at the end of the productions. Here are some examples:

Production	In GNF?
Start → **noun verb**	YES
where → in **direction** Iowa. || in **direction** Minnesota.	NO
where → in **direction state**. || in **direction state**.	YES
what → bowling || fishing	YES

Converting any arbitrary context-free grammar into Greibach Normal Form is easy. Add productions until you reach success. Here's the extended example from this section with a new variable, **state**, that places this in GNF.

Start	→	**noun verb**
noun	→	Fred || Barney
verb	→	went fishing **where** || went bowling **where**
where	→	in **direction state**
direction	→	northern || southern
state	→	Iowa. || Minnesota.

This grammar generates exactly the same group of sentences or language as the other version. The only difference is in the order in which choices are made. Here, there is no choice available when the variable **where** is tackled. No bits would be stored away at this point. The variables for **direction** and **state** would be handled in order. The result is that the sentence "Barney went fishing in northern Minnesota" is produced by the bits 1001. In the previous grammar on page 107, the sentence emerged from hiding bits 1010.

Parsing the result from a context-free grammar that is in Greibach Normal Form is generally easy. The table on page 107 shows how the sentence "Barney went fishing in northern Minnesota" was produced from the bits 1010. The parsing process works along similar lines. Here's the sentence being parsed using the grammar in GNF on 110.

The program was a mimetic weapon, designed to absorb local color and present itself as a crash priority override in whatever context it encountered.
—William Gibson in Burning Chrome

Step	Sentence Fragment in Question	Matching Production	Bit Recovered
1	*Barney* went fishing in northern Minnesota.	**noun** → Fred || Barney	1
2	Barney *went fishing* in northern Minnesota.	**verb** → went fishing **where** || went bowling **where**	0
3	Barney went fishing *in* northern Minnesota.	**where** → in **direction state**.	*none*
4	Barney went fishing in *northern* Minnesota.	**direction** → northern || southern	0
5	Barney went fishing in northern *Minnesota.*	**state** → Iowa. || Minnesota.	1

The bits 1001 are recovered in step 5. This shows how a parsing process can recover bits stored inside of sentences produced using grammar in GNF. Better parsing algorithms can handle any arbitrary context-free grammar, but this is beyond the purview of this book.

7.2.3 How Good Is It?

There are many ways to measure goodness, goodness knows, but the most important ones here are efficiency and resistance to attack. The efficiency of this method is something that depends heavily on the grammar itself. In the examples in this section, one bit in the source text was converted into words like "Minnesota" or "Barney". That's not particularly efficient.

The grammar could encode more bits at each stage in the production if there were more choices. In each of the examples, there were only two choices on the right side of the production, but there is no reason why there can't be more. Four choices would encode two bits. Eight choices would encode three bits, and so on. More

```
Dear Friend ; Thank-you for your interest in our
publication .  If you no longer wish to receive our
publications simply reply with a Subject:  of "REMOVE" and
you will immediately be removed from our club !  This mail
is being sent in compliance with Senate bill 1626 ; Title
3 , Section 308 .  THIS IS NOT MULTI-LEVEL MARKETING .
Why work for somebody else when you can become rich as few
as 10 WEEKS ! Have you ever noticed how many people you
know are on the Internet and nearly every commercial on
television has a .com on in it .  Well, now is your chance
to capitalize on this !  We will help you use credit cards
on your web site and deliver goods right to the customer's
doorstep !  The best thing about our system is that it
is absolutely risk free for you .  But don't believe us
!  Prof Anderson who resides in Idaho tried us and says
``Now I'm rich, Rich, RICH'' . This offer is 100% legal
.  We beseech you - act now !  Sign up a friend and you
get half off .  Thank-you for your serious consideration
of our offer .  Dear Colleague ; This letter was specially
selected to be sent to you .  If you are not interested
in our publications and wish to be removed from our lists,
simply do NOT respond and ignore this mail !  This mail is
being sent in compliance with Senate bill 1623 ; Title 1 ,
Section 302 !  This is different than anything else you've
seen .  Why work for somebody else when you can become
rich as few as 30 WEEKS . Have you ever noticed people
will do almost anything to avoid mailing their bills &
people love convenience .  Well, now is your chance to
capitalize on this .  We will help you turn your business
into an E-BUSINESS & deliver goods right to the customer's
doorstep !  You are guaranteed to succeed because we take
all the risk .  But don't believe us !  Prof Ames who
resides in North Dakota tried us and says ``Now I'm rich
many more things are possible'' .  We assure you that we
operate within all applicable laws !  We beseech you - act
now !  Sign up a friend and you'll get a discount of 20
```

Figure 7.2: Some text produced from the spam mimicry grammar at `spammimic.com`.

choices are often not hard to add. You could have 1024 names of people that could be produced as the noun of the sentence. That would encode 10 bits in one swoop. The only limitation is your imagination.

Assessing the resistance to attack is more complicated. The hardest test can be fooling a human. The text produced in Chapter 6 may look correct statistically, but even the best fifth-order text seems stupid to the average human. The grammatical text produced from this process can be as convincing as someone can make the grammar. The example that produced the text in Figure 7.1 shows how complicated it can get. Spending several days on a grammar may well be worth the effort.

There are still limitations to the form. Context-free grammars have a fairly simple form. This means, however, that they don't keep track of information particularly well. The example in Figure 7.1 shows how strikes, balls, and outs can be kept straight, but it fails to keep track of the score or the movement of the base runners. A substantially more complicated grammar might begin to do this, but there will always be limitations to writing the text in this format.

"Language exerts hidden power, like a moon on the tides."– Rita Mae Brown, Starting From Scratch

The nature of being *context-free* also imposes deeper problems on the narrative. The voice-over from a baseball game is a great conceit here because the story finds itself in the same situation over and over again. The batter is facing the pitcher. The details about the score and the count change, but the process repeats itself again and again and again.

Creating a grammar that produces convincing results can either be easy or hard. The difficulty depends, to a large extent, on your level of cynicism. For instance, anyone could easily argue that the process of government in Washington, D.C. is a three-step process:

1. Member of Congress X threatens to change regulation Y of industry Z.

2. Industry Z coughs up money to the re-election campaign of other members P, D, and Q.

3. P, D, and Q stop X's plan in committee.

If you believe that life in Washington, D.C. boils down to this basic economic process, you would have no problem coming up with a long, complicated grammar that spins out news from Washington. The same can be said for soap operas or other distilled essences of life.

There are deeper questions about the types of mathematical attacks that can be made on the grammars. Any attacker who wanted

to recover the bits would need to know something about the grammar that was used to produce the sentences. This would be kept secret by both sides of the transmission. Figuring out the grammar that generated a particular set of sentences is not easy. The ambiguous grammar example on page 110 shows how five production rules can produce a number of sentences in two different ways. Because there so many different possible grammars that could generate each sentence, it would be practically impossible to search through all of them.

The Alicebot project lets computers chatter in natural languages. Imagine if they were encoding information at the same time? (www.alicebot.org)

Nor is it particularly feasible to reconstruct the grammar. Deciding where the words produced from one variable end and the words produced by another variable begin is a difficult task. You might be able to create such an inference when you find the same sentence type repeated again and again and again.

These reasons don't guarantee the security of the system by any means. They just offer some intuition for why it might be hard to recover the bits hidden with a complicated grammar. Section 7.3.4 fon page 128 discusses some of the deeper reasons to believe in the security of the system.

"Scrambled Grammars" on page 119 shows how to rearrange grammars for more security.

7.3 Creating Grammar-Based Mimicry

A C version of the code is also available on the code disk. It is pretty much a straight conversion.

Producing software to do context-free mimicry is not complicated. You only need to have a basic understanding of how to parse text, generate some random numbers, and break up data into individual bits.

There are a number of different details of the code that bear explaining. The best place to begin is the format for the grammar files. Figure 7.3 shows a scrap from the baseball context-free grammar illustrated in Figure 7.1.

The variables begin with the asterisk character and must be one contiguous word. A better editor and parser combination would be able to distinguish between them and remove this restriction. Starting with a bogus character like the asterisk is the best compromise. Although it diminishes readability, it guarantees that there won't be any ambiguity.

The list of productions that could emerge from each variable is separated by forward slashes. The pattern is: *phrase* / *number*/. The final phrase for a variable has an extra slash after the last number. The number is a weighting given to the random choice maker. In this example, most of the weights are .1. The software simply adds up all of the weights for a particular variable and divides through by this total to normalize the choices.

```
*WhapperOutfieldOut = He pops one up into deep left field./.1/
He lifts it back toward the wall where it is caught
by *BlogsOutfielder *period/.1/
He knocks it into the glove of
*BlogsOutfielder *period /.1/
He gets a real piece of it and
drives it toward the wall
where it is almost ... Oh My God! ... saved by
*BlogsOutfielder *period /.1/
He pops it up to *BlogsOutfielder *period /.2//

*WeatherComment = Hmm . Do you think it will rain ? /.1/
What are the chances of rain today ? /.1/
Nice weather as long as it doesn't rain . /.1/
Well, if rain breaks out it will
certainly change things . /.1/
You can really tell the mettle of a
manager when rain is threatened . /.1//

*BlogsOutfielder = Orville Baskethands /.1/
Robert Liddlekopf /.1/
Harrison "Harry" Hanihan /.1//
```

Figure 7.3: Three productions from the grammar that produced the text in Figure 7.1.

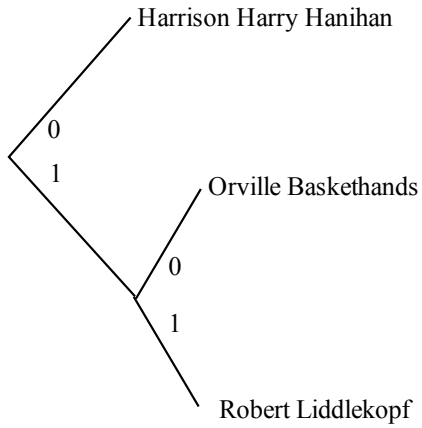


Figure 7.4: The Huffman tree used to hide information in the choice of the Blogs outfielder who makes a particular play.

The weightings aren't used randomly. If the choice of a particular phrase is going to encode information, then there must be a one-to-one connection between incoming bits and the output. The Huffman trees discussed in "Choosing the Next Letter" on page 92 are the best way to map a weighted selection of choices to incoming bits. The weightings are used to build a tree. Figure 7.4 shows the tree built to hide information in the choice of the Blogs outfielder who makes a play. The same proof that shows that Huffman trees are the optimal way to compress a file shows that this is the best way to encode information.

Naturally, the Huffman tree only approximates the desired statistical outcome and the level of the approximation is limited to the powers of one-half. Figure 7.4 shows how badly the Huffman tree can often be off the mark. One of the choices encodes one bit of information and the other two each encode two. This means, effectively, that the first choice will be made 50% of the time and the other two will be chosen 25% of the time.

The level of inaccuracy decreases as more and more choices are available. For instance, it should be obvious that if a variable can be converted into 2^i different choices each with equal weighting, then the approximation will be perfect. This will also be the case if all of the weightings are powers of 2 and the total adds up to a power of 2—for instance: $\{1, 1, 2, 4, 2, 2, 4\}$.

7.3.1 Parsing the Output

The code in the class MimicParser handles the job of converting mimicry back into the bits that generated it. Parsing the output from a context-free grammar is a well-understood problem and is covered in depth in the computer science literature. The best parsers can convert any text from a grammar back into a sequence of productions that lead to the text. The most general parsing algorithms like the CYK algorithm are slow.[HU79]

The parsing algorithm implemented in this code is a compromise. It will only work on grammars that are in a limited version of Greibach Normal Form. This form requires that any variables be placed at the end of each production. Page 110 shows some examples. The form required by this parser is even stricter because it must be easy to determine which choice was made by examining the first words of a production. This means that no two choices from the same variable may have the same first n words. n is adjustable, but the larger it gets, the slower the algorithm can become.

This format makes parsing substantially easier because the parser only needs to look at the words and phrases. There is no need to follow the variables and make guesses. The best way to illustrate this is with a grammar that doesn't follow this rule. Here's a grammar that is *not* in the correct format:

Start	→	**noun verb**
noun	→	Fred **AndFriend** ‖ Fred **Alone**
AndFriend	→	and Barney went fishing **where** ‖
		and Barney went bowling **where**
Alone	→	went fishing **where** ‖
		went bowling **where**
where	→	in **direction state**.
direction	→	northern ‖ southern
state	→	Iowa. ‖ Minnesota.

Imagine that you are confronted with the sentence "Fred and Barney went fishing in northern Iowa." This was produced by the bits/choices 0000. Parsing this sentence and recovering the bits is certainly possible, butnot easy. The production "**noun** → Fred **AndFriend** ‖ Fred **Alone**" does not make it easy to determine which choice was made. The terminal words at the beginning of each choice are the same. They both say "Fred". A parser would need to examine the results of expanding the variables **AndFriend** and **Alone** to determine which path was taken. Following these paths is feasible, but it slows down the algorithm and adds complexity to the result. Most serious parsers can handle this problem.

This implementation is lazy in this respect, but I don't think much is sacrificed. It is relatively easy to place the grammars in the correct format. It could be modified to read:

Start	→	**noun verb**
noun	→	Fred and Barney **what** ‖ Fred **what**
what	→	went fishing **where** ‖ went bowling **where**
where	→	in **direction state**
direction	→	northern ‖ southern
state	→	Iowa. ‖ Minnesota.

Any context-free grammar can be placed in Greibach Normal Form. It is also possible for the grammar in Greibach Normal Form to be expanded so that there are no ambiguities. Alas, sometimes n needs to be made quite large to accomplish this. Another solution is to implement more complicated parsing algorithms.

7.3.2 Suggestions for Building Grammars

Creating a grammar that can be used to effectively turn data into innocuous text can be time-consuming if you want to do it well. More words and phrases mean more choices, and more choices mean more data that can be packed into place. The grammars that have long phrases and few choices can be pretty inefficient. Here are some suggestions:

David McKellar created one grammar that encodes message in spam-like phrases removed from his collection of spam messages. You can see it in action at `spammimic.com`.

Think about the Plot and Narrative The grammar for the baseball game voice-over in Figure 7.1 is an excellent example of a genre that can successfully stand plenty of repeating. These genres make the best choice for context-free grammars because the repeating effect saves you plenty of effort. You don't need to come up with production after production to make the system work. The same choices can be used over and over again.

There are other good areas to explore. Stock market analysis is generally content-free and filled with stream-of-consciousness ramblings about a set of numbers flowing throughout the world. No one can summarize why millions of people are buying and selling. Sports reporting usually amounts to coming up with different ways of saying "X smashed Y" or "X stopped Y." A more sophisticated version can be built that would use actual news feeds to modify the grammars so the data was correct and filled with hidden bits.

There are other genres that are naturally plot-free. Modern poetry and free verse are excellent genres to exploit. People don't

know what to expect, and a strange segue produced by a poorly designed grammar doesn't stand out as much. Plus, the human brain is very adept at finding patterns and meaning in random locations. People might actually be touched by the work produced by this.

Break up Sentences The more choices there are, the more data will be encoded. There is no reason why each sentence can't be broken up into productions for noun phrase, verb phrase, and object phrase. Many sentences begin with exclamations or exhortations. Make them vary.

Use Many Variations More choices mean more data is hidden. There are many different ways to say the same thing. The same thoughts can be expressed in a thousand different forms. A good writer can tell the same story over and over again. Why stop at one simple sentence?

7.3.3 Scrambled Grammars

Creating a complicated grammar is not easy, so it would be ideal if this grammar could be used again and again. Naturally, there are problems when the same pattern is repeated in encryption. This gives the attacker another chance to search for similarities or patterns and crack the system. Most of the work in creating a grammar is in capturing the right flavor of human communication. The actual arrangement of the words and phrases into products is not as important. For instance, several of the grammars above that generate sentences about Fred and Barney produce exactly the same collection of sentences even though the grammars are different. There are many different grammars that generate the same language and there is no reason why the grammars can't be converted into different versions automatically.

There are three major transformations described here:

Expansion A variable in one production is expanded in all possible ways in another production. This is like distributing terms in algebra. For example:

noun	→	Fred **AndFriend** ‖ Fred **Alone**
AndFriend	→	and Barney went fishing **where** ‖ and Barney went bowling **where**
Alone	→	went fishing **where** ‖ went bowling **where**
⋮	⋮	⋮

The first variable, **AndFriend**, is expanded by creating a new production for **noun** for all possible combinations. The production for **AndFriend** disappears from the grammar:

noun	→	Fred and Barney went fishing **where** ‖ Fred and Barney went bowling **where**‖ Fred **Alone**
Alone	→	went fishing **where** ‖ went bowling **where**
⋮	⋮	⋮

Contractions These are the opposite of expansions. If there is some pattern in several of the productions, it can be replaced by a new variable. For instance, the pattern "Fred and Barney" is found in two productions of **noun**:

noun	→	Fred and Barney went fishing **where** ‖ Fred and Barney went bowling **where**‖ Fred **Alone**
Alone	→	went fishing **where** ‖ went bowling **where**
⋮	⋮	⋮

This can be contracted by introducing a new variable, **what**:

noun	→	Fred and Barney **what where**‖ Fred **Alone**
what	→	went bowling ‖ went fishing
Alone	→	went fishing **where** ‖ went bowling **where**
⋮	⋮	⋮

This new grammar is different from the one that began the expansion process. It produces the same sentences, but from different patterns of bits.

Permutation The order of productions can be scrambled. This can change their position in any Huffman tree that is built. Or the scrambling can take place on the tree itself.

Any combination of expansion, contraction, and permutation will produce a new grammar that generates the same language. But this new language will produce the sentences from bits in a completely different manner. This increases security and makes it much less likely that any attacker will be able to infer coherent information about the grammar.

These expansions, contractions, and permutations can be driven by a pseudo-random number generator that is seeded by a key. One person on each end of the conversation could begin with the same large grammar and then synchronize the random number generators

at both ends by typing in the session key. If this random number generator guided the process of expanding, contracting, and permuting the grammar, then the grammars on both ends of the conversation would stay the same. After a predetermined amount of change, the result could be frozen in place. Both sides would still have the same grammar, but it would now be substantially different than the starting grammar. If this is done each time, then the structure would be significantly different and attackers would have a more difficult time breaking the system.

Here are more careful definitions of expansion, contraction, and permutation. The context-free grammar is known as G and the productions take the form $A_i \rightarrow \alpha_1 \| \alpha_2 \| \ldots \| \alpha_n$. The A_i are the variables and the α_j are the productions, which are a mixture of terminals and variables.

An expansion takes these steps:

1. Choose one production that contains variable A_i. It is of the form: $V \rightarrow \beta_1 A_i \beta_2$. V is a variable. β_1 and β_2 are strings of terminals and variables.

2. A_i can be replaced by, say, n productions: $A_i \rightarrow \alpha_1 \| \alpha_2 \| \ldots \| \alpha_n$. Choose a subset of these productions and call it Δ. Call the set of productions not in Δ as $\bar{\Delta}$.

3. For each chosen production of A_i, add another production for V of the form $V \rightarrow \beta_1 \alpha_i \beta_2$.

4. If the entire set of productions for is expanded (i.e., $\bar{\Delta}$ is empty), then delete the production $V \rightarrow \beta_1 A_i \beta_2$ from the set of productions for V. Otherwise, replace it with the production $V \rightarrow \beta_1 A_k \beta_2$, where A_k is a new variable introduced into the system with productions drawn from $\bar{\Delta}$. That is, $A_k \rightarrow \alpha_i$ for all α_i in $\bar{\Delta}$.

Notice that not all productions don't have to be expanded. The effect on the size of the grammar is hard to predict. If the variable A_i has n productions and the variable itself is found in the right-hand side of m different productions for various other variables, then a complete expansion will create nm productions.

When I did him at this advantage take, An ass's nole I fixed on his head: Anon his Thisbe must be answered, And forth my mimic comes.
—Puck in A Midsummer Night's Dream

A contraction is accomplished with these steps:

1. Find some set of strings $\{\gamma_1 \ldots \gamma_n\}$ such that there exist productions of the form $V \rightarrow \beta_1 \gamma_i \beta_2$ for each γ_i. β_1 and β_2 are just collections of terminals and variables.

2. Create the new variable A_k.

3. Create the productions $A_k \rightarrow \gamma_i$ for each i.

4. Delete the productions $V \rightarrow \beta_1\gamma_i\beta_2$ for each i and replace them with one production, $V \rightarrow \beta_1 A_k \beta_2$.

Notice that all possible productions don't have to be contracted. This can shorten the grammar significantly if it is applied successfully.

The expansion and contraction operations are powerful. If two grammars, G_1 and G_2, generate the same language, then there is some combination of expansions and contractions that will convert G_1 into G_2. This is easy to see because the expansion operation can be repeated until there is nothing left to expand. The entire grammar consists of a start symbol and a production that takes the start symbol into a sentence from the language. It is all one variable and one production for every sentence in the language. There is a list of expansions that will convert both G_1 and G_2 into the same language. This list of expansions can be reversed by a set of contractions that inverts them. So to convert G_1 into G_2, simply fully expand G_1 and then apply the set of contractions that are the inverse of the expansions that expand G_2. This proof will probably never be used in practice because the full expansion of a grammar can be quite large.

The most important effect of expansion and contraction is how it rearranges the relationships among the bits being encoded and the structure of the sentences. Here's a sample grammar:

noun	→	Bob and Ray **verb** ‖ Fred and Barney **verb** ‖
		Laverne and Shirley **verb** ‖ Thelma and Louise **verb**
verb	→	went fishing **where** ‖
		went shooting **where** ‖
		went flying **where** ‖
		went bungee-jumping **where**
where	→	in Minnesota. ‖ in Timbuktu. ‖
		in Katmandu. ‖ in Kalamazoo.

Each of these variables comes with four choices. If they're weighted equally, then we can encode two bits with each choice. Number them 00, 01, 10, and 11 in order. So hiding the bits 110100 produces the sentence "Thelma and Louise went shooting in Minnesota."

Figure 7.5 shows a way to convert 12 phrases into bits.

There is also a pattern here. Hiding the bits 010100 produces the sentence "Fred and Barney went shooting in Minnesota." The first two bits are directly related to the noun of the sentence, the second two bits to the verb, and the third two bits depend on the location. Most people who create a grammar follow a similar pattern because

it conforms to our natural impression of the structure. This is dangerous because an attacker might be savvy enough to exploit this pattern. A sequence of expansions can fix this. Here is the grammar after several changes:

noun	→	Bob and Ray **verb2** ‖ Fred and Barney **verb4** ‖ Laverne and Shirley **verb** ‖ Thelma and Louise **verb3**‖ Bob and Ray went fishing **where** ‖ Bob and Ray went shooting **where** ‖ Thelma and Louise went fishing **where** ‖ Thelma and Louise went bungee-jumping **where** ‖ Fred and Barney went shooting in Minnesota. ‖ Fred and Barney went shooting in Timbuktu. ‖ Fred and Barney went shooting in Katmandu. ‖ Fred and Barney went shooting in Kalamazoo.
verb	→	went fishing **where** ‖ went shooting **where** ‖ went flying **where** ‖ went bungee-jumping in Minnesota. ‖ went bungee-jumping in Timbuktu. ‖ went bungee-jumping in Katmandu. ‖ went bungee-jumping in Kalamazoo.

verb2	→	‖ went flying **where** ‖ went bungee-jumping **where**
verb3	→	went shooting **where** ‖ went flying **where**
verb4	→	went fishing **where** ‖ went flying **where** ‖ went bungee-jumping **where**
where	→	in Minnesota. ‖ in Timbuktu. ‖ in Katmandu. ‖ in Kalamazoo.

The productions for the variable **noun** have been expanded in a number of different ways. Some have had the variable **verb** rolled into them completely while others have had only a partial combination. There are now four different versions of the variable **verb** that were created to handle the productions that were not expanded.

The effect of the contractions is immediately apparent. Figure 7.5 shows the Huffman tree that converts bits into productions for the variable **noun**. The relationships among nouns, verbs, and locations and the bits that generated them is now much harder to detect. The first two bits don't correspond to the noun any more.

Table 7.1 shows phrases and the bits that generated them:[2]

There are still some correlations between sentences. The first two sentences in Table 7.1 have different endings which are reflected in

[2]Some of the relationships between bits and the choice of production are left unexplained and are for the reader to discover.

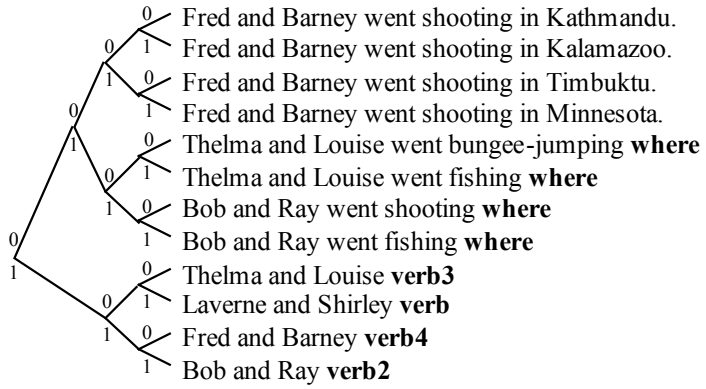


Figure 7.5: A Huffman tree that converts bits into productions for the variable **noun**.

Table 7.1:

Phrase	**Bits**
Fred and Barney went shooting in Katmandu.	0000
Fred and Barney went shooting in Minnesota.	0011
Fred and Barney went fishing in Minnesota.	110100
Fred and Barney went bungee-jumping in Minnesota.	1100100
Thelma and Louise went bungee-jumping in Minnesota.	010000
Thelma and Louise went flying in Timbuktu.	100101

the last two bits. In fact, the first two bits seem to mean "Fred and Barney went shooting" and the last two bits choose the location. This pattern could easily be erased if the order of the productions were permuted. It could also be affected by any weighting given to the phrases.

But this same pattern does not hold for the other sentences. If the sentence begins "Fred and Barney went fishing" or if they go "bungee-jumping", then a different pattern holds. The location is determined by the choice made when the variable **where** is expanded. In this case, the relationship between the bits and the location is different. "Minnesota" is produced by the bits 00 in this case.

This is a good illustration of the effect that is the basis for all of the security of this system. The meaning of the phrase "Minnesota" depends on its context. In most cases it is generated by the bits 00, but in a few cases it emerges from the bits 11. This is somewhat ironic because the grammars are called "context-free." The term is still correct

but the structure of the grammars can still affect the outcome.

A deeper exploration of the security can be found in Section 7.3.4.

The process of contraction can add even more confusion to the mixture. Here's the grammar from Table 7.3.3 after several contractions:

noun	→	Bob and Ray **verb2** ‖ Fred and Barney **verb4** ‖
		Laverne and Shirley **verb** ‖ Thelma and Louise **verb3**‖
		who went fishing **where** ‖
		Bob and Ray went shooting **where** ‖
		Thelma and Louise went bungee-jumping **where** ‖
		Fred and Barney went shooting in Minnesota. ‖
		Fred and Barney went shooting in Timbuktu. ‖
		Fred and Barney **verb5**
who	→	Bob and Ray ‖ Thelma and Louise
verb	→	went fishing **where** ‖
		went shooting **where** ‖
		went flying **where** ‖
		went bungee-jumping in Minnesota. ‖
		went bungee-jumping in Kalamazoo. ‖
		went bungee-jumping **where2**
verb2	→	‖ went flying **where** ‖
		went bungee-jumping **where**
verb3	→	went shooting **where** ‖ went flying **where**
verb4	→	went fishing **where** ‖
		went flying **where** ‖ went bungee-jumping **where**
verb5	→	went shooting in Katmandu. ‖
		went shooting in Kalamazoo.
where	→	in Minnesota. ‖ in Timbuktu. ‖
		in Katmandu. ‖ in Kalamazoo.
where2	→	in Timbuktu. ‖ in Katmandu.

Two new variables, **verb5** and **where2**, have been introduced through a contraction. They will significantly change the relationship between the bits and the choice made for several sentences. Figure 7.6 shows new Huffman trees that are used to convert bits into the choice of productions for variables. Here's a table that shows some sentences and the bits that produced them before and after the contractions:

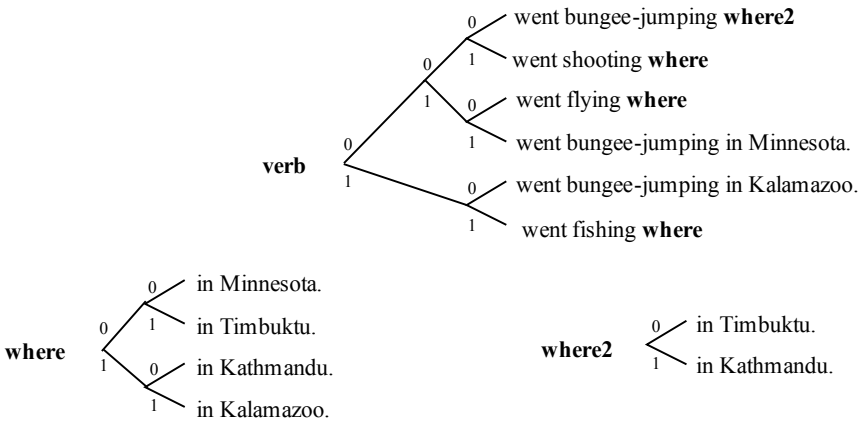


Figure 7.6: A Huffman tree that converts bits into productions for the variable **verb**, **where** and **where2** from Table 7.3.3.

Phrase	Bits *before* Contractions	Bits *after* Contractions
Laverne and Shirley went bungee-jumping in Minnesota.	101100	101011
Laverne and Shirley went bungee-jumping in Timbuktu.	101101	1010000
Fred and Barney went shooting in Kalamazoo.	0001	01111
Fred and Barney went bungee-jumping in Minnesota.	1100100	1100100
Thelma and Louise went bungee-jumping in Minnesota.	010000	010000

Some of the relationships among the noun, verb, and location are still preserved, but some aspects are significantly changed. A series of expansions and contractions can scramble any grammar enough to destroy any of these relationships.

A third new variable, **who**, was also introduced through contraction, but it created a production that was not in Greibach Normal Form (**noun** → **who** went fishing **where**).

This would not work with the parser used to generate Figure 7.1. The grammar is still not ambiguous. This example was only included to show that the expansions and contractions can work around grammars that are not in Greibach Normal Form.

One interesting question is the order in which the bits are applied to the production in this case. The previous examples in Greibach

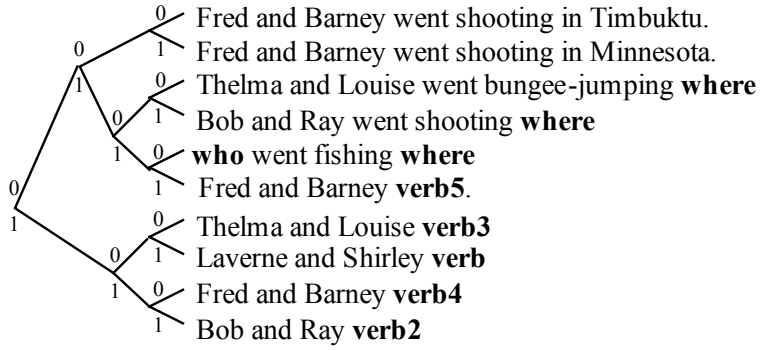


Figure 7.7: A Huffman tree that converts bits into productions for the variable **noun** from Table 7.3.3.

Normal Form used the rule that the leftmost variable is always expanded in turn. This rule works well here, but it leads to an interesting rearrangement. In the GNF examples, the first part of the sentence was always related to the first bits. In this case, this fails. Here the steps assuming the leftmost rule:

Starting	Bits of Choice	Produces
noun	0010	**who** went fishing **where**
who went fishing **where**	0	Bob and Ray went fishing **where**
Bob and Ray went fishing **where**	11	Bob and Ray went fishing in Kalamazoo.

There is no reason why the sequence of productions needs to be related with a leftmost first rule. A general parsing algorithm would be able to discover the three different choices made in the creation of this sentence but the GNF-limited parser could not. They could be arranged in any predefined order used by both ends of the communications link. So the sentence "Bob and Ray went fishing in Kalamazoo" could be said to be generated by any of the six combinations 0010011, 0010110, 0001011, 0110010, 1100010, or 1100100.

This last section on expansions and contractions has ignored the feature that allows a user to weight the choices according to some predetermined agenda. These weights can be carried accurately throughout the expansion and contraction process. If there is an expansion, the terms are multipled through. If there is a contraction, they are gathered. Here's an example of expansion. The weightings are shown as variables in parentheses.

Before:

noun → Thelma and Louise **what** (a_1) ‖ Harry and Louise **what** (a_2)
what → went shooting. (a_3) ‖ went to the hospital. (a_4)

Before expansion:

noun	→	Thelma and Louise **what** (a_1) ‖
		Harry and Louise went shooting. $(\frac{a_2 a_3}{a_3+a_4})$ ‖
		Harry and Louise went to the hospital. $(\frac{a_2 a_4}{a_3+a_4})$
what	→	went shooting. (a_3) ‖ went to the hospital. (a_4)

Here's the same example reworked for contraction. Before:

noun	→	Thelma and Louise **what** (a_1) ‖
		Harry and Louise went shooting. (a_2) ‖
		Harry and Louise went to the hospital. (a_3)
what	→	went shooting. (a_4) ‖
		went to the hospital. (a_5)

After contraction:

noun	→	Thelma and Louise **what** (a_1) ‖
		Harry and Louise **what2** $(a_2 + a_3)$
what	→	went shooting. (a_4) ‖
		went to the hospital. (a_5)
what2	→	went shooting. $(\frac{a_2}{a_2+a_3})$ ‖
		went to the hospital. $(\frac{a_3}{a_2+a_3})$

These rules can be expanded arbitrarily to handle all expansions and contractions. Weightings like this can significantly affect the way that bits are converted into phrases using Huffman trees. The trees work perfectly only if the weights are structured correctly, so it is highly likely that most trees will produce imperfect approximations of the weights. As the expansions and contractions change the tree structure, the weights will significantly alter the patterns produced.

7.3.4 Assessing the Theoretical Security of Mimicry

Determining the strengthof mimic functions based on context-free grammars is not an easy task. There are two basic approaches and both of them can leave you with doubt. The first is to analyze the structure of the system on a theoretical level and use this to compare it to other systems. This can indicate that it can often be quite hard to break through the mimicry in these systems, but it can't prove to

you that there are no holes out there. You just know that in the past, others have tried and failed to break similar systems. This is good news, but it is not conclusive. There might be new holes that are easy to exploit in these grammar-based mimic functions, but that are hard to use in other systems.

The principal difficulty of your case lay in the fact of there being too much evidence. What was vital was overlaid and hidden by what was irrelevant.
—Arthur Conan Doyle in The Naval Treaty

These holes are fairly common in theoretical approaches. For instance, there are very few proofs that show how hard it is to solve some mathematical problems. Sorting numbers is one of the few examples. It has been shown that if you have a list of n numbers, then it takes time proportional to $cn \log n$ where c is some machine-based constant [AHU83]. This is a nice result, but it doesn't make a good theoretical basis for a cryptographically secure system. There are other algorithms for sorting that can succeed in a time proportional to kn, where k is a different machine-based constant. These algorithms only work if you can place absolute bounds on the size of the numbers before beginning (64 bits is usually enough).

The other approach is to create different attacks against the system and see if it is strong enough to withstand them. This can certainly show the strength of the system, but it too can never be conclusive. There is no way to be sure that you've tried all possible attacks. You can be thorough, but you can't be complete.

Still, probing the limits of grammar-based mimic functions is an important task. The best theoretical bounds that exist are based on work exploring the limits of computers that try to learn. In this area, many researchers have based their work on Les Valiant's PAC model from probablistic learning [Val84]. In it, a computer is given some examples from a particular class and it must try to learn as much as possible about the class so it can decide whether a new example is part of it. The computer's success is measured probabilistically and it succeeds if it starts getting more right than wrong.

There are many different forms of PAC algorithms. In some, the computer is given examples that are just in the class. In others, the computer gets examples from within and without the class. Sometimes, the computer can even create examples and ask whether that example is in or out of the class. This type of algorithm has the potential to be the most powerful, so it helps if the theoretical bounds can defend against it.

Michael Kearns and Les Valient show that "learning" boolean formulas, finite automata, or constant-depth threshold circuits is at least as difficult as inverting RSA encryption or factoring Blum integers (x, such that $x = pq$, where p and q are prime, and $p, q = 3 \bmod 4$). The proof shows this by casting the factoring process into each of these different models of computation. [KV89, Kea89]

Dana Angluin and Michael Kharitonov [AK91] extended the work of Kearns and Valient as well as the work of Moni Naor and M. Yung [NY89, NY90]. Their work shows that there are no known algorithms run in polynomial time that predict membership in a class defined by finite unions or interesections of finite automata or context-free grammars.

These bounds deal with learning to predict whether a sentence is in a class defined by a grammar, not to discover its parse tree. But the results can apply to the grammar-based system here if there is someway that a parse tree-discovering algorithm can be used to predict membership.

Imagine that such an algorithm exists. Here is how to apply it to predicting membership in some language defined by grammar G_1, known as $L(G_1)$. The start symbol for G_1 is S_1. Now suppose there is another grammar G_2 with start symbol S_2. Create a new grammar, G, that is the union of G_1 and G_2 by creating a new start symbol, S, and the production $S \rightarrow S_1 \| S_2$. Take a set of strings $a_i \in L(G)$. They are either in $L(G_1)$ or $L(G_2)$. Apply the algorithm that can learn to predict parse trees and feed it this set of strings. If such an algorithm can learn to predict the parse tree grammar, then it can predict whether a string is in $L(G_1)$. If such an algorithm runs in polynomial time, then it can be used to break RSA, factor Blum integers and solve other problems. Therefore, there is no known algorithm to predict even the first branch of a parse tree.

This result applies to the hardest grammars that might exist but it does not offer any clues on how to actually produce such a grammar. An algorithm that could construct such a grammar and guarantee that it was hard to discover would be quite a find. There are some minor observations, however, that can be satisfying.

You can easily imagine a grammar that would be easy to break. If each word or substring was visible in one production, then it would be relatively easy to isolate the string of productions that produced a long section of text. The boundaries of the productions are simple to establish by accumulating enough sample text so that each production is used twice. The two occurrences can be compared to reveal the different parts of the production.

This leads to the observation that each word should appear in multiple productions. The section beginning on page 119 describes how contractions and expansions can be applied automatically to change grammars so they fit this requirement.

How much contraction and expansion is enough? [Way95a] gives one set of equations that can be used to measure the "randomness" or "entropy" of a grammar. The equations are modelled on Shannon's

measure of entropy of a bit stream. If one word is quite likely to follow another, then there is not much information bound in it. If many words are likely, then there is plenty of information bound in this choice.

The equations measure the entropy of the entire language generated by a grammar. If the amount is large, then the information capacity of the grammar is also large and a great deal of information should be able to be transmitted before significant repetition occurs. This practical approach can give a good estimate of the strength of a grammar.

Both of these approaches show that it can be quite difficult to discover the grammar that generated a text. They do not guarantee security, but they show that the security may be difficult to achieve in all cases. It can be even more difficult if the grammar is modified in the process through expansions and contractions. These can be chosen by both sides of a channel in a synchronized way by agreeing on a cryptographically secure pseudo-random number generator.

7.3.5 Efficient Mimicry-Based Codes

The one problem with the mimicry system described in this chapter is that it is inefficient. Even very complicated grammars will easily double, triple, or quadruple the size of a file by converting it into text. Less complicated grammars could easily produce output that is ten times larger than the input. This may be the price that must be paid to achieve something that looks nice, but there may be other uses for the algorithm if it is really secure.

Efficient encryption algorithms using the techniques of this chapter are certainly possible. The results look like ordinary binary data, not spoken text, but they do not increase the size of a file. The key is just to build a large grammar. Here's an example:

Terminals Let there be 256 terminal characters, that is, the values of a byte between 0 and 255. Call these $\{t_0 \ldots t_{255}\}$.

Variables Let there be n variables, $\{v_0 \ldots v_n\}$. Each variable has 256 productions.

Productions Each variable has 256 productions of the form $v_i \rightarrow t_j v_{a_1} \ldots v_{a_k}$. That is, each variable will be converted into a single terminal and k variables. Some productions will have no variables and some will have many. Each terminal will appear on the right side of only one production for a particular variable. This ensures that parsing is easy.

This grammar will not increase the size of the file when it is encoded. Each variable has 256 different productions available to it, so 8 bits are consumed in the process of making the choice. The result is one new terminal added to the stream which takes 8 bits to store.

There are potential problems with this system. The biggest one is ensuring that the average string of terminals in the language is finite. If there are too many variables on the right-hand side of the productions, then the generating process could never end. The stack of pending variables would continue to grow with each production. The solution is to make sure that the average number of variables on the right-hand side of the production is less than one. The relationship between the average number of variables and the average length of the phrases in the language defined by the grammar is direct. A smaller number of average variables means shorter phrases. As the average number of variables approaches one, the average length tends toward infinity.[3]

The average length of a phrase in the language is not as important in this particular example. The bits can be recovered easily here because the grammar is in Greibach Normal Form and there is no need to place parsing decisions on hold. Each terminal only appears on the right hand side of one production per variable, so the final file does not need to be a complete phrase produced by the grammar. It could just be a partial one. There is no reason why the grammars need to be as easy to parse, but more complicated grammars need to have the entire phrase produced from the starting symbol.

7.4 Summary

This chapter has described simple ways to produce very realistic texts by using a system of rules defined by a human. Complicated grammars can hide large volumes of data in seemingly human babble. This babble could be posted to some Internet newsgroup, and it will be hard to tell the difference between this and the random flames and cascading comments that float through the linguistic ether.

There are still other levels of abstraction that are possible. MUDs (Multiple-User Dungeons) allow users to meet up in a text-based world defined and built up by textual architects. It is possible to meet people in the MUD rooms and hold conversations in the same way that you might ordinarily talk. Some MUDs now sport computer programs that pretend to be human in the spirit of the great Eliza [Wei76]. These programs use complicated grammars to guide the

[3] This might be modeled with queuing theory.

response of the computer and the computer can turn data into the random choices that guide the grammars.

The sun's a thief, and with his great attraction Robs the vast sea: the moon's an arrant thief, And her pale fire she snatches from the sun: The sea's a thief, whose liquid surge resolves The moon into salt tears... —William Shakespeare in Timons of Athens

Here's an extreme example. You want to set up a conversation with a friend across the country. Ordinarily, you might use the basic `talk` protocol to set up a text-based link. Or you might use one of the Internet phone programs to exchange sound. In either case, the bits you're exchanging can be monitored.

What if your talk program didn't contact the other person directly but logget into a MUD somewhere on the Net as a persona? The other person's talk program could do the same thing and head for the same room. For the sake of atmosphere, let's make it a smoke-filled room with leather chairs so overstuffed that our textual personae get lost in them. There are overstuffed mastodons on the wall to complement the chairs.

Instead of handing your word bits over to the other person's persona directly, your talk program encodes them into something innocuous like a discussion about last night's baseball game. It might be smart enough to access the online database to get an actual scorecard to ensure that the discussion was accurate. When the other person responded, his talk program would encode the data with a similar grammar. The real conversation might be about very private matters, but it might come out sounding like baseball to anyone who happened to be eavesdropping on the wires.

Both sides of the conversation can use the same grammar. This convention would make it possible for both sides to hold a coherent conversation. After one persona commented about the hitting of Joe Swatsem, the other could say something about Swatsem because the same grammar would control what came afterward.

The entire system is just an automated version of the old gangster-movie conceit about talking in code. One gangster says, "Hey, has the shipment of tomatoes arrived yet?" The other responds, "Yeah. It'll cost you 10,000 bananas." The potentials are amazing.

The Disguise Grammar-based mimicry can be quite realistic. The only limitation is the amount of time that someone puts into creating the grammar.

How Secure Is It? At its best, the grammar-based system here can be as hard to break as RSA. This assessment, though, doesn't mean that you can achieve this security with the same ease as you can with RSA. There is no strong model for what is a good key. Nor has there been any extensive work done on breaking the system.

How to Use It? The code for the mimic system is available from the author.. Or you can just go to `spammimic.com` if you want your message to look like spam.

Further Work There are a number of avenues to pursue in this arena. A theory that gives a stronger estimates of the brute force necessary to recognize a language would be nice. It would be good to have a strong estimate of just how many strings from a language must be uncovered before someone can begin to make sense of it. If someone could program the entropy estimates from [Way95a] or come up with better ones, then we could experiment with them to see how well they assess the difficulty of attack.

It would also be nice to have an automatic way of scanning texts and creating a grammar that could be used by the system here. There are many basic constructs from language that are used again and again. If something could be distilled from the raw feed on the Net, then it could be pushed directly into a program that could send out messages. This could lead to truly automated broadcast systems. One part would scan newsgroups or the net for source text that could lead to grammars. The other would broadcast messages using them. I imagine that it could lead to some truly bizarre AI experiences. You could set up two machines that babble to each other, mimicking the Net but really exchanging valuable information.

Further Reading

- A number of papers from Purdue's large group extend these grammar techniques with a comprehensive database of synonyms for words. Their work suggests that the most ambiguous words where possible because more specific words may not be easily swapped into a sentence. [ARC+01, ARH+02, TTA06, AMRN00, ARH+03, TTA06]
- Cuneyt M. Taskiran, Umut Topkara, Mercan Topkara and Edward J. Delp developed a tool for finding text generated by methods like the grammar-based tool described in this chapter. They compare the statistics gathered from analyzing when words are found adjacent to each other with their general occurrence in the general language and use it to train statistical classifiers. [TTTD06]
- The BLEU score, a rough statistical comparison of the phrase length, measures the quality of automated transla-

tion. Mercan Topkara, Guiseppe Riccardi, Dilek Hakkani-Tur and Mikhail Atallah use it to evaluate the effects of their textual transformation on the readability.[TRHTA06] [TRHTA06]

- Compris sells TextHide and TextSign, a software programs that hide information by changing the structure of sentences. See `compris.com/subitext/`.

Chapter 8

Turing and Reverse

8.1 Doggie's Little Get Along

One weekend I messed with the guts of my jukebox.
I wanted to zip it up to tweet like a bird
When the wires got crossed and the records spun backward
And this is the happy voice that I heard:

Whoopee Tie Yi Yay,
    The world's getting better and your love's getting strong
Whoopee Tie Yi Yay,
    Your lame dog will walk by the end of this song.

The music was eerie, sublime and surreal,
But there was no walrus or devil.
The notes rang wonderfully crystalline clear
Telling us that it was all on the level:

Whoopee Tie Yi Yay
    This weekend your sixty-foot yacht will be floated.
Whoopee Tie Yi Yay
    The boss just called to tell you, "You're promoted."

So after a moment I began to start thinking
What if I rewired the touch tone?
After a second of cutting and splicing, it suddenly rang.
This was voice that came from the phone:

Whoopee Tie Yi Yay
    This is the Publisher's Clearing House to Tell You've Won
Whoopee Ti Yi Yay
    A new car, an acre of dollars and a house in the sun.

A few minutes later my lost sweetheart called:
The guy she ran off with wasn't worth Jack.
He wore a toupee and the truck was his mother's.
Now she could only beg for me back.

Whoopee Tie Yi Yay
Why spend your grief on a future that's wrecked
Whoopee Tie Yi Yay
Why look backward when hindsight is always so perfect.

8.2 Running Backward

The song that introduces this chapter is all about what happens to a man when he finds a way to play the country music on his jukebox backward. His dog walks, his girlfriend returns, and the money rolls in. The goal of this chapter is to build a machine that hides data as it runs forward. Running it in reverse allows you to recover it. The main advantage of using such a machine is that some theoretical proofs show that this machine can't be attacked by a computer. These theoretical estimates of the strength of the system are not necessarily reliable for practical purposes, but they illustrate a very interesting potential.

Chapter 7 described how to use grammars to hide data in realistic-sounding text. The system derived its strength from the structure of the grammars and their ability to produce many different sentences from a simple collection of inputs. The weaknesses of the system were also fairly apparent. Grammars that were context-free could not really keep track of scores of ballgames or other more complicated topics. They just produced sentences with no care for the context. A bit of cleverness could go a long way, but anyone who has tried to create complicated grammars begins to understand the limitations of the model.

This chapter will concentrate on a more robust and complete model known as the *Turing machine*, a concept was named after Alan Turing, who created it in the 1930s as a vehicle for exploring the limits of computation. Although the model doesn't offer a good way to whip up some good mimicry, it does offer a deeper theoretical look at just how hard it may be to break the system.

A good way to understand the limits of context-free grammars is to examine the type of machine that is necessary to recognize them. When testing this, I built a parser for recovering the data from the mimicry using a model of a *push-down automata*. The automata refers to a mechanism that is a nest of if-then and goto statements. The push-down refers to the type of memory available to it—in this case a push-down stack that can store information by pushing it onto a stack of data and retrieve it by pulling it off. Many people compare this to the dishracks that are found in cafeterias. Dishes are stored in a spring-loaded stack. The major limitation of this type of memory is

the order. Bits of information can only be recalled from the stack in the reverse order in which they were put onto the stack. There is no way to dig deeper.

It is possible to offer you solid proof that push-down automata are the ideal computational model for describing the behavior of context-free grammars, but that solution is a bit dry. A better approach is to illustrate it with a grammar:

You can find a good proof in [AHU83].

start	→	Thelma and Louise **what when** ‖ Harry and Louise **what when**
what	→	went shooting **with where** ‖ bought insurance **with where**
with	→	with Bob and Ray ‖ with Laverne and Shirley
when	→	on Monday. ‖ on Tuesday. ‖ on Wednesday. ‖ on Thursday.
where	→	in Kansas ‖ in Canada

A typical sentence produced by this grammar might be "Thelma and Louise went shooting with Bob and Ray in Kansas on Monday." This was produced by making the first choice of production from each variable and thus hiding the six bits 000000. But when the first choice was made and Thelma and Louise became the subjects of the sentence, the question about the date needed to be stored away until it was needed later. You can either think of the sentence as developing the leftmost variable first or you can think of it as choosing the topmost variable from the stack. Here's a table showing how a sentence was produced. It illustrates both ways of thinking about it.

Stack	Pending Sentence	Pending with Variables
start		**noun**
what when	Thelma and Louise	Thelma and Louise **what when**
with where when	Thelma and Louise went shooting	Thelma and Louise went shooting **with where**
where when	Thelma and Louise went shooting with Bob and Ray	Thelma and Louise went shooting with Bob and Ray **where when**
when	Thelma and Louise went shooting with Bob and Ray in Kansas	Thelma and Louise went shooting with Bob and Ray in Kansas **when**
empty	Thelma and Louise went shooting with Bob and Ray in Kansas on Monday.	Thelma and Louise went shooting with Bob and Ray in Kansas on Monday.

Both metaphors turn out to be quite close to each other. The context-free grammars and the stack-based machines for interpreting them are equivalent. This also illustrates why it is possible to imitate certain details about a baseball game like the number of outs or the number of strikes, while it is much harder, if not impossible, to give a good imitation of the score. There is no way to rearrange the information on the stack or to recognize it out of turn.

The Turing machine is about as general a model of a computer as can be constructed. Unlike the push-down automata, a Turing machine can access any part of its memory at any time. In most models, this is described as a "tape" that is read by a head that can scan from left to right. You can also think of the "tape" as regular computer memory that has the address 0 for the first byte, the address 1 for the second byte, and so on.

The main advantage of using a Turing machine is that you access any part of the memory at any time. So you might store the score of the baseball game at the bytes of memory with addresses 10140 and 10142. Whenever you needed this, you copy the score to the output. This method does not offer any particularly great programming models that would make it easier for people to construct a working Turing mimicry generator. Alas.

The real reason for exploring Turing machines is that there are a wide variety of theoretical results that suggest the limits on how they can be analyzed. Alan Turing originally developed the models to explore the limits of what computers can and can't do [Tur36a, Tur36b]. His greatest results showed how little computers could do when they were turned against themselves. There is very little that computers and the programs they run can tell us definitively about another computer program.

These results are quite similar to the work of Kurt Gödel, who originally did very similar work on logical systems. His famous theorem showed that all logical systems were either incomplete or inconsistent. The result had little serious effect on mathematics itself because people were quite content to work with incomplete systems of logic— they did the job. But the results eroded the modernist belief that technology could make the world perfect.

Turing found that the same results that applied to Gödel's logical systems could apply to computers and the programs that ran on them. He showed, for instance, that no computer program could definitively answer whether another computer program would ever finish. It might be able to find the correct answer for some subset of computer programs, but it could never get the right answer for all of them. The program was either incomplete or inconsistent.

Others have extended Turing's results to show that it is practically impossible to ask the machines to say anything definitive about computers at all. Rice's theorem showed that computers can only answer *trivial* questions about other computers [HU79]. Trivial questions were defined as those that were either always true or always false.

Abraham Lincoln was really the first person to discover this fact when he told the world, "You can fool some of the people all of the time and all of the people some of the time. But you can't fool all of the people all of the time." The same holds true if you substitute "computer program" or "turing machine" for "people."

To some extent, these results are only interesting on a theoretical level. After all, a Macintosh computer can examine a computer program written for an IBM PC and determine that it can't execute it. Most of the time, a word processor might look at a document and determine that it is in the wrong format. Most of the time, computers on the Internet can try to establish a connection with other computers on the Internet and determine whether the other computer is speaking the right language. For many practical purposes, computers can do most things we tell them to do.

The operative qualifier here is "most of the time." Everyone knows how imperfect and fragile software can be. The problems caused by the literal machines are legendary. They do what they're told to do, and this is often incomplete or imperfect—just like the theoretical model predicted they would be.

The matter for us is compounded by the fact that this application is not as straightforward as opening up word processing documents. The goal is to hide information so it can't be found. There is no cooperation between the information protector and the attacker trying to puncture the veil of secrecy. A better model is the world of computer viruses. Here, one person is creating a computer program that will make its way through the world and someone else is trying to write an anti-virus program that will stop a virus. The standard virus-scanning programs built today look for tell-tale strings of commands that are part of the virus. If the string is found, then the virus must be there. This type of detection program is easy to write and easy to keep up to date. Every time a new virus is discovered, a new tell-tale string is added to the list.

Can you abort a virus? Can you baptize one? How smart must a virus be to be a virus?

But more adept viruses are afoot. There are many similar strings of commands that will do a virus's job. It could possibly choose any combination of these commands that are structured correctly. What if a virus scrambled itself with each new version? What if a virus carried a context-free grammar of commands that would produce valid viruses? Every time it copied itself into a new computer or program, it would spew out a new version of itself using the grammar as its copy. Detecting viruses like this is a much more difficult proposition.

You couldn't scan for sequences of commands because the sequences are different with each version of the virus. You need to

build a more general model of what a virus is and how it accomplishes its job before you can continue. If you get a complete copy of the context-free grammar that is carried along by a virus, you might create a parser that would parse each file and look for something that came from this grammar. If it was found, then a virus would be identified. This might work sometimes, but what if the virus modified the grammar in the same way that the grammars were expanded and contracted on page 119? The possibilities are endless.

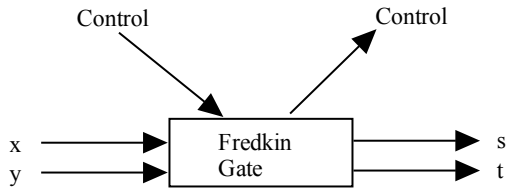
The goal for this chapter is to capture the same theoretical impossibility that gives Turing machines their ability to resist attacks by creating a cipher system that isn't just a cipher. It's a computing machine that runs forward and backward. The data is hidden as it runs forward and revealed as it runs backward. If this machine is as powerful as a Turing machine, then there is at least the theoretical possibility that the information will never be revealed. Another computer that could attack all possible machines by reversing them could never work in all cases.

8.2.1 Reversing Gears

Many computer scientists have been studying reversible computers for some time, but not for the purpose of hiding information. Reversible machines have a thermodynamic loophole that implies that they might become quite useful as CPUs become more and more powerful. Ordinary electronic circuits waste some energy every time they make a decision, but reversible computers don't. This wasted energy leaves a normal chip as heat, which is why the newest and fastest CPUs come with their own heat-conducting fins attached to the top. Some of the fastest machines are cooled by liquid coolants that can suck away even more heat. The build up of waste heat is a serious problem—if it isn't removed, the CPU fails.

The original work on reversible computers was very theoretical and hypothetical. Ed Fredkin offered a type of logic gate that would not expend energy. [Fre82] Normal gates that take the AND of two bits are not reversible. For instance, if $x\ AND\ y$ is 1, then both x and y can be recovered because both must have been 1. But if $x\ AND\ y$ is 0, then nothing concrete is known about either x or y. Either x or y might be a 1. This makes it impossible to run such a normal gate in reverse.

The Fredkin gate, on the other hand, does not discard information so it can be reversed. Figure 8.1 shows such a gate, and the logic table that drives it. There are three lines going in and three lines leaving. One of the incoming lines is a control line. If it is on, then the other two lines are swapped. If it is off, then the other lines are re-



IN			OUT		
Control	x	y	Control	s	t
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	1	1	1

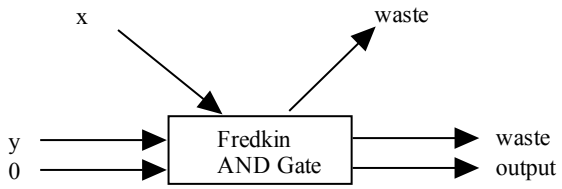
Figure 8.1: An illustration of a Fredkin gate. If the control line is on, then the output lines are switched. Otherwise, they're left alone. (Drawn from Bennett's figure.[BL85])

The Scientific American *article by Charles Bennett and Rolf Landauer makes a good introduction to reversible machines. [BL85]*

versed. This gate can be run in reverse because there is only one possible input for each output.

Figure 8.2 shows an AND gate built out of a Fredkin gate. One of the two input lines from a normal AND gate is used as the control line. Only one of the output lines is needed to give us the answer. The other two bits are wasted. Ordinarily, the information here would be thrown away by sending the bits to ground, where they would heat up the chip. A truly reversible machine would store the bits at this location until the computer was run in reverse. Then the gate would have all of the information ready to compute the inverse. An OR gate would be built in the same way, but it would have one input fixed to be a 1.

There are a number of other mechanical approaches to building a reversible computer. Ed Fredkin and Tommaso Toffoli developed a billiard ball computer that could be made to run in reverse *if* a suitable table could be found [FT82]. It would need to be perfectly smooth so the balls would move in synchrony. The table itself must be frictionless and the bumpers would need to return all of the en-



IN			OUT		
			waste	waste	output
x	y				
0	0	0	0	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	1	0	1	0	1

Figure 8.2: An AND gate built out of a Fredkin gate. The extra waste bits must be stored at the gate so that computation can be reversed later. [BL85]

ergy to the balls so that nothing would be lost and there would be just as much kinetic energy at the beginning of the computation as at the end.

Figure 8.3 shows how two billiard balls can build an AND gate. The presence of a ball is considered to be the *on* state, so if both balls are there, they will bounce off each other and only one ball will continue on its way. If the balls reach the end of the computation, then they can bounce off a final wall and make their way back. It should be easy to see that this gate will work both forward and backward. OR gates are more complicated and include extra walls to steer the balls.

This is an interesting concept, but it is hardly useful. No one can build such a frictionless material. If they could, it might be years before we got to actually trying to use it to compute. There would be too many other interesting things to do, like watching people play hockey on it. More practical implementations, however, use cellular automata that come before and after it. Toffoli described reversible cellular automata in his Ph.D. thesis [Tof77a] and in other subsequent articles [Tof77b, TM87]. N. Margolus offers one solution that implements the billiard ball models. [Mar84]

David Hillman has written about reversible one-dimensional cellular automata [Hil91b, Hil91a].

The key result about reversible computers comes from Charles Bennett who showed that any computation can be done with a reversible Turing machines. He created a few basic examples of reversible Turing machines with well defined commands for moving the read/write head of the tape and changing the state of the machine.

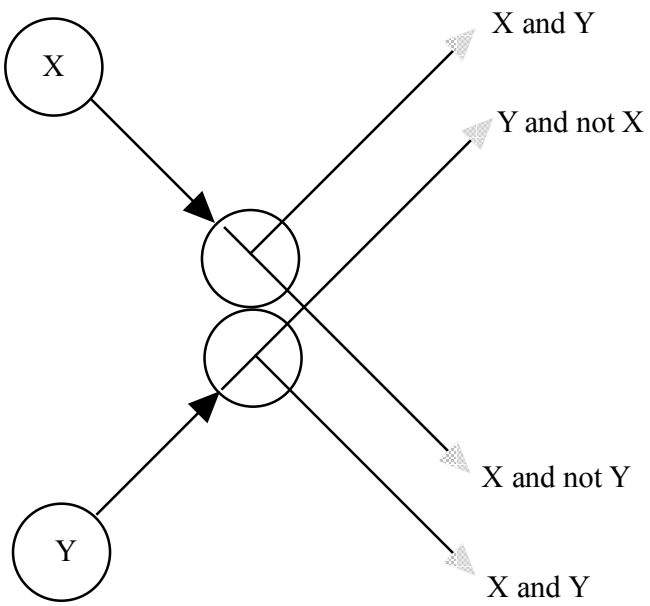


Figure 8.3: The three possible outcomes of a billiard ball AND gate. The presence of a ball indicates an *on* signal. If only one ball is present, then no bounce occurs and it continues on its way. If both are present, then they bounce off each other. If none are present, then nothing happens. (Adapted from Bennett)

The transition rules for these machines often look quite similar to the Fredkin gate. There is just as much information coming out of each step as going into it. This is balanced correctly so the position that leads to another position can always be inferred and the machine can be reversed.

Reversible computation is also great *for debugging programs.*

This result shows that anything that can be done with a computer can be done with a reversible computer. All that is necessary is finding a way to save the information from each step so it can be effectively run in reverse. But what does this mean if you want to hide information? It means that any computation can be used to hide information in the final outcome. How much can be stored? It all depends on the calculation. Ordinarily, any random number generator that is used to add realism or to scramble the outcome of a game can be replaced by a collection of data to be hidden. This data can be recovered as the machine runs in reverse.

How would such a system work? One obvious solution is to create a universal, reversible Turing machine format. A standard program

running on everyone's computer would be able to read in a Turing machine and run it forward or backward. If you wanted to send a message, you would pack in the data and run the machine until it stopped. The result would be the output, perhaps some computer-generated poetry, and a pile of waste data that must be kept around in order to run the machine in reverse.

At the other end, the recipient would load this information into the same universal, reversible Turing machine and run it backward to recover the data. The one problem with this scenario is that any attacker could also have the same universal, reversible Turing machine. They could intercept the message and reverse it. For the technique to be successful, some data must be kept secret from the attacker. This could travel separately. In the grammar machine from Chapter 7, the grammar acts as the key. It must be distributed separately.

One solution is to keep the structure of the Turing machine secret and let it act as a key. Only the output and the extra, "waste" bits of information must be transmitted to the recipient. Anyone can intercept the message, but they cannot read it without a copy of the program that created it.

How difficult can this be? Obviously, there will be some programs that are pretty easy to crack. For instance, a program that merely copies the data to be hidden and spits it out would be easy to deduce. The output would maintain all of the structure of the original document. More and more complicated programs would get more and more complicated to deduce. Eventually, something must be too hard to crack. The tough question is whether there is some threshold that can be established where it is positively known that programs that are beyond it are completely safe.

Such a threshold can never be well-defined. That is, there can be no neat machine that will examine any program and say, "This can't be broken." There might be machines that could point out flaws in programs and show how they could be broken, but they would not be guaranteed to find all flaws.

This uncertainty is a pain, but it affects the enemy in the same way. The enemy can not come up with an arbitrary machine that will be able to examine every message you send and discover the program that was used to hide the data. It may be able to find some solutions, but there will be no brute-force attack that will work in all cases.

This is a nice beginning for security, but it is not absolute. The one-time pad offers a similar security blanket. No brute-force attack that will break the system, as long as the key bits are completely

See Section 2.2 on page 22 for details.

random. What is completely random? In practice, it means that the attacker can't build a machine that will predict the pattern of the bits. This is substantially easier to achieve for one-time pads than it is for reversible Turing machines. There are numerous sources of completely random information and noise that can be used as the basis for a one-time pad. The random snow from a noisy section of the radio spectrum can make the good beginning for such a pad.

The rest of this chapter will concentrate on actually constructing a reversible Turing machine that could be used to create hidden messages in text. It will be based, in part, on the grammar approach from Chapter 7 because text is a good end product for the process. There is no reason why the work couldn't be adapted to produce other mimicry.

8.3 Building a Reversible Machine

Some reversible machines are inherently error limiting and self-resynchronizing in both directions, as shown by Peter Neumann[Neu64] for David Huffman's information-lossless sequential machines. [Huf59]

If every Turing machine can be reconstructed in a reversible manner, then every possible machine is a candidate for being turned into a vehicle for hidden information. Obviously, though, some machines are more interesting than others. For instance, loan companies use computer programs to evaluate the credit of applicants and these programs respond with either "qualified" or "unqualified". That's just one bit of information and it seems unlikely that anyone will be able to hide much of anything in that bit. On the other hand, programs that produce complex worlds for games like Doom spit out billions of bits. There is ample room in the noise. Imagine if some secret information was encoded in the dance of an attacking droid. You might get your signal by joining an internet group game of Doom. The information would come across the wire disguised as instructions for where to draw the attacker on the screen. Your version of Doom could extract this.

This chapter will show how to build two different reversible machines. The first, a simple reversible Turing machine, is provided as a warm-up. It is based on the work of Charles Bennett and it shows how to take the standard features of a Turing machine and tweak them so that there is only one possible state that could lead to another. This makes it possible to rewind the behavior.

The second machine is an extension of the grammar-based mimicry from Chapter 7. That system used only context-free grammars. This one lets you simulate any arbitrary computation to add realism to the text that it produces. The data hidden by the system won't be recovered by parsing. It will come by running the machine in reverse.

This means that a practical way needs to be constructed to ship the extra variables and "waste" bits along.

8.3.1 Reversible Turing Machines

An ordinary Turing machine consists of a set of states, S, a set of symbols that can appear on the tape, Σ, and a set of transition rules, δ, that tell the machine what happens when. For instance, δ might specify that if the machine is in state s_2, and the symbol σ_4 is on the tape underneath the read/write head, then the read/write head should write the symbol σ_5 on the tape, move the head to the right one notch and change to state s_{42}. This is how you program a Turing machine. The abstraction is fairly crude, but it makes it simpler to keep track of all the possibilities.

If a certain puritanical tradition, for instance, is profoundly suspicious of the novel, this is because the novel is felt to celebrate and encourage misconduct, rather than censure and repress it.
—D.A. Miller in The Novel and The Police

Converting such a machine to run backward is pretty straightforward. The main problem is looking for combinations of states and tape symbols that lead to the same states. That is when it is impossible to put the machine in reverse because there are two different preceding situations that could have led to the present one. The easiest solution is to keep splitting up the states until there is no confusion.

For each state, $s_i \in S$, construct a list of triples of states, tape symbols, and direction (s_j, σ_k, L) that could lead to the state s_i. That is, if the machine is in state s_j with the read/write head over the symbol σ_l, then it will write σ_k and move to the left one step. In other words, if the machine is running backward and it finds itself in state, s_i , with symbol σ_k to the right of it, then it can move to the right, change to state s_j, and overwrite σ_k with σ_l and not violate the program. That is, this is a correct move.

There will be a conflict if there are triples of the form (s_*, σ_*, L) and (s_*, σ_*, R) in the same set. (Let s_* stand for any element s_i from S.) This is because it is possible that the machine will end up someplace with one of the symbols to the left and one to the right. You might be able to make meta-arguments that such a combination could never exist because of the structure of the program, but these are often hard to prove.

If such a conflict occurs, then create a new state and split apart the actions. All of the triples that moved left into the old state, s_i, can stay pointing to state, s_i. The triples that moved *right*, however, will be moved to point to the new state s_j. The transition rules out of s_j will be a duplicate of s_i.

To a large extent, splitting these states is the same as finding a place to keep a "waste" bit around. The Fredkin AND gate generates some waste bits that must be stored. Splitting the state creates a

waste bit.

The same splitting process must be done if there are two triples of the form: (s_a, σ_k, L) and (s_b, σ_k, L). Both of these states, s_a and s_b, lead to the same symbol existing to the right of the current position of the read/write head. Choosing is impossible. Again, a new state must be added and the transition rules duplicated and split.

It should be obvious that a Turing machine will grow substantially as it is made reversible. This growth can even be exponential in many cases. There is no reason why anyone would want to program this way. The complications are just too great. But this example is a good beginning.

8.3.2 Reversible Grammar Generators

The goal of this book is to produce something that seems innocuous but hides a great deal of information from plain sight. Chapter 7 did a good job of this with a context-free grammar, but there are numerous limitations to that approach. This part of the book will build a reversible, Turing-equivalent machine that will be able to do all basic computations, but still be reversible. It will get much of its performance by imitating the Fredkin gate, which merely swaps information instead of destroying it.

Numerous problems that need to be confronted in the design of this machine. Here are some of them:

Extra State At the end of the computation, there will be plenty of extra "waste" bits hanging around. These need to be conveyed to the recipients so they can run their machines in reverse.

There are two possible solutions. The first is to send the extra state through a different channel. It might be hidden in the least significant bits of a photo or sent through some other covert channel. The second is to use a crippled version of the machine to encode the bit as text without modifying any of the state. That is, reduce the capability of the machine until it acts like the context-free grammar machine from Chapter 7.

Ease of Programmability Anyone using the machine will need to come up with a collection of grammars to simulate some form of text. Constructing these can be complicated, and it would be ideal if the language could be nimble enough to handle many constructions.

The solution is to imitate the grammar structure from Chapter 7. There will be variables and productions, but you can change the productions en route using reversible code.

Minimizing Extra State Any extra bits must be transported through a separate channel at the end and so they should be kept to a minimum. For that reason, all strings should be predefined as constants. They can't be changed. If they could be changed, then the final state of all strings would need to be shipped to the recipient. This would be too much baggage.

Arithmetic Arithmetic is generally not reversible. $3 + 2$ is not reversible, but it can be reversed if one half of the equation is kept around. So adding the contents of register A_1 and register A_2 and placing the result in register A_1 *is* reversible. The contents of A_2 can be subtracted from A_1 to recover the original value of A_1.

For the most part, addition, subtraction, multiplication, and division are reversible if they're expressed in this format. The only problem is multiplication by zero. This must be forbidden.

Structure of Memory What form will the memory take? Ordinary computers allow programmers to grab and shift blocks of memory at a time. This is not feasible because it would require too many extra waste bits would need to be stored. Block moves of data are not reversible. Swaps of information are.

For that reason, there is an array of registers. Each one holds one number that can be rounded off in some cases. The final state of the registers will be shipped as extra state to the recipient so any programmer should aim to use them sparingly. Unfortunately, the rules of reversibility can make this difficult.

Conditional Statements Most conditional statements that choose between branches of a program can be reversed, but sometimes they can't be. Consider the case that says, "If x is less than 100, then add 1 to x. Otherwise, add 1 to p." Which path do you take if you're running in reverse and x is 100? Do you subtract 1 from p or not? Either case is valid.

One solution is to execute each branch storing results in temporary variables. When the conditional statement is encountered, the proper choices are swapped into place.

Another solution is to forbid the program to change the contents of the variables that were used to choose a branch. This rules out many standard programming idioms. Here's one way to work around the problem:

```
if x<100 then {
swap x,k;
k=k+1}
else {
p=p+1;
swap x,k;}
swap x,k;
```

Loops Loops may be very handy devices for a programmer, but they can often be an ambiguous obstacle when a program must run in reverse. One easy example is the while loop often written to find the last element in a string in C. That is, a counter moves down the string until the termination character, a zero, is found. It may be easy to move backward up the string, but it is impossible to know where to stop.

The problems with a loop can be eliminated if the structure is better defined. It is not enough to give a test condition for the end of the loop. You must specify the dependent variable of the loop, its initial setting, and the test condition. When the loop is reversed, the machine will run through the statements in the loop until the dependent variable reaches its initial setting.

This structure is often not strong enough. Consider this loop:

```
i=1;
j=i;
while (i<2) do {
j=j+.01;
i=floor(j);}
```

The `floor(x)` function finds the largest integer less than or equal to `x`. This function will execute 100 times before it stops. If it is executed in reverse, then it will only go through the loop twice before `i` is set to its initial value, one. It is clear that `i` is the defining variable for the loop, but it is also clear that `j` plays a big part.

There are two ways to resolve this problem. The first is to warn programmers and hope that they will notice the mistake before they use the code to send an important message. This leaves some flexibility in their hands.

Another solution is to further constrain the nature of loops some more. There is no reason why they can't be restricted to `for` loops that specify a counter that is incremented at each iteration and to `map` functions that apply a particular function to

every element in a list. Both are quite useful and easy to reverse without conflicts.

Recursion Recursion is a problem here. If procedures call themselves, then they are building a de facto loop and it may be difficult to identify a loop's starting position. For instance, here is an example of a loop with an open beginning:

```
procedure Bob;
x=x+1;
if x<100 then Bob;
end;
```

This is just a while loop and it is impossible to back into it and know the initial value of `x` when it began.

One solution is to ban recursion altogether. The standard loop constructs will serve most purposes. This is, alas, theoretically problematic. Much of the theoretical intractability of programs comes from their ability to start recursing. While this might make implementing reversible programs easier, it could severely curtail their theoretical security.

Another solution is to save copies of all affected variables before they enter procedures. So, before the procedure Bob begins, the reversible machine will save a copy of `x`. This version won't be destroyed, but it will become part of the waste bits that must be conveyed along with the output.

Ralph Merkle also notes that most assembly code is reversible and predicts that in the future smart compilers will rearrange instructions to ensure reversibility. This will allow the chips to run cooler once they're designed to save the energy from reversible computations [Mer93].

In the end, the code for this system is quite close to the assembly code used for regular machines. The only difference is that there is no complete overwriting of information. That would make the system irreversible. Perhaps future machines will actually change the programming systems to enhance reversibility. That may come if reversible computers prove to be the best way to reduce power consumption to an acceptable level.

8.3.3 The Reversible Grammar Machine

Although the structure will be very similar to machine code, I've chosen to create this implementation of the Reversible Grammar Machine (RGM) in LISP, one of the best experimental tools for creating new languages and playing around with their limits. It includes many of the basic features for creating and modifying lists plus there are many built-in pattern-matching functions. All of this makes it

relatively easy to create a reversible machine, albeit one that doesn't come with many of the features of modern compilers.

Here are the major parts of the system:

Constant List The major phrases that will be issued by the program as part of its grammar will be stored in this list, `constant-list`. It is officially a list of pairs. The first element of each pair is a tag-like `salutation` that is used as a shorthand for the phrase. The second is a string containing the constant data. This constant list is part of the initial program that must be distributed to both sides of the conversation. The constants do not change; so there is no need to transmit them along with the waste state produced by running a program forward. This saves transmission costs. The main purpose of the constant list is to keep all of the phrases that will be output along the way. These are often long, and there is little reason for them to change substantially. Defining them as constants saves space. The constant list can also include any data like the variable list. The data just won't change.

Variables The data is stored in variables that must be predefined to hold initial values. These initial values are the only way that information can actually be assigned to a variable. The rest of the code must change values through the swap command. The variables are stored in the list `var-list`, which is as usual a list of pairs. The first element is the variable tag name. The second is the data stored in the variable.

There are five types of data available here: lists, strings, integers, floating-point numbers, and tags. Lists are made up of any of the five elements. There is no strict type checking, but some commands may fail if the wrong data is fed to them. Adding two strings, for instance, is undefined.

Some care should be taken with the choice of variables. Their contents will need to be sent along with the output so the recipient can reverse the code. The more variables there are, the larger this section of "waste" code may be.

Procedure List At each step, some code must be executed. These are procedures. For the sake of simplicity, they are just lists of commands that are identified by tags and stored in the list `proc-list`. This is a list of pairs. The first element is the tag identifying the procedure, and the second is the list of commands. There are no parameters in the current implementation, but they can be added.

Commands These are the basic elements for manipulating the data. They must be individually reversible. The set of commands includes the basic arithmetic, the swap command, the if statement and the for loop tool. These commands take the form of classic LISP function calls. The prefix notation that places the command at the front is not as annoying in this case because arithmetic is reversible, so addition looks like this: `(add first second)`. That command adds `first` and `second` and places the result in `first`.

There are three other arithmetic commands: `sub`, `mul` and `div`, which stand for subtraction, multiplication and division, respectively. The only restriction is that you can't multiply a number by zero because it is not reversible. This is reported by an error message.

Output Commands There is one special command, `chz`, that uses the bits that are being hidden to pick an output from a list. When this command is run in reverse by the recipient, the hidden bits are recovered from the choice. The format is simple: `(chz ( tag tag...tag))`. The function builds up a Huffman tree like the algorithm in Chapter 7 and uses the bits to make a choice. The current version does not include the capability to add weights to the choices, but this feature can be added in the future.

The tags can point to either a variable or a constant. In most cases, they'll point to strings that are stored as constants. That's the most efficient case. In some cases, the tags will contain other tags. In this case, the `choose` function evaluates that tag and continues down the chain until it finds a string to output.

For practical reasons, a programmer should be aware of the problems of reversibility. If two different tags point to the same string, then there is no way for the hidden bits to be recovered correctly. This is something that can't be checked in advance. The program can check this on the fly, but the current implementation doesn't do it.

Code Branches There is an `if` statement that can be used to send the evaluation down different branches. The format is `(if (test if-branch else-branch)`. The program evaluates the `test` and if it is true, then it follows the `if-branch` otherwise it follows the `else-branch`.

The format of the test is quite similar to general LISP. For instance, the test `(gt a b)` returns true if `a` is greater than `b`. The

other decision functions are `lt`, `le`, `ge` and `eq`, which stand for less than, less than or equal to, greater than or equal to, and equal to.

The current implementation watches for errors that might be introduced if the two variables used to make a decision were changed along one of the branches. It does this by pushing the names onto the `Forbidden-List` and then checking to see the list before the evaluation of each operation.

Program Counter and Code This machine is like most other software programs. There will be one major procedure with the tag `main`. This is the first procedure executed, and the RGM ends when finished. Other procedures are executed as they're encountered and a stack is used to keep track of the position in partially finished procedures.

The source code is available from the author.

8.4 Summary

Letting a machine run backward is just one way to create the most complicated computer-generated mimicry. You could also create double-level grammars or some other modified grammar-based system.

The Disguise The text produced by these reversible machines is as good as a computer could do. But that may not be so great. Computers have a long way to go before they can really fool a human. Still, static text can be quite realistic.

How Secure Is It? Assessing the security of this system is even more complicated than understanding the context-free grammars used in Chapter 7. Theoretically, there is no Turing machine that can make nontrivial statements about the reversible Turing machine. In practice, there may be fairly usable algorithms that can assemble information about the patterns in use. The question of how to create very secure programs for this reversible machine is just as open as the question of how to break certain subclasses.

How to Use It The LISP software is available from the author. It runs on the XLISP software available for free at many locations throughout the Internet.

Further Work The LISP code is very rudimentary. It's easy to use if you have access to a LISP interpreter. A better version would offer a wider variety of coding options that would make it easier to produce complicated text.

A more interesting question is how to *guarantee* security. Is it possible to produce a mechanism for measuring the strength of a reversible grammar? Can such a measuring mechanism be guaranteed? An ultimate mechanism probably doesn't exist, but it may be possible to produce several models for attack. Each type of attack would have a corresponding metric for evaluating a grammar's ability to resist is. Any collection of models and metrics would be quite interesting.

Further Reading

The field of reversible computing continues to draw some attention from people trying to build quantum computers. Some recent papers include a survey by Michael Frank and the papers from the tracks at the conference devoted to the topic. [Fra05] In particular, see the work of Daniel B. Miller, Ed Fredkin, Alexis De Vos and Yvan Van Rentergem. [VR05, MF05, TL05, BVR05, Fra05]

Chapter 9

Life in the Noise

9.1 Boy-Zs in Noizy, Idaho

Scene: A garage with two teens and guitars.

Teen #1 No. I want it to go, "Bah, dah, dah, dah, bah, screeeeech, wing, zing…"

Teen #2 How about, "Bah, dah, dah, dah, bah, screeeeech, screech, wing, zing…"

Teen #1 Hey, let's compromise: "Bah, dah, dah, screeech, zip, pop, screeech?"

Teen #2 Oh; I don't know anymore.

Teen #1 What's the problem?

Teen #2 I just get tired of trying to say something with noise.

Teen #1 Hey. We agreed. Mrs. Fishback taught us in English class that the true artist challenges contemporary society. We need to expose its fallacies through the very force of our artistic fervor. Our endeavor must course through the foundations of society like an earthquake that gets a 10.0 on the Richter scale.

Teen #2 Yeah. So what? She's just a hippie chick. That's her idea.

Teen #1 Come on. Join the clambake. We have to confront the conformity of the adults with an urgency that heretofore has not been seen on this planet. We need to demand that culture come alive with a relevance that can speak truth to the young and the restless. There are paradigms to shatter.

Teen #2 Would you shut up with that science stuff? Mr. Hornbeam said that Thomas Kuhn wasn't going to be on the final.

Besides people managed to have a good time even before Copernicus and Galileo broke the paradigms apart. What about melody and harmony?

Teen #1 We can make our fuzz circuits do everything for us. Suburbia is just sleepwalking through life. Only our harsh notes can wake them to the discord that lies beneath the greenswept swards of our existence. That's what Mrs. Fishback says.

Teen #2 Cut it out. You like your dad's car as much as I do. How would you like your father to awaken you from your sleepwalking and force you to do some actual walking to the mall? I don't want to make Mrs. Fishback's music.

Teen #1 Why not? She obviously understands the evil hegemony proffered by a corporate culture intent on creating a somnolent adolescence. We are not people merely because we consume.

Teen #2 Nirvana, Pearl Jam, and the rest live on major labels sold at full list price at our mall.

Teen #1 Whoa! Perhaps we're being led to rebel in the hopes that anticulture will sell even more than traditional culture?

Teen #2 Yes. You got it.

Teen #1 It's true. Mrs. Fishback just wants us to create a youth she never had when she was running between classes and earning good grades. The revolution always ended in the 1960s when the exams came around. They smoked a little pot, went to a protest, but most of it was just grooving to the music and searching for someone to do some loving. Then they got married and got jobs. We're just doing what her generation wants. They're marketing to us through their dreams of what they wished their childhood had been.

Teen #2 You're getting the hang of it.

Teen #1 The pervasive drive to explode the previous is just another marketing move. Unknowingly, we're channeling our rebellious energy through a marketing path created by a cynical corporate structure intent on destroying the potential for upheaval in every youth. Instead of remaking the world with our passion, we're simply consuming anticultural icons constructed as pseudo-rebellious pabulum.

Teen #2 Bonzai!

Teen #1 So what do we do?

Teen #2 I have this Beethoven music here. It has no copyright.

Teen #1 Excellent. By reinvigorating the classic music, we'll be

subverting the corporate music world that uses the laws of intellectual property to milk our youth. Instead of working long hours at McDonald's to save for a new $17.95 Nirvana Retrospective CD, we'll truly shatter the power structure by playing music long freed from the authorial and corporate imperative.

Teen #2 And there are some great bass chords in this Ninth Symphony.

9.2 Hiding in the Noise

Noise, alas, is part of our lives. The advertisements for digital this and digital that try to give the world the impression that digital circuits are noise-free and thus better, but this is only half true. The digital signal may be copied and copied without changing the message. thanks to error-correcting codes and well-defined circuitry, but this doesn't eliminate much of the original noise. Digital photographs, digitized music, and digital movies all have a significant amount of noise that is left over from their original creation. When the voices, sounds, and photographs are converted into bits, the circuits that do the job are often less than perfect. A bit of electrical noise might slightly change the bits and there is no way to recover. This noise is something that will always be with us.

This noise is also an opportunity. If it doesn't really matter whether the bits are exactly right, then anyone who needs to hide information can take advantage of the uncertainty. They can claim the bits for their own through squatter's rights. This is probably the most popular form of steganography and the one with the most potential. There are millions of images floating about the Net used as window dressing for Web sites and who knows what. Any one could hijack the bits to carry their own messages.

The principle is simple. Digitized photos or sounds are represented by numbers that encode the intensity at a particular moment in space and/or time. A digital photo is just a matrix of numbers that stands for the intensity of light emanating from a particular place at a particular time. Digitized sounds are just lists of the pressure hitting a microphone at a sequence of time slices.

All of these numbers are imprecise. The digital cameras that generate images are not perfect because the array of charge-coupled devices (CCDs) that convert photons to bits is subject to the random effects of physics. In order to make the devices sensitive enough to work at normal room levels, they must often respond to only a few photons. The randomness of the world ensures that sometimes a few

too many photons will appear and sometimes a few too few will arrive. This will balance out in the long run, but the CCD must generate an image in a fraction of a second. So it is occasionally off by a small amount. Microphones suffer in the same way.

"God is in the details." –Mies van der Rohe

The amount of noise available for sending information can be truly staggering. Many color digital photographs are stored with 32 bits allocated for each pixel. There are 8 bits used to encode either the amount of red, blue, and green or the amount cyan, magenta, and yellow of each pixel. That's 24 bits. If only one pixel from each of the colors was allocated to hiding information, then this would be $1/8^{th}$ or about 10% of the file. At the top of the scale, a Kodak photo-CD image is 3072 by 2048 pixels and takes up about 18 megabytes. That leaves about 1.8 megabytes to hold information. The text of this book is well under half a megabyte, so there is plenty of room for hiding more information in a *single* snapshot. Many people won't want to spend 18 megabytes of storage space on a single snapshot. Less precise versions of images can run between 200k to 600k and still devote about 10% of their space to hidden data.

But if about 12.5% is devoted, how much does this affect the appearance of the image? Each of these 8 bits stores a number between 0 and 255. The last bit in each group of 8 bits is known as the least significant bit. It's value is 1. The most significant bit, the first one, contributes 128 to the final number if it is a one. This means that the least significant bit can change the intensity of a pixel in the final image by about .5 to 1% at the most. Trading 12.5% of the image data in a way that will only affect the final image by about 1% is a good solution.

There is no reason why more data can't be stored away. If the two least significant bits are given over to hidden data, then each pixel cannot change by more than 3 units. That is still about 1 to 3% of the value of a pixel. But this is 25% of the final image size. This is a huge amount of bandwidth waiting to be captured and used.

9.2.1 Problems with the Noise

The amount of bandwidth available in the least significant bits of an image or a sound file is large, but it is not always easy to exploit. Unfortunately, potential steganographers must fight for the hidden spaces with compression algorithm architects who want to create compression algorithms that strip away the extra space.

Chapter 5 discusses compression algorithms and how they can enhance steganography.

The basic image format may use 24 bits to encode the color of each pixel, but this basic format is used less and less frequently. Compression algorithms like JPEG do a good job and often use one or

two bits per pixel. Basic compression algorithms like JPEG can easily save a factor of 10 without significantly distorting an image. Most digital cameras, for instance, now come with built in JPEG compression chips to save space and allow people to take more pictures. 24-bit color may be slightly more accurate, but no one wants to waste the space on it.

The same holds true for music. Today, MP3 files are much more common than files that record the intensity at each time slice. Compression algorithms like MP3 can easily save a factor of 10 over raw digitized data. Newer algorithms can save even more. This is great news if you're storing your CD collection on your computer, but not if you want an easy channel to exploit for steganography.

This effect, incidentally, is what leads some steganographers to hide information in the most "perceptually significant" parts of a file. That is, they want to ignore the noise and hide the information in the part that the humans can perceive. The noise will eventually extracted and removed by some compression algorithm, but the perceptually important parts will live on. [CKLS96] Instead of hiding information in subtle changes of the intensity, hide the information in the position of a person's nose or the length of the hair.

This is a good point, but it is more of a challenge for researchers and a loose design principle. Even if basic mechanisms for exploiting the noise in a file may not be as robust as possible, they are still worth exploring. The rest of this chapter is devoted to noise. Following chapters attempt more robust solutions.

9.2.2 Good noise?

A practical problem is finding good noise. Most image and sound files include enough natural noise

Chapter 17 discusses how some cameras don't provide good enough noise to mask hidden bits.

to hide a 3% change, but this noise is rarely as pure as can be. Figure 9.1 shows a black-and-white scanned image of a photograph taken of a computer on a desk. Figure 9.2 shows just the least significant bits. It is obvious that there is a highly random pattern to them caused by the noise in the digitizing circuit on the scanner. It is random, but it is not as random as it could be.

Many images and sound files probably have enough inherent noise to hide data. The image in Figure 9.1 has plenty of junk so small variations don't show up. But there are some images that do not handle the imposed noise as well. Many images are created entirely on the computer in applications like Adobe Illustrator. These produce pure, consistent fields of color. Even modifying them a bit can stand

Figure 9.1: This is a black-and-white photo of the author's desk. There is plenty of junk on the desk that is hiding secret documents. The noise in the image lends itself to hidden data as well.

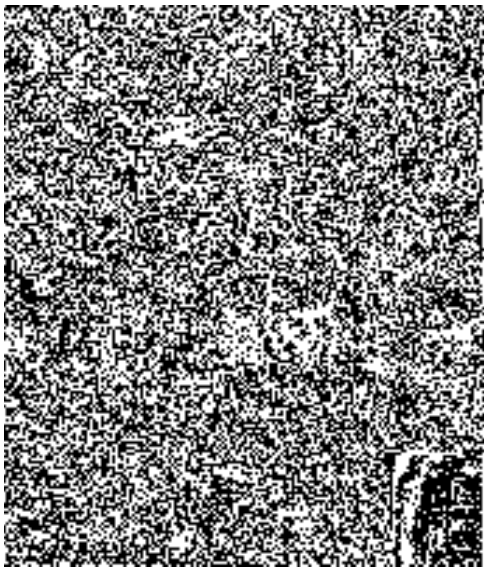


Figure 9.2: These are the least significant bits of the photo in Figure 9.1. The most significant bits were deleted to show the randomness that exists at this level. (See Figure 9.4.)

out because a pure tone is converted to one with a bit of noise.[1]

"Putting JPEG to Use" on page 183 shows how compression can identify just how much space can be exploited in an image.

9.2.3 Independence Problems

One of the deeper problems is defining good noise. The least significant bits of a music or image file often seem close to random, at least to the average eye or ear, but they often contain hidden patterns and structure. Many of the microphones, cameras or scanners used to generate the files are far from perfect and they often introduce their own patterns.

One of the most common is a correlation between the high order bits and the least significant bit. A picture of a bright day might include a number of intense reflections of the sun. These pure white points usually saturate the sensor and produce patches of maximum values of 255. There are relatively few values of 254. If the least

[1] The pure colors are often jarring to the eye and this is why artists often use textures and slight imperfections to make the image more appealing. It is anyone's guess why the optic nerve seems to react this way, but perhaps it is an effect like the moiré patterns produced when one pattern is digitized at too coarse a level. If you've ever looked at anyone wearing a fine checked shirt or tie on television you may have seen this effect.

significant bits were truly random and uncorrelated with the higher order bits, then there would be equal numbers of 255s and 254s. This often happens to a lesser degree in many other parts of the image where only one color may appear completely saturated.

Chapter 17 offers a number of basic statistical tests for identifying the presence of steganography.

This is just a simple example. Good scientists with a deep knowledge of the physics behind the sensors can often detect more sophisticated patterns. Digital copier manufacturers, for instance, can tune their toner mechanisms to do a good job with either fine white lines or fine black lines. Doing well with both is difficult.

All of these subtle statistical patterns can be destroyed when you replace the least significant bits with your message. Simply pouring in a well-compressed and encrypted message puts in white noise with no correlations to the higher order bits.

There is no easy way to avoid this problem. The mimic functions from Chapter 6 can be used in complicated ways to imitate the patterns, but this is largely a cat-and-mouse, spy-vs-spy game. The attacker may have some model of the statistical correlations in the file. If you can anticipate this model or come up with your own that encompasses it, then you can mold the data in your message to fit it. If you choose incorrectly or the eavesdropper/attacker changes their model, your data could stick out like a sore thumb.

There is no solution to winning this game, but it is possible to minimize the dangers of playing it. The best defense against statistical problems like this is to avoid getting greedy by packing in too much data. An 800kB file has 800kB least significant bits available that can store up to 100kB bytes of a message. Using all 100kB of the channel, however, will completely destroy all statistical correlations but inserting 1k message will leave 99% of the least significant bits unchanged. Most of the statistical patterns will also be unchanged and thus indistinguishable from a pure file.

All of the statistical techniques for detecting steganography usually become much less sensitive when only a small fraction of the available bandwidth is used to hold a message.

See Chapter 17 for a description of some of these algorithms for detecting hidden messages.

Another more sophisticated solution is to embed the information in several pixels at the same time. A simple way is to just choose several pixels and embed the information in the parity of these pixels. Let a zero be encoded by ensuring that there's an even number of 1s in the least significant bits of the chosen pixels. A one can be encoded with an odd number of 1s. This spreads out the change.

All of the analytical attacks on steganography work best when the steganographer gets greedy and saturates the hidden channel. Leaving most of the image unchanged is the best defense.

9.2.4 File Format Grief

File formats are a serious problem for anyone who would like to routinely use bit-level steganography to hide images. Many image or audio file formats were designed to squeeze out some of the extra noisy details to save space. This might be done by a special, efficient file format like GIF or by an aggressive compression program that does not care if it reconstructs an image that is not exactly the same. Both of these make it hard to just hide information in the least significant bits of an image.

Jessica J. Fridrich and Rui Du propose hiding information in the parity of the sum of the colors of multiple pixels. The more pixels used for each color, the smaller the changes that need to be made.[FD99, AP98]

The GIF file format and its 8-bit color standard is a significant impediment because 8-bit color is quite different from 24-bit color. It uses a table of 256 different colors that best represent the image as a color map. The color of each pixel is described by giving the closest color from this 256-entry table. The bits do not correspond to the intensity of the colors at each pixel. This means that changing the least significant bits doesn't necessarily change the intensity at a pixel by less than 1%. Entry 128 of the table might be a saturated ruby red while entry 129 might be a pale, washed-out indigo. They may only differ in the least significant bit, but that can be enough to cause major changes in the final outcome.

Yes, the newspapers were right: snow was general all over Ireland. —James Joyce in The Dead

There are a number of different solutions to this problem. The first is to use a smaller number of colors in the table. Instead of choosing the 256 colors that do the best job representing the colors of an image, the software could choose 128 colors and then choose 128 colors that are quite similar to the original 128. They might even be the same, but that could be too suspicious. The table could be arranged so that the two very similar colors only differ in one bit. Here's an abbreviated example of such a table:

Entry Num.	Binary	Red Intensity	Green Intensity	Blue Intensity
0	00000000	150	20	10
1	00000001	151	20	12
2	00000010	14	150	165
3	00000011	16	152	167
4	00000100	132	100	10
5	00000101	135	67	15
⋮	⋮	⋮	⋮	⋮

So if you want to hide the value 1 in a pixel, you would find the closest color in the table and then choose the version of it with the least significant bit set to one. If the closest color had red set to 15, green set to 151, and blue set to 167, and you wanted to hide the bit

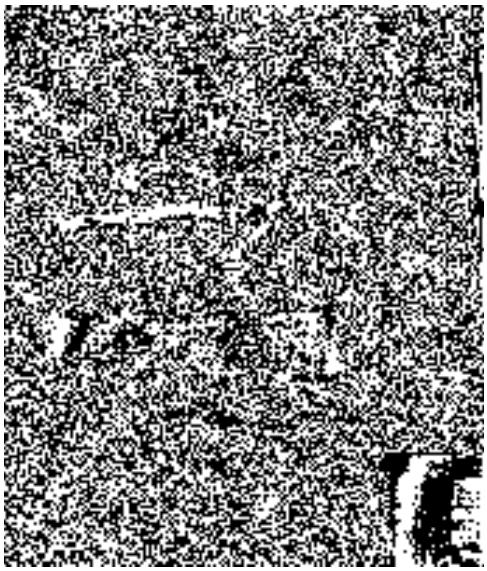


Figure 9.3: Even the second-to-least significant bits appear fairly random. These are the second-to-least significant bits of the photo in Figure 9.1.

1, then you would choose color 3 for that pixel. If you wanted to hide a 0, then you would choose color 2.

There is no reason why the least significant bit needs to be used to separate pairs. It might very well be any of the bits. Here is another table where the third bit is used to mix pairs:

Entry Num.	Binary	Red Intensity	Green Intensity	Blue Intensity
0	00000000	150	20	10
⋮	⋮	⋮	⋮	⋮
2	00000010	14	150	165
⋮	⋮	⋮	⋮	⋮
4	00000100	132	100	10
⋮	⋮	⋮	⋮	⋮
33	00100001	151	20	12
⋮	⋮	⋮	⋮	⋮
35	00100011	16	152	167
⋮	⋮	⋮	⋮	⋮
37	00100101	135	67	15

Figure 9.4: These are the most-significant bits of the photo in Figure 9.1. The seven least significant bits were deleted to contrast the images in Figure 9.2.

Nor is there any reason why only 1 bit is encoded in each pixel. The same algorithms that choose 256 or 128 best colors for an image can be used to find the closest 64 colors. Then 2 bits per pixel could be allocated to hidden information. Obviously this can lead to a degraded image, but the hidden information can often moderate the amount of degradation. Imagine, for instance, that we tried to be greedy and hide 4 bits per pixel. This would leave 4 bits left over for actually specifying the color and there could only be 16 truly different colors in the table. If the photo was of a person, then there is a good chance that one of the colors would be allocated to the green in the background, one of colors would go to a brown in the hair and maybe two colors would be given over to the skin color. A two-toned skin could look very fake.[2] But each of these two tones might also be hiding 4 bits of information. This would be mean that there were 16 surrogates for each of these two tones and these 16 surrogates would be used fairly ran-

[2]Recent work suggests that human eyes pick up skin tones more than most colors. So the best algorithms devote more colors in the table to skin colors in the hopes of better representing them. The eyes don't really seem to care much about the shade of green in a tree.

domly. This would add a significant amount of texture that might mitigate some of the effects.[3]

Another significant hurdle for image- and sound-based steganography is the compression function. The digital representations of these data are so common that many specialized compression functions exist to pack the data into smaller files for shipping across the network. JPEG (Joint Photographic Experts Group) is one popular standard algorithm for compressing photographs. MPEG is a similar standard designed for motion pictures.

Both of these are dangerous for bit-level infopacking because they are *lossy* compression functions. If you take a file, compress it with JPEG, and then uncompress it later, the result will not be exactly the same as the original. It will look similar, but it won't be the same. This effect is quite different than the *lossless* compression used on many other forms of data like text. Those functions reproduce the data verbatim. Lossy compression functions are able to get significantly more compression because they take a devil-may-care attitude with the details. The end result looks close enough. The JPEG algorithm itself is adjustable. You can get significantly more compression if you're willing to tolerate more inaccuracies. If you turn up the compression significantly, the pixels begin to blend into big blocks of the same color.

JPEG compression can also help. See page 183.

9.2.5 Deniability

Deniability is one of the greatest features of hiding information in images from Web pages. If you structure your information correctly, you can spread it out among a number of unrelated locations. If the information is discovered, it will be impossible to tell exactly where it came from.

Imagine that you have some bits that you want to distribute to the world. You could hide these bits in an image file and place it on your home page for all to download. If unintended people discover the bits, however, they know the information came from you because it is on your Web page.

Instead, you can split up the information into n parts using the basic tricks from Chapter 4. These n files, when they're XORed together, will reveal the hidden data. Ordinarily, you would create $n-1$ files at random and then compute the last file so that everything adds up. But why bother using files at random when there is a great source of randomness on the network? You could snarf $n-1$ different GIF images from the Net and use them. One might be a picture of Socks

[3] Random noise has been used to make quantization look more realistic. Too many discrete levels look artificial [Rob62].

the Cat from the White House page. Another might be a quilt from the page of some quilting club that is the picture of innocence. Anyone who wanted to recover the information would get all of these GIFs from the Net, recover the secret bits from all of them, and add them up to recover the hidden data.

The net effect of this trick is deniability. No one can be sure who was the one who was hiding the secret bits. Could it be the White House? They've been known to sponsor covert missions based in the Old Executive Office Building. Could it be the quilting circle? No one knows who injected the secret bits into the file. Even the person who recovers the bits might not know who was sending the message. It's quite a ruse.

The section beginning on page 99 describes sophisticated ways of matching patterns in the least significant bits.

There are some practical problems associated with this technique. First, you must keep the file creation dates secret. The one GIF that actually contains the message will be the newest file. HTTP doesn't usually ship this information to Web browsers so there is little problem with keeping the information secret. But you can also fake it by resetting the clock on your machine.

Second, you should search out GIF files that seem to have the right structure for storing secret bits. This will prevent someone from examining the files and discovering that only one of them has the right structure to hide bits. That is, all but one of the n files are 8-bit color with color tables filled with 256 different shades.

Third, you should worry about one of the images disappearing from the Net. It's tempting to use images from other web sites for parts because it will deflect attention and hide the source of the message, but this could be thwarted if someone redesigns a web site.

You can add some error-correcting features to this scheme if you want to create, say, three different sets of files. When each set of the three sets of files are combined, then three versions of the hidden bits emerges. Any disparities between the files can be resolved by choosing the value of the bit in question that is correct in two out of three files.

More complicated error-correcting schemes like the ones described in Chapter 3 can also be used successfully. For instance, a file to be hidden could be encoded with an error-correcting code that converts every 8 bits into, say, a 12-bit block that can recover errors. One bit from each of the 12-bit blocks could be placed into 12 separate files that were then hidden in 12 different GIFs sprinkled around the network. If someone could not recover all 12 GIFs because of network failures, then the error-correcting code will allow the information to be recovered.

9.2.6 Finding Edges

The Digital Invisible Ink Toolkit written by Kathryn Hempstalk includes several algorithms for encoding information in the most visual significant parts of the image. How does it identify them? By using a Sobel filter or a Laplace filter, two standard averaging tools that identify the pixels that are most different from their neighbors. These are usually called the *edges* of the item.

The software package locates these edges and then tweaks the least significant bits at these locations. The algorithms only feed the most significant bits into the filters to avoid any problems caused by side effects from changing the least significant bits. It is possible that the change of the least significant bit would turn a pixel from an edge into a non-edge making it invisible to the decoding algorithm.

One of the version called *BattleSteg* will choose pixels at random until it finds one that is significantly edgy. Then it tweaks the least significant bits of the pixels around it. The random selection process is driven by the password so it the recovery process can repeat the same sequence. FIgure 9.5 shows an image with a hidden information and the locations where the data is stored.

9.3 Bit Twiddling

Perhaps the best way to begin experimenting with hiding information in the noise is to download one of the experimental packages floating around the Net. There are easily more than one hundred programs that have been circulated publically, but many of them seem to disappear as quickly as they appear.

Appendix A offers a list of some of the more prominent steganography packages.

9.3.1 Working with GIFs

GIF files store their images by creating a pallette of 2^n different colors and then mapping the colors from the image to the closest color. This can make it harder to store data by flipping the least significant bits because two adjacent entries in the table of colors may wildly different even though they only differ by one significant bit.

One of the earlier programs, Hide and Seek worked around this problem.

One version, `grey.exe`, converted color GIFs into grayscale GIFs that would not show any of the artifacts associated with 8-bit color

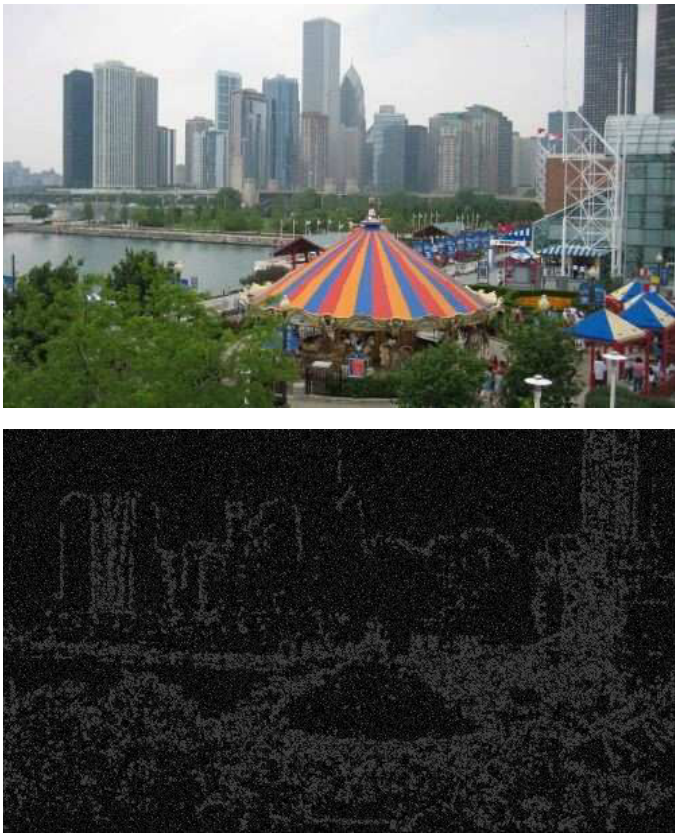


Figure 9.5: The top photo shows an image after the BattleSteg algorithm hides information in the edges identified by a Laplace Filter. The bottom image shows these most interesting or salient pixels.

Figure 9.6: This shows how adding information to the least significant bit of a 24-bit color image has little effect. The image was later converted to black and white for this book. (Photo courtesy of the Lacrosse Foundation.)

Figure 9.7: 8-bit color images can make poor candidates for adding information into the least significant bit because entries in the color table might not be next to each other. The image was later converted to black and white for this book.

steganography.The lack of color, though, could be a bit of a indication that something had changed.

The second program, `reduce.exe`, shrank the color table from 256 colors to 128 colors and then duplicates these 128 colors so that adjacent entries in the color table were exactly the same. If this is done, hiding information in the least significant bit won't affect the look of the image.

There are a number of different programs for reducing the size of the palette. The simplest is:

1. Create a two-dimensional matrix containing the distances between all pairs of colors in the palette. The distance between two colors, (R_1, G_1, B_1) and (R_2, G_2, B_2) is

   $$\delta((R_1, G_1, B_1), (R_2, G_2, B_2)) = \sqrt{(R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2}$$

   .

2. Find the best color to delete. That is, find the color with the closest neighbor and remove it.

3. Repeat until the palette is the desired size.

Unfortunately, this technique leaves a large red flag for anyone scanning GIFs looking for hidden information. While the eye may not see any change after flipping the least significant bits, an 8-bit color table with only 128 different colors is easier to detect automatically than a bad image with plenty of artifacts.

Chapter 17 describes a several techniques for detecting a steganographic message.

Another early program, EzStego's, solution for dealing with GIF palettes was a bit more wily. The software written by Romana Machado sorts the palettes so that the 2^n colors in an n-bit file flow smoothly from one to another. That is, each jump from the i-th color to the $i + 1$ is relatively small. Any hidden information that changes the least significant bit will introduce small changes. Ordering the colors in the palette is trivial with one-dimensional color spaces in black and white images, but it is much more difficult in three dimensions.

EzStego treats the challenge as a version of the Travelling Salesman Problem. The colors are cities in three dimensions (RGB) and the goal is to find the shortest path through all of stops.

There are no easy answers for Travelling Salesman problems so EzStego uses a basic approximation that works pretty well. It may not find the optimal solution, but it will often find one that works very well. Here's the algorithm:

Other approximations for the travellings salesman problem can be found in [Tah92, Cla83].

Begin with two colors in the list, $\{c_0, c_1\}$. Set the first one to be C. Repeat this until all colors are inserted.

1. Find the color that is farthest from C. Call it d.

2. Find the best place to insert this color in the list. That is, scan the list and find i such that $\delta(C, c_i) + \delta(C, c_{i+1})$ is minimized.

3. Insert d and set it to be the new C.

Finding the farthest color ensures that algorithm does not get "trapped" in one corner of the three-dimensional color cube only to find it must make big leaps to reach the other colors.

This sorting algorithm is not perfect and there's no guarantee that the jumps will be small. While the overall distance is minimized, some distances are more important than others. Adding information in the least significant bit will only produce some changes. In a three-bit system, 001 can only change to 000, not 010. Sorting the entire palette may leave one of the longer jumps between color 000 and 001 and put one of the shorter ones between 001 and 010. A more sophisticated approach will try to find the best way to pair up all of the colors in the palette so that the total distance of all of the pairs is minimized.

Jessica J. Fridrich offered an easy tweak to the algorithm. Instead of sorting the palette, the algorithm looks for the closest color with the right parity for representing the bit being hidden. The parity of the bit is $R + G + B \; mod \; 2$. [Fri99]

EzStego tries to thwart steganalysis by shipping the palette unsorted. Here are the steps:

1. Begin with an unsorted palette produced by a program like Photoshop.

2. Sort the palette so the closest colors fall next to each other in order.

3. Encode the message by twiddling the least significant bit. Make sure to encrypt the message with a cryptographically secure random number stream.

4. Unsort the palette by renumbering all of the colors with their original values from the original palette.

5. Ship the image. Any attacker looking at the palette will only see one produced by a non-steganographic program.

6. The receiver can re-sort the palette using the same sorting algorithm. If the algorithm is deterministic, it will produce the same results.

7. The receiver can now extract the bits by using the sorted palette values assigned to the colors. The least significant bits now encode the message.

This solution removes the problem with a highly structured palettes because it uses one produced by the image's creating software. Any attacker will have trouble determining the existence of the message by looking at the structure of the palette.

Andreas Westfeld and Andreas Pfitzmann created the pallettes in Figure 9.8 for their paper [WP99] with only 61 short lines of Postscript!

9.3.2 Smarter Color Reduction

There are two different types of image steganography supported by S-Tools a package written by Andy Brown. The software, now up to version 4.0, can hide information in images stored as either GIFs or BMP files orin sound stored as WAV files. Earlier versions even offered to store data in the unallocated sectors of a disk.

Wilson MacGyver Liaw wrote a good introduction to the GIF file format in [Lia95].

While the software can hide information in the least significant bits of 24-bit BMP files, the software can also reduce the image to 256 colors with an algorithm designed by Paul Heckbert [Hec82] to reduce the number of colors in an image in the most visually nondisruptive way possible. The algorithm plots all of the colors in three dimensions and then searches for a collection of n boxes that contains all of the colors in one of the boxes. When it is finished, it chooses one color to represent all of the colors in each box. S-Tools offers three different options for how to choose this one color: the center of the box, the average box color, or the average of all of the pixels in the box.

The process for constructing the set of boxes is described in detail in Heckbert's thesis. The process begins with the complete $256 \times 256 \times 256$ space as one box. Then it begins to recursively subdivide the boxes by splitting them in the best way possible. It continues this splitting process until there are n boxes representing the space. Heckbert developed this algorithm to correct some of the defects he found in the "popularity" algorithms being used. These algorithms would clump together nearby colors until only n clumps were left. Then it would choose some color, usually the center of the clump, to represent all of the colors. This works quite well for colors in tight clumps, but it can be disastrous for colors that are part of big, gaseous clumps. In those cases, the difference between the colors and their chosen representative was too large. This would lead to big shifts in the colors used in the details.

David Charlap wrote a good introduction to the BMP format in [Cha95a, Cha95b].

Heckbert suggests that a good way to understand the two approaches is by comparing them to the "quantization" methods used in choosing the representatives for the two houses of the Congress of

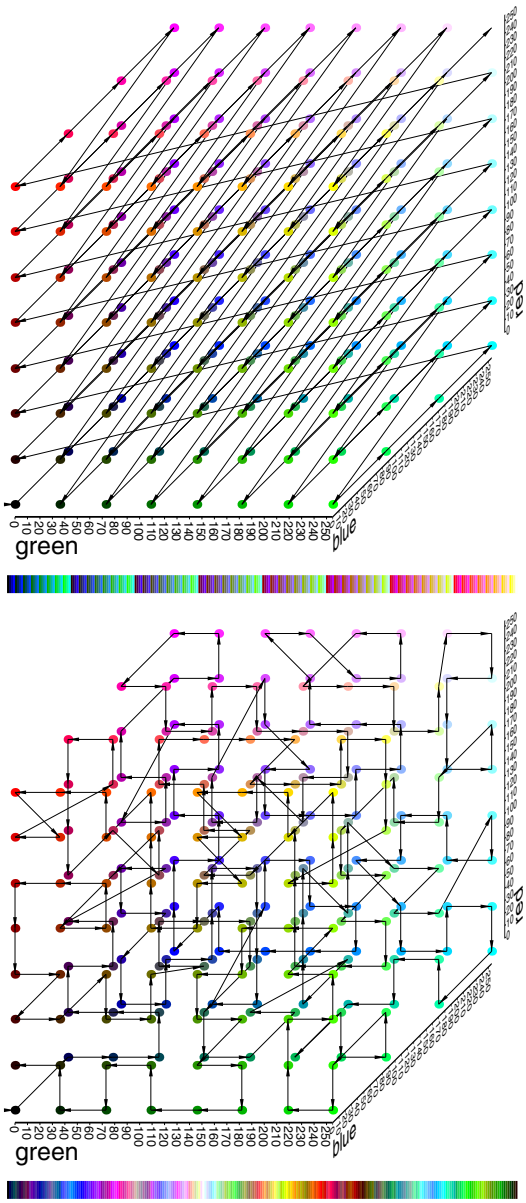


Figure 9.8: A generic palette is shown on the top and the sorted version is shown on the bottom.

the United States. The Senate gets two members from each state and Heckbert compares this to his algorithm. It spreads out the representation so no part of the color space is over- or under-represented. The House of Representatives, on the other hand, gets one representative for each unit of population. This works well if you're from heavily populated areas like Manhattan. These have representatives for each part of town. Western states like Nevada, however, have only one representative and thus have little power in the House. Heckbert compares this approach to the "popularity" algorithms.

The subdivision algorithm used by S-Tools can use two different ways to cut the boxes. In one way, the largest dimension is chosen by measuring the greatest difference in RGB values. In the other way, the largest dimension is found by comparing the luminosity of the different choices. Here is the basic algorithm in detail:

1. Place all of the colors from the image in one box.
2. Repeat this until there are n boxes that will represent the final n colors.
   (a) For each box, find the minimum and maximum value in each dimension. That is, find the smallest and largest value of red for any color in the box, the smallest and largest value of green, and the smallest and largest value of blue.
   (b) For each dimension of each box, measure the length. This might be the difference in absolute length or it might be the difference in luminosity.
   (c) Find the longest dimension and split this particular box. Heckbert suggests this can be done by either finding the median color in the box along this dimension, or you can choose the geometric middle.
3. Choose a representative color for all of the original colors in each box. S-Tools offers three choices: center of the box, average of the colors, or average of the pixels.

When the new set of n colors is chosen, S-Tools can use "dithering" to replace the old colors with new ones.

The algorithm attempts to find the best number of new colors, n, through a limbo process. It slowly lowers the number of colors until it ends up with less than 256 colors after the data is mixed into the least significant bits. Often, it must repeat this process several times until the right number is found. S-Tools cannot predict the number of final colors ahead of time because it constantly tries to

add 3 bits to each pixel. That is, it takes the red, green, and the blue values for each pixel and changes the least significant bit of each one independently. That means one color could quite possibly become eight. This is quite likely to happen if that color is common in the image because each pixel is handled differently. On average, each of the eight slightly different colors should appear after ten to twelve pixels of the same color are mapped.

The program MandelSteg, developed by Henry Hastur, hides information in the least significant bit of an image of the Mandelbrot Set. This synthetic image is computed to seven bits of accuracy and then the message is hidden in the eighth. *See page 319.*

This means that it is impossible for the algorithm to predict the final number of colors it needs. It might try to reduce the number of final colors in the image to 64. Then, after the data is mixed in, it might end up with 270 colors or 255. If it was 255, then it could save the file. Otherwise, it would start the process again and reduce the colors some more. The entire process is iterative. S-Tools attempts to predict the correct number through extrapolation, but it has taken several iterations every time I modified a file.

9.3.3 Sound Files

The simplest way to hide information in the noise is to use uncompressed sound files in formats like the WAV. S-Tools can store data in the least significant bits of a WAV file—one of the standard sound formats for Microsoft Windows. These files can use either 8 or 16 bits of data to represent each instance. People with Sound Blaster cards will have no problem generating these files from any source.

S-Tools hides one bit per either 8 or 16 bits and will also use a random number generator to choose a random subset of bits. This spreads the distortion throughout the sound file. The program will display a graph representing the sound and also play it for you. After data is hidden, the graph shows all of the changes made to the wave form in red and leaves the unchanged parts in black. This is, in effect,

revealing where the pattern of ones and zeros in the hidden file differed from the least significant bits of the sound file. Figure 9.9 shows a screen shot from the program.

9.4 Random Walks and Subsets

This chapter has discussed hiding information in image or sound files by grabbing all of the least significant bits to hold information. There is no reason why all of them need to be used. Both Hide and Seek and S-Tools use random number generators to choose the bytes that are actually drafted to give up their least significant bits to the

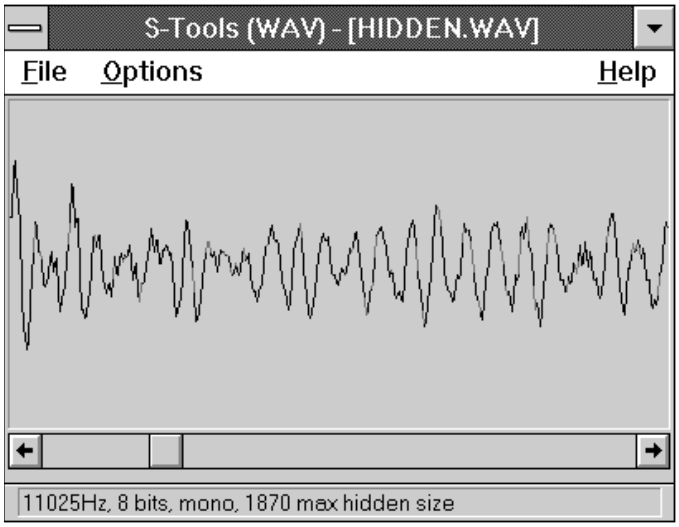


Figure 9.9: The main window of `st-wav.exe`, the S-Tools program for hiding information in a WAV file. The program displays the changed parts of the waveform in red, but this detail is lost in a black-and-white book.

cause. This random process guarantees that the distortion is distributed throughout the file so it is not so apparent. It also makes it difficult for some attacker to figure out which bits are important. S-Tools uses the MD5 algorithm to ensure that the random numbers are cryptographically secure. A more modern hash function like SHA256 may make more sense now that researchers are discovering a number of weaknesses in MD5 .

In fact, a random subset can have some other uses. First, if a person selects a small, random subset to store information, then another person could do the same thing. If both use different sources of randomness to choose small subsets, then there is a good chance that very few bits will end up in both subsets. Error-correcting codes can help recover from this overlap. This could allow several people to use the same file to pass information to several different other ones.

Steve Walton suggested this approach in his article, "Image Authentication for a Slippery New Age" [Wal95b]. This approach uses a general, two-dimensional random walk that weaves around a picture. Occasionally, the path may wrap around itself which requires keeping track of where the path has been before. Hide and Seek, in contrast, views the picture as a one-dimensional list of pixels and chooses a random number of pixels to jump ahead.

The source code was also included with the earlier versions of Hide and Seek, which makes it possible to look at the guts of the program. This makes it possible to examine how information is *dispersed* throughout the bits. If a smaller file is going to be packed into the bits, then the program tries to randomly arrange the bits so that they're not adjacent to each other. This has two effects. First, the noise is more randomly distributed throughout the image instead of being clustered on the top half. Second, you must know the location of bits to find the data and the location is governed by a random number generator driven by a user-chosen key. Both of these enhance the security of the system.

The dispersion is controlled by an 8-byte header for the file. The first 2 bytes are the length of the file, the second 2 bytes are a random number seed that is chosen at random when the information is packed and the third pair of bytes is the version number. The fourth pair is not used, but is included to fill out an 8-byte block for the IDEA cipher. This block is encrypted with the IDEA cipher using an optional key and then stored in the first 64 pixels of the image. If you don't know the key, then you can't recover the header information that controls the dispersion of the data throughout the image.

The actual dispersion is random. At the beginning, the random number generator is seeded with the second pair of bytes from the header block. The code from Hide and Seek 4.1 just uses the built-in C random number generator which may be adequate for most intents and purposes. A stronger implementation would use a cryptographically secure random number generator. Or, perhaps, it would use IDEA to encrypt the random bit stream using a special key. Either method would add a a great deal of security.

Here's the section of C code devoted to handling the dispersion. The code for getting the color table entry for pixel (x, y) and flipping it appropriately is removed. This code will store an entire byte in the eight pixels. The amount of the variable `dispersion` controls how many pixels are skipped on average. It is set to be the rounded off amount of 19000 divided by the length of the incoming data.

```
int used=0,disp=0,extra=0;

for(i=0;i<8;i++) {

// Code removed here for flipping LSB of (x,y)

disp=(random(dispersion+extra)+1);
```

```
used+=disp;

extra=((dispersion*(i+1))-used);

x+=disp;

// Move over x pixels. Code removed to handle

// wraparound.

}
```

There is a certain rough, frontier aspect to just choosing arbitrary random walks throughout the data. It does not require that there be any prior communication between two people who happen to be hiding information in the same picture. They're just both keeping their fingers crossed that the collisions will be minor and the error-correcting codes will be able to fix them.

Here is a more principled way to create multiple channels in an image. If all parties coordinate their use ahead of time, they can ensure that their random walks will not collide. This saves space because error-correcting codes do not need to be used, but it does increase the complexity of the process.

To create n channels, divide the file into n-byte blocks of data. One byte from each block will be given to each channel. In the simplest and most transparent approach, the assignment of byte and channel number is hard coded. Channel 1 gets byte 1, channel 2 gets byte 2, and so on. A better approach shuffles the bytes by using a set of permutations of the values between 1 and n. Here's a good way to generate a sequence of random permutations of the set:

1. Start with the ordered set $(1, 2, \ldots, n)$.
2. To generate a new random permutation repeat this j times. A larger j is better, but less efficient.
   (a) Choose two items in the set at random using the output of a cryptographically secure random number generator.
   (b) Swap their positions. For example, if the set is $(5, 1, 3, 2, 4)$ before and the second and fourth values are chosen by the random number generator, then the result will be $(5, 2, 3, 1, 4)$.
3. Output this permutation. Goto step 2 to keep going.

The i-th permutation spit out by this permutation generation routine can determine which channel gets which byte in the i^{th} block. This ensures that no two users will collide.

Another approach can mangle the process even more. Why should blocks be made up of adjacent bytes? In the most basic approach, byte i from channel k in an n-channel system is assigned to byte $in + k$ in the file. This can be scrambled using exponentiation modulo the length of the file. So if the file is p bytes long and p just happens to be prime, then $(in + k)^e \bmod p$ will scramble the bytes so they are not adjacent.

Walton imagines that the least significant bits in his random walk can be used to construct a seal for the image. That is, you can "sign" the image by embedding some digital signature of the image in the least significant bits. Naturally, this digital signature would only be computed of the non-least significant bits because those bits are the only ones that would remain unchanged during the process. This sealing system could be used by professional photographers to attach their mark to a photograph.

Some have argued that this approach is a waste. Appending the signature data to the end of the photo made more practical sense. This type of signature would be able to handle all types of photo formats including binary images without enough significant bits to hide data. Also, there would be no need to avoid the least significant bits while encoding the information and so the signature would be even better.

Another solution is to create a random permutation of the bits. Tuomas Aura describes this in [Aur95].

These suggestions are certainly correct. The only advantage that the surreptitious approach would have is secrecy. Presumably photographers would sign images to protect their copyright. They could prove conclusively that the photo was stolen. If the signature is appended to the file, then someone could remove it or tamper with it. If it is hidden with a random walk in the least significant bits, then someone has to find it first. Of course, malicious people could just write over the least significant bits of a photo as a precaution.

9.4.1 Empty Disk Space

The algorithms for choosing a random subset of an image or sound file by walking around them at random can be used to hide information in other ways.

Earlier versions of S-Tools included a program, `st-fdd.exe`, to hide information in the unallocated areas of a floppy disk. Each disk is broken into sectors and the sectors are assigned to individual files by the file allocation table (FAT). The unused sectors are just

sitting around not doing anything. If someone tries to open them up with an editor like a word processor or even tries to examine them with a File Manager, they'll find nothing. This is just empty space to the operating system. But this doesn't mean it can't hold anything. Information can be written into these sectors and left around. The only way it can be corrupted is if someone writes a new file to the disk. The operating system may assign those sectors to another file because it thinks the space is free.

S-Tools stores information in this free space by choosing empty sectors at random. The first sector gets the header of the file which specifies the length and the random number seed that was used to choose the sectors. Then the information is just stored in this string of sectors selected at random.

If encryption is used, the random number generator uses the encryption key as a seed. This means that a different selection of random sectors will be chosen. The data itself is encrypted with any of the five algorithms offered in the other two implementations of S-Tools.

At the end, S-Tools offers to write random noise in the extra space that is not taken up by the hidden file. This is often a good idea because the empty space may often have some pattern to it left over from the last file it stored. Ordinarily, disk space is not actually cleared off when a file is erased.

The entry in the FAT table is just changed from "assigned" to "empty." The old data and its pattern are still there. This means that someone could identify the sectors of a floppy disk containing hidden information by looking for the ones that have random information. The ones that contain scraps of text files or image or ordinary data would be presumed innocent.[4] S-Tools will overwrite this to convert the unallocated sectors into a sea of noise. This is equivalent to using a new disk.

9.5 Putting JPEG to Use

The first part of this chapter lamented the effects of JPEG on image files holding data in the least significant bits. The lossy compression algorithm could just mush all of that information into nothingness because it doesn't care if it reconstructs a file correctly. Although this can be problem if someone uses JPEG to compress your file, it doesn't

[4] You could first use `st-bmp.exe` or `st-wav.exe` to hide the information in a picture or a sound bite. Then you could store it in the unallocated sectors. Then it would look like random discarded information.

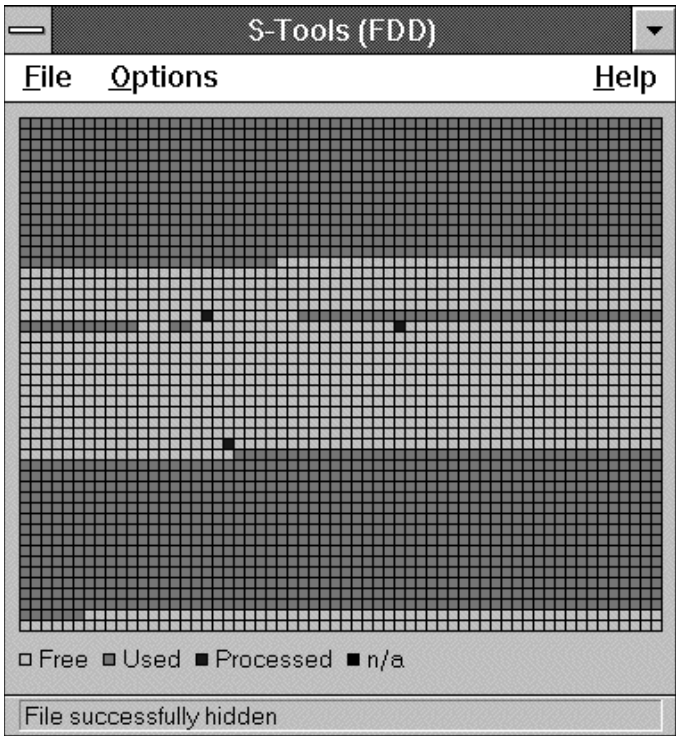


Figure 9.10: The user interface for `st-fdd.exe` shows the allocated sectors in red, the unallocated sectors in gray, and the ones that have been converted to hide information in yellow.

mean that the JPEG algorithm is useless to the person who would like to hide information successfully in images.

There are two possible ways that you can use the JPEG algorithm to store information. The first is to use it as a tool to identify the complexity of an image. This section discusses that approach. The second way is to use some hidden parts of the standard to hide information. That is described in the next section.

The JPEG algorithm can be a good tool for identifying the level of detail in a scene. This level of detail can be used to find the noisiest corners of the image where data can be stored. In the first part of the chapter, the basic algorithm for hiding information would just use the n least significant bits to hide information. If $n = 1$, there would only be a small but uniform effect that was randomly distributed throughout the image. If n was larger, then more information could be stored away in the image, but more distortion would also emerge. In any case, the distortion would be uniformly distributed across the entire image, even if this wasn't practical.

For instance, imagine a picture of a person sitting on a red-and-white checked picnic blanket in the middle of a grassy field. It might make sense to set $n = 4$ over a grassy section because it is out of focus and not particularly filled with important detail. On the other hand, you would only want to use $n = 1$ in the areas of the face because the detail was so significant to the photo. Naturally, you could go through the photo by hand and identify the most significant and fragile sections of the photo, but this would defeat the purpose of the algorithm. Not only would it be time-consuming, but you would need to arrange for someone on the other end of the conversation to construct exactly the same partition. This is the only way that they would know how to recover the bits.

The JPEG compression algorithm offers an automatic way to segment the photo and identify the most important or salient portions of an image. It was designed to do this to increase compression. The algorithm's creators tweaked the algorithm so it would provide visually satisfying images even after some of the detail was lost to compression.

The application is basic. Let f be a 24-bit image file waiting to have data hidden in some of its least significant bits. Let $JPEG^{-1}(JPEG(f))$ be the result of first compressing f with JPEG and then uncompressing it. The differences between f and $JPEG^{-1}(JPEG(f))$ reveal how much noise is available to hide information. For each pixel, you can compare f with $JPEG^{-1}(JPEG(f))$ and determine how many of the bits are equal. If only the first 4 bits of the 8 bits encoding the blue intensity are the same, then you can

conclude that the JPEG algorithm doesn't really care what is in the last 4 bits. The algorithm determined that those 4 bits could be set to any value and the resulting image would still look "good enough." That means that 4 bits are available to hide information. Elsewhere in the image, all 8 bits of f might agree with $JPEG^{-1}(JPEG(f))$. Then no information can be hidden in these bits.

This algorithm makes it possible to identify the locations of important parts of the image. You can choose the right accuracy value for JPEG as well. If you need a good final representation, then you should use the best settings for JPEG and this will probably identify a smaller number of bits available to hide data. A coarser setting for JPEG should open up more bits.

There are many other compression algorithms being developed to hide information. The fractal compression algorithms from Barnsley's Iterated Systems [BH92] are some of the more popular techniques around. Each could be used in a similar fashion to identify sections of the image that can be successfully sacrificed.

Other solutions that are tuned to different types of images can also be used successfully. For instance, there are some algorithms designed to convert 24-bit color images into 8-bit color images. These do a good job of identifying 256 colors that represent the image. You can identify the number of free bits at each pixel by comparing the 24-bit value with the entry from the 256-color table that was chosen to replace it. Some of these algorithms are tuned to do a better job on faces. Others work well on natural scenes. Each is applicable in its own way and can do a good job with the system.

If you use JPEG or a similar lossy algorithm to identify the high-noise areas of an image, then you must change one crucial part of the system. When a GIF file is used to hold information, then the recipient doesn't need to have a copy of the original image. The n least significant bits can be stripped away and recovered. They can be used verbatim. If JPEG is going to point out the corners and crevices of the image waiting for more data, then both the sender and the recipient must have access to the same list of corners and crevices. Probably the easiest way to accomplish this is to make sure that both sides have copies of the original image. This is a limitation if you're going to communicate with someone whom you've never met before. You must somehow arrange to get the image to them.

9.5.1 Hiding Information in JPEG Files

There is no doubt that the JPEG's lossy approach to hiding information is a problem that confounds the basic approach to stega-

nography. The compression algorithm is lossy so it will usually destroy any of the subtle changes you might make to embed a message. Derek Upham recognized that it was possible to tweak the JPEG coefficients instead of the individual pixels, an approach, he called *Jsteg*. The JPEG algorithm compresses data in two steps. First, it breaks the image into 8×8 blocks of pixels and fits cosine functions (Figure 14.9) to these pixels to describe them. That is, it finds a weighted average of the functions that adds up to something pretty close to the original block of 8×8 pixels. Then it stores the weights of these cosine functions to serve as a description of this block of pixels. The amount of compression can be increased or decreased by setting more or fewer of the weights to be zero. Upham recognized that you could tweak the least significant bits of the weights to store information.

His solution is coded in C and distributed as a `diff` file that can be added to the standard JPEG version 4 distributed on the Net. His code adds an additional command line feature for UNIX machines that allows you to hide a file as you compress an image. This is a nice approach because it builds on the standard JPEG distribution. Also, it is important because the JPEG image format more common on the Net. It is much more efficient than the GIF format for photographs, although GIF is usually better for graphics when there are a limited number of colors and run-length compression works well.

There are also a number of physiological reasons why this approach may actually generate better effects than tweaking the least significant bits of the data. The programs like S-Tools and Stego jump through many hoops to handle 8-bit color images. They end up with clusters of colors in the color table that are quite similar to each other. This can be accomplished quite well, but it may be easy to detect by someone scanning the color tables.

Tweaking the frequencies of the discrete cosine transform that models the 8×8 block of data has a different effect. Although these tweaks can harm the quality of the final image, it is hard to distinguish their effects. After all, the discrete transformation is already an approximation and it is harder to notice changes in an approximation. In essence, the bits are hidden by controlling whether the JPEG program rounds up or rounds down. Rounding up is a 1 and rounding down is a 0. These numbers can be recovered by looking at the least significant bits of the frequencies.

Upham chose one interesting approach to hiding the information. There must be some header at the beginning of the block of data to tell how many bits are there. Ordinarily, this would be a single number. So the first 32 bits would be devoted to a number that would say that there are, say, 8,523 bytes stored in the least signif-

icant bits that follow. Upham noted that this number would often have a great number of zeros at the front of it. Since these bits would normally be somewhat randomly distributed, a block of zeros could look suspicious. Sixteen zeros in a row should only occur about 1 out of 2^{16} times.

His solution was to have two fields in the header. The first consisted of a 5-bit number that specified the number of bits in the second field. The second field would contain the number of bits in the entire file. This would remove any large blocks of bits at the beginning of the number while leaving the flexibility for extremely large files. He also suggests that the number of bits in the second field be padded with an extra 0 about half of the time. This prevents the sixth bit of the file from always being a 1. This is a very subtle attention to detail.[5]

9.5.2 Outguess

The Outguess software written by Niels Provos tweaks least significant bits like all of the other tools, but it does it carefully to avoid introducing statistical signatures that might alert attackers looking for the presence of the message.

This attention to statistical detail has its costs. The program identifies potential bits and then rules out using half of them in order to have potential corrections. This cuts the capacity of the channel in half but increases the security dramatically.

In the most abstract sense, the Outguess algorithm is straight forward. Every time you change a bit to hide information, you search for an equivalent bit and change it too to maintain a balanced statistical profile. If you change a 0 to a 1, then change a 1 to a 0 at the same time. The Outguess software can be modified to work with any data format given routines for identifying good places to hide data. [Pro01b, Pro01a]

These balanced changes can be used in any of the steganographic solutions. In practice, Provos implements the algorithm by changing the JPEG compression coefficients. Just changing the least significant bits may introduce higher order changes to the JPEG compression coefficients and these changes can be relatively easy to detect. [JJ98a]

[5] John Marsh, a reader, suggests that techniques like *Rice Coding* and *Golumb Coding* are effective here because they do well encoding small integers while handling very large ones too.

9.5.3 F4 and F5

Jsteg was the first generation of programs designed to hide information in the JPEG image format, but attackers soon discovered it left a serious statistical signature. The coefficents in JPEG compression normally fall along a bell curve and the information hiding process distorts this. In most photographs, the most common coefficient is 0 followed by 1 and -1, followed again by 2 and -2, etc. The occurrence of coefficients drops off as the coefficients get larger.

Jsteg ruins this smooth curve when information is hidden. The coefficients of 2 and 3, for instance, are interchanged by the process of tweaking the least significant bit leaving both of them occurring in equal proportions. [JJ98a, JJ98b, WP99, Wes01] The same happens with coefficient pairs like 4 and 5 or -1 and -2. Finding a JPEG file with many pairs of coefficients that occur in equal probability is a sure sign that a message may be hidden with the Jsteg algorithm.

One solution is to change the probability of zeros and ones in the information being hidden. The algorithms from Chapter 6 make it possible to change the statistical probability to mimic any distribution. Let's say that our analysis of JPEG image shows that the coefficient of 3 occurs about 60% as often as 2. A 3 corresponds to a hidden bit of 1 and a 2 corresponds to a hidden bit of 0. If we could arrange for the number of 1s to be hidden about 60% as often as the number of 0s, then the coefficients would balance out.

Here's one basic solution. Create a collection of n bit words and uses these as characters for the algorithm in section 6.2.1. Use the number of zeros and ones in the word to determine the weight. If a zero is assigned weight $.625$ and a one is assigned $.375$, the values will emerge in something approximating a distribution of 1 to $.6$.

For example, let $n = 8$. There are 256 characters in the alphabet. Give 00000000 a weight of $8 \times .625$, 00000001 a weight of $7 \times .625 + .375$, etc. These can be used to build a Huffman tree to change the statistics of the incoming data. If this pre-processing is done, the statistics of the Jsteg algorithm begin to come much closer to real images. This isn't perfect, but it doesn't need to be because the values of the coefficients vary from image to image.

Andreas Westfeld proposes another solution to the statistical gap. His algorithm, F4, encodes the data with more care avoiding the statistical distortions. [Wes01]

Here's the mechanism for hiding a coefficient C:

- If $C = 0$, skip over it and don't hide any information in this coefficient. This decreases the effectiveness of the algorithm, but there's no choice. Jsteg also does this.

- If $C > 0$ and odd leave C unchanged to encode a 1 and decrement it to encode a 0.
- If $C > 0$ and even leave C unchanged to encode a $-$ and decrement it to encode a 1.
- if $C < 0$ and odd, leave C unchanged to hide a 0 and increment it to hide a 1.
- if $C < 0$ and even, leave C unchanged to hide a 1 and increment it to hide a 0.

The only problem occurs when the decrementing or the incrementing produce a new coefficient of zero. In those cases, the bit is repeated because the information is effectively lost. The coding process ignores the coefficients with zero.

Here's a table that shows a slightly different technique for encoding information.

Coefficient	After encoding 0 (skip)	1
2	2	1
3	2	3
4	4	3
-1	-1	0 (skip)
-2	-1	-2
-3	-3	-2
-4	-3	-4

Here's a table showing the bits 01010 being hidden. The five bits require seven coefficients because some are skipped.

Hidden Bit	Coefficient Before	Coefficient After
0	4	4
1	2	1
0 (skip)	0	0
0 (again)	-2	-1
1	1	1
0 (skip)	1	0
0 (again)	2	2

Decoding the file is simple. A coefficient of 1, for instance, is either produced by hiding a 1 in a coefficient of 2 or by hiding a 1 in a coefficient of 1. In either case, the hidden bit is 1. The same pattern holds throughout the encoding process.

This solution avoids the gross statistical problems of Jsteg, but it still changes the profile. Some bits are hidden by shrinking the absolute values of the coefficients. This means that there are more values clustered around 0 than before and the distribution is now

tighter. This is not as glaring because the distribution naturally varies between images, but it is still worth combating.

Westfeld further enhanced the process to minimize the disruption to the file in the next version of the algorithm, F5. He uses a process he calls matrix encoding to spread the information out among more bits. This reduces the density and decreases the amount of distortion.

Imagine you want to store $n = 4$ bits of data. One solution is to pick $2^n = 16$ different locations in the image and only change one of the locations. A change at position $3 = 0011$ would mean you wanted to store the message 0011. The algorithm F5 chooses the best value of n to accommodate the data being stored. The positions are chosen with a cryptographically secure pseudo-random bit stream.

9.6 Summary

Placing information in the noise of digitized images is one of the most popular methods of steganography. The different approaches here guarantee that the data will be hard to find if you're careful about how you use the tools. The biggest problem is making sure that you handle the differences between 24-bit and 8-bit images correctly.

The Disguise The world is filled with noise. There is no reason why some of the great pool of randomness can't be used to hide data. This disguise is often impossible for the average human to notice.

How Secure Is It? These systems are not secure if someone is looking for the information. But many of the systems can produce images that are indistinguishable form the original. If the data is compressed and encrypted before it is hidden, it is impossible to know whether the data is there or not. This can be subverted if a special header is used to identify details about the file. Chapter 17 describes how these simple approaches can be detected.

How to Use the Software There are many different versions of the software available on the Net. The Cypherpunks archive is a good location for the programs. Others circulate throughout the Net.

Chapter 10

Anonymous Remailers

10.1 Dr. Anon to You

Host: On this week's show, we have Anonymous, that one-named wonder who is in the class of artists like Madonna, Micheangelo and the Artist Formerly Known as Prince, who are so big they can live on one name alone. He, or perhaps she, is the author of many of the most incendiary works in the world. We're lucky we could get him or her on the show today, even though he or she would only agree to appear via a blurred video link. [beat] Mr. Anonymous (or should I say Ms. Anonymous?), it's great to have you on the show.

Anon: Make it Dr. Anonymous. That will solve the gender problem. I was just granted an honorary doctorate last June.

Host: Congratulations! That must be quite an honor. Did they choose you because of your writings? It says here on my briefing sheet that you've written numerous warm and romantic novels like the *Federalist Papers*. Great stuff.

Anon: Actually, the *Federalist Papers* weren't a book until they were collected. It really wasn't a romantic set of papers, although it did have a rather idealistic notion of what Congress could be.

Host: Sexual congress. Now that's a euphemism I haven't heard for a while. You're from the old school, right? That's where you got the degree?

Anon: Well...

Host: This explains why you're so hesitant to get publicity, right? It's too flashy.

Anon: No. It's not my style. I prefer to keep my identity secret because some of what I write can have dangerous repercussions.

Host: What a clever scheme! You've got us all eating out of your hand. Every other author would fall over himself to get on this show. We're just happy to be talking with you.

Anon: I was a bit hesitant, but my publisher insisted on it. It was in the contract.

Host: Do you find it hard to be a celebrity in the modern age? Don't you feel the pull to expand your exposure by, say, doing an exercise book with Cher? She's got one name too. You guys would get along great. You could talk about how the clerk at the Motor Vehicles department gives you a hard time because you've left a slot on the form blank.

Anon: Well, that hadn't crossed my mind.

Host: How about a spread in *Architectural Digest* or *InStyle*? They always like to photograph the famous living graciously in large, architecturally challenging homes. Or how about *Lifestyles of the Rich and Famous*? They could show everyone where and, of course, how you live. It's a great way to sell your personality.

Anon: Actually, part of the reason for remaining anonymous is so that no one shows up at your house in the middle of the night.

Host: Oh, yeah, groupies offering themselves. I have that problem.

Anon: Actually to burn the place down and shoot me.

Host: Oh, okay. I can see you doing a book with Martha Stewart on how to give a great masquerade party! You could do some really clever masks and then launch it during Mardi Gras in New Orleans. Have you thought about that?

Anon: No. Maybe after I get done promoting my latest book. It's on your desk there. The one exposing a deep conspiracy that is fleecing the people. Money is diverted from tax accounts into a network of private partnerships where it fills the coffers of the very rich.

Host: What about a talk show? I guess I shouldn't ask for competition. But you could be a really spooky host. You could roam the audience wearing a big black hood and cape. Just like in that Mozart movie. Maybe they could electronically deepen your voice so everyone would be afraid of you when you condemned their shenanigans. Just like in the *Wizard of Oz*.

It would be really hot in those robes under the lights, but I could see you getting a good share of the daytime audience. You would be different.

Anon: My book, though, is really showing the path toward a revolution. It names names. It shows how the money flows. It shows which politicians are part of the network. It shows which media conglomerates turn out cheerful pabulum and "mind candy" to keep everyone somnolescent.

Host: Whoa! Big word there. Speaking of big words, don't you find "Anonymous" to be a bit long? Do you go by "Anon"? Does it make you uncomfortable if I call you "Anon"? Or should I call you "Dr. Anon"?

Anon: Either's fine. I'm not vain.

Host: I should say not. Imagine not putting your name on a book as thick as this one. Speaking of vain, are you into horse racing? I wanted to ask you if you were the person in that Carly Simon song, "You're so vain, you probably think this song is about you." She *never* told anyone who it was about. I thought it might be you. The whole secrecy thing and all.

10.2 Anonymous Remailers

There are many reasons why people would want to write letters or communiqués without attaching their names. Some people search for counseling through anonymous suicide prevention centers. Other people want to inquire about jobs without jeopardizing their own. Then there are the times that try our souls and drive us to write long, half-mad screeds that ring with the truth that the people in power don't want to hear. These are just a few of the reasons to send information anonymously. Even a high government official who is helping to plan the government's approach to cracking down on cryptography and imposing key escrow admitted to me over lunch that he or she has used pay phones from time to time. Just for the anonymity.

Much of what we do, or did in the past, is largely anonymous. Keeping track of who did what when is a waste of time. People only recorded data that made sense and the rest was quickly forgotten providing a cloud of forgiveness that covered the past.

On the Internet, *anonymous remailers* are one solution for letting people communicate anonymously. These mail programs accept incoming mail with a new address and some text for the body of the letter. They strip off the incoming header that contains the real identity of the sender and remail the content to the new address. The re-

cipient knows it came from an anonymous remailer, but they doesn't know who sent it there.

In some cases, the remailer creates a new pseudonym for the outgoing mail. This might be a random string like "an41234". Then it keeps a secret internal log file that matches the real name of the incoming mail with this random name. If the recipient wants to reply to this person, they can send mail back to "an41234" in care of the anonymous remailer who then repackages the letter and sends it on. This allows people to hold a conversation over the wires without knowing each other's identity.

There are many legitimate needs for services like this one. Most of the newspapers that offer personal ads also offer anonymous mailboxes and voicemail boxes so that people can screen their responses. People may be willing to advertise for a new lover or friend if the anonymous holding box at the newspaper gives them a measure of protection. Some people often go through several exchanges of letters before they feel trusting enough to meet the other person. Or they may call anonymously from a pay phone. There are enough nasty stories from the dating world to make this anonymous screening a sad, but very necessary, feature of modern life.[1]

Of course, there are also many controversial ways that anonymous remailers can be used. Someone posted copyrighted documents from the Church of Scientology to the Internet using an anonymous remailer based in Finland [Gro95]. This raised the ire of the church, which was able to get the local police to raid the site and force the owner to reveal the sender's name. Obviously, remailers can be used to send libelous or fake documents, documents under court seal, or other secret information. Tracking down the culprit depends on how well the owner of the remailer can keep a secret.

There are a wide variety of anonymous remailers on the Internet and the collection is growing and shrinking constantly. One current source of a good list can be found at `http://www.chez.com/frogadmin/`. These also include pointers to the software and instructions on how to start up your own remailer.

10.2.1 Enhancements

There are a number of ways that the anonymous remailers can be enhanced with features. Some of the most important ones are:

[1] Strangely enough, in the past people would rely on knowing other people extensively as a defense against this type of betrayal. People in small towns knew everyone and their reputations. This type of knowledge isn't practical in the big city, so complete anonymity is the best defense.

Encryption The remailer has its own public-key pair and accepts the requests in encrypted form. It decrypts them before sending them out. This is an important defense against someone who might be tapping the remailer's incoming and outgoing lines of a remailer.

Latency The remailer will wait to send out the mail in order to confound anyone who is watching the traffic coming in and out. This delay may either be specified by the incoming message or assigned randomly.

Padding Someone watching the traffic in and out of a remailer might be able to trace encrypted messages by comparing the size. Even if the incoming and outgoing messages are encrypted with different keys, they're still the same size. Padding messages with random data can remove this problem.

Reordering The remailer may get the messages in one order, but it doesn't process them in the same first-in-first-out order. This adds an additional measure of secrecy.

Chaining Remailers If one anonymous remailer might cave in and reveal your identity, it is possible to chain together several remailers in order to add additional secrecy. This chain, unlike the physical basis for the metaphor, is as strong as its *strongest* link. Only one machine on the list has to keep a secret to stop the trail.

Anonymous Posters This machine will post the contents to a newsgroup anonymously instead of sending them out via e-mail.

Each of these features can be found in different remailers. Consult the lists of remailers available on the net to determine which features might be available to you.

10.2.2 Using Remailers

There are several different types of anonymous remailers on the network and there are subtle differences between them. Each class was written by different people and they approached the details in their own way. The entire concept isn't too challenging, though, so everyone should be able to figure out how to send information through an anonymous remailer after reading the remailer's instructions.

One of the more popular remailers in history was run by Johan Helsingius in Helsinki, Finland, at `anon@anon.penet.fi` until legal troubles exhausted his patience. Composing e-mail and sending it

through the remailer was simple. You created the letter as you would any other, but you addressed it to `anon@anon.penet.fi`. At the top of the letter, you added two fields, `X-Anon-Password:` and `X-Anon-To:`. The first held a password that you used to control your anonymous identity. The second gave the address to which the message would go. Here's a short sample:

```
Mime-Version: 1.0
Content-Type: text/plain; charset="us-ascii"
Date: Tue, 5 Dec 1995 09:07:07 -0500
To: anon@anon.penet.fi
From: pcw@flyzone.com (Peter Wayner)
Subject: Echo Homo

X-Anon-Password: swordfish
X-Anon-To: pcw@access.digex.net

Le nom de plume de la rose est <<Pink Flamingo.>>
```

When the message arrives in Finland, the remailer strips off the header and assigns an anonymous ID to my address `pcw@flyzone.com`. The real name and the anonymous name are placed in a table and bound with a password. You don't need to use a password, but this adds security. Anyone with a small amount of technical expertise can fake mail so that it arrives looking like it came from someone else. The password prevented anyone from capturing your secret identity. If they did't know your password, then they could't assume your identity.

The password was also necessary for dissolving your identity. If you wanted to remove your name and anonymous identity from the system, then you needed to know the password. This remailer placed a waiting period on cancellation because it did't want people to come in, send something anonymously, and then escape the flames. If you send something, then you should feel the heat, was the philosophy– a philosophy that eventually wore out the welcome.

If you wanted to post anonymously to a newsgroup, you could put the newsgroup's name in the `X-Anon-To:` field like this:

```
Mime-Version: 1.0
Content-Type: text/plain; charset="us-ascii"
Date: Tue, 5 Dec 1995 09:07:07 -0500
To: anon@anon.penet.fi
From: pcw@flyzone.com (Peter Wayner)
Subject: Stupidity

X-Anon-Password: swordfish
X-Anon-To: alt.flames
```

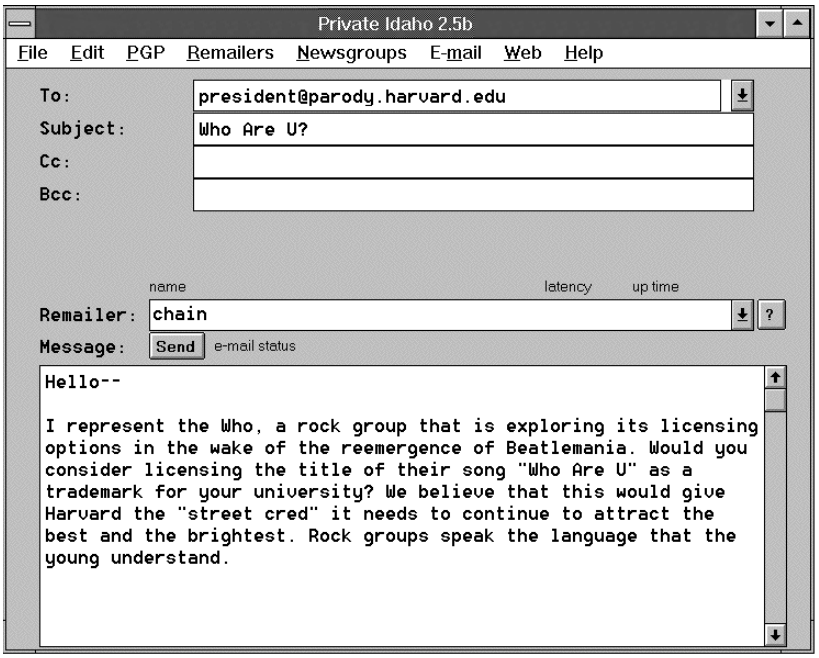


Figure 10.1: A screenshot of the Windows program Private Idaho. You can use it to send encrypted or anonymous mail.

```
In <412A9231243@whitehouse.gov>, Harry Hstar writes:
> Why you're so dumb, I can't believe that someone
> taught you how to type.

You're so stupid, that you probably don't understand
why this is such a great insult to you.
```

This would get posted under your anonymous identity. If someone wanted to respond, they could write back through the remailer. It would protect your identity to some degree.

10.2.3 Using Private Idaho

One of the nicer e-mail packages for the Windows market is Private Idaho, first written by Joel McNamara The original program is still available as freeware, but some of the more advanced development is now bundled as a US$30 product. (`www.itech.net.au/pi/`)

Private Idaho is just a shell for composing the email message and

encrypting it with the necessary steps. The final product can be handed off directly to an SMTP server or to another email package like Eudora. The original product was just a shell for handling many of the chores involved in choosing a path, handling the keys, and encrypting the message. The latest is more of a full-fledged tool.

You can get copies of the original software directly from `www.eskimo.com/~joelm/pi.html` and versions of the latest from `www.itech-.net.au/pi/`.

10.2.4 Web Remailers

A number of web sites offer remailers for reposting information. Adding one level of anonymity is easy for web designers to include and many do. Some of the most popular are now pay services. The Anonymizer (`http://www.anonymizer.com/`) offers tools for both sending anonymous email and browsing the web anonymously. They deliberately keep few log files that might be used to break the veil of secrecy. After the September 11,2001 attacks on the World Trade Center and the Pentagon, the company made news by offering to help anonymous tipsters turn in the terrorists. The site argued that if the terrorists were ruthless enough to kill 5,000, they would not hesitate to track down and kill anyone who turned them in.

Many people use the free email services like Hotmail, Popmail, Excite, or Netscape as pseudononymous drop boxes. These services may work well for basic situations, but they often keep voluminous log files that can reveal the identity of user. Some, like Microsoft's Hotmail, are pushing new services such as the Passport in an effort to assign fixed identities to people.

10.3 Remailer Guts

Designing the inside of a remailer is fairly easy. Most UNIX mail systems will take incoming mail and pass it along to a program that will do the necessary decoding. Repackaging it is just a matter of rearranging the headers and re-encrypting the information. This process can be accomplished with some simple scripts or blocks of C code. Moving this to any platform is also easy.

Designing better, smarter remailer systems is more of a challenge. Here are some of the standard attacks that people might use to try to follow messages through a web of remailers:

In and Out Tracking The attacker watches as messages go in and out of the remailer and matches them up by either order or size. The defense against this is to keep n messages in an internal queue and dispense them in random order. The messages are either kept all the same size or randomly padded at each iteration.

Remailer Flooding Imagine that one remailer receives a letter and the attacker wants to know where it is going. The remailer keeps n letters in its queue and dispenses them randomly. The attacker can send n messages to the remailer just before the message in question arrives. The attacker knows the destination of her own n messages, so she can pick out the one message different from the flow. If the messages are sent out randomly, then the attacker must send another n messages to ensure that subsequent messages won't confuse her.

One defense against this approach is remailer broadcasting. Instead of sending each subsequent message to a particular remailer using one-to-one mail delivery, the remailer would broadcast it to a group of other remailers. Only one remailer would have the right key to decrypt the next address. The others would simply discard it.

Replay Attack An attacker grabs a copy of the message as it goes by. Then it resends it later. Eventually the letter will make its way through the chain of remailers until it arrives at the same destination as before. If the attacker keeps track of all of the mail going to all of the destinations and replays the message several times, then only one consistent recipient will emerge. This is the destination.

The best solution is to require that each message contain an individual ID number that is randomly generated by the sender. The remailer stores this ID in a large file. If it encounters another message with the same ID, then it discards the message. The size of this ID should be large enough to ensure that two IDs will almost certainly not match if they're chosen at random.

Forged Mail Attack It is relatively easy to fake mail sent to an SMTP. Someone could pretend to be you when they sent the anonymous message containing something illegal. If the police were willing to pressure the remailer operator into revealing names, then you could be fingered for something you didn't do.

The passwords used by many remailers are a good defense against this problem. The anonymous remailer won't send

along mail that is supposed to be from a certain site unless the correct password is included. A more sophisticated system would require that the mail be signed with the correct digital signature.

Each of these solutions came from a paper by David Chaum [Cha81] that describes a process called a *mix*. The details of this paper were used as the architecture for the most sophisticated type of remailer currently operating on the Net. Lance Cottrell wrote Mixmaster, a UNIX-based program that will send anonymous mail packages using the more robust structure described in the paper.

The State may, and does, punish fraud directly. But it cannot seek to punish fraud indirectly by indiscriminately outlawing a category of speech, based on its content, with no necessary relationship to the danger sought to be prevented.
—From the majority opinion by Justice Stevens in Joseph McIntyre v. Ohio Election Committee

The main difference is in the structure of the address information. The first class of remailers packaged their data up in nesting envelopes. Each remailer along the chain would open up an envelope and do the right thing with the contents. Mixmaster maintains a separate set of addressing blocks. Each travels through the entire chain of remailers. It is more like a distribution list that offices often use to route magazines through a list of different recipients. Each recipient crosses off its name after it receives it.

There are two advantages to arranging the contents of the messages in this form. The first is that there is no natural reason for the size of the messages to shrink. If the outer envelopes are merely stripped off, then the size of the letter will shrink. This can be compensated by adding padding, but getting the padding to be the right size may be complicated because of the different block sizes of ciphers like DES. The second advantage is reduced encryption time. The block of the encryption does not have to be encrypted or decrypted for each stage of the remailer chain. Only the address blocks need to be manipulated.

Imagine that a message will take five hops. Then the header for a Mixmaster will contain a table that looks something like this if all of the encryption was removed:

Remailer's Entry	Next Destination	Packet ID	Key
Bob	Ray	92394129	12030124
Ray	Lorraine	15125152	61261621
Lorraine	Carol	77782893	93432212
Carol	Gilda	12343324	41242219
Gilda	Final Location	91999201	93929441

The encryption was removed to show how the process works. This header specifies that the mail should go from the remailer run

by Bob, to Ray to Lorraine to Carol to Gilda before heading to its final destination. The Packet ID is used by each remailer to defend against replay attacks.

There are two types of encryption used in Mixmaster. First, each entry in the header is encrypted with the public key of the remailer. So the Next Destination, the Packet ID, and the Key for Ray are encrypted with Ray's public key. Only the rightful recipient of each remailer will be able to decode its entry.

The second encryption uses the keys stored in the table. The best way to understand it is to visualize what each remailer does. Here are the steps:

1. Decodes its packet using its secret key. This reveals the next destination, the ID, and the Key.
2. Uses its Key to decrypt every entry underneath it. Mixmaster uses triple DES to encode the messages.
3. Moves itself to the bottom of the list and replaces the remailer name, the destination information, and the ID with a random block of data. This obscures the trail.

I was the shadow of the waxwing slain by the false azure of the window pane.
—John Shade in Pale Fire

If this is going to be repeated successfully by each remailer in the list, then the initial table is going to have to be encrypted correctly. Each entry in the header will need to be encrypted by the key of each of the headers above it. For instance, the entry for Carol should look something like this:

$$E_{12030124}(E_{61261621}(E_{93432212}(PK_{Carol}(\ldots))).$$

Bob's remailer will strip off the first level of encryption indicated by the function $E_{12030124}$, Ray's will strip off the second and Lorraine's will strip off the third. The final block left is encrypted by Carol's public key.

When the header finally arrives at the last destination, each block will have been re-encrypted in reverse order. This forms something like the signature chain of a certified letter. Each step must be completed in order and each step can only be completed by someone holding the matching secret key. The final recipient can keep this header and check to see that it was processed correctly.

The last key in the chain, in this case the one in the entry for Gilda, is the one that was used to encrypt the message. There is no reason for the remailer to decrypt the message at each step.

Mixmaster currently appends a block of 20 header entries to the top of each entry. Each block takes 512 bytes. If the letter is only going

through five remailers, for instance, then the others are filled with random noise. Each entry in the table contains a bit that identifies it as a "final" hop. If that bit is set, then the key is used to decrypt the main block.

The main block of each message is also kept the same size. If a current message is too short, then padding is added until it is 20k long. If it is too long, then it is broken into 20k blocks. This size is flexible, but it should be set to a constant for all messages. This prevents anyone from identifying the messages from their size or the change of size.

Mixmaster software and much more information can currently be found at `obscura.com`.

10.3.1 Other Remailer Packages

One of the nicest, feature-rich programs for UNIX-based machines is Mailcrypt, written in emacs-lisp for use with the popular GNU Emacs program distributed by the GNU project. The software, created by Patrick LoPresti, will handle all of the basic encryption jobs for mail including encrypting outgoing mail, decrypting incoming mail, and evaluating signatures. The software interacts with the major UNIX mail reading programs like MH-E, VM, and Rmail.

The software also includes a good implementation that will create chains of remailers. When you choose this option, it will automatically create a nested packet of encrypted envelopes that will be understood by the remailers on the list maintained by Raph Levien.

You can create lists of possible remailer chains for future use. These can either be hard coded lists or they can be flexible. You can specify, for instance, that Mailcrypt should choose a different random ordering of four remailers everytime it sends something along the chain. You could also request that Mailcrypt use the four most reliable remailers according to the list maintained by Raph Levien. This gives you plenty of flexibility in guiding the information. To get Mailcrypt, go to `http://cag-www.lcs.mit.edu/mailcrypt/`.

Mailcrypt also makes it easy to use pseudonyms very easily. You can create a PGP key pair for a secret identity and then publicize it. Then if you want to assume a name like Silence Dogood, you could send off your messages through a chain of remailers. The final readers would be able to verify that the message came from the one and only original Silence Dogood because they would be able to retrieve the right public key and check the signature. Some people might try and imitate him or her, but they would not own the corresponding secret key so they couldn't issue anything under this pseudonym.

Another program developed just for chaining together the necessary information for remailers is Premail written by Raph Levien. The software is designed as a replacement for Sendmail, the UNIX software that handles much of the low-level SMTP. Premail can take all of the same parameters that modify its behavior including an additional set of commands that will invoke chains of remailers. So you can drop it in place of Sendmail any place you choose.

Premail has several major options. If you just include the line `key: user_id` in the header with the recipient's user_id, then Premail will look up the key in the PGP files and encrypt the file using this public key on the way out the door. If you include the header line `Chain: Bob; Ray; Lorraine`, then Premail will arrange it so that the mail will head out through Bob, Ray, and Lorraine's anonymous remailers before it goes to the final destination. You can also specify an anonymous return address if you like by adding the `Anon-From:` field to the header. Premail is very flexible because it will randomly select a chain of remailers from the list of currently operating remailers. Just specify the number of hops in a header field like this: `Chain:3`. Premail will find the best remailers from Raph Levien's list of remailers.

10.3.2 Splitting Paths

The current collection of remailers is fairly simple. A message is sent out one path. At each step along the line, the remailers strip off the incoming sender's name and add a new anonymous name. Return mail can follow this path back because the anonymous remailer will replace the anonymous name with the name of the original sender.

This approach still leaves a path—albeit one that is as strong as its strongest link. But someone can certainly find a way to discover the original sender if they're able to compromise every remailer along the chain. All you need to know is the last name in the chain, which is the first one in the return chain.

A better solution is to use two paths. The outgoing mail can be delivered along one path that doesn't keep track of the mail moving along its path. The return mail comes back along a path specified by the original sender. For instance, the original message might go through the remailer `anon@norecords.com` which keeps no records of who sends information through it. The recipient could send return mail by using the return address in the encrypted letter. This might be `my-alias@freds.remailer.com`. Only someone who could decode the message could know to attack `my-alias@freds.remailer.com` to follow the chain back to the sender.

The approach defends against someone who has access to the header which often gives the anonymous return address. Now, this information can be encoded in the body. The plan is still vulnerable because someone who knows the return address `my-alias@freds.remailer.com` might be able to coerce Fred into revealing your name.

A different solution is to split up the return address into a secret. When you opened an account at `freds.remailer.com`, you could give your return address as R_1. This wouldn't be a working return address, it would just be one half of a secret that would reveal your return address. The other half, R_2, would be sent along to your friends in the encrypted body of the letter. If they wanted to respond, they would include R_2 in the header of their return letter. Then, `freds-.remailer.com` could combine R_1 and R_2 to reveal the true return address.

The sender's half of the return address can arrive at the anonymous drop box at any time. The sender might have it waiting there so the letter can be rerouted as soon as possible or the sender might send it along three days later to recover the mail that happened to be waiting there.

This split secret can be created in a number of different ways. The simplest technique is to use the XOR addition described in Chapter 4. This is fast to implement, and perfectly secure. The only practical difficulty will be converting this into suitable ASCII text. email addresses are usually letters and some punctuation. Instead of creating a full 8-bit mask to be XORed with the address, it is probably easier to think of offsets in the list of characters. You could come up with a list of the 60-something characters used in all email addresses and call this string, C. Splitting an email address would consist of doing the following steps on a character-by-character basis:

1. Choose a new character from C. Store this in R_1. Let x be its position in C.

2. To encode a character from the email address, find the character's position in C and move x characters down x. If you get to the end, start again.

3. Store this character in R_2.

The reverse process is easy to figure out. This will produce a character-only split of the email address into two halves, R_1 and R_2. R_1 is deposited at an anonymous remailer and attached to some pseudonym. R_2 is sent to anyone whom you want to respond to you.

They must include R_2 in their letter so the remailer can assemble the return address for you.

An even more sophisticated approach can use the digital signature of the recipient. The initiator of the conversation could deposit three things at the return remailer: the pseudonym, one half of the return address, R_1, and the public key of the person who might be responding. When that person responds, they must send $f_e(R_2)$. This is the other half of the secret encoded with the private key. The remailer has the corresponding public key so it can recover R_2 and send the message on its way.

The systems can be made increasingly baroque. A remailer might want to protect itself against people banging down its door asking for the person who writes under a pseudonym. This can be accomplished by encrypting the remailer's files with the public keys of the recipient. This is better explained by example. Imagine that Bob wants to start up an anonymous communication channel with Ray through `freds.remailer.com`. Normally, `freds.remailer.com` would store Bob's return address, call it B, and match it with Bob's pseudonym, `maskT-AvEnGrr`. Naturally, someone could discover B by checking these files.

`freds.remailer.com` can protect itself by creating a session key, k_i, and encrypting it with Ray's public key, $f_{ray}(k_i)$. This value is sent along to Ray with the message. Then it uses k_i to encrypt B using some algorithm like triple DES before discarding k_i. Now, only Ray holds the private key that can recover k_i and thus B. `freds.remailer.com` is off the hook. It couldn't reveal B even if it wanted to.

This solution, unfortunately, can only handle one particular ongoing communication. It would be possible to create different session keys for each person to whom Bob sends mail. This increases the possibility that B could be discovered by the remailer who keeps a copy of B the next time that mail for `maskT-AvEnGrr` comes through with a session key attached.

10.4 Anonymous Networks

Anonymous remailers move single packets. Some will fetch webpages anonymously. It should come as no surprise that ambitious programmers extended these ideas to provide seamless tools for routing all packets to the Internet. They have essentially created TCP/IP proxies that encrypt all data leaving the computer and then bounce it through a network of servers that eventually kick it out into the Internet at large.

Each of these systems offer a great deal of anonymity against many attackers. The packets from all of the users form a cloud that effectively obscures the begining and the end of each path. None of the solutions, however, are perfect against omniscient and omnipotent attackers who can monitor all of the nodes in the network while probing it with their own packets. Each of the systems has some definite strengths but a few weaknesses that may be exploited in extreme cases.

10.4.1 Freedom Network

Zero Knowledge Systems designed and built the Freedom Network, a collection of servers joined by a sophisticated protocol for encrypting packets. The network lasted until 2001 when the company shut it down for financial reasons. The network remains one of the most ambitious tools for providing privacy on the Internet.

The Freedom Network drew heavily on the inspiration of the Onion Routing Network developed at the Naval Research Labs by Paul Syverson, Michael Reed and David Goldschlag. [SRG00, STRL00, SGR97, RSG98] See Section 10.7.

The network consisted of a collection of *Anonymous Internet Proxies* that would decrypt and encrypt messages while forwarding the data on to other proxies. If a computer wants to establish a path to the Internet, it takes these steps:

1. At the center of the network is the NISS or the Network Information Status Server, a central computer that maintains a list of operating AIPs and their public keys.

2. The computer takes a list of these machines and chooses a random path through a collection of machines. This may use information about distance and load to optimize the process. Shorter chains offer better service while longer chains offer more resistance to detection. Chains running through different countries may offer some extra legal protection.

3. The computer uses Diffie-Hellman key exchange to negotiate a key with each AIP in the chain.

4. The data going out the chain is encrypted with each key in turn. If f_k is the encryption function using key k, then $f_{k_1}(f_{k_2}(\ldots f_{k_n}(data)))$ is sent down the chain. k_i is the key for the $i-th$ AIP in the chain.

5. Each AIP receives its packet of data and uses the negotiated session key to strip away the top layer before passing it on.

6. The last AIP in the chain sends the packet off to the right destination.

7. The return data follows the same chain in reverse. Each AIP uses the session key to encrypt the data.

8. The computer strips away the n layers.

Zero Knowledge refers to this process as *telescope encryption*. The actual process is more involved and sophisticated. Providing adequate performance while doing so much encryption is not an easy trick.

10.4.2 PipeNet

PipeNet is another anonymous network created by Wei Dai. It also rests on a network of computers that route encrypted packets. The principle difference lies in the synchronized mechanism for coordinating the flow of the packets. At each clock step, all of the computers in the network receive a packet, perform the necessary encryption, and then pass it on. If a packet does not arrive, one is not sent.

This solution prevents an omniscient attacker from watching the flow of all of the packets in the hope to figuring out who is communicating with whom. In the Freedom network, a heavy user may inadvertantly give away their path by shipping a large amount of data along it. The omniscient attacker may not be able to break the encryption, but jus counting the size of the packets could reveal the destination. Ideally, a large user base would provide enough cover.

The PipeNet's strict process for sending information ensures that each link between machines only carries the same amount of information at each step. The data moves along the chain in a strictly choreographed process like soldiers marching across the square.

This process, however, has its own weaknesses. If one packet is destroyed or one node in the network locks up, the entire chain shuts down. If data doesn't arrive, it can't go out. [BMS01]

10.4.3 Crowds

The Crowds tool developed by Michael Reiter and Aviel D. Rubin offers a good mechanism for webbrowsing that provides some of the same anonymity as the Freedom Network or PipeNet, but without as much security. It's simplicity, however, makes it easy to implement and run. [RR98]

The protocol is very simple. Each computer in the network accepts a URL request for a document on the web and it makes a random choice to either satisfy the request or pass it along to another randomly selected user. If you want to see a document, your request

may pass through a number of different people before finally being fetched from the right server and passed back through the chain.

This process offers a high degree of anonymity, but not one that can begin to fool an omniscient attacker watching all sites. The simplicty offers a strong amount of confusion. Your machine may receive a request from Alice, but there's no way to know if Alice is actually interested in the information itself. Her machine may just be passing along the request from someone else who might be passing it along from someone else etc. Each individual in the chain can only know that *someone* out there is interested in the information, but they can't be certain who that person is.

10.4.4 Freenet

One of the most ambitious and successful anonymous publication systems is Freenet, a peer-to-peer network originally designed by Ian Clarke. The project itself is highly evolved and open source distributions of the code are available from `freenet.sourceforge.net`. [CSWH00, Cla99]

The system distributes information across a random collection of servers donating their spare diskspace to people seeking to publish documents. The network has no central server that might be compromised so all searches for information fan out across the network. Each machine remembers a certain amount about previous searches so it can answer requests for popular documents.

Each document is known within the network by three different keys which are really 160-bit numbers created by applying the SHA hash function. If you want to retrieve a document, you ask for it with one of the three key numbers. The search process is somewhat random and unorganized, but also resistant to damage to the network. Here are the steps:

1. You start a search by specifying the key value and a "time to live" number which limits the depth of the nodes you want to search.

2. You choose one node in the network to begin the search.

3. This node checks to see if the key matches any files stored locally. If there's a match, the node returns the file.

4. If there's no local match, the node checks a cache of recent searches. If the key is found there, the node retrieves the document. In some cases, this document is already stored locally. In

others, the node must return to the original source to retrieve it.

5. If there's no match, the server asks another in a chain and the process repeats. At each step, the "time to live" counter is reduced by one. When it reaches zero, the search fails.

The caching of this depth-first search speeds up the retrieval of the most popular documents.

The keys assigned to each document are generated in three different ways. The author begins by assigning a title to the document, T. This string of characters is converted into a *keyword-signed key*. This value is hashed by computing $SHA(T)$ and then used to both encrypt and sign the document. Using $SHA(T)$ to encrypt the document ensures that only someone who knows T can read a file. The individual servers can hold any number of files each encrypted by the hash of their titles, but only the person who knows the title can read them. This provides a certain amount of deniability to the server owner who never really understands the material on their hard disk.

The hash of the title is also used as the seed to a pseudo-randomly driven public/private key generation routine. While most public keys are chosen with true random sources, this algorithm uses the hash of T to ensure that everyone can generate the same key pair if they know T. This public key is then used to sign the document providing some assurance that the title and the document match.

This mechanism is far from perfect. Anyone can think up the same title and an attacker may deliberately choose the same title for a replacement document. Freenet fights this by creating a *signed subspace key* connected to the author posting the document. The creation of this key is a bit more involved:

1. First the author publishes a public key bound to their identity.
2. The public key and the title are hashed independently.
3. The results are XOR'ed together and hashed again: $SHA(SHA(T) \oplus SHA(public\,key))$.
4. The private key associated with the public key is used to sign the file.
5. The file is published with both the signed subspace key and the signature.

Retrieving the file now requires knowing both T and the public key of the author. Only the author, however, knows the private key so only the author can generate the right signature.

A third key, the *content-hash key*, is created by hashing the entire document. The author can decide which keys to include with the document when publishing it.

Obviously maintaining some central index of keys, documents and the servers holding them would make life easier for everyone including those who seek to censor the system. Freenet avoids this process, but does take one step to make the searching process easier. When new documents are inserted into the network, the author places them on servers which hold similar keys. The storing procedure searches the network looking at similar keys and then places the document there.

10.4.5 OceanStore

One of the more ambitious projects for persistent, robust, distributed storage is OceanStore developed by a large group of faculty and students at the University of California at Berkeley. Many of the basic ideas and the flavor of the project are clearly inspired by the Freenet project, but embellished with more sophisticated tools for upgrading and duplicating documents. [KBC+00]

The most significant addition is a mechanism for ensuring that documents aren't destroyed when they are updated. Blocks of new data can be inserted into documents and these changes propagate through the network until all copies are current. The mechanism also contains a more sophisticated routing structure to speed the identification and location of documents. All of these details are beyond the current scope of the book.

10.5 Long term storage

Anonymous remailers normally do their work in a few split seconds, deliver their message to the right person, and then move on. What if we want a send a message that lasts? What if we want to post a message where it will be available for a long time? You could always buy a server, pay for a connection and keep paying the bills but that may not work. Controversial material can be shut down with many different types of legal challenges and most server farms aren't willing to spend too much time or energy against determined opponents.

This technique is becoming increasingly controversial because the algorithms are used to store and exchange copyrighted music and video files.

10.5.1 Eternity Server

One solution is to split up the document into many pieces and store these piece on many computers, a suggestion that Ross Anderson explored in his design for an *Eternity server.*[And96a] The design outlines several of the tricks that might allow a network of n machines to store m files and allow the file's owner to both pay the cost of storage and recover the files when necessary.

In the system, each file, F_i, is given a name, N_i, and stored on a set of servers, $\{S_j, S_k, \ldots\}$ with these steps:

1. A general key, key, is chosen.
2. The key for encrypting the data for server S_j is computed from $h(key + name(S_j))$ where $name(S_j)$ is some function that generates a name for a server like the DNS system.
3. A unique name for each file is computed with $h(N_i, name(S_j))$. The data is stored under this unique name on S_j.
4. A random amount of padding is added to F_i in a way that can be easily removed. It might consist of appending the true length of the file to the beginning of the file and then adding extra random information to the end.
5. The file is encrypted for S_j with this key and sent to the server.

This stores a separate copy on the set of servers. Each copy is encrypted with a different key and each copy has a different length.

Another option is to split each file up into smaller, standard sizes, a technique that eliminates the need for padding while introducing more complexity. It is also possible to add the secret sharing technique from Chapter 4 to add the requirement that any k parts of the file must be recovered before the file can be found. This would eliminate some of the need for encryption.

Anderson imagines paying the server owners, a process that can involve increasingly complex amounts of anonymous payment. He imagines that each server might submit a bill every so often to some central payment office. This could be audited to prevent an errant server from claiming to be storing a file without actually doing so. One technique would be for the central office to send the server a challenge nonce, c, and ask the server to compute a keyed hash, $h(c, F_i)$ to prove that they know the data in F_i. This audit would require the central auditing office have a copy of F_i and the key used to scramble it, key. If the auditing passes, the payment office would send the right money to the server.

Searching for the file could be accomplished by broadcasting N_i to all of the servers, asking them to compute $h(N_i + name(S_j))$, and see if they have the file. This might help someone recover a file after some time or allow a central payment mechanism to figure out who to pay for storage.

10.5.2 Entanglement

One technique for preventing the destruction of some documents is to tie their existence to other documents so that one can't be deleted without others being destroyed in the process. Some say that the future of the documents becomes *entangled*.

James Aspnes, Joan Feigenbaum, Aleksandr Yampolskiy, and Sheng Zhong consider some of the theoretical aspects of All or Nothing Entanglement in [AFYZ04].

Here's a simple algorithm for mixing the fate that is based on some of the simplest secret sharing algorithms from Chapter 4. Given a file, F, break it into n parts $F_1+F_2+\ldots+F_n$ so that F can only be reconstructed by someone who holds all n pieces. (Let $+$ stand for exclusive-or.) Ordinarily, $n-1$ pieces would be built with a random number source. In this case, let's choose parts of other documents for these pieces.

That is, let $P_1, P_2, P_3, \ldots$ be a sequence of parts to documents that acts as the storage for the entangled documents. To store F, use these steps:

1. Choose a set of $n-1$ parts from the set of m parts of files previously locked away in storage, $P_1, P_2, P_3, \ldots P_m$. Use some random number generator driven by the file name. This might mean using $h(key+name(F))$ as a seed where key is an optional key and $name(F)$ is some name given to the file. Call these $P_{r1}, P_{r2}, P_{r3}\ldots$ where $r_1, r_2, \ldots$ represents the choices made by the random number generator driven by $h(key + name(F))$.
2. Compute $P_{m+1} = F + P_{r1} + P_{r2} + \ldots + P_{rn-1}$.
3. Create a table entry that links P_{m+1} with $name(F)$.

This solution makes it impossible to delete a file without endangering some of the files that were stored after it. There are still some weaknesses from this approach.The table linking P_{m+1} with $name(F)$ lets someone know exactly which block to delete to destroy the file. It won't be possible to tell which other files are endangered unless there's no key used to compute the sequence of random blocks chosen for each file, $r_1, r_2, \ldots$.

One possible solution is to use a hash table[2] with holes for k

[2]In this case, the word *hash* is used as a data structure designer might use it, not a cryptographer.

blocks $P_1 \ldots P_k$. In the beginning, all of the blocks are initialized to be empty. Then a random collection of blocks are filled with random numbers so that there will be random data to be used for the $n-1$ blocks used to store a file.

A keyed random number sequence, $r_1, r_2, \ldots$, is still used to select the parts, but it is modified to take into account the fact that many of the blocks may hold no data. In this case, the $n-1$ parts used in the secret sharing algorithm will be the first $n-1$ *full* parts identified by the random number sequence. That is, P_{r1} and P_{r2} might be unfilled yet, but P_{r3} and P_{r4} have random data available either from another file or from the initial seeding of random information. So we use those two parts. The final part, $F + P_{r1} + P_{r2} + \ldots + P_{rn-1}$, is not stored at P_{m+1}, but at the first *empty* part identified by the random number stream, in this case, P_{r1}.

This technique eliminates the need for any table linking P_{m+1} with the name for the file, but it adds other complexity. It is possible to identify the empty and full blocks when the file is being stored but this will change as more blocks are added to the mixture. One solution is to add some additional information to the final part that includes the name of the file, perhaps encrypted with a key, and the locations of the $n-1$ part needed to assemble the final part. The file could then be recovered by computing the random number stream, $r_1, r_2, \ldots$, and testing each part identified by the stream until the final part is found. The name attached to the final part will make it possible to identify it and the locations of the $n-1$ parts stored with that name will make it possible to recover the entire file.

An approach like this is still vulnerable to some simple denial-of-service attacks like storing a large number of garbage files. This will make it much more likely that deleting one file will not damage anything of value.

10.6 Publius

Another popular tool for storing documents securely is the *Publius* system designed by Marc Waldman, Aviel D. Rubin and Lorrie Faith Cranor.

The system uses secret sharing algorithms to split up a document between n different servers so that any m pieces are sufficient to recover it. The tool uses a protocol for recovering the document that rests on top of HTTP making it possible for it to work fairly innocuously with standard webservers. The solution is notable because it offers the publisher and only the publisher an opportunity to delete and update the documents after they're distributed.

Publius is named after the pseudonym chosen by Alexander Hamilton, James Madison, and John Jay. [Pub88] They chose the pseudonym from the name of Publius Valerius Publicola, a prominent Roman consul known for helping establishing a republic.

A server, S_i, holds a document, F under a name, $N_{i,j}$ by storing the following:

1. A share of a symmetric cipher key, key, call it key_i. This is constructed by using a secret sharing algorithm to split the key into n parts so that any m are sufficient to recover it.

2. The result of encrypting F with some strong symmetric cipher, $ENC(F, key)$. (The published paper stores the same encrypted version on all servers, but it might make sense to add another layer of encryption so that the encrypted version is different on all servers.)

3. A name by which the part will be known. In this case, $name_i = h(F, k_i)$. The original paper shortens the result of h by splitting the result of the hash function in half and XORing the two parts together.

4. A password that will give the creator the ability to delete and update the block of data stored here. In this case, the password itself is not stored, but the hash of the password concatenated with the server's name: $h(S_i, password)$.

Storing the file involves encrypting it at least once with a random key and splitting the key up between the servers. Recovering it involves retrieving at least m shares of the key and the copies of the encrypted file before decrypting them.

In the original version, the n servers are chosen from the names themselves: $name_i \bmod servers$ where $servers$ is a static number of servers in the system. If by some coincidence, the various values of $name_i$ only choose a small subsection of the set of servers, a different key is chosen until the parts will be distributed successfully.

This process makes it possible to create short URLs produced by concatenating the values of $name_i$ into one manageable block of text. Here's one of the original URLs from the paper:

```
http://!anon!/AH2LyMOBWJrDw=GTEaS2GlNNE=NIBsZlvUQ-
P4=sVfdKF7o/kl=EfUTWGQU7LX=OCk7tkhWTUe=GzWiJyio75b=-
QUiNhQWyUW2=fZAX/MJnq67=y4enf3cLK/0=
```

The values of $name_i$ are BASE64 encoded and separated by equals signs. Any user recovering a document would choose a subset of m names and compute $name_i mymod\ servers$ to identify where the parts might be stored before requesting them.

The password allows the creator to change the stored data by proving knowledge of the password, perhaps by sending

$$h(nonce, h(S_i, password))$$

where $nonce$ is a random string concatenated with the current time. When the server validates the result by duplicating the calculation, the server can either delete the file entirely or just add an update.

The original paper suggests using a new URL to store any updated content, effectively replacing an old file with a completely new version with a new name and new URL. Any request for an old URL would receive a copy of the new URL. The client would compare the new URLs delivered by the m servers and, if the new URLs match, request the new URL. Another technique is to add a list of changes to the original file, a newtermdiff that adds the changes. This could also be encrypted by key and stored along side. The client package would be responsible for integrating these changes after recovering the original file and the diff.

The original Publius system also embedded a flag in the URL which would effectively say, "This URL can't be updated." If this flag appeared in the URL, the client would refuse to follow any redirections offered by the servers. This could prevent any redirection by a man in the middle. Publishing the diffs alongside would not be affected to some system wide redirection.

10.7 Onion Routing

One of the most successful anonymous routing tools is the *Onion Routing* protocol built by by Paul Syverson, David Goldschlag, Michael Reed, Roger Dingledine and Nick Mathewson. The protocol has been revised and extended since its introduction in 1996 and it is implemented by a rich collection of tools including a version of the Firefox browser.

The system can protect a user from an eavesdropper tracking their browsing by sending the requests through a series of randomly chosen proxy servers. Each link in the path is encrypted with a different layer of encryption a process that gives the protocol its name. Each proxy server along the chain will strip off its layer of encryption until the last proxy server will spit out an unencrypted packet of data to the eventual source. All of the proxy servers along the path can confound an eavesdropper by making it impossible to know which data is coming and which is going. Anyone watching a user may see only encrypted data leaving the machine and encrypted data return-

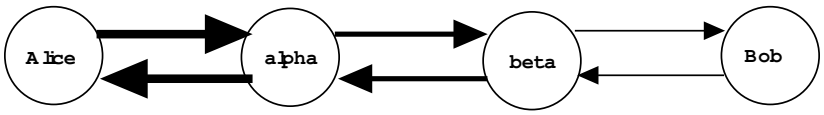


Figure 10.2: A simple circuit established between Alice's and Bob's computers might pass through two proxies, $alpha$ and $beta$. This circuit will be hidden by the confusion of the network described in Figure 10.3.

ing effectively making it impossible to track where the packets go and who sends back a response.

The security of the system depends heavily on the confusion provided by the network of proxy machines– a network that is said to include at least 1000 servers at some times. If an eavesdropper can't watch all of the machines, then the eavesdropper may not be able to track where requests go. Encrypted requests enter the cloud on the edges and emerge somewhere else as cleartext data going to the real destination. Matching the clients with the cleartext requests that eventually emerge is not an easy task. [DS03, Ser07]

A powerful eavesdropper, though, can often figure out information from watching carefully. An omniscient eavesdropper can often match clients with the cleartext packets that leave the cloud of proxy servers by timing them. If an encrypted packet enters from Alice's machine at 10:30, 10:32, and 10:35 and some cleartext packets of the same general size leave a distant proxy at 10:31, 10:33, and 10:36, then there is a good chance that those were Alice's packets. The quality of this analysis depends heavily on the performance of the network and the number of other users. A fast network that doesn't introduce very large delays will also make the matching process more precise.

The system is also vulnerable to other accidental leaks of data. If a proxy server shuts down for some reason then it will break all of the paths that use it. All of the clients that used it will need to renegotiate new paths, effectively identifying some of the past traffic. This won't link up people immediately, but it can be revealing if it is repeated several times.

There is also one inherent limitations to the protocol that is often forgotten: onion routing only protects information to the last proxy in the chain. After that, the last proxy will communicate in the clear with the final destination for the data packet. Some suggest that the people who volunteer to operate the last proxies in the chain, the edge servers, may be doing so to peek at all of the data flowing past.

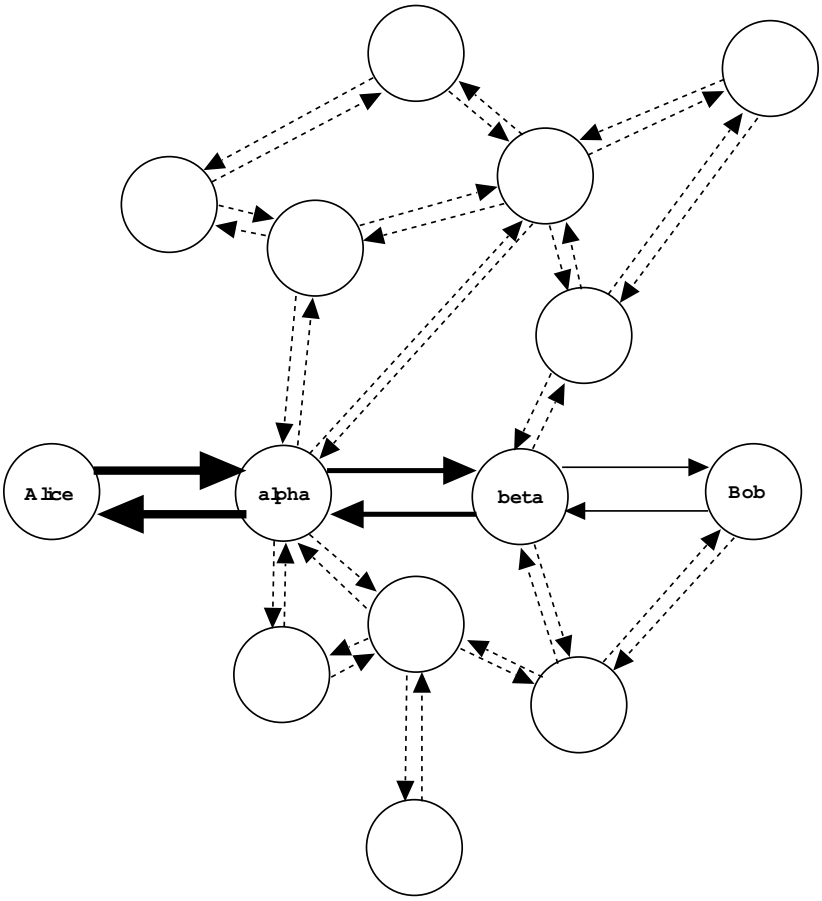


Figure 10.3: The circuit between Alice and Bob from Figure 10.2 will be hidden by the traffic between all of the other circuits flowing through the network of proxies. In this case, S_{alpha} is the *entry node* and S_{beta} is the *exit node*. There will probably be other servers in between in an actual example, but they are elided here.

This won't tell them who initiated the data packet, but it will give them a view of the data itself and this can be quite useful. This problem can be reduced by using SSL to negotiate a separate encrypted channel between the sender and the final destination for the data.

10.7.1 Establishing a Circuit

The Onion Routing protocol builds its circuits up step by step by negotiating with each step along the circuit. Here are the steps involved in building the chain illustrated in Figures 10.2 and 10.3:

1. Alice decides she wants to send some packets of data to Bob using the Onion Routing network.
2. She chooses server S_{alpha} at random from a list of servers that accept incoming circuits. This is often called the *entry node*.
3. Alice and S_{alpha} negotiate a key. The latest version of the Onion Routing software uses ElGamal key exchange with established Diffie-Hellman keys because it has proven to be faster than the RSA algorithms used in the original version. Call this $key_{alice,alpha}$. [ØS07a]
4. Alice now has a secure path between her computer and S_{alpha}. She can extend this by choosing another server at random from the network, S_{beta}.
5. Alice does not communicate with S_{beta} directly; she uses S_{alpha} as her proxy. S_{beta} doesn't even know that Alice exists because all of $Sbeta$'s communications are with S_alpha. Alice negotiates a key by sending her half of the key establishment protocols through her encrypted tunnel with S_{alpha} who sends them on to S_{beta} on her behalf. Let's call the result of this negotiation: $keyalice, beta$.
6. After Alice completes the key negotiation process with S_{beta}, she checks the signature on $key_{alice,beta}$ provided by S_{beta}. This prevents S_{alpha} from cheating and pretending to open up a new circuit for S_{beta} or just setting up a fake man-in-the-middle attack.
7. If the circuit was longer, Alice would repeat this negotiation phase with a number of servers in the circuit.
8. Alice now has a circuit constructed with S_{beta}, her proxy for the general Internet also called the *exit node*. If she sends out a

packet, it will be S_{beta} that delivers it to the final destination and it will be S_{beta} that will accept any communication for her as her proxy.

This description leaves out a number of details of how the negotiation is accomplished but they can be found in the original papers. [ØS07a, DMS04, SRG00, STRL00, SGR97, RSG98]

After building these keys, Alice now has a way to communicate with Bob. A round trip might look like this:

1. Alice encrypts the packet of data, M, with $key_{alice,beta}$ producing $E_{key_{alice,beta}}(M)$.
2. Alice encrypts this encrypted packet again with $key_{alice,alpha}$ producing: $E_{key_{alice,alpha}}(E_{key_{alice,beta}}(M))$
3. Alice sends this to S_{alpha}.
4. S_{alpha} strips away the outer layer to get $E_{key_{alice,beta}}(M)$ and sends this to S_{beta}.
5. S_{beta} strips away the inner layer and uncovers M. This packet of data could be any low-level TCP packet like an HTTP web page request.
6. S_{beta} sends off M and gets back any response, call it R.
7. S_{beta} encrypts the response with the key shared with Alice producing: $E_{key_{alice,beta}}(R)$ and pass this on to S_{alpha}.
8. S_{beta} encrypts the response with the key shared with Alice producing: $E_{key_{alice,alpha}}(E_{key_{alice,beta}}(R))$ and pass this on to Alice.
9. Alice strips away both of the layers of encryption to get R.

This process is often called *telescoping*, a reference to the old collapsible spyglasses built from nesting tubes.

10.7.2 More Indirection: Hidden Services

The basic Onion Routing system hides the identity of one end of a conversation, call it the client, from the other by passing the bits through a cloud of proxies. There's no reason why the system can't also hide the destination, the so-called server, from the client if the proxy servers in the middle of the cloud can be trusted to keep some secrets about the destination of the information. This is a reasonable

assumption if the number of proxies is large and the eavesdropping capabilities of any adversary are small.

The latest versions of the system can also offer *hidden servers* through the cooperation of special proxy nodes acting as *rendezvous points*, *directory servers*, introduction points, and *valet servers*. All of these live on top of the normal network of entry nodes and exit nodes, often with the same machine offering multiple services. The directory server contains a list of hidden servers out on the network and the introduction points that will act as their proxies and hide their existence. When a connection is established, the rendezvous points will act as a midway point, hiding the identies of the client and the now hidden server from each other.

Here is the rather complicated first structure for hiding hidden servers behind multiple layers of introduction as illustrated by Figure 10.4 [ØS07b].:

1. When a hidden server wants to join the onion network, it looks around for an introduction point. When it finds a trustable server that will do the job, it sets up a secure link and waits.

2. The hidden server tells the *directory server* about the introduction point. This directory server can assign a special name much like the DNS system used for normal domains. In practice, the domains with the suffix `.onion` are used to indicate these addresses.

3. When Alice wants to find the hidden server, she sends the `.onion` address to the directory server which sends her to the *introduction point*.

4. Alice shops around for a *rendezvous point* to act as the mid point for the communications. (This is marked as "meet" in Figure 10.4 to save space.) Note that the rendezvous point will not ordinarily know who Alice may be because the communication is hidden by the chain of proxies in the circuit that Alice established with the rendezvous point. So Alice will need to identify her connection with some sort of pseudonym. Note, this could be a weak point if Alice is able to control the rendezvous point too.

5. Alice begins a key negotiation with the hidden server and forwards this information with the location of the rendezvous point to the introduction point.

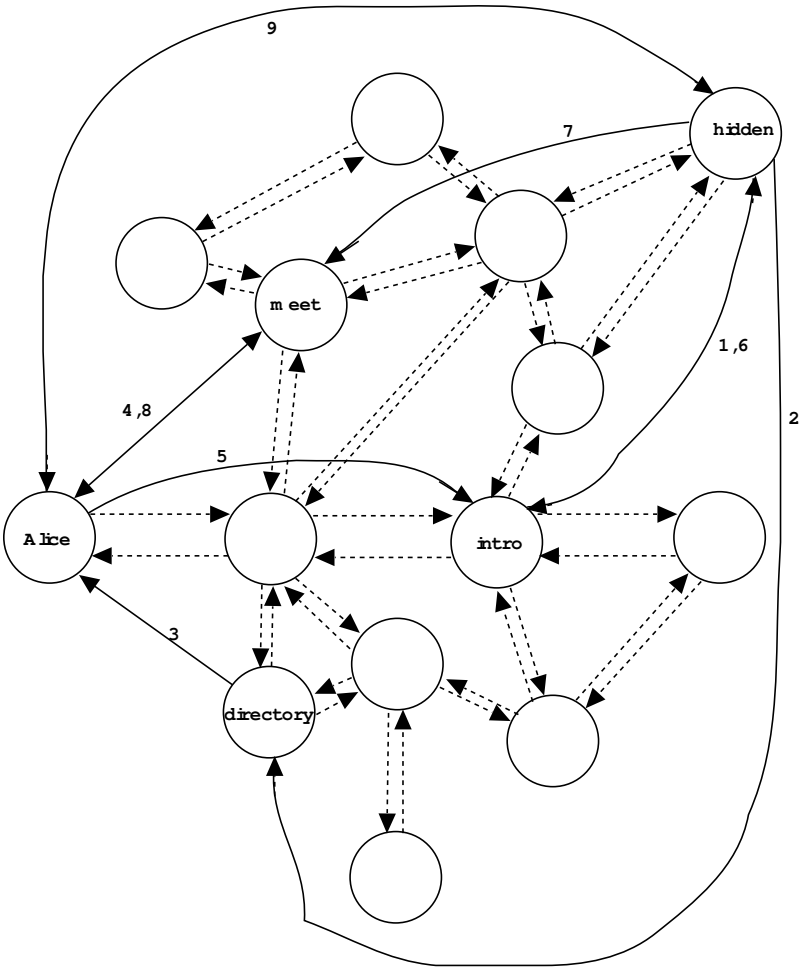


Figure 10.4: The circuit between a client (Alice) and a hidden server (perhaps Bob) will work through an introductory server (marked “intro”), a directory server and a rendezvous server (marked “meet”). The dashed lines aren’t directly part of the protocol; they just indicate that the normal onion routing encryption is the basis for the interaction. [ØS07b]

6. The introduction point passes this information on to the hidden server who then decides whether to complete the key exchange and meet at the rendezvous point.

7. Assuming the hidden server likes Alice's request, it will set up its own tunnel to the rendezvous point and tell the rendezvous point that it wants to communicate with Alice. Well, it won't know Alice by name. It will only know Alice's pseudonym.

8. The rendezvous point will notify Alice that the hidden server wants to talk too.

9. Alice completes the key negotation with the hidden service and she now has a secure link to the hidden server without knowing who the hidden server happens to be.

This entire exchange is a bit like the kind of negotiation that teenagers use in courting when asking their friends to find out if so-and-so likes me. If the entire onion network is trustworthy, the link is trustworthy too. The rendezvous point doesn't know the identity of either the client (Alice) or the server because the layers of proxies in between hide this information. Also, the introductory server and the directory server won't know the hidden server because the layers of proxies hid the sender during the initiation. The directory server just knows that there's some hidden server out there with a name and an identity.

Still, this has limitations. Imagine someone decides to operate a hidden server `good-stuff.onion`. Once the name becomes known and passed around among people, there's no easy way for the original owner of `good-stuff.onion` to prevent someone else from setting up shop and broadcasting the name to directory servers. How will the directory server know which is the rightful owner of the name? If the battles over domain services are brutal when privacy isn't involved, they're close to impossible when it's not possible to identify people.

One solution is to tie the identity to some public key not a domain name like `good-stuff.onion`. Only the person who created the original key pair should be able to lay claim to this public key later. The only problem is that this key is not as easy to remember as the simple domain name `good-stuff.onion`.

This problem can be reduced but not eliminated if the directory servers have a long memory and maintain long, stable connections with the onion routing network. They can link the domain name with the public key when the entry is created and as long as they honor this link, the network will be able to find one and only one

hidden server that is the original owner of `good-stuff.onion`. Only the owner of a matching key can re-establish the link.

The usual problems of timing attacks are also possible in this world. If a user can control how a rendezvous point re-broadcasts information to a hidden server, a sophisticated client might be able to identify a hidden server by noting when the data leaves the rendezvous point and when it arrives. If the data packets leave and arrive in a pattern that stands out from the noise, it would be possible to find a hidden server in much the same way that an eavesdropper can identify both sides of a communication.

If hidden servers don't introduce enough layers of indirection and proxied communication, then you might also look to *valet nodes*. These replace the *directory servers* with a cloud of flexible, irregular servers that know the right *introduction points*. After a hidden server negotiates a connection with an introduction point, the introduction point turns around and finds some valet nodes on its own. The hidden server doesn't contact a directory server and it doesn't broadcast its own information. This reduces the weakness posed by a directory server.

How does the client find a valet node? The information for connecting with the valet node and the various public keys for negotiating a tunnel with the introduction point are bundled together and circulated, perhaps out of the network.

10.7.3 Stopping Bad Users

Bad users of the onion routing network can ruin the reputation of other users. The Wikipedia, for instance, often blocks TOR exit nodes complete because some people have used the network to hide their identities while defacing the wiki's entries. Is it possible to build up anonymous reputations for users that follow them from visit to visit, effectively banning the bad user? There are no simple solutions because there's little way to establish that a new user to the system isn't someone who acted poorly in the past. But it is possible to put some hurdles in the way by giving returning users some additional powers if they present some form of anonymous credentials.

One straight-forward solution is to use some form of certificates signed with a blind signature, a technique that borrows from some of the early solutions for building anonymous digital cash. [SSG97, DMS03] When you register, you get an anonymous coin to be spent redeeming services in the future. If you behave well, you can get a new coin when you turn in the old one.

Section 12.5.1 uses blind signatures for zero knowledge proofs.

For the sake of simplicity, let the coin be some random number,

m, chosen by the newly arriving user who wants to be anonymous in the future. When registering for the service, the user doesn't present m per se, just m after being modified by some *blinding factor*, a value that is chosen differently depending on the digital signature algorithm that will be used to certify the coin. In the case of an RSA signature, this would be another random value b encrypted by the public key. The signature will reverse this encryption allowing the blinding factor to be stripped out later:

1. Alice wants to register for anonymous access to the server– a process through which the server might ask for a real identity. This happens outside the TOR cloud.

2. Alice downloads the server's public key. In the case of RSA, this would be a modulus n and an exponent e.

3. Alice chooses a random blinding factor b and a random serial number for the coin, m, computes $b^e m \bmod n$ and sends this to the server while registering.

4. The server responds by signing this value. In the case of RSA, that means computing $(b^e m \bmod n)^d \bmod n = b^{de} m^d \bmod n = bm^d \bmod n$ where d is the corresponding private exponent. The server returns this to Alice.

5. Alice strips out the blinding factor. In the case of RSA, this means multiplying by $b^{-1} \bmod n$. This produces $m^d \bmod n$, a valid digital signature on m that was produced without revealing m to the server.

Alice is free to use this anonymous token at any time by submitting both m and $m^d \bmod n$ at any time. (The server might impose time limits by changing the values of the public and private keys, d, e, and n, from time to time.) If Alice behaves well, she can get another anonymous token by working through the same algorithm again with a new serial number, m'. The server would keep track of the spent serial numbers to prevent reuse.

There are limitations to this approach too. The server must make a decision about Alice's behavior *before* giving her a new coin for another visit. If Alice's misbehavior comes to light after this coin is generated, well, there's nothing that can be done. The chain of abuse will continue.

A more robust algorithm called *Nymble* proposes a way of constructing the coins so they can be linked together. It adds, in essence, a trap door to the blinding mechanism that is minded by a separate

reputation server. If bad behavior comes to light, the service can approach the reputation server and ask it to spring open the trap door. This will allow the server to ban the bad person when they return. [JKTS07]

The field of digital cash is a rich world of mathematical techniques for settling debts anonymously. All of the algorithms are much more sophisticated and secure than the simple one presented here. See [CFN93, Way95b, Way97a] and many others.

The actual algorithm is too complicated to describe in this space, but it is based on a technique known as *hash chains*, a sequence of values produced by repeatedly hashing a number. That is, m_0 is chosen at random and then $m_i = h(m_{i-1}$. This technique might be combined with the anonymous coin by presenting m_i with the anonymous coin on the first visit. The server does not need to renew the service at each visit by providing another anonymous coin because it can use the hash chain to keep track of the good users.

On the next trip, Alice presents m_{i-1} to the server which verifies it by looking at to see that $h(m_{i-1}) = m_i$. If it does, the server discards m_i and replaces it with m_{i-1} on the list of hash chains corresponding to good anonymous users. If problems emerge later, the server just shuts down a particular chain. Bad behavior is banned again.

10.8 Anonymous Auction Protocols

In anonymous and semi-anonymous networks, it is often difficult to conduct traditional business when the traditional solution requires some form of identification. Auctions, for instance, usually demand that all of the interested parties appear before each other so everyone can watch the action. Real estate auctions are held on the steps of the court house for just this reason. When people want to remain anonymous during an auction, they usually need to send some proxy or rely on the auction house to keep their identity a secret.

Ross Anderson and Frank Stajano showed how the Diffie-Hellman key exchange algorithm could be extended to offer anonymous bidding over a network. [SA00] Their auction takes place continuously and requires the various bidders to chime in with their bids at the right time, much like a traditional auction held in a hall. The identities, though, are hidden from each other and even from the seller too. When a winner is finally chosen, the seller and the buyer can quickly establish an encrypted communication channel to negotiate when and how to trade the goods for cash. If the goods are digital, the buyer and seller don't even need to reveal their identities to each other.

In the original paper, the i^{th} round of the auction begins at time it where t is the amount of time between rounds. For the sake of simplicity, let the price be $f(i)$. Since this is a Diffie-Hellman-based

protocol, the group must also choose some prime p and a generator g for the field.

If buyer j wants to pay $f(i)$, they choose a random number, $x_{i,j}$, and announce $g^{x_{i,j}} \mod p$ at time it. The seller chooses one of the submitted values, perhaps at random or perhaps by choosing the first to arrive if it can be determined fairly. This becomes the winning bid at time i.

The seller announces the chosen value and broadcasts it. If the seller wants to be clever, the seller can also choose a random value, y_i, construct a key from $g^{y_i} \mod p$ and $g^{x_{i,j}y_i} \mod p$, and encrypt a message. Each of the bidders can try to read the message but only the bidder who wins round i will succeed.

This continues until there's only one winning bidder left standing. The bidder who wins round i will usually drop out of round $i+1$ to avoid being duped into continuing bidding because there's no easy way to for an anonymous bidder to see when there are no other bidders. When no bids arrive for a round, the seller and the buyer can establish a key and begin any negotiations to exchange the goods. If physical items and real cash are part of the transaction, this will usually involve stripping away the anonymity from each other. But the losing bidders don't find out any information.

Anderson and Stajano point out, often humorously, that there are many ways for bidders to ensure that the auction is honest. While the bidders are kept anonymous, they can claim ownership of a bid by revealing a value of $x_{i,j}$.

It is possible to extend the protocol to remove the requirement to hold the bidding in rounds by allowing each bidder to include their maximum bid. The bidder can also sign this with their private key, $x_{i,j}$ and $g^{x_{i,j}} \mod p$. If this can be reliably distributed to all bidders, perhaps through a system like the Dining Cryptographers' network (see Chapter 11), then the seller can choose a winning bid through a dutch auction.

10.9 The Future

In the short-term future, every machine on the Internet will be a first-class citizen that will be able to send and receive mail. The best solution for active remailers is to create tools that will turn each SMTP port into an anonymous remailer. To some extent, they already do this. They take the incoming mail messages and pass them along to their final destination. It would be neat, for instance, to create a plug-in MIME module for Eudora or another email program that

would recognize the MIME type "X-Anon-To:" and resend the mail immediately.

To a large extent, these tools are not the most important step. The tools are only useful if the remailer owner is willing to resist calls to reveal the hidden identity.

There is also a great need for anonymous dating services on the Net. Although many of the remailers are clothed in the cyberpunk regalia, there is no doubt that there are many legitimate needs for remailers. An upscale, mainstream remailer could do plenty of business and help people in need of pseudonymous communication.

10.10 Summary

The Disguise The path between sender and recipient is hidden from the recipient by having an intermediate machine remove the return address. More sophisticated systems can try to obscure the connection to anyone who is watching the mail messages entering and leaving the remailing computer.

How Secure Is It? Basic anonymous remailers are only as secure as the strongest link along the chain of remailers. If the person who runs the remailer chooses to log the message traffic, then that person can break the anonymity. This may be compelled by the law enforcement community through warrants or subpoenas.

The more sophisticated remailers that try to obscure traffic analysis can be quite secure. Anyone watching the network of remailers can only make high-level statements about the flow of information in and out of the network. Still, it may be quite possible to track the flow. The systems do not offer the unconditional security of the dining cryptographers networks described in Chapter 11.

Digital Mixes must also be constructed correctly. You cannot use RSA to sign the message itself. You must sign a hash of the message. [PP90] shows how to exploit the weakness.

How to Use the Software The Cypherpunks archive offers all of the software necessary to use chaining remailers or Mixmaster. The WWW pages are the easiest options available to most people.

Further Reading

- There are a number of different products available now from companies offering various levels of anonymous and pseudonymous accounts. Some include MuteMail (`mute-mail.com`), HushMail (`hushmail.com`), Guardster (`guard-ster.com`), Safe Mail, (`safe-mail.net`), and Anonymizer (`anonymizer.com`). These operations are often secure, but they have been known to turn over subscriber information and, in some cases, decryption keys in response to subpoenas. [And07] HushMail, for instance, says that it will not release information without a subpoena from the Supreme Court of British Columbia, Canada, a significant legal hurdle but one that can be met.
- George Danezis, Roger Dingledine and Nick Mathewson have an anonymous remailer, Mixminion, that uses Mix-like routing to obscure the content and path of the messages. The latest version is available from `mixminion.net`. [DDM03]
- MorphMix is a tool developed by Marc Rennhard and Bernhard Plattner that asks each user's node to actively contribute to the mix, a choice that helps force the network infrastructure to scale while also requiring a mechanism for ranking the reputation of the users.[RP04, TB06]
- Salsa, a tool from Arjun Nambiar and Matthew Wright, selects the nodes in a path in a random way in order to prevent an attacker from infiltrating the circuits. This forces the attacker to control a significant proportion of the nodes in order to have a good chance of intercepting a message. [NW06]

Chapter 11

Secret Broadcasts

11.1 Table Talk

Chris: I heard that Bobby got fired?
Leslie: Fired?
Pat: I heard he was let go because of a drop in sales.
Chris: Yes, the sales he's supposed to be making.
Leslie: But everyone's sales are down.
Pat: He said he was having a good quarter.
Chris: Good? His sales are down even more.
Leslie: Down more than what?
Pat: Maybe they're just down relative to last year, which was a good year for everyone.
Chris: I think they're down relative to everyone else'.
Leslie: Maybe it was something else. I heard he was drinking too much.
Pat: I heard because he couldn't take his boss's stupidity.
Chris: Actually, his boss is brilliant. Bobby's the problem.
Leslie: This doesn't add up.
Pat: Well, it does add up. Maybe the truth is somewhere in between everything we've said. You just need to add it together.

11.2 Secret Senders

How can you broadcast a message so everyone can read it but no one can know where it is coming from? Radio broadcasts can easily be located with simple directional antenna. Anonymous remailers

(see Chapter 10) can cut off the path back to its source, but they can often be compromised or traced. Practically any message on the Net can be traced because packets always flow from one place to another. This is generally completely impractical, but it is still possible.

None of these methods offers unconditional security, but there is one class of algorithms created by David Chaum that will make it impossible for anyone to detect the source of a message. He titled the system the "dining cryptographers" which is a reference to a famous problem in computer system design known as the "dining philosophers." In the Dining Philosophers problem, n philosophers sit around the table with n chopsticks set up so there is one chopstick between each pair. To eat, a philosopher must grab both chopsticks. If there is no agreement and no schedule, then no one will eat at all.

Chaum phrased the problem as a question of principle. Three cryptographers are eating dinner and is from the National Security Agency. The waiter arrives and tells them that one person at the table has already arranged for the check to be paid, but he wouldn't say who left the cash. The cryptographers struggle with the problem because neither of the two nongovernment employees want to accept even an anonymous gratuity from the NSA. But, because they respect the need for anonymity, they arrange to solve the problem with a coin-tossing algorithm. When it is done, no one will know who paid the check, but they'll know if the payer is from the NSA.

This framing story is a bit strained, but it serves the purpose. In the abstract, one member will send a 1-bit message to the rest of the table. Everyone will be able to get the same message, but no one will be able to identify which person at the table sent it. There are many other situations that seem to lend themselves to the same problem. For instance, a father might return home to find the rear window smashed. He suspects that it was one of the three kids, but it could have been a burglar. He realizes that none will admit to doing it. Before calling the police and reporting a robbery, he uses the same dining cryptographer protocol so one of the kids can admit to breaking the window without volunteering for punishment.[1]

If a 1-bit message can be sent this way, then there is no reason why long messages cannot come through the same channel. One problem is that no one knows when someone else is about to speak, since no one knows who is talking. The best solution is to never interrupt someone else. When a free slot of time appears, participants should wait a random amount of time before beginning. When they start broadcasting something, they should watch for corrupted mes-

Random protocols for sharing a communication channel are used by the Ethernet developed at Xerox PARC.

[1] This may be progressive parenting, but I do not recommend that you try this at home. Don't let your children learn to lie this well.

sages caused by someone beginning at the same time. If that occurs, they should wait a random amount of time before beginning again.

The system can also be easily extended to create a way for two people to communicate without anyone being able to trace the message. If no one can pinpoint the originator of a message with the Dining Cryptographers protocol, then no one can know who is actually receiving the message. If the sender encrypts the communication with a key that is shared between two members at the table, then only the intended recipient will be able to decode it. The rest at the table will see noise. No one will be able to watch the routing of information to and fro.

The system for the dining cryptographers is easy to understand. In Chaum's initial example, there are three cryptographers. Each one flips a coin and lets the person on his right see the coin. Now, each cryptographer can see two coins, determine whether they're the same or different, and announce this to the rest of the table. If one of the three is trying to send a message—in this case that the NSA paid for dinner—then they swap their answer between same and different. A 1-bit message of "yes" or "on" or "the NSA paid" is being transmitted if the number of "different" responses is odd. If the count is even, then there is no message being sent.

There are only three coins, but they are all being matched with their neighbors so it sounds complex. It may be best to work through the problem with an example. Here's a table of several different outcomes of the coin flips. Each column shows the result of one diner's coin flip and how it matches that of the person on their right. An "H" stands for heads and a "T" stands for tails. Diner #1 is to the right of Diner #2, so Diner #1 compares the coins from columns #1 and #2 and reports whether they match or not in that subcolumn. Diner #2 is to the right of Diner #3 and Diner #3 is to the right of Diner #1.

Diner #1		Diner #2		Diner #3		
Coin	Match	Coin	Match	Coin	Match	Message
H	Y	H	Y	H	Y	none
T	N	H	Y	H	N	none
T	*Y*	H	Y	H	N	*yes*
T	N	H	*N*	H	N	*yes*
T	N	H	N	T	Y	none
T	*Y*	H	N	T	Y	*yes*

In the first case, there are three matches and zero differences. Zero is even so no message is sent. But "no message" could be considered the equivalent of 0 or "off". In the second case, there are two

differences, which is even so no message is sent. In the third case, a message is sent. That is, a "1" or an "on" goes through. The same is true for the fourth and sixth cases.

The examples don't need to be limited to three people. Any number is possible and the system will still work out the same. Each coin flipped is added into the final count twice, once for the owner and once for the neighbor. So the total number of differences will only be odd if one person is changing the answer.

What happens if two people begin to send at once? The protocol fails because the two changes will cancel each other out. The total number of differences will end up even again. If three people try to send at once, then there will be success because there will be an odd number of changes. A user can easily detect if the protocol is failing. You try to broadcast a bit, but the final answer computed by everyone is the absence of a bit. If each person trying to broadcast stops and waits a random number of turns before beginning again, then the odds are that they won't collide again.

Is this system unconditionally secure? Imagine you're one of the people at the table. Everyone is flipping coins and it is clear that there is some message emerging. If you're not sending it, then can you determine who is? Let's say that your coin comes up heads. Here's a table with some possible outcomes:

You		Diner #2		Diner #3	
Coin	Match	Coin	Match	Coin	Match
H	Y	H	N	?	Y
H	Y	*H*	*N*	H	Y
H	Y	H	N	*T*	*Y*
H	Y	H	Y	?	N
H	Y	*H*	*Y*	T	N
H	Y	H	Y	*H*	*N*
H	N	T	Y	?	Y
H	N	*T*	*Y*	H	Y
H	N	T	Y	*T*	*Y*
H	N	T	N	?	N
H	N	*T*	*N*	T	N
H	N	T	N	*H*	*N*

We were never that concerned about Slothrop qua *Slothrop.*
—Thomas Pynchon in Gravity's Rainbow

There are four possible scenarios reported here. In each case, your coin shows heads. You get to look at the coin of Diner #2 to your right. There are an odd number of differences appearing in each case so someone is sending a message. Can you tell who it is?

The first entry for each scenario in the table has a question mark for the flip of the third diner's coin. You don't know what that coin is.

In the first scenario, if that hidden coin is heads then Diner #2 is lying and sending a message. If that hidden coin is tails, then Diner #3 is lying and sending the message. The message sender for each line is shown in *italics*.

As long as you don't know the third coin, you can't determine which of the other two table members is sending the message. If this coin flip is perfectly fair, then you'll never know. The same holds true for anyone outside the system who is eavesdropping. If they don't see the coins themselves, then they can't determine who is sending the message.

There are ways for several members of a dining cryptographers network to destroy the communications. If several people conspire, they can compare notes about adjacent coins and identify senders. If the members of the table announce their information in turn, the members at the end of the list can easily change the message by changing their answer. The last guy to speak, for instance, can always determine what the answer will be. This is why it is a good idea to force people to reveal their answers at the same time.

The Dining Cryptographers system offers everyone the chance to broadcast messages to a group without revealing anyone's identity. It's like sophisticated anonymous remailers that can't be compromised by simply tracing the path of the messages. Unfortunately, there is no easy way to use system available on the Internet. Perhaps this will become more common if the need emerges.

11.3 Creating a DC Net

The Dining Cryptographers (DC) solution is easy to describe because many of the difficulties of implementing the solution on a computer network are left out of the picture. At a table, everyone can reveal their choices simultaneously. It is easy for participants to flip coins and reveal their choices to their neighbors using menus to shield the results. Both of these solutions are not trivial to resolve for a practical implementation.

The first problem is flipping a coin over a computer network. Obviously, one person can flip a coin and lie about it. The simplest solution is to use a one-way hash function like MD5 or Sneferu.

Manuel Blum described how to flip coins over a network in [Blu82]. This is a good way to build up a one-time pad or a key.

The phone book is a good, practical one-way function but it is not too secure. It is easy to convert a name into a telephone number, but it is hard to use the average phone book to convert that number back into a name. The function is not secure because there are other ways around the problem. You could, for instance, simply dial the number and ask the identity of the person who answers. Or you could gain

access to a reverse phone directory or phone CD-ROM that offered the chance to look up a listing by the number, not the name.

The solution to using a one-way function to flip a coin over a distance is simple:

1. You choose x, a random number and send me $h(x)$ where h is a one-way hash function that is easy to compute but practically impossible to invert.

2. I can't figure out x from the $h(x)$ that you sent me. So I just guess whether x is odd or even. This guess is sent back to you.

3. If I guess correctly about whether x is odd or even, then the coin flip will be tails. If I'm wrong, then it is heads. You determine whether it is heads or tails and send x back to me.

4. I compute $h(x)$ to check that you're not lying. You can only cheat if it is easy for you to find two numbers, x , which is odd, and y, which is even, so that $h(x) = h(y)$. No one knows how to do this for good one-way hash functions.

The protocol can be made stronger if I provide the first n bits of x. A precomputed set of x and y can't be used.

This is the algorithm that the two neighbors can use to flip their coins without sitting next to each other at the dinner table. If you find yourself arguing with a friend over which movie to attend, you can use this algorithm with a phone book for the one-way function. Then the flip will be fair.

The second tool must allow for everyone to reveal at the same time whether their coin flips agree or disagree. Chaum's paper suggests allowing people to broadcast their answers simultaneously but on different frequencies. This requires more sophisticated electronics than computer networks currently have. A better solution is to require people to commit to their answers through a *bit commitment* protocol.

The solution is pretty simple. First, the entire group agrees on a stock phrase or a collection of bits. This should be determined as late as possible to prevent someone from trying to use computation in advance to game the system. Call this random set of bits B. To announce their answers, the n participants at the table:

1. Choose n random keys, $\{k_1, \ldots, k_n\}$, in secret.

2. Individually take their answers, put B in front of the answer, and encrypt the string with their secret key. This is $f_{k_i}(Ba_i)$, where f is the encryption function, k_i is the key, and a_i is the answer to be broadcast.

3. Broadcast their encrypted messages to everyone in the group. It doesn't matter what order this happens.

4. When everyone has received the messages of everyone else, everyone begins sending their keys, k_i, out to the group.

5. Everyone decrypts all of the packets, checks to make sure that B is at the beginning of each packet, and finally sums the answers to reveal the message.

These bit commitment protocols make it nearly impossible for someone to cheat. If there were no B stuck at the beginning of the answers that were encrypted, a sophisticated user might be able to find two different keys that reveal different answers. If he wanted to tell the group that he was reporting a match, then he might show one key. If he wanted to reveal the other, then he could send out another key. This might be possible if the encrypted packet was only one bit long. But it would be near impossible if each encrypted packet began with the same bitstring, B. Finding such a pair of keys would be highly unlikely, which is why the bitstring B should be chosen as late as practical.

The combination of these two functions makes it easy to implement a DC network using asynchronous communications. There is no need for people to announce their answers in synchrony. Nor is there any reason for people to be adjacent to each other when they flip the coin.

11.3.1 Cheating DC Nets

There are a wide variety of ways for people to subvert the DC networks, but there are adequate defenses to many of the approaches. If people conspire to work together and reveal their information about bits to others around the table, then there is nothing that can be done to stop tracing. In these situations, anonymous remailers can be more secure because they're as secure as their strongest link.

Another major problem might be jamming. Someone on the network could just broadcast extra messages from time to time and thus disrupt the message of someone who is legitimately broadcasting. If, for instance, a message is emerging from the network, a malicious member of the group could start broadcasting at the same time and destroy the rest of the transmission. Unfortunately, the nature of DC networks means that the identity of this person is hidden.

If social problems become important, then it is possible to reveal who is disrupting the network by getting everyone on the network to reveal their coin flips. When this is done, it is possible to

determine who is broadcasting. Presumably, there would be rules against broadcasting when another person is using the DC network so it would be possible to unwind the chain far enough to reveal who that person might be.

This can be facilitated if everyone sends out a digital signature of the block of coin flips. Each person in the network has access to two keys— theirs and their neighbor's. The best solution is to have someone sign the coin flips of their neighbor. When the process is unwound, this prevents them from lying about their coin flips to protect themselves. Forcing everyone to sign their neighbor's coin flips prevents people from lying about their own coin flips and changing the signature.

This tracing can be useful if only one person uses the DC network for a purpose that offends a majority of the participants—perhaps to send an illegal threat. If one person is trying to jam the communications of another, however, then it reveals both senders. The only way to determine which one is legitimate is to produce some rules for when members can start broadcasting. The first one would be the legitimate sender. The one who began broadcasting afterward would be the jammer.

11.4 Summary

Dining Cryptographers networks offer a good opportunity to provide unconditional security against traffic analysis. No one can detect the broadcaster if the nodes of the network keep their coin flips private. Nor can anyone determine the recipient if the messages are encrypted.

The major limitation to DC nets is the high cost of information traffic. Every member of the network must flip coins with their neighbor and then broadcast this information to the group. This can be done in blocks, but it is still a significant cost. n people mean that network bandwidth increases by a factor of $2n$.

The Disguise DC nets offer an ideal way to obscure the source of a transmission. If this transmission is encrypted, then only the intended recipient should be able to read it.

How Secure Is It? The system is secure if all of the information about the coin flips is kept secret. Otherwise, the group can track down the sender by revealing all of this information.

Further Reading

- Philippe Golle and Ari Juels updated the algorithm making it easier to identify the cheating players without requiring extra rounds. The data for checking the calculations is bundled with each broadcast making it noninteractive. [GJ04]

Chapter 12

Keys

12.1 The Key Vision

A new directive from the National Science Foundation ordered all researchers to stop looking for "keys" that will dramatically unlock the secrets behind their research. The order came swiftly after a new study showed that a "wholistic vision" was a more powerful metaphor than the lock and key. Administrators at the National Science Foundation predicted the new directive would increase discoveries by 47% and produce significant economies of effort.

The recent news of the metaphoric failure of the lock and key image shocked many researchers. Landon P. Murphy, a cancer specialist at Harvard's Women and Children's Hospital, said, "We spent our time searching for one key insight that would open up the field and provide us all of nature's abundant secrets. One key insight can do that for you."

In the future, all researchers will train their minds to search for a "wholistic picture" that encompasses all of their knowledge. An inclusive understanding is thought to yield more discoveries in a faster time period because the grand vision can often see the entire forest not just the trees.

"Let's say you're on top of a mountain. You can see much further than you can at the base– even if you have a key to the tunnel under the mountain." said Bruce Konstantine, the Executive Deputy Administrative Aide at the NSF. "We want our researchers to focus on establishing the grand vision."

Some scientists balked at the new directive and countered with a metaphor of their own. "Sure you can see for miles but you can't see detail." said Martin Grubnik, a virologist at the University of Pitts-

burgh. "Some of us need to focus on the small things and the details to make progress. That's the key."

Konstantine dismissed this objection and suggested that the virologists might make more progress if they avoided a narrow focus.

"The key insight, or perhaps I should say the true vision, is that scientists who focus too narrowly avoid seeing the big picture. We want more big pictures. If that means abandoning the hope for one key, so be it."

12.2 Extending Control

Most of the game of steganography involves finding a set of algorithms that can make one chunk of data look like another. In some instances, camouflaging the existence of the data is not enough. Stronger attacks require stronger measures and one of the most versatile is adding some *key* bits to the algorithm.

The key is some relatively small collection of bits that plays a strong role in the algorithm. If someone doesn't hold the right key, they can't unlock certain features of the algorithm. The bits of information in the key are somehow essential for manipulating the data.

Most of the keying techniques used in steganography are extensions of the solutions used in basic cryptography. Some of the basic types include:

Secret Keys One key is used to hide the information and the same key must be available to uncover the information. This is often called *symmetric* or *private key* steganography. The second term isn't used in this book to avoid confusion with public-key approaches.

Public Mechanisms or Public Keys One key hides the information and a different key uncovers it. Some call these *asymmetric* because the algorithms separate the functions. These solutions are often useful for watermarking information because someone can uncover the information without the power to hide new information— that is, the power to make new copies with the person's watermark.

None of the algorithms described here offer a solution with any of the simplicity of the best public-key encryption systems. They often rely on difficult problems or some obscurity to provide some form of a "private key". In many cases, there is no private key at all. The public key is just used to verify the signature.

Some of the other solutions provide public-key behavior by encrypting and decrypting the hidden information with standard public-key algorithms.

Zero-Knowledge Proofs These systems allow the information hider to secure information so that it's existence can be revealed without revealing the information itself. The technique can also be useful for watermarking because the information hider can prove they hid the information without giving someone else the power to hide the same information. That is, to make copies with the same watermark.

Collision Control Codes Let's say you have several copies of a document. These codes try to prevent you from combining the copies in ways to obscure the information hidden inside. Some basic attacks on watermarking information, for instance, involve averaging several different copies. These solutions resist these attacks.

There are a number of approaches for implementing algorithms that fall into these classes. The simplest is to use basic secret-key or public-key algorithms on the data before it is handled by the steganography algorithm. The encryption algorithm and the hiding algorithm are kept separate and distinct from each other.

Separating the functions is easier to understand and it has one practical advantage: encryption algorithms naturally make data look more random and random data is often the best kind for steganography. Encrypting the data often makes sense even if the secrecy is not necessary because encryption is one of the simplest ways to increase the randomness of the data. Of course, this approach can occasionally produce data that is too random, a problem discussed in depth in Chapter 17.

Keeping the encryption separate from the hiding is more intellectually straight-forward. Good encryption algorithms can be mixed or matched with good steganographic solutions as the conditions dictate. There's no need to make compromises.

But splitting the two processes is also limiting because the hiding algorithm is the same for everyone. Anyone can recover the hidden bits because the algorithm is essentially public. Anyone who uses it to recover bits on one occasion can now use it again and again in other situations. They may not be able to do anything with the bits because the encryption is very strong, but they will be able to find them relatively easily *and* replace them with bits of their own.

Keying the hiding process ensures that only people with the right key can either hide or recover bits. This restricts attackers by adding

an additional layer of complexity to the process.

Many of the basic algorithms in this book use generic keys to control the random choices made by them. If an arbitrary decision needs to be made, then a cryptographically secure random number generator driven by a key is one of the simplest mechanisms for adding a key to the scheme.

The algorithms in Chapter 9 hide information in the least significant bits of image and sound files by selecting a subset of elements. This selection process is driven by a random number generator that repeatedly hashes a key. In Chapter 13, the functions used to compute the sorted list of data elements can include a key. If the same stream of random numbers isn't available, the bits can't be extracted.

More sophisticated systems integrate the key even deeper into the algorithm. Some try to constrain how the answer to some hard problem is constructed. Others try to limit how it is encoded in the data.

Many of these newer advanced systems show how just about any computational processes can be tweaked or distorted to include a few extra bits. Most algorithms include some arbitrary decisions about location, order, or process and which can be driven by some key. In the best cases, the authors understand the problem well enough to provide some actual arguments for believing that the process is hard to decrypt without the key.

12.3 Signing Algorithms

Many of the keying algorithms provide some kind of assurance about the documents authenticity by acting like digital signatures for the document. These solutions are quite useful in all situations where digital signatures on arbitrary files provide some certainty. They're also especially useful for watermarking. The ideal algorithm allows the file's creator to embed a watermark in such a way that only the proper key holders can produce that watermark.

The basic solution involves separating the image or audio file into two parts. The first holds the details that will remain unchanged during the steganography. If the information is hidden in the least significant bits, then this part is the other bits, the most significant ones. The second part is the bits that can be changed to hide information. This set may be defined and hence controlled by a key.

A digital signature on the file can be constructed by hashing the unchangeable part, signing the hash value with a traditional digital signature function, and then encoding this information in the second part reserved for hidden information. The digital signature may

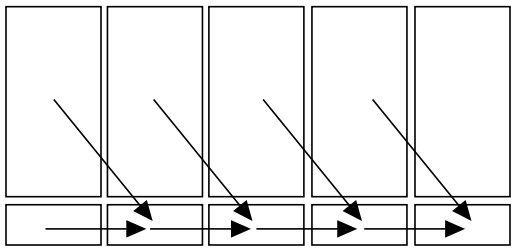


Figure 12.1: An embedded digital signature can be woven into a file. In this visual allegory, the file is broken up into five chunks. Each chunk is broken up into two part– one that remains unchanged and the other that hides information. The data from section i is used to create a digital signature that is embedded in section $i+1$.

be computed with traditional public-key algorithms like RSA, or it may use more basic solutions with secret key algorithms or even hash functions.[Won98, Wal95a]

This process uses two keys that may or may not be drawn from the same set of bits. The first is used to define the parts of the file that may be changed. It could be a random number stream that picks pixels to hide information. The second key actually constructs the signature.

Jessica J. Fridrich and Miroslav Goljan suggest self-embedding a copy of an image in itself. Details from one block are embedded in another block across the image. Cropping or other tampering can be reversed. [FG99]

This approach is direct, and easy to code, but it hides the information in the most succeptable part of the file. Compression algorithms and other watermarks can damage the information by changing the data in this section. [CM97]

The mechanism can also be extended by splitting the file into numerous sections. The signature for section i can be embedded into section $i+1$. Figure 12.1 shows how this might be done with a file broken into five sections. The data from one section is hidden in the next section. Separating the hiding also can provide an indication of where any tampering took place.

12.4 Public-Key Algorithms

Many researchers are trying to develop "public-key" algorithms for hiding information that provide many of the same features as public-key cryptography. The systems allow a user to hide information in such a way that anyone can recover it, but ideally in a way that no one can create new data in the right format.

This feature is desirable for watermarks because it is ideal for

the average user (or the user's computer) to check information for watermarks.

All of the algorithms are new and relatively untested, but they still offer exciting possibilities for watermarking files. If they can be developed to be strong enough to resist attack, people will be able to use them to track royalties and guarantee the authenticity of the files.

12.4.1 Leveraging Public-Key Cryptography

The standard encryption algorithms can be mixed with steganography to offer many of the features of public-key cryptography. Imagine that you want to embed a message into a file so that only the person with the right secret key can read it. [And96c]

Here's a straight-forward example:

1. Choose a secret key, x.
2. Encrypt the plaintext data with this key using the public key of the recipient: $E_{pk}(x)$.
3. Hide this value in a predetermined spot.
4. Use x as a standard key to determine where the other information is hidden.

If the public-key algorithm and the infrastructure are trustworthy, only the correct recipient should be able to decrypt $E_{pk}(x)$ and find x.

12.4.2 Constraining Hard Problems

One strategy is to use problems that are hard to solve but easy to verify as the basis for public-key signature. The knowledge of how to solve the difficult problem acts like a private key and the knowledge of how to verify it acts like the public key. The real challenge is finding problems that behave in the right way.

The class of NP-complete problems includes classic computer science challenges like boolean satisfiability, graph coloring, the travelling salesman problem, and many others. [GJ79] They are often hard to solve but always easy to verify. Unfortunately, it is not always easy to identify a particular problem with the right mixture of strength.

This approach has not been historically successful. An early public-key system created by Ralph Merkle and Martin Hellman used an NP-complete problem known as the knapsack. (Given n items

with weights $\{w_1, ldots, w_n\}$, find a subset that weighs W pounds.) Their algorithm created custom knapsacks that appeared to offer a guarantee that they were hard to pack exactly. Only someone with a secret value, the private key, could determine the right subset of objects to put in the knapsack with ease. After some equally hard work, a human, Adi Shamir found a general algorithm that will break the system without threatening the general NP-complete problems. It exploited the key generation process that tried but failed to find enough complicated sizes for the knapsacks. [Sha82, Lag84, Odl84]

No one else has had luck with NP complete problems. They seem theoretically secure because there's no general algorithms for solving them quickly, there are a number of fast heuristics that find solutions in almost all cases. Indeed, no one has a good mechanism for identifying a small subset of problems that are difficult enough to offer some guarantees.

In one solution, Gang Qu imagines that the ability to create these solutions is what is being protected by a watermark. A person who knows how to solve the travelling salesman problem optimally may market their solutions to airlines, businesses or traveling salesforces. To protect against piracy of their custom itineraries, they could hide some signature information in the file that proves the solutions are theirs. A competing business may develop their own solutions, but anyone can check the provenance by looking for this hidden information.

The technique hides information in the solutions to the difficult problems by forcing certain parameters to take certain values. [Qu01, KQP01] Anyone can examine the solution and check the parameters, but it should be difficult to assemble a new solution because the problem is presumably hard. The approach does not really require a set of bits labeled a "public key", but it still behaves in much the same way.

Information is hidden in the solution to the problem by introducing new constraints on the solution. The classic boolean satisfiability problem, for instance, takes n boolean variables, $\{x_1, \ldots, x_n\}$ and tries to assign true or false values to them so that a set of boolean clauses are all true. Information can be encoded in the answer by forcing some of the variables to take particular values— say $x_{14} = T, x_{25} = F, x_{39} = F, x_{59} = T$ and $x_{77} = T$ might encode the bitstring 10011. Of course, adding this extra requirement may make a solution impossible, but that is a chance the designers take.

The technique can be applied to many NP-complete problems and other problems as well. Many problems have multiple solutions and information can be hidden by choosing the appropriate solution

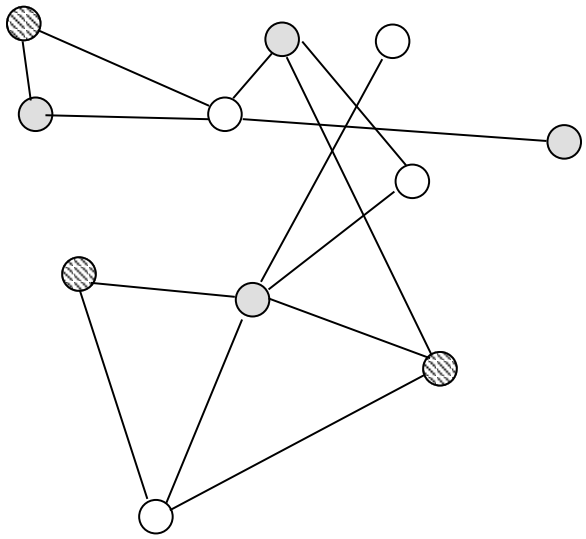


Figure 12.2: A graph colored so that no two adjacent nodes have the same color. Information can be encoded in this solution by forcing certain nodes to certain colors.

that also encodes the information. Figure 12.2 shows a graph colored so that several nodes take fixed colors. One way to encode the information is to grab pairs of nodes and force them to have either the same or different colors. Two nodes can be forced to have the same color by merging them while the algorithm looks for a solution. Two nodes can be kept different by adding an edge that connects them. Other problems usually have some latitude.

The process can be strengthened by hashing steps along the way. Let K_0 be a creator's original watermarking information. It might be their name, it might be an email address, or it could be a reference to some big table of known creators. Let H be a strong hash function like MD5 or SHA. Let $\{C_1, \ldots, C_n\}$ be a set of constraints to the problem that are open for twiddling to hold information. These may be extra variables in a boolean satisfiability problem that can be set to true or false. Or they could be nodes in a graph that might hold different values. For the sake of simplicity, assume that each constraint can encode one bit. The following loop will encode the watermark:

1. Let $K_i = H(K_{i-1}, C_{i-1})$. The hashing function should be fed both the information encoded in C_{i-1} and some data about the structure of C_{i-1} itself. This data might include the node numbers, the variable names, or the boolean formulae. (Set C_0

to be a null string.)

2. Extract one bit from K_i and encode it in C_i.

Repeat this for all n constraints.

The watermark can be tested by repeating the process. Any skeptical inquirer can get K_0 from the creator and step through the process, checking the bits at each point.

This approach can be used to protect complicated engineering designs where the creator must solve a difficult problem. Chip designers, for instance, often solve large NP-complete problems when they choose how to lay out transistors. This technique allows them to encode a watermark in the chip design that essentially says, "Copyright 2001 Bob Chiphead".

The technique can be applied to more general information-hiding problems or watermarking problems, but it has limitations. The creator must have the ability to find solutions to difficult problems– solutions that are difficult for the average person to aquire. The simplest way to accomplish this is to create a large computer that is barely able to find solutions to large, difficult problems. Only the owner of the computer (or one of similar strength) would be able to generate solutions. Ron Rivest and Silvio Micali discuss using a similar solution to mint small tokens, *Peppercoin*. [Riv04]

12.4.3 Using Matrix Multiplication

Recovering information from a file requires finding a way to amplify the information and minimize the camouflaging data. One solution is to rely on the relatively random quality of image or sound information and design a recovery function that strips away relatively random information. Whatever is left could hold the information in question.

Joachim J. Eggers, Jonathan K. Su, and Bernd Girod suggest using a mechanism where matrix multiplication dampens the covering data but leaves distinctive watermarks untouched. [ESG00b, ESG00a] They base their solution on eigenvectors, the particular vectors that are left pointing in the same direction even after matrix multiplication. That is, if M is a matrix and w is an eigenvector, then $Mw = \lambda w$, where λ is a scalar value known as an eigenvalue. Every eigenvector has a corresponding eigenvalue. Most vectors will be transformed by matrix multiplication, but not the eigenvectors. For the sake of simplicity, we assume that the eigenvectors are one unit long— that is, $w^h w = 1$.

Exploiting this property of matrix multiplication requires assuming that the data in the file is relatively random and comes with a mean value of zero. Sound files fit this model, but image files usually fail because they consist of bytes that range between 0 and 255. Any file can be converted into one with a zero mean value by calculating the mean and subtracting it from each entry. The result may not be sufficiently random for this algorithm, but we will assume that it is.

Let x be a vector of data where the watermark will be hidden. We assume that it comes with a zero mean and is sufficiently random. What is sufficiently random? Perhaps it's better to define this by describing the effect we want. Ideally, we want $x^h Mx = 0$ or at least sufficiently close to zero. Then it will drop out of the computation and only the watermark will be left.

Let w be an eigenvector of some matrix, M, and λ be the corresponding eigenvalue. $Mw = \lambda w$. This vector can be used as a watermark and added to the camouflaging data, x, with a weight, β. Ideally, β is chosen so that $x + \beta w$ is perceptually identical to x and the watermark can be extracted.

The watermark is extracted from the data by computing

$$(x + \beta w)^h M(x + \beta w) = x^h Mx + x^h M\beta w + \beta w Mx + \beta^2 wMw.$$

If the assumption about the randomness of x holds, the first three terms will be close to zero leaving us with $\beta^2\lambda(w_w^h) = \beta^2\lambda$.

A public-key system can be established if the values of M, β, and λ are distributed. Anyone can test a file, y, for the presence or absence of the watermark by computing $y^h My$ and determining whether it matches $\beta^2\lambda$.

This approach still has a number of different limitations. First, the number of elements in x and w must be relatively large. Eggers, Su and Girod report results with lengths of about 10,000 and about 100,000. Larger values help guarantee the randomness that pushes $x^h Mx$ to zero.

Second, finding the eigenvectors of M is just as easy for the message creator as any attacker. One solution is to choose an M which has many different eigenvectors, $\{w_1, \ldots, w_n\}$ that all come with the same eigenvalue λ. The attacker may be able to identify all of these eigenvectors, but removing the watermark by subtracting out the different values of w_i, one after another, could be seen as a brute-force attack.

Of course, the ease of finding the eigenvectors means that someone can insert a fake watermark by choosing any eigenvector, w_i, that comes with an eigenvalue of λ. This means that the algorithm can't be used to generate digital signatures like many of the classic pub-

lic key algorithms, but it might still be of use for creating watermarks that prevent copying. A pirate would find little use in adding a signal that indicated that copying should be forbidden.

More sophisticated attacks may be possible. The authors credit Teddy Furon for identifying a trial-and-error approach to removing watermarks with this tool by changing the scale for the part of the signal in the space defined by the eigenvectors, $\{w_1, \ldots, w_n\}$, with length λ.

The algorithm can be tuned by choosing different values of M. The authors particularly like permutation matrices because an $n \times n$ matrix can be stored with $n - 1$ values. "Multiplication" by a permutation matrix is also relatively easy. Using the matrix designed to compute the discrete cosine transform is also a good choice because the computation is done frequently in image and sound manipulation.

This solution is far from perfect and its security is not great. Still, it is a good example of how a function might be designed to minimize the camouflaging data while amplifying the hidden data. The process is also keyed so that the value of M must be present to extract the hidden message.

12.4.4 Removing Parts

Many of the algorithms in this book hide information by making a number of changes to a number of different locations in the file and then averaging these changes to find the signal. The proceding section (12.4.3), for instance, may add hundreds of thousands of small values from the watermark eigenvector into the file. The spread-spectrum-like techniques from Chapter 14 will spread the signal over many different pixels or units from a sound file. The signal is extracted by computing a weighted average over all of them.

One insight due to Frank Hartung and Bernd Girod is that the information extractor does not need to average all of the locations to extract a signal. [HG97] The algorithms already include a certain amount of redundancy to defend against either malicious or accidental modifications to the file. If the algorithms are designed to carry accurate data even in the face of changes to an arbitrary number of elements, why not arrange for the receiver to skip that arbitrary number of elements all together?

Consider this algorithm:

1. Create n "keys", $\{k_1, \ldots, k_n\}$.

2. Use cryptographically secure random number generators to

use these keys to identify n different sets of m elements from the file.

3. Tune the hiding algorithm so it can hide information in mn elements in such a way that the information can be recovered even if $(n-1)m$ elements are not available or damaged.

4. Hide the information.

5. Distribute the n "keys" to the n different people who might have a reason to extract the information. The algorithm will still work because the information was hidden with such redundancy that only one subset is necessary.

Hartung and Girod use the term "public-key" for the n values of $\{k_1, \ldots, k_n\}$, even though they do not behave like many traditional public keys. The keys do offer a good amount of antifraud protection. Anyone possessing one k_i can extract the information, but they cannot embed new information. If the holder of the value of k_i tries to embed a new signal, it will probably be invisible to the holders of the other $n-1$ keys because their keys define different subsets. The holder of k_i can change the elements as much as possible, but this won't change the information extracted by others.

Of course, this approach does have limitations. The values of k_i are not truly public because they can't circulate without restrictions. They also can't be used to encrypt information so that they can only be read by the holder of a corresponding private key. But the results still have some applications in situations where the power of keys needs to be reined in.

12.5 Zero Knowledge Approaches

Zero knowledge proofs are techniques for proving you know some information without revealing the information itself. The notions began evolving in the 1980s as cryptographers and theoretical computer scientists began exploring the way that information could be segregated and revealed at optimal times. In one sense, a zero knowledge proof is an extreme version of a digital signature.

Here's an example of a zero knowledge proof. Let G be a graph with a set of nodes ($\{v_1, v_2, \ldots, v_n\}$) and a set of edges connecting the nodes ($\{(v_i, v_j), \ldots\}$). This graph can be $k-$colored if there exists some way to assign one of k different colors to the n nodes so that no edge joins two nodes with the same color. That is, there does not exist an i and a j such that $f(v_i) = f(v_j)$ and (v_i, v_j) is in the set of

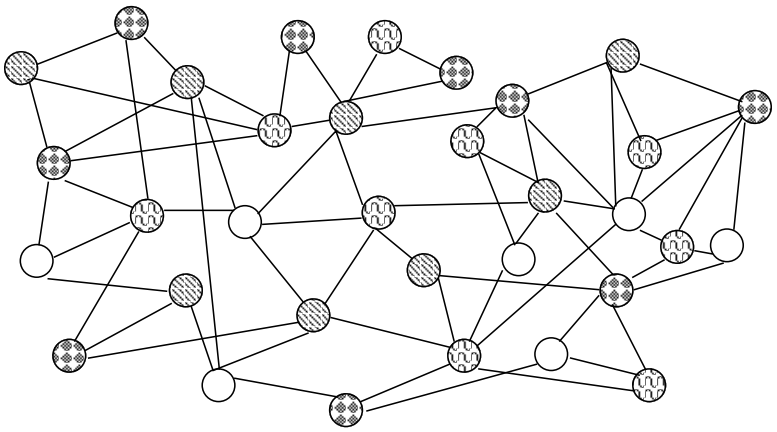


Figure 12.3: A graph where the nodes are assigned one of four colors.

edges. f is the coloring function. Figure 12.3 shows one graph that is 4-colored.

Finding a $k-$coloring of an arbitrary graph can be a complicated and difficult process in some cases. The problem is known to be NP-complete, [GJ79] which means that some instances seem to grow exponentially more difficult as more nodes and edges are added. In practice, a coloring for many graphs can be found relatively quickly. It's often harder to find difficult graphs and this is one of the practical limitations of using zero knowledge proofs in applications. There are a number of details that make it difficult to produce a working and secure implementation of this idea.

Let's say you know a coloring of some graph and you want to prove that you know it without actually revealing it to another person, the skeptical inquirer. Here's an easy way to prove it:

1. Create a random permutation of the coloring, f'. That is, swap the various colors in a random way so that $f'(v_i) \neq f'(v_j)$ for all (v_i, v_j) in the graph.

2. The skeptical inquirer can give you a random bit string, S. Or S might be established from an unimpeachable source by hashing an uncorruptible document. If the zero knowledge proof is going to be embedded in a document, this hash might be of the parts of the document that will not be changed by the embedding.

3. Create n random keys, $p_1 \ldots p_n$ and use an encryption function to scramble the string $S + i + f'(v_i)$ for each node in the graph

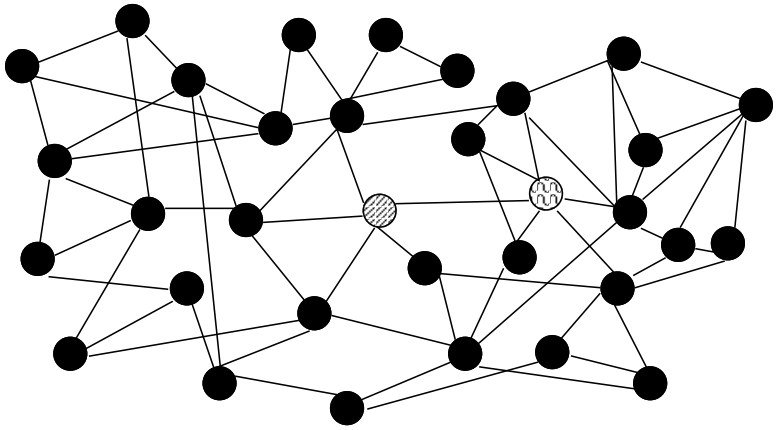


Figure 12.4: In a zero-knowledge proof, the colors of the nodes are encrypted and sent to the skeptical inquirer who asks to have two adjacent nodes revealed.

where $+$ stands for concatenation. Ship the encrypted versions to the skeptical inquirer.

4. The skeptical inquirer chooses an edge at random from the graph, (v_a, v_b) and presents it to you.

5. You ship p_a and p_b to the skeptical inquirer, who uses them to decrypt the encrypted version of the colors, $f'(v_a)$ and $f'(v_b)$. If the two are different, the skeptical inquirer knows that you've proven that you know how to color at least one part of the graph. Figure 12.4 shows two adjacent nodes being revealed.

This process should be repeated until the skeptical inquirer is satisfied. If the edges are truly chosen at random, any mistakes in the coloring stand an equal chance of being revealed at each step. The randomness prevents the prover from anticipating the choice and selecting f'. Eventually, any weakness or miscoloring will show through.

Zero knowledge proofs like this may be useful in the world of hidden information because they allow the information hider to control how and when the information is revealed to another person. The structure of the proof, however, guarantees that no information is revealed. The proof can't be repeated by someone else or used as a basis to remove the information. This solution may be useful for watermarking applications where a copyright holder may want to prove

that they are the rightful owner without revealing the technique to others.

In theory, any zero-knowledge proof can be used as a watermark. Every proof begins with a set of bits that identify the instance of the problem. Imagine that m iterations of the encrypted graph procedure described above are sufficient to prove knowledge of the graph coloring. The proof can be embedded in a document with these steps:

1. Create a value of S by hashing the parts of the document that won't be modified during the embedding process. These might be the most significant bits or some other high-pass filtering of the document.
2. Create m permutations of the coloring: $f'_1, f'_2, \ldots f'_m$.
3. Create $m \times n$ keys, $p_{i,j}$, where i stands for the iteration of the proving algorithm, $(1 \leq i \leq m)$, and j stands for the node, $(1 \leq j \leq n)$.
4. Scramble the coloring of the n nodes for each iteration of the proof by encrypting the string $S + i + j + f'_i(v_j)$, where $+$ stands for concatenation.
5. Embed each of the encrypted colors and the description of the graph in the document. If necessary, the encrypted colors can be embedded in hidden locations described by $p_{i,j}$. For instance, this value could be used as a seed for a cryptographically secure random number generator that chooses a string of pixels to hide the signal.

If someone challenges the document, they can ask the information hider to prove that they know how to color the graph hidden away inside of it. Presumably, this problem is difficult to solve and only the person who hid the graph would know how to color it. The skeptic could repeat these steps m times. Let i stand for the iteration.

1. The skeptic recovers the graph.
2. The skeptic assembles S by examining the unchanged parts of the document.
3. The skeptic chooses a random edge, (v_a, v_b).
4. The prover reveals $p_{i,a}$ and $p_{i,b}$. These values can be used to locate the encrypted colors and then decrypt them. If they don't match, the process continues. If they do match, the prover is toast.

This solution offers a number of advantages if a suitable graph can be found. The person hiding the information can prove they hid it without revealing enough information for someone to duplicate the proof. Each skeptic is going to choose the edges at random, and it's unlikely that the all of them will be repeated.

This static version is not as powerful as the interactive version because the prover must lock in the colorings. If a skeptic is able to get the prover to repeat the algorithm a number of times, eventually the skeptic will gather enough of the keys, $p_{i,j}$, to stand a good chance of proving they know the graph coloring. Eventually, the prover will reveal the entire colorings. This slow leak of information means that the proofs are not zero-knowledge. This process may still be practical if m and n are large enough and the number of times the proof is executed is kept small enough.

Finding a suitable graph is not an easy process. There are many different kinds of zero-knowledge proofs that can be embedded in similar situations, but most of them are not particularly practical. The proofs rely on problems that can be hard in the worst cases, but are usually rather easy to compute. Finding and identifying guaranteed difficult instances of the problem is not easy. One of the first public-key encryption systems developed by Ralph Merkle and Martin Hellman used one NP-complete problem known as the knapsack. Eventually, all versions of the algorithm were broken and many grew skeptical that basic NP-complete problems could be trusted to produce difficult instances.

12.5.1 Discrete Logs for Proofs

Scott Craver developed one mechanism for zero knowledge watermarking that relies on the difficulty of the discrete log problem. (Given y, find x such that $y = g^x \ mod \ q$, where q is a prime chosen according to some increasingly strict guidelines for ensuring the strength of cryptographic systems based on the difficulty of the discrete log problem.) This solution succeeds because it uses a number of fake watermarks as decoys. Ideally, anyone trying to tamper with the watermark must remove the real one as well as all of the decoys. Ideally, there will be so much information that removing them all will substantially change the image.

Let x be the secret key that acts as a watermark. This algorithm will allow the information hider to prove they know x without revealing x itself. The prover hides $y_1 = g^x \ mod \ q$ in the document using a key, p_1, to choose the locations in the document.

The prover then extracts $n-1$ fake watermarks that happen to be

in the document. That is, the prover chooses $n-1$ keys, $p_2, \ldots, p_n$, to extract $n-1$ values, $y_2, \ldots, y_n$. Using fake decoys derived from the document can help increase the strength of the watermark. The more there are, the more likely that removing them will damage the image. In some cases, the prover may actually insert them if this makes the task easier.

When a skeptic shows up with the document, the prover can show knowledge of x without giving the skeptic the power to repeat this process. The decoys prevent the skeptic from ever learning which value of y_i has a known log. Here are the steps.

1. The prover presents the keys, $p_1, \ldots, p_n$, to the skeptic.
2. The skeptic recovers the values of y_i.
3. This loop is repeated as many times as necessary.
   (a) The prover generates n *blinding factors*, $b_1, \ldots, b_n$, that are used to scramble the watermarks by computing $w_i = g^{b_i} y_i$.
   (b) The prover scrambles up these values of w_i before shipping them to the skeptic.
   (c) The skeptic flips a coin and makes one of two demands to the prover:
      i. Tell me all of the blinding values, $b_1, \ldots, b_n$. Knowing these, the skeptic can check to make sure that there is one legitimate value of w_i for each value of y_i.
      ii. Prove you know the log to one of these values. The prover accomplishes this by revealing $x+b_1 \; mod \, (q-1)$. This is the log of $w_1 = g^{b_1} y_1 = g^{b_1} g^x = g^{x+b_1} \; mod \, q$.

At each iteration, the prover must reveal one half of the solution. The structure makes it unlikely that a skeptic will turn around and successfully repeat the proof to someone else. It is not possible to take the answers from one and use the information to answer the other. Only someone who knows the value of x can do it.

There are still limitations to this algorithm. The prover must reveal the location of the different blocks of bits, something that leaves them succeptible to destruction. The decoys increase the workload of the attacker and increase the probability that removing all of the information will substantially change the document.

One way to improve the strength of the algorithm is to mix in a number of real marks with the decoys. That is, the prover arranges to hide multiple values of y_i where the prover knows the value of x_i such that $y_i = g^{x_i} \; mod \; q$. The prover can then use a different

subset of values with each skeptic as long as the prover knows one value of x_i for the values in the subset. Of course, multiple copies and multiple blocks and help reduce this danger but they do not eliminate it. Eventually, all of the marks will be revealed.

12.6 Collusion Control

Another technique for "keying" watermarks is occasionally called "collusion control" for lack of a better term. Imagine that you create two documents, d_1 and d_2, with watermark bits w_1 and w_2 encoding the true owner's identity. Watermarking is an imperfect science, so despite your best efforts someone trying to cheat the system compares the files and finds where they are different. The cheater injects random noise into these locations, effectively changing half of the bits. What happens to the watermarks?

Basic systems would fail here. If $w_1 = 001010111$ and $w_2 = 001010100$, then only the last two bits are different. A new watermark, 00101010101 would implicate someone else.

Collusion control systems that can combat this were first introduced by Dan Boneh and James Shaw.[BS95] Their mechanism acts like an extension of error-correcting codes.

Error-correcting codes are described in Chapter 3.

Here's a example. Let S be the set of n-bit code words with only a single bit set to 1. If $n = 4$, then $S = \{1000, 0100, 0010, 0001\}$. Let one code word be assigned to each document. If two document holders collude, they will only find that their watermarks differ by two bits. Boneh and Shaw call this a *frameproof* code and note that any code that is frameproof for n users must come with n bits. So this construction is optimal.

For example, let Alice's watermark be 0100 and Bob's be 0001. Someone tries to erase the watermark by creating a synthetic one blending the two files. After identifying all bits that are different, the attacker chooses half from Alice's file and half from Bob's. What happens if someone compares a file from Alice and a file from Bob to find differences? If someone creates a new file with the watermark 0101, then both Alice and Bob will be implicated. Anyone examining the file can trace it back to both of them. Changing 50 percent of the bits will produce one of the two watermarks. One will remain implicated.

Of course, a clever attacker may flip both to zero to produce the watermark 0000. This will implicate no one, but it is not easy to generate. Anyone trusting the watermark will not choose bit vectors like this example. They may XOR them with a secret password and the attacker won't know what is up and down, so to speak. A zero in

the fourth position may identify Bob after it is XORed with the secret vector.

Boneh and Shaw extend this idea by combining it with ideas from the world of error-correcting codes. Each attacker will need to flip many bits before the watermark is effectively erased, because the error-correcting codes compensate for random flips.

12.7 Summary

Most algorithms in this book can add additional security with a key. This key can be used to create a pseudo-random bit stream by computing successive values of some encryption or hash function:

$$f(key), f(f(key)), f(f(f(key))), \ldots.$$

This bit stream can be used to either choose a subset of the file or to control how it is encoded at each location. Only someone with the same key can recover the information. Of course, it also makes sense to encrypt the file with another key and a cryptographically strong algorithm before it is inserted.

Keying the algorithm defends against the often public nature of computer standards. If the software is going to be used often and distributed to many people, then the algorithms inside of it become public. Adding a key to the algorithm increases its strength dramatically.

The Disguise Many of the algorithms add an extra layer of disguise by using the key to modify the behavior of the algorithm. This means that any attacker will not be able to extract the data with knowledge of the algorithm alone.

How Secure Is It? Pseudo-random bit streams created by repeated encryption or hashing can be quite secure.

How to Use It Replace any random number generator used to add noise with a pseudo-random keyed stream, which can also be used to extract particular subsets or to rearrange the data itself. The algorithms in Chapter 13, for instance, use a keyed encryption function to change the data's order.

Further Reading

- Luis von Ahn and Nicholas J. Hopper provide a good theoretical foundation for understanding when steganography can offer public-key-like behavior. [vAH04]

- Michael Backes and Christian Cachin suggest that a good public-key steganography system is built from a good public-key crypto system offering output indistinguishable from a random source. This allows them to produce a system that can resist active attacks. [BC05]

Chapter 13

Ordering and Reordering

13.1 Top 10 Reasons Why Top 10 Lists Fail

10. Who wants to be last on the list? On the other hand, making the list is much better than being 11th.

9. Is nine really better than 10?

8. There are usually twenty to thirty people who say they are in "the top ten."

7. "Lists provide a simulacrum of order that reifies our inherent unease over the chthonic forces of disorder"– Guy de Montparnasse, doctoral thesis.

6. Sixth is a comfortable position. It's not high enough to provide endless anxiety about slippage, but it's not low enough to slip off of the list. Slipping off the list would be terrible.

5. Five golden rings.

4. There is no number 4.

3. All good things come in threes. But it's not the same if you're number 3 in a list of 10. Being third in a list of 100, however, is pretty good. Of course, every top 10 list is really a list of 100 or 1000 with the other 90 or 990 left off. Where do you draw the line?

2. No one ever remembers the silver medalist.

1. Being number one would be much more fun if everyone wasn't gunning to unseat you.

13.2 Introduction

Most of the algorithms in this book hide information in data formats that are relatively rigid. Image files must describe the color of each pixel in a well-defined order. Audio files are pretty much required to describe the sound at each point in time. Hiding the information in specific places in the noise is a pretty good gamble because the files aren't going to be rearranged.

Some data is not as rigid. Text documents can have chapters, sections, paragraphs and even sentences rearranged without changing the overall meaning. That is to say, the overall meaning of a text document isn't changed by many ways of rearranging sentences, paragraphs, sections or chapters. There's no reason why the elements in a list can't be scrambled and rescrambled without the reader noticing. Many of the chapters in this book can be reordered a bit without hurting the overall flow.

Even files with a strong fixed relationship with time and space can be rearranged. Image files can be cropped, rotated, or even arranged in a tile. Songs can be rearranged or edited, even after being fixed in a file. All of these changes can rearrange files.

When the data can be reordered, an attacker can destroy the hidden message without disturbing the cover data. Mikhail Atallah and Victor Raskin confronted this problem when designing a mechanism for hiding information in natural language text. If they hid a bit or two by changing each sentence, then they could lose the entire message if the order of each sentence was rearranged.[ARC+01] Many of the text mechanisms in Chapters 6, 7, and 8 are vulnerable to this kind of attack. One change at the beginning of the datastream could confound the decoding of everything after it. Many of the solutions in Chapter 9 store the bits in random locations dictated by some pseudo-random stream of values. Destroying the bits early in this stream can trash the rest of the file.

"There it was, hidden in alphabetical order"– Rita Holt

The information stream has two components: the data and the location where it is stored. Most of the algorithms in this book concentrate on styling and coiffing the data until it assumes the right disguise. The location where the data goes is rarely more than a tool for adding some security and the algorithms pay little attention to the problem.

This chapter focuses on the second component: how to defend and exploit the location where the information is held. Some of the algorithms in this chapter insulate data against attackers who might tweak the file in the hope of dislocating the hidden information underneath. They provide a simple way to establish a canonical order

for a file. The attackers can rearrange the file as much as they want but the sender and the receiver will still be able to re-establish the canonical order and send a message.

Other algorithms exploit the order of the content itself and make no changes to the underlying information. There are $n!$ ways that n items can be arranged. That means there are $\log_2 n!$ bits that can be transmitted in a list with n items. None of the items themselves change, just their order in a list.

Still other algorithms mix in false data to act as decoys. They get in the way until they're eliminated and the canonical order is returned.

All of these algorithms rely on the fact that information does not need to flow in a preset pattern if the correct order can be found later. This fact is also be useful if parts of the message travel along different paths.

13.3 Strength Against Scrambling

Many of the attacks on steganographic systems try to destroy the message with subtle reordering. Image warping is one the most complex and daunting parts of the StirMark benchmarks used to measure the robustness of techniques for hiding data in images.

Here's a very abstract summary of a way to make any steganographic system resist re-ordering:

- Break up the data stream into discrete elements: $\{x_1, x_2, \ldots, x_n\}$. This may be words, pixels, blocks of pixels, or any subset of the file. These blocks should be small enough to endure any tweaking or scrambling by an opponent but large enough to be different.
- Choose a function, f, that is independent of the changes that might be made in the hiding process. If the least significant bit is changed to hide a message, then f should only depend on the other bits.

  Many of the same principles that go into designing a hash function can be applied to designing this sorting function. This function, f, can also be keyed so that an additional value, k, can change the results.
- Apply f to the elements.
- Sort the list based on f.
- Hide the information in each element, x_i, with appropriate solutions.

- Unsort the list or somehow arrange for the receipient to get all of the values of $\{x_1, \ldots, x_n\}$ through any means.

In a one-time hash system, the sender changes the cover data, x, until $f(x)$ sends the right message. [Shi99] The message is usually short because searching for a x requires brute force. This is similar to the mechanisms in Chapter 9 which permute the color of a pixel until the parity sends the right message.

Information can be hidden in multiple locations and reordered by the recipient without any communication from the source. The only synchronization necessary is the choice of f and perhaps the key, k. The good news is that the multiple containers can move through different channels on independent schedules and arrive in any order without compromising the message.

Sorting the data allows the sender to scramble the cover data in any random fashion, secure in the knowledge that the receiver will be able to reorder the information correctly when it arrives.

Sorting is also a good alternative to choosing a subset of the cover data to hold the hidden message. Other solutions use some key and a random number generator to choose a random subset of the message. Another solution is to apply some function, f, to the elements in the data stream, sort the stream and then choose the first n to hold the hidden message.

Of course, the success of these solutions depends entirely on the choice of the function, f. Much of the problem is usually solved once the data is converted into some digital format. It's already a big number so sorting the values is easy. The identity function, $f(x) = x$, is often good enough.

One potential problem can occur if multiple elements are identical or produce the same value of f That is, there exist i and j such that $f(x_i) = f(x_j)$. In many normal sorting operations, the values of i and j are used to break the tie, but this can't work because the sender and the receiver may get the values of x in a different order.

If $x_i = x_j$, then it doesn't matter in which order they occur. But if $x_i \neq x_j$, then problems may emerge if the sender and the receiver do not place them in the same order. After the receiver extracts the data from x_i and x_j, it could end up in the wrong order, potentially confusing the rest of the extracted information.

There are two solutions to this problem. The simplest is to be certain that $f(x_i) = f(x_j)$ only happens when $x_i = x_j$. This happens when f is a pure encryption automorphism. This is often the best solution. The other is to use some error correcting coding to remove the damage caused by decoding problems. Some error-correcting codes can do well with several errors in a row and this kind is essential. If x_i and x_j are misordered because $f(x_i) = f(x_j)$, then they'll generate two wrong errors in order. If there are multiple values that produce the same f, then there will be multiple problems.

If it is not possible to guarantee that the values of f will be unique, it makes sense to ensure that the same amount of information is

packed into each element x_i. This guarantees that the damage misordering will not disturb the other packets.

13.4 Invariant Forms

One of the biggest challenges in creating the functions f is the fact that the message encoder will use f to choose the order *before* the data is hidden. The receiver will use f *after* the data is inserted. If the hiding process can change the value of f, then the order could be mangled.

There are two basic ways to design f. The first is to ensure that f does not change when data is hidden. Some simple invariant functions are:

- If the elements are pixels where data is inserted into the least significant bit, then the value of f should exclude the least significant bit from the calculations.
- If the elements are compressed versions of audio or image data, then the function f should exclude the coefficients that might be changed by inserting information. JPEG files, for instance, can hide data by modifying the least signficant bit of the coefficients. f should depend on the other bits.
- If the data is hidden in the elements with a spread-spectrum technique that modifies the individual elements by no more than $\pm\epsilon$, then the values can be normalized. Let x_i be the value of an element. This defines a range $x_i - \epsilon < x_i \leq x_i + \epsilon$. Let the set of normalized or canonical points be $\{0, 2\epsilon, 4\epsilon, 6\epsilon, \ldots\}$. One and only one of these points is guaranteed to be in each range. Each value of x_i can be replaced by the one canonical point that lies within $\pm\epsilon$ of x_i.

  To compute f use the canonical points instead of the real values of x_i.

13.5 Canonical Forms

Another solution is to create a "canonical form" for each element. That is, choose one version of the element that is the same. Then, the data is removed by converting it into the canonical form and the result is used to compute f.

Here are some basic ones:

- If the data is hidden in the least significant bit of pixels or audio file elements, then the canonical form can be found by setting the least significant bit to zero.

- If the information-hiding process modifies the elements by no more than $\pm\epsilon$ then canonical points can be established as they were above. Let $\{0, 2\epsilon, 4\epsilon, 6\epsilon, \ldots\}$ be the set of canonical points. Only one will lie in the range $x_i - \epsilon < x_i \leq x_i + \epsilon$.

- Sentences can be put into canonical form. Mikhail Atallah and Victor Raskin use natural language processing algorithms to parse sentences.[ARC+01] The sentence "The dog chased the cat" takes this form in their LISP-based system:

  ```
  (S
  (NP the dog)
  (VP chased
  (NP the cat)))
  ```

  The letter 'S' stands for the beginning of a sentence, the letters 'NP' stands for a noun phrase, and the letters 'VP' stands for a verb phrase. If the sentence can't be parsed, it is ignored.

  Their solution hides information by applying a number of transformations like switching between the active and passive voice. One bit could be encoded by switching this example to read "The cat was chased by the dog". Others solutions they use include moving the adjunct of a sentence, clefting the sentence, and inserting extra unnecessary words and phrases like "It seems that..."

  A canonical form can be defined by choosing one version of the transformation. In this example, the active voice might be the canonical form for the sentence.

13.6 Packing in Multiple Messages

Sorting can also let you store multiple messages in a collection of data. If f_1 defines one order and f_2 defines another order, then both orderings can be used to hide information if the sizes are right. If the number of elements in the camoflauging data is much larger than the amount of information being hidden, then the chances of a collision are small.

The chances of a collision are easy to estimate if you can assume that the sorting functions, f_1 and f_2, behave like random number

generators and place each element in the order with equal probability. The easiest way to create this is to use a well-designed cryptographically secure hash function like SHA. The designers have already worked hard to ensure that the results of these functions behave close to a random number source.

If you can assume that f_1 and f_2 are random enough, then the odds of a collision are simple. If data is hidden in the first n_1 elements in the list produced by f_1 and the first n_2 elements in the list produced by f_2, then the estimated chance of collisions is:

$$n_1 \times \frac{n_2}{n}.$$

The damage can be reduced by using error-correcting codes like the ones described in Chapter 3.

13.7 Sorting to Hide Information

The algorithms in the first part of this chapter use sorting as a form of camoflage. The information is hidden and then scrambled to hide it some more. Correctly sorting the information again reveals it.

Another solution is to actually hide information in the choice of the scrambling. If there are n items, then there are $n!$ ways that they can be arranged. If the arrangements are given numbers, then there are $\log n!$ bits. Matthew Kwan used this solution to hide information in the sequence of colors in the palette of a GIF image. A freely-available program called GifShuffle implements the solution.

Here's a simple version of the algorithm:

1. Use a keyed pseudo-random bit stream to choose pairs of pixels or data items in the file.

2. For each pair, let D be the difference between the two values.

3. If the difference is greater than some threshold of perception, ignore the pair. That is, if the differences between the pixels or data items are noticable by a human, then ignore the pair.

4. If $D = 0$, ignore the pair.

5. Encode one bit of information in the pair. Let $D > 0$ stand for a zero and $D < 0$ stand for a one. If the pixels aren't arranged in the right order, swap them.

The locations of transistors on a circuit can be sorted in different ways to hide information that may track the rightful owner.[LMSP98]

This basic mechanism hides bits in pairs of pixels or data items. The solution does not change the basic statistical profile of the underlying file, an important consideration because many attacks on steganography rely on statistical analysis. Of course, it *does* change some of the larger statistics about which pixels of some value are near other pixels of a different value. Attackers looking at basic statistics won't detect the change, but attackers with more sophisticated models could.

The algorithm can also be modified to hide the information statistically. An early steganographic algorithm called Patchwork repeats this process a number of times to hide the same bit in numerous pairs. The random process chooses pairs by selecting one pixel from one set and another pixel from a different set. The message is detected by comparing the statistical differences between the two sets. The largest one identifies the bit being transmitted. There's no attempt made to synchronize the random selection of pixels. [BGML96, GB98].

In this simple example, one bit gets hidden in the order of 2^1 items. The process can be taken to any extreme by choosing sets of n pixels or items and hiding information in the order of all of them. Here's one way that a set of n items, $\{x_0, x_1, \ldots, x_{n-1}\}$, can encode a long $\log n!$-bit number, M. Set $m = M$ and let S be the set $\{x_1, x_2, \ldots, x_n\}$. Let the answer, the set A, begin as the empty set. Repeat this loop for i taking values beginning with n and dropping to 2.

1. Select the item in S with the index $m \; mod \; i$. The indices start at 0 and run up to $i - 1$. There should only be i elements left in S at each pass through the loop.
2. Remove the item from S and stick it at the end of A.
3. Set $m = \frac{m}{i}$. Round down to the nearest integer.

The value of M can be recovered with this loop. Begin with $m = 1$.

1. Remove the first element in A.
2. Convert this element into a value by counting the values left in A with a subscript that is less than its own. That is, if you remove x_i, count the values of x_j still in A where $j < i$.
3. Multiply m by this count and set it to be the new value of m.

Using this steganographically often requires finding a way to assign a true order to the element. This algorithm assumes that the

elements in the set S come ordered from 0 to $n-1$. Kwan, for instance, orders the RGB colors in the palette of the GIF by using what he calls the "natural" order. That is, every color is assigned the value $2^{16} \times Red + 2^8 \times Blue + Green$ and then sorted accordingly.

Many of the sorting strategies used above can also be used to sort the elements. If they are encrypted with a particular key before the sorting then the key acts as a key for this steganography.

This algorithm can be particularly efficient in the right situations. Imagine you have all 65,536 two byte values arranged in a list. This takes $128k$ bytes. This list can store 119255 bytes with this algorithm—close to 90% of the payload. Of course, good steganographic opportunities like shopping lists are rarely so compact, but the theoretical potential is still astonishing.

13.8 Word Scrambling

One commonly cited observation from cognitive science is that humans can read words even if the order of some of the internal letters are reordered. Apparently the words are usually understandable if the first and last letters are left unchanged. The mind seems to pay particular attention to the first and last letter while ignoring the absolute order of the interior letters.

Here's a sample:

> *At Pairs, jsut atfer dark one gusty eivnneg in the auutmn of 18, I was einonyjg the tofolwd luxury of mittoiaedn and a mehuseacrm, in cnpmaoy wtih my fnired C. Augsute Diupn, in his lttile back lrraiby, or book-coelst, au tiosrime, No. 33 Rue Donot, Fuuoarbg St. Giearmn. For one hour at lesat we had maniienatd a pofunord sielcne; wilhe ecah, to any caasul osvreebr, might hvae seemed intetly and eilesulcvxy oipcecud with the cilunrg eiddes of sokme taht oppressed the apoestrhme of the ceahmbr. For melsyf, heovewr, I was mletnlay dusiscisng cietarn tipocs wichh had foermd mettar for ciestnovoarn bteeewn us at an eielarr pioerd of the eiennvg; I mean the aiffar of the Rue Mogure, and the mestryy aintentdg the meudrr of Miare Rogt. I leookd on it, torfeerhe, as sioetnhmg of a coniiccende, wehn the door of our apetnarmt was torwhn open and aittedmd our old aincqatcanue, Mioesunr G, the Pfeecrt of the Piasairn piolce.*

Here's the original from Edgar Allan Poe's *Purloined Letter*:

> *At Paris, just after dark one gusty evening in the autumn of 18, I was enjoying the twofold luxury of meditation and a meerschaum, in company with my friend C. Auguste Dupin, in his little back library, or book-closet, au troisime, No. 33 Rue Donot, Faubourg St. Germain. For one hour at least we had maintained a profound silence; while each, to any casual observer, might have seemed intently and exclusively occupied with the curling eddies of smoke that oppressed the atmosphere of the chamber. For myself, however, I was mentally discussing certain topics which had formed matter for conversation between us at an earlier period of the evening; I mean the affair of the Rue Morgue, and the mystery attending the murder of Marie Rogt. I looked on it, therefore, as something of a coincidence, when the door of our apartment was thrown open and admitted our old acquaintance, Monsieur G, the Prefect of the Parisian police.*[Poe44]

The algorithm used the original order as the canonical order for the letters in each word and then hid information in the order of the $n-2$ interior letters of every n-letter word.

13.9 Adding Extra Packets

Another way to scramble the order is to add fake packets. Ron Rivest suggested this idea as one way to evade a ban on cryptography. [Riv98] He suggested letting every packet fly in the clear with a digital signature attached. Valid packets come with valid signatures, while invalid packets come with invalid signatures.

Rivest suggested using keyed message authentication codes generated by hash functions. If x is the message, then the signature consisted of $f(kx)$, where k is a key known only to those who can read the message. Let $f'(x)$ be some bad signature-generating function, perhaps a random number generator. Let the message be $\{x_1, x_2, \ldots, x_n\}$. Rivest called these the *wheat*. Let the *chaff* be $\{r_1, \ldots, r_k\}$, random numbers of the correct size. The message consists of pairs like $(x_1, f(kx_1))$ mixed in with distractions like $(r_i, f'(r_i))$.

Many standard digital signature algorithms can also be used if we relax the notion that the signatures can tested by anyone with access to a public key. If the verification key is kept secret, only the sender and any receiver can tell the wheat from the chaff. Traditional public

key algorithms can still be useful here because the receiver's key can not be used to generate the signatures themselves.

Mihir Bellare and Alexandra Boldyreva take this one step further with all or nothing transforms that provide more efficiency.[BB00a]

At the time Rivest's article was written, the U.S. government restricted the export of algorithms designated as "cryptography" while placing no limitations on those used for "authentication". Hash function-based message authentication codes were typically assumed to offer no secrecy and thus were freely exportable. Rivest suggested that his solution pointed to a loophole that weakened the law.

The security of this solution depends to a large extent on the structure of x and the underlying data. If x contains enough information to be interesting in and of itself, the attacker may be able to pick out the wheat from the chaff without worrying about f or k.

One solution is to break the file into individual bits. This mechanism is a bit weak, however, because there will be only two valid signatures: $f(k0)$ and $f(k1)$. Rivest overcomes this problem by adding a counter or nonce to the mix so that each packet looks like this: $(x_i, i, f(kix_i))$. This mechanism is not that efficient because each bit may require a 80-200-bit-long packet to carry it.

This solution can easily be mixed with the other techniques that define the order. One function, f, can identify the true elements of the message and another function, g, can identify the canonical order for the elements so the information can be extracted.

13.10 Port Knocking

When bits travel across the Internet, they carry two major pieces of information that act as the address: the *IP address* and the *port*. The IP address has traditionally been four bytes written as four base 10 numbers separated by periods (e.g. 55.123.33.252), an old format that is gradually being eclipsed by *IPv6* a newer version with longer addresses to solve the problems of overcrowding. The IP address generally makes a clear distinction between machines, while the port generally makes a distinction between functions.

So information sent to a different IP address will generally go to a different machine, although this is often confused by the way that some machines, especially servers, will do the work of multiple machines. Information sent to different ports will generally end up in the control of different software packages that will interpret it differently. Telnet generally uses port 25, while webservers generally answer requests on port 80. There are dozens of standard and not so standard choices that computers generally follow but don't have to

do so. Some web servers answer port 8080 while some users try to route data through port 80 trying to act like the data from a web site.

Network administrators have tried to curtail some behavior on their branches of the Internet by blocking particular ports. If the administrator doesn't like web sites operating inside the subnetwork, the administrator can program the routers to refuse to deliver all information heading to port 80. Spammers can usually be locked out by shutting down all traffic going to port 25, a technique that also effectively blocks all legitimate messages too.

What's a network administrator to do? There's no easy way to tell the difference between good and bad information by looking at the port numbers alone. Good and bad people use them for good and bad reasons.

One neat idea is *port knocking*, a virtual implementation of the idea used by speakeasies and other places of ill repute to exclude police and other undesirables by only opening a door if the person uses the right pattern of knocks. The right pattern of sounds, say "rap rap RAP [pause] RAP RAP", is equivalent to a computer trying to open a connection to ports 152, 222, and 13 in short succession. A firewall might deny access to all outsiders who don't present the right pattern of data (ports 152, 222, and 13) at the beginning of a session and block data from other IP addresses that don't present the right sequence. The firewall doesn't need to read the data itself, just the quick pattern of requests for connection to particular ports. [Krz03b, Krz03a]

A nice collection of articles and pointers to software packages can be found at `portknocking.org`.

This technique is an ideal application of the way to hide information inside of a list described in this chapter. A user and a firewall could agree on the correct pattern for the day by computing $h(salt|port|password)$ for each available port. (It would make sense to exclude the ports that might be left open, say 23, 25 and 80.) Then these hashed values could be sorted and the first n used as the port knocking sequence. The $salt$ could consist of some random information and, perhaps, the date or time that the sequence would work.

The port knocking could also encrypt a message by permuting the available port numbers according to the algorithms in this chapter.

13.10.1 Enhancing Port Knocking

In recent years, port knocking has captured the imagination of a number of protocol designers who've found a number of ways to revise and extend the idea to enhance its security [Jea06, deG07]:

One Time Pads The desirable sequence can only be used once and then it is forgotten. This might be implemented by adding a

counter to the $salt$ used to compute $h(salt|port|password)$. After each successful use, the counter in the salt is incremented and the old version is no longer useful.

Challenge-Response Knocking If one knocking sequence is good, then two or three might be more secure. If Alice wants Bob to open up, Alice can present the right sequence of knocks for the day. Then Bob could return the favor and send some challenge value back to Alice. It could go in the clear or it could be further encrypted by turning it into a sequence of knocks that Bob makes on Alice's firewall. Alice decodes this challenge value or *nonce* and then uses it to compute a new sequence of knocks to return. Perhaps Alice must compute $h(salt|port|password|nonce)$ for each $port$. Then Bob can really be certain that Alice is trustworthy.

Letting Bob challenge Alice to create a new sequence of knocks removes the danger that a casual eavesdropper will be able to figure out the right sequence of ports by simply replaying the traffic pattern. Alice's second set of knocks will be different with each different $nonce$ and only someone who knows the correct values of $salt$ and $password$ will be able to compute the right sequence of ports.

Monotonically Increasing Knocks If port requests are sent across the Internet in quick succession, they may arrive in a different order than they departed. This scrambling can confound an algorithm that is depending on a particular sequence. One trick is to remove the ordering and just look at the choice of ports. If Alice requests access by computing the first n ports after sorting $h(salt|port|password)$, then it will suffice for Alice to present those first n ports in *any* sequence. That is, if the correct order is ports $1552, 299, 441,$ and 525, then any of the 24 ways of knocking on these four ports will be enough to gain access.

It should be clear that this cuts down the brute-force attack by a factor of $n!$, making it seem a bit weaker, but this approach can increase the stealth by reducing the chance for a consistent pattern. Alice may knock on ports $1552, 299, 441,$ and 525 at one moment and $441525, 299,$ and 1552 at the next.

Fake Ports There's no reason why the sequence of ports needs to be limited to the correct sequence and only that sequence. Bob might allow Alice to mix in as many as m different spurious values chosen at random. As long as the n correct ports arrive within a window of $n + m$ knocks, then Bob will assume that

Alice is correct. This can also increase the stealth by preventing the same pattern of n ports from emerging.

Subsets Another way to make the process even trickier is to group together sets of ports, call them $P_1, P_2, \ldots$. When the algorithm says knock on port i, it chooses one of the ports from set P_i, adding more confusion.

There is no end to the complexity and deception that can be added to these algorithms. The only limitation is that port knocking is relatively expensive. A "knock" consists of a UDP or TCP packet that may have several hundred bits even though it is only conveying 16 bits of information, the port number. Mixing in fake ports with a challenge and response can really slow things down. This is why some are examining packing all of the authentication information into a single packet, a technique often called, unsurprisingly, *single packet authentication*.

13.11 Continuous Use and Jamming

There is no reason why the information needs to be encoded in one set of n items and $n!$ different permutations. R. C. Chakinala, Abishek Kumarasubramanian, R. Manokaran, G. Noubir, C.Pandu Rangan, and Ravi Sundaram imagined using the technique to send information by distorting the order of packets traveling over a TCP-IP network. These packets are not guaranteed to arrive in the same sequence in which they left the sender and so the packets include a packet number to allow the recipient to put them back in the correct order. Sometimes packets take different paths through the network and the different paths reorder things. This is an opportunity to hide information by purposely misordering the packets as they leave and hiding a signal in this misordering. [CKM+06]

In a long file transfer, there's no obvious beginning and end of the set of objects and so it's possible to imagine a more continuous transfer of data by modifying the order of successive groups. It would be possible, for instance, to reorder every 5 packets in groups and send along $log_2 5!$ bits.

If some intermediary wants to distort the flow of packets to try and jam any communications, the game becomes a bit more interesting. Imagine that the sender can only modify the position of a packet by a few slots, say $+/-2$. The jammer can only change the position of a few. The result can be analyzed by game theory to determine the maximal data throughput that the sender can produce and the maximum amount of data that the jammer can stop.

13.12 Summary

The order of objects in a set is a surprisingly complex stream of information that offers a correspondingly large opportunity for steganography. Sometimes an adversary may choose to change the order of a list of objects in the hope of destroying some steganography. Sometimes the data objects take different asynchronous paths. In either case, a hidden canonical order defined by some keyed function, f, is a good way to restore the order and withstand these attacks.

The idea can also be turned on its head to store information. Changing the order of objects requires no change in the objects themselves, eliminating many of the statistical or structural forms of steganalysis described in Chapter 17. There are no anolomies created by making subtle changes. The result can hold a surprisingly large amount of information. n objects can store $\log n!$ bits.

Of course, sometimes subtle changes may need to be made to add cover. Many lists are sorted alphabetically or according to some other field of the data. This field could be an artificially generated system t act as a cover. A list of people, for instance, might be sorted by a membership number generated randomly to camoflage the reason for sorting it.

The Disguise Any set of n items, $\{x_1, \ldots, x_n\}$, can conceal $\log n!$ bits of information in the sorted order. This information can be keyed or hidden by sorting on $f(x_i)$ instead of x_i, where f is an encryption or hash function.

Some decoy packets can also distract eavesdroppers if another function, g, can be used to create a secure signature or message authentication code that distinguishes between valid and invalid packets.

Sometimes the attacker will rearrange the order of a set of objects to destroy a message. Sorting the objects with the function f before processing them is a good defense.

How Secure Is It? The security depends on both the quality of the camouflaging data and the security of f. If the set of objects is a list that seems completely random, then it is unlikely to receive any scrutiny. The information can only be extracted if the attacker discovers the true order of the set— something that a solid encryption or hash function can effectively prevent.

How to Use It? Choose your objects for their innocence, choose an encryption function, choose a key, compute $f(x_i)$ for all ob-

jects, sort on this value to get the so-called "true order", use the reordering function to insert the data, and ship with this order.

Further Reading

- Rennie deGraaf, John Aycock and Michael Jacobson add some authentication to port knocking by mixing in standard authentication algorithms and public-key cryptography. [dAJ05]
- David Glaude and Didier Barzin created SteganoGifPaletteOrder which hides information in the permutation of the colors in the GIF palette in the same manner as GifShuffle. They have a nice description of the algorithm. See `http:/users.skynet.be/glu/sgpo.htm`
- There's been a great deal of exploration of how to hide basic information in TCP/IP protocols. (And much of this applies to other network protocols because it relies upon general weaknesses in exchanges.) See, for instance, work by Joanna Rutkowska [Rut06],

  Steven J. Murdoch and Stephen Lewis [ML05, Mur06], and

  Eugene Tumoian and Maxim Anikeev. [TA05]
- Mihir Bellare and Alexandra Boldyreva looked at the security of chaffing and winnowing in [BB00b].

Chapter 14

Spreading

14.1 A New Job

We open with a shot of a fishing boat where a father and son work on tempting the fish with seductive offers of worms and minnows.

Father: It makes me happy to have my son come home from college and stay with us.

Son: Ah, quit it.

Father: No. Say something intelligent. I want to see what I got for my money.

Son: Quit it.

Father: No. Come on.

Son: The essential language of a text is confusion, distraction, misappropriation, and disguise. Only by deconstructing the textual signifiers, assessing and reassessing the semiotic signposts, and then reconstructing a coherent yet divergent vision can we truly begin to apprehend and perhaps even comprehend the limits of our literary media.

The son reels in his line and begins to switch the lures.

Father: It makes me proud to hear such talk.

Son: I'm just repeating it.

Father: A degree in English literature is something to be proud of.

Son: Well, if you say so.

Father: Now, what are you going to do for a job?

Son: Oh, there aren't many openings for literature majors. A friend works at a publishing house in New York...

The son trails off, the father pauses, wiggles in his seat, adjusts his line and begins.

Father: There's no money in those things. College was college. I spoke to my brother Louie, your uncle, and he agrees with me. We want you to join the family business.

Son: The import/export business? What does literature have to do with that?

Father: We're not technically in import and export.

Son: But.

Father: Yes, that's what the business name says, but as you put it, there's a bit of confusion, distraction and disguise in that text.

Son: But what about the crates of tomatoes and the endless stream of plastic goods?

Father: Subtext. We're really moving money. We specialize in money laundering.

Son: What does money have to do with tomatoes?

Father: Nothing and everything. We move the tomatoes, they pay us for the tomatoes, someone sells the tomatoes, and in the end everyone is happy. But when everything is added up, when all of the arithmetic is done, we've moved more than a million dollars for our friends. They look like tomato kings making a huge profit off of their tomatoes. We get to take a slice for ourselves.

Son: So it's all a front?

Father: It's classier if you think of the tomatoes as a language filled with misdirection, misapprehension, and misunderstanding. We take a clear statement written in the language of money, translate it into millions of tiny tomatoe sentences, and then, through the magic of accounting, return it to the language of cold, hard cash.

Son: It's a new language.

Father: It's an old one. You understand addition. You understand the sum is more than the parts. That's what we do.

Son: So what do you want from me?

Father: You're now an expert on the language of misdirection. That's exactly what we need around the office.

Son: A job?

Father: Yes. We would pay you three times whatever that publishing firm would provide you.

Son: But I would have to forget all of the words I learned in college. I would need to appropriate the language of the stevedores and the longshoremen.

Father: No. We want you to stay the way you are. All of that literature stuff is much more confusing than any code we

could ever create.
Father and son embrace in a happy ending unspoiled by confusion or misdirection.

14.2 Spreading the Information

Many of the algorithms in this book evolved during the modern digital era where bits are bits, zeros are zeros and ones are ones. The tools and the solutions all assume that the information is going to be encoded as streams of binary digits. Tuning the applications to this narrow view of the world is important because binary numbers are the best that the modern machines can do. They are still capable of hair-splitting precision, but they do it by approximating the numbers with a great deal of bits.

The algorithms in this chapter take a slightly different approach. While the information is still encoded as ones or zeros, the theory involves a more continuous style. The information at each location can vary by small amounts that may be fractions like $.042$ or 1.993. This detail is eventually eliminated by rounding them off, but the theory embraces these fractional values.

This is largely because the algorithms imitate an entire collection of techniques created by radio engineers. Radio engineers attacked a similar problem of hiding information in the past when they developed spread-spectrum radio using largely analog ideas. In the beginning, radios broadcast by pumping power in and out of their antenna at a set frequency. The signal was encoded by changing this power ever so slightly. Amplitude modulated radios (AM) changed the strength of signal while frequency modulated radios (FM) tweaked the speed of the signal a slight amount. To use the radio engineers words, all of the "energy" was concentrated at one frequency.

spread-spectrum radio turned this notion on its head. Instead of using one frequency, it used several. All of the energy was distributed over a large number of frequencies— a feat that managed to make radio communication more secret, more reliable, more efficient and less susceptible to jamming. If the signal lived on many different frequencies, it was much less likely that either intentional or unintentional interference would knock out the signal. Several radios could also share the same group of frequencies without interfering. Well, they might interfere slightly, but a small amount wouldn't disrupt the communications.

Many of the techniques from the spread-spectrum radio world are quite relevant today in digital steganography. The ideas work well

if the radio terminology is translated into digital speak. This is not as complicated as it looks. Plus, using digital metaphors can offer a few additional benefits which may be why most spread-spectrum radios today are entirely digital.

The basic approach in radio lingo is to "spread the energy out across the spectrum". That is, place the signal in a number of different frequencies. In some cases, the radios hop from frequency to frequency very quickly in a system called time sequence. This frequency hopping is very similar to the technique of using a random number generator to choose the locations where the bits should be hidden in a camouflaging file. In some cases, the same random number generators developed to help spread spectrum radios hop from frequency to frequency are used to choose pixels or moments in an audio file.

Sometimes the systems use different frequencies at the same time, an approach known as direct sequence. This spreads out the information over the spectrum by broadcasting some amount of information at one frequency, some at another, etc. The entire message is reassembled by combining all of the information.

The way the energy is parceled out is usually pretty basic. At each instance, the energy being broadcast at all of the frequencies is added up to create the entire signal. This is usually represented mathematically by an integral. Figure 14.1 shows two hypothetical distributions. The top integrates to a positive value, say 100.03, and the bottom integrates to 80.2. Both functions look quite similar. They have the same number of bumps and the same zero values along the x-axis. The top version, however, has a bit more "energy" above the x-axis than the other one. When all of this is added up, the top function is sending a message of "100.03"

spread-spectrum radio signals like this are said to be resistant to noise and other interference caused by radio jammers. Random noise along the different frequencies may distort the signal, but the changes are likely to cancel out. Radio engineers have sophisticated models of the type of noise corrupting the radio spectrum and they use the models to tune the spread-spectrum algorithms. Noise may increase the signal at one frequency, but it is likely to decrease it somewhere else. When everything is added together in the integral the same result comes out. Figure 14.2 shows the same signals from Figure 14.1 after a bit of noise corrupts them.

A radio jammer trying to block the signal faces a difficult challenge. Pumping out random noise at one frequency may disrupt a signal concentrated at that single frequency, but it only obscures one small part of the signal spread out over a number of frequencies. The

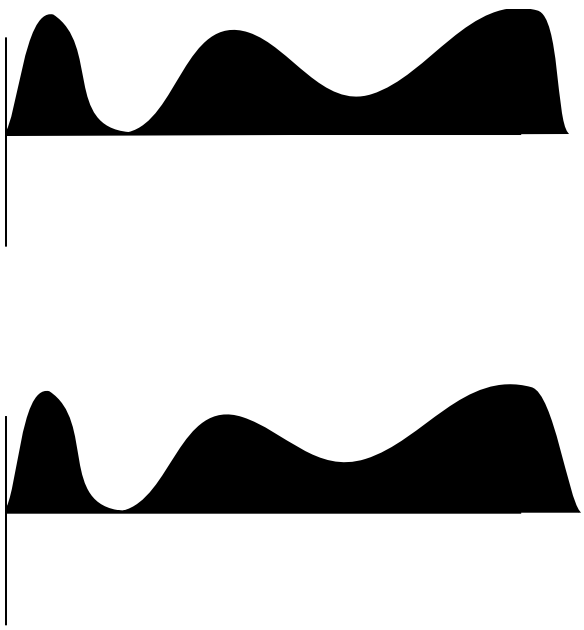


Figure 14.1: The energy spread over a number of different frequencies is computed by integration. The top function integrates to say 100.03 and the bottom integrates to 80.2. Both look quite similar but the top one is a bit more top-heavy.

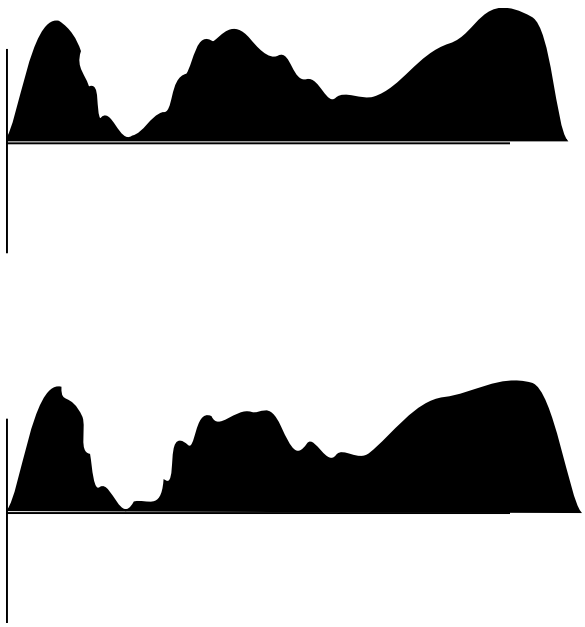


Figure 14.2: The signals from Figure 14.1 after being changed by a bit of random noise. The integrals still come out to 100.03 and 80.2.

effects of one powerful signal are easy to filter out by placing limits on the amount of power detected at each frequency.

If a jammer is actually able to identify the band of frequencies used for the entire spread-spectrum signal, it is still hard to disrupt it by injecting random noise everywhere. The noise just cancels out and the signal shines through after integration. In practice, there are ways to jam spread-spectrum signals, but they usually require the jammer to pump out significantly more power. Large amounts of noise can disrupt the system.

These algorithms for spreading information out over a number of bits are very similar to the error-correcting codes described in Chapter 3.

14.3 Going Digital

The spread-spectrum solutions use the analog metaphors of continuous functions measuring "energy" that are evaluated using integration. In the digital world, integers measuring any quantity define discrete functions. In many cases, they measure energy. The integers that define the amount of red, blue and green color at any particular pixel measure the amount of red, green and blue light coming from that location. The integers in a sound file measure the amount of energy traveling as a pressure wave. Of course, the same techniques also work for generic numbers that measure things besides energy. There's no reason why these techniques couldn't be used with an accounting program counting money.

A "spread-spectrum" digital system uses the following steps:

1. **Choose the Locations** If the information is going to be spread out over a number of different locations in the document, then the locations should be chosen with as much care as practical. The simplest solution is choose a block of pixels, a section of an audio file, or perhaps a block of numbers. More complicated solutions may make sense in some cases. There's no reason why the locations can't be rearranged, reordered, or selected as directed by some keyed algorithm. This might be done with a sorting algorithm from Chapter 13 to find the right locations or a random number generator that hops from pixel to pixel finding the right ones.

2. **Identify the Signal Strength** Spread spectrum solutions try to hide the signal in the noise. If the hidden signal is so much larger than the noise that it competes with and overwhelms the main signal, then the game is lost.

   Choosing the strength for the hidden signal is an art. A strong signal may withstand the effects of inexact compression algo-

rithms but it also could be noticeable. A weaker signal may avoid detection by both a casual user and anyone trying to recover it.

3. **Study the Human Response** Many of the developers of spread-spectrum solutions for audio files or visual files study the limits of the human's senses and try to arrange for their signals to stay beyond these limits. The signal injected into an audio file may stay in the low levels of the noise or it may hide in the echoes.

   Unfortunately, there is a wide range in human perception. Some people have more sensitive eyes and ears. Everyone can benefit from training. Many spread-spectrum signal designers have developed tools that fooled themselves and their friends, but were easily detected by others.

4. **Inject the Signal** The information is added to the covering data by making changes simultaneously in all of the locations. If the file is a picture, then all of the pixels are changed by a small amount. If it's a sound file, then the sound at each moment gets either a bit stronger or a bit weaker.

The signal is removed after taking the same steps to find the right locations in the file.

Many approaches to spread-spectrum signal hiding use a mathematical algorithm known as the Fast Fourier Transform. This uses a collection of cosine and sine functions to create a model of the underlying data. The model can be tweaked in small ways to hide a signal. These approaches might be said to be everything in parallel. More modern variations use only the cosine (Discrete Cosine Transform), the sine (Discrete Sine Transform), or more complicated waveforms altogether (Discrete Wavelet Transform).

14.3.1 An example

Figure 14.3 shows the graph of .33 seconds from a sound file with samples taken 44,100 times per second. The sound file records the sound intensity with a sequence of $14,976$ integers that vary from about $30,000$ to about $-30,000$. Call these x_i, where i ranges between 0 and 14975.

How can information be distributed throughout a file? A spread-spectrum approach is to grab a block of data and add small parts of the message to every element in the block. Figure 14.4 shows a graph of the data at the moments between 8200 and 8600 in the sound file. A message can be encoded by adding or subtracting a small amount

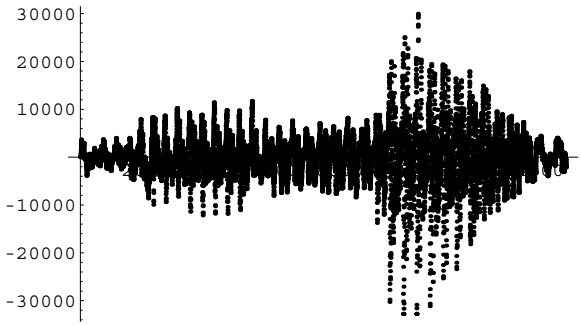


Figure 14.3: A graph of a .33 seconds piece from a sound file.

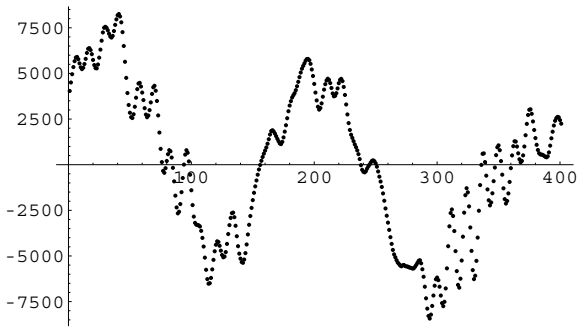


Figure 14.4: A small section of Figure 14.3 from range (8200,8600).

from each location. The same solution can be used with video data or any other data source that can withstand small pertubations.

Here are the raw values between 8250 and 8300: 2603, 2556, 2763, 3174, 3669, 4140, 4447, 4481, 4282, 3952, 3540, 3097, 2745, 2599, 2695, 2989, 3412, 3878, 4241, 4323, 4052, 3491, 2698, 1761, 867, 143, -340, -445, -190, 203, 575, 795, 732, 392, -172, -913, -1696, -2341, -2665, -2579, -2157, -1505, -729, 6, 553, 792, 654, 179, -548, -1401, -2213 .

The values in the range $(8200, 8600)$ add up to $40,813$, or an average of about 102.

A basic algorithm encodes one bit by choosing some strength factor, S, and then arranging for the absolute value of the average value of the elements to be above S if the message is a 1 and below S if the message is a 0.

Choosing the right value of S for this basic algorithm is something of an art that is confounded by the size of the blocks, the strength of the real signal, the nature of the sound, and several other factors. Let's imagine $S = 10$. If the message to be encoded is 1, then nothing

needs to be done. The average value of 102 is already well above 10.

If the message is 0, however, the average value needs to be reduced by at least 92 and perhaps more if there's going to be any margin of error. Subtracting 100 from each element does not distort the signal too much when the values range between ± 7500. Of course, some elements have small values like 6 or -190, and they will be distorted more, but this is well below the threshold of our perception.

A more sophisticated mechanism spreads the distortion proportionately. This can be calculated with this formula:

$$\frac{x_i = x_i - x_i \times total\ change}{\sum |x_i|}$$

If this is reduced to each value, x_i, then the sum moves by the amount of total change.

This approach has several advantages over simply encoding the information in the least signficant bit because the data is spread over a larger block. Any attacker who just flips a random selection of the least signficant bits will wipe out the least significant bit message, but will have no effect on this message. The random changes will balance out and have no net effect on the sum. If the absolute value of the average value is over S, then it will still be over S. If it was under, then it will still be under.

Random noise should also have little affect on the message if the changes balance out. A glitch that adds in one place will probably be balanced out by a glitch that subtracts in another. Of course, this depends on the noise behaving as we expect. If the size of the blocks is big enough, the odds suggest that truly random noise will balance itself.

The mechanism does have other weaknesses. An attacker might insert a few random large values in places. Changing several small elements of 100 to $30,000$ is one way to distort the averages. This random attack is crude and might fail for a number of reasons. The glitches might be perceptable and thus easily spotted by the parties. They could also be eliminated when the sound file is played back. Many electronic systems remove short, random glitches.

Of course, there are also a number of practical limitations. Many compression algorithms use only a small number of values or quanta in the hope of removing the complexity of the file. 8-bit μ-law encoding, for instance, only uses 256 possible values for each data element. If a file were compressed with this mechanism, any message encoded with this technique could be lost when the value of each element was compressed by converting it to the closest quantized value.

There are also a great number of practical problems in choosing the size of the block and the amount of information it can carry. If the

blocks are small, then it is entirely possible that the signal will wipe out the data.

The first part of Figure 14.4, for instance, shows the data between elements x_{8200} and x_{8300}. Almost all are above 0. The total is $374,684$ and the average value is 3746.84. Subtracting this large amount from every element would distort the signal dramatically.

Larger blocks are more likely to include enough of the signal to allow the algorithm to work, but increasing the size of the block reduces the amount of information that can be encoded.

In practice, blocks of 1000 elements or $1/44^{th}$ of a second seem to work with a sound file like the one displayed in Figure 14.3. The following table shows the average values in the blocks. The average values are about 1% of the largest values in the block. If S is set to be around 3%, then the signal should be encoded without a problem.

x_1 to x_{1000}	19.865
x_{1000} to x_{1999}	175.589
x_{2000} to x_{2999}	-132.675
x_{3000} to x_{3999}	-354.728
x_{4000} to x_{4999}	383.372
x_{5000} to x_{5999}	-111.475
x_{6000} to x_{6999}	152.809
x_{7000} to x_{7999}	-154.128
x_{8000} to x_{8999}	-59.596
x_{9000} to x_{9999}	153.62
$x_{10,000}$ to x_{10999}	-215.226

14.3.2 Synchronization

This mechanism is also susceptible to the same synchronization problem that affects many watermarking and information hiding algorithms, but it is more resilient than many. If a significant number of elements are lost at the beginning of the file, then the loss of synchronization can destroy the message.

This mechanism does offer a gradual measure of the loss of synchronization. Imagine that the block sizes are 1000 elements. If only a small number, say 20, are lost from the beginning of the file, then it is unlikely that the change will destroy the message. Spreading the message over the block ensures that there will still be 980 elements carrying the message. Clearly, as the amount of desynchronization increases, the quality of the message will decrease, reaching a peak when the gap reaches one half of the block size. It is interesting to note that the errors will only occur in the blocks where the bits being encoded change from either a zero to a one or a one to a zero.

Synchronization can also be automatically detected by attempting to extract the message. Imagine that the receiver knows that a hidden message has been encoded in a sound file but doesn't know where the message begins or ends. Finding the correct offset is not hard with a guided search.

Chapter 3 discusses error-detecting and error-correcting codes.

The message might include a number of parity bits, and a basic error-detecting solution. That is, after every eight bits, an extra parity bit is added to the stream based on the number of 1s and 0s in the previous eight bits. It might be set to 1 if there's an odd number and 0 if there's an even number. This basic error detection protocol is used frequently in telecommunications.

This mechanism can also be used to synchronize the message and find the location of the starts of the blocks through a brute-force search. The file can be decoded using a variety of potential offsets. The best solution will be the one with the greatest number of correct parity bits. The search can be made a bit more intelligent because the quality of the message is close to a continuous function. Changing the offset a small amount should only change the number of correct parity bits by a correspondingly small amount.

Many attacks like the StirMark tests destroy the synchronization. Chapter 13 discusses one way to defend against it.

More sophisticated error-correcting codes can also be used. The best offset is the one that requires the fewest number of corrections to the bit stream. The only problem with this is that more correcting power requires more bits, and this means trying more potential offsets. If there are 12 bits per word and 1000 samples encoding each bit, then the search for the correct offset must try all values between 0 and 12000.

14.3.3 Strengthening the System

Spreading the information across multiple samples can be strengthened by using another source of randomness to change the way that data is added or subtracted from the file. Let α_i be a collection of coefficients that modify the way that the sum is calculated and the signal is extracted. Instead of computing the basic sum, calculate the sum weighted by the coefficients:

$$\sum \alpha_i x_i.$$

The values of α can act like a key if they are produced by a cryptographically-secure random number source. One approach is to use a random bit stream to produce values of α equal to either $+1$ or -1. Only someone with the same access to the random number source can compute the correct sum and extract the message.

The process also restricts the ability of an attacker to add random glitches in the hope of destroying the message. In the first example, the attacker might always use either positive or negative glitches and drive the total to be either very positive or very negative. If the values of α are equally distributed, then the heavy glitches should balance out. If the attacker adds more and more glitches in the hope of obscuring any message, they become more and more likely to cancel each other out.

Clearly, the complexity can be increased by choosing different values of α such as $2, 10$ or 1000, but there do not seem to be many obvious advantages to this solution.

14.3.4 Packing Multiple Messages

Random number sources like this can also be used to select strange or discontinuous blocks of data. There's no reason why the elements x_1 through x_{1000} need to work together to hide the first bit of the message. Any 1000 elements chosen at random from the file can be used. If both the message sender and the receiver have access to the same cryptographically secure random number stream generated by the same key, then they can both extract the right elements and group them together in blocks to hide the message.

This approach has some advantages. If the elements are chosen at random, then the block sizes can be significantly smaller. As noted above, the values between x_{8200} and x_{8300} are largely positive with a large average. It's not possible to adjust the average up or down to be larger or smaller than a value, S, without signficantly distorting the values in the block. If the elements are contiguous, then the block sizes need to be big to ensure that they'll include enough variation to have a small average. Choosing the elements at random from the entire file reduces this problem significantly.

The approach also allows multiple messages to be packed together. Imagine that Alice encodes one bit of a message by choosing 100 elements from the sound file and tweaking the average to be either above or below S. Now, imagine that Bob also encodes one bit by choosing his own set of 100 elements. The two may choose the same element several times, but the odds are quite low that there will be any significant overlap between the two. Even if Alice is encoding a 1 by raising the average of her block and Bob is encoding a 0 by lowering the average of his block, the work won't be distorted too much if very few elements are in both blocks. The averages will still be close to the same.

Engineering a system depends on calculating the odds and this

process is straight forward. If there are N elements in the file and k elements in each block, then the odds of choosing one element in a block is $\frac{k}{N}$. The odds of having any overlap between blocks can be computed with the classic binomial expansion.

In the sound file displayed in Figure 14.3, there are $14,976$ elements. If the block sizes are 100 elements, then the odds of choosing an element from a block is about $\frac{1}{150}$. The probability of choosing 100 elements and finding no intersections is about $.51$, one intersection, about $.35$, two intersections, about $.11$, three intersections, about $.02$, and the rest are negligible.

Of course, this solution may increase the amount of information that can be packed into a file, but it sacrifices resistance to synchronization. Any complicated solution for choosing different elements and assembling them into a block will be thwarted if the data file loses information. Choosing the 1st, the 189th, the 542nd, and the 1044th elements from the data file fails if even the first one is deleted.

14.4 Comparative Blocks

The first section spread the information over a block of data by raising the average so it was larger or smaller than some value, S. The solution works well if the average is predictable. While the examples used sound files with averages near zero, there's no reason why other data streams couldn't do the job if their average was known beforehand to both the sender and the recipient.

Another solution is to tweak the algorithm to adapt to the data at hand. This works better with files like images, which often contain information with very different statistical profiles. A dark picture of shadows and a picture of a snow-covered mountain on a sunny day have significantly different numbers in their file. It just isn't possible for the receiver and the sender to predict the average.

Here's a simple solution:

1. Divide the elements in the file into blocks as before. They can either be contiguous groups or random selections chosen by some cryptographically secure random number generator.
2. Group the blocks into pairs.
3. Compute the averages of the elements in the blocks.
4. Compare the averages of the pairs.
5. If the difference between the averages is larger than some threshold, ($|B_1 - B_2| > T$), throw out the pair and ignore it.

6. If the difference is smaller than the threshold, keep the pair and encode information in it. Let a pair where $B_1 > B_2$ signify a 1 and let a pair where $B_1 < B_2$ signify a 0. Add or subtract small amounts from individual elements to make sure the information is correct. To add some error resistance, make sure that $|B_1 - B_2| > S$, where S is a measure of the signal strength.

Throwing out wildly different pairs where the difference is greater than T helps ensure that the modifications will not be too large. Both the sender and the receiver can detect these wildly different blocks and exclude them from consideration.

Again, choosing the correct sizes for the blocks, the threshold T and the signal strength S requires a certain amount of artistic sense. Bigger blocks mean more room for the law of averages to work at the cost of greater bandwidth. Decreasing the value of T reduces the amount of changes that might need to be made to the data in order to encode the right value at the cost of excluding more information.

One implementation based on the work of Brian Chen and Greg Wornell hid one bit in every 8×8 block of pixels.[CW00, CW99, Che00, CMBF08] It compared the 64 pixels in each block with a reference pattern by computing a weighted average of the difference. Then it added or subtracted enough to each of the 64 pixels until the comparison came out to be even or odd. Spreading the changes over the 64 pixels reduces the distortion.

This reference pattern acted like a key for the quantization by weighting it. Let this reference pattern be a matrix, $w_{i,j}$, and the data be $m_{i,j}$. To quantize each block, compute:

$$x = \sum_{i,j} w_{i,j} m_{i,j}.$$

Now, replace x with a quantized value, q, by setting $m_{i,j} = m_{i,j} - w_{i,j}(x - q)/b$, where b is the number of pixels or data points in the block. In the 8×8 example above, b would be 64. The distortion is spread throughout the block and weighted by the values of $w_{i,j}$.

The reference pattern does not need to be the same for each block and it doesn't need even to be confined to any block-like structure. In the most extreme case, a cryptographically secure pseudo-random stream could be used to pick the data values at random and effectively split them into weighted blocks.

14.4.1 Minimizing Quantization Errors

In one surprising detail, the strategy of increasing the block size begins to fail when the data values are tightly quantized. Larger blocks

mean the information can be spread out over more elements with smaller changes. But after a certain point, the changes become too small to measure. Imagine, for instance, the simplest case of a grayscale image, where the values at each pixel range from 0 to 255. Adding a small amount, say $.25$, is not going to change the value of any pixel at all. No one is going to be able to recover the information because no pixel is actually changed by the small additions.

This problem can be fixed with this algorithm. Let S be the amount to be spread out over all of the elements in the block. If this was spread out equally, the amount added to each block would be less than the quantized value.

While S is greater than zero, repeat the following:

1. Choose an element at random.
2. Increase the element by one quanta. That is, if it is a simple linearly encoded data value like a pixel, add one quanta to it. If it is a log-encoded value like an element in a sound file, select the next largest quantized value.
3. Subtract this amount from S.

The average of all of the elements in the block will still increase, but only a subset of the elements will change.

14.4.2 Perturbed Quantization

Another solution to minimizing the changes to an image is to choose the pixels (or other elements) that don't show the change as readily as the others. Here's a basic example. Imagine that you're going to turn a nice grayscale image into a black and white image by replacing each gray value between 0 and 255 with a single value, 0 or 255. This process, often called *half-toning*, was used frequently when newspapers could only print simple dots.

The easiest solution is to round off the values, turning everything between 0 and 127 into a 0 and everything between 128 and 255 into a 255. In many cases, this will be a good approximation. Replacing a 232 with a 255 doesn't change the result too much. But there may be a large collection of gray pixels that are close to the midpoint. They could be rounded up or down and be just as inaccurate. The *perturbed quantization* algorithms developed by Jessica J. Fridrich, Miroslav Goljan, Petr Lisonek and David Soukal focus on just these pixels. [FGS04, FGLS05, Spi05]

The problem is that while the sender will know the identities of these pixels, the recipient won't be able to pick them out. There's no

way for the decoder to know which pixels could have been rounded up *or* down. The trick to the algorithms is to use a set of linear equations and working with the perturbable pixels to ensure that the equations produce the right answer. If the sender and the receiver both use the same equations, the message will get through.

These equations are often called wet paper codes and are based on solutions developed to work with bad memory chips with stuck cells. [FGS04]

Let there be n pixels in the image. Here's an algorithm for a way to change the values that are close in order to hide a message, $m_0, m_1, m_2, \ldots$:

1. Run your quantization algorithm to quantize all of the pixels. This will produce a set of bits, b_i, for $0 \leq i < n$. In our black and white example, the value of b_i is the value after the rounding off. In practice, it might be the least-significant bit of the quantization. When these bits are places in a vector, they are called b.

2. Examine the quantization algorithm and identify k bits that are close and could be quantized so that b_i could be either a 0 or a 1. In our black and white example, these might be gray values of $128 \pm \epsilon$. The number of bits that are close enough, that is within $\pm\epsilon$, determine the capacity of the image.

3. Construct a $k \times n$ binary matrix, D. This might be shared in public, computed in advance, or constructed with a cryptographically secure pseudo-random stream dictated by a shared password.

4. Arrange for row i of this matrix to encode message m_i. Since this is a binary matrix with bit arithmetic, this means that the n entries in row i select a subset of the pixels in the image. When the bits from these pixels are added up, they should equal m_i. In other words,
$$Db = m$$

5. The problem is that Db will not equal m unless we arrange for some of the b values to change. If we could change all of the bits in b, then we could simply invert D and compute $D'm$. If we can only change k values, we still produce the right answer. In other words, we need to find some slightly different values, $\hat{b}$, where $\hat{b}_i = b_i$ when b_i is not one of the k values that can be changed. On average, only 50% of the k values will actually be changed.

6. Gaussian elimination can solve these equations and find the values for $\hat{b}$.

7. Change the values of the k changeable bits so that they're equal to $\hat{b}$. The recipient will be able to use D to find m without knowing which of the bits were actually changed.

In practice, it often makes sense to work with subsets of the image when the implementation computes $\hat{b}$ in $O(k^3)$ time.

14.5 Fast Fourier Solutions

Many of the spread-spectrum solutions use a branch of mathematics known *Fourier Analysis* or a more modern revision of this known as *Wavelet Analysis*. The entire branch is based on the work of Jean-Baptiste Fourier, who came up with a novel way of modeling a functions using a set of sine and cosine functions. This decomposition turned out to be quite useful for finding solutions to many differential equations, and it is often used to solve engineering, chemistry and physics problems.

The mechanism he proposed is also quite useful for steganography because it provides a basic way to embed several signals together in a larger one. The huge body of scholarship devoted to the topic makes it easier to test theories and develop tools quickly. One of the greatest contributions, the so-called *Fast Fourier Transform* is an algorithm optimized for the digital data files that are often used to hide information today.

The basic idea is to take a mathematical function, f, and represent it as the weighted sum of another set of functions, $\alpha_1 f_1 + \alpha_2 f_2 \alpha_3 f_3 \ldots$. The choice of the values of f_i is something of an art and different choices work better for solving different problems. Some of the most common choices are the basic harmonic functions like sine and cosine. In fact, the very popular *discrete cosine transform* which is used in music compression functions like the MP3 and the MPEG video compression functions uses $f_1 = cos(\pi x)$, $f_2 = cos(2\pi x)$, $f_3 = cos(3\pi x)$, etc. (Figure 14.6 shows the author's last initial recoded as a discrete cosine transform.) Much research today is devoted to finding better and more sophisticated functions that are better suited to particular tasks. The section on wavelets (Section 14.8) goes into some of the more common choices.

Much of the mathematical foundation of Fourier Analysis is aimed at establishing several features that make them useful for different problems. The cosine functions, for instance, are just one set that is *orthogonal*, a term that effectively means that the set is as efficient as possible. A set of functions is orthogonal if no one function, f_i, can

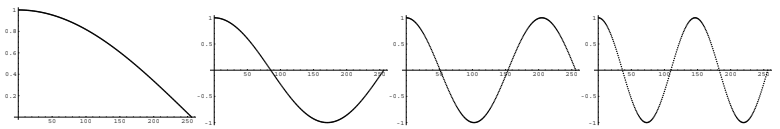


Figure 14.5: The first four basic cosine functions used in Fourier series expansions: $\cos(x), \cos(2x), \cos(3x)$ and $\cos(4x)$.

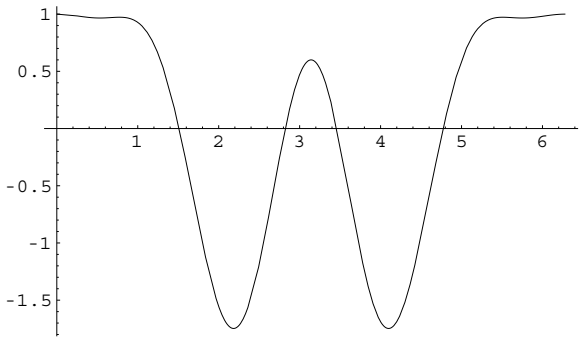


Figure 14.6: The author's last initial recreated as a Fourier series adding together the four functions shown in Figure 14.5: $1.0\cos(x) + .5\cos(2x) - .8\cos(3x) + .3\cos(4x)$.

be represented as the sum of the others. That is, there are no values of $\{\alpha_1, \alpha_2, \ldots \alpha_{i-1}, \alpha_{i+1}, \ldots\}$ that exist so that

$$f_i = \sum_{i \neq j} \alpha_j f_j.$$

The set of cosine functions also forms a *basis* for the set of sufficiently continuous functions. That is, for all sufficiently continuous functions, f, there exists some set of coefficients, $\{\alpha_1, \alpha_2, \ldots\}$, such that

$$f = \sum \alpha_j f_j.$$

The fact that the set of cosine functions is both orthogonal and a basis means that there is only one unique choice of coefficients for each function. In this example, the basis must be infinite to represent all sufficiently continuous functions, but most discrete problems never require such precision. For that reason, a solid discussion of what it means to be "sufficiently continuous" is left out of this book.

Both of these features are important to steganography. The fact that each function can be represented only by a unique set of values $\{\alpha_1, \alpha_2, \ldots\}$ means that both the encoder and the decoder will be

working with the same information. The fact that the functions form a basis mean that the algorithm will handle anything it encounters. Of course, both of these requirements for the set of functions can be relaxed, as they are later in the book, if other steps are taken to provide some assurance.

Incidentally, the fact that there are many different basis functions means that there can be many different unique representations of the data. There is no reason why the basis functions can't be changed frequently or shared between sender and receiver. A key could be used to choose a particular basis function and this could hamper the work of potential eavesdroppers. [FBS96]

14.5.1 Some Brief Calculus

The foundation of Fourier analysis lies in calculus, so a brief introduction is provided in the original form. If we limit $f(x)$ to the range $0 \leq x \leq 2v$, then the function f can be represented as the infinite series of sines and cosines:

$$f(x) = \frac{c_0}{2} + \sum_{j=-\infty}^{\infty} c_j \sin(\frac{j\pi x}{v}) + d_j \cos(\frac{j\pi x}{v}).$$

Fourier developed a relatively straight forward solution for computing the values of c_j and d_j, again represented as integrals:

$$c_j = \frac{1}{v}\int_0^{2v} f(x)\cos(\frac{j\pi x}{v})dx \quad d_j = \frac{1}{v}\int_0^{2v} f(x)\sin(\frac{j\pi x}{v})dx$$

The fact that these functions are orthogonal is expressed by this fact:

$$\int \cos(i\pi x)\cos(j\pi x)dx = 0, \forall i \neq j.$$

The integral is 1 if $i = j$.

In the past, many of these integrals were not easy to compute for many functions, f, and entire branches of mathematics developed around finding results. Today, numerical integration can solve the problem easily. In fact, with numerical methods it is much easier to see the relationship between the functional analysis done here and the vector algebra that is its cousin. If the function f is only known at a discrete number of points $\{x_1, x_2, x_3, \ldots, x_n\}$, then the equations for c_j and d_j look like dot products:

$$c_j = \sum_{i=1}^{n} f(x_i)\cos(j\pi x_i/v) \quad d_j = \sum_{i=1}^{n} f(x_i)\sin(j\pi x_i/v).$$

Discrete approaches are almost certainly going to be more interesting to modern steganographers because so much data is stored and transported in digital form.

14.6 The Fast Fourier Transform

The calculus may be beautiful, but digital data doesn't come in continous functions. Luckily, mathematicians have found versions of the equations suitable for calculation. In fact, the version for discrete data known as the Fast Fourier Transform (FFT) is the foundation for many of the digital electronics used for sound, radio, and images. Almost all multimedia software today uses some form of the FFT to analyze data, find the dominant harmonic characteristics, and then use this information to enhance or perhaps compress the data. Musicians use FFT-based algorithms to add reverb, dampen annoying echoes, or change the acoustics of the hall where the recording was made. Record companies use FFTs to digitize music and store it on CDs. Teenagers use FFTs again to convert the music into MP3 files. The list goes on and on.

The details behind the FFT are beyond the scope of this book. The algorithm uses a clever numerical juggling routine often called a "butterfly algorithm" to minimize the number of multiplications. The end result is a long vector of numbers summarizing the strength of various frequencies in the signal.

To be more precise, an FFT algorithm accepts a vector of n elements, $\{x_0, x_1 \ldots x_{n-1}$, and returns another vector of n elements $\{y_0, y_1 \ldots y_{n-1}$, where

$$y_s = \frac{1}{\sqrt{n}} \sum_{r=1}^{n} x_r e^{2\pi i(r-1)(s-1)/n}.$$

This equation is used by Mathematica and its Wavelet Explorer package, the program that created many of the pictures in this section. Others use slight variations designed to solve particular problems.

The vector that emerges is essentially a measure of how well each function matches the underlying data. For instance, the fourth element in the vector measures how much the graph has in common with $\cos(4 \times 2\pi x) + i\sin(4 \times 2\pi x)$.

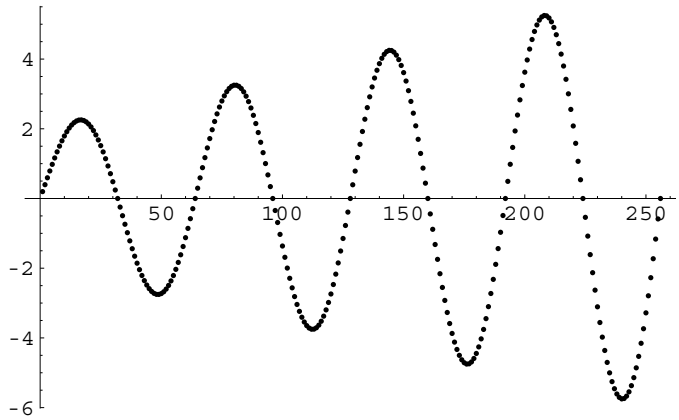


Figure 14.7: 256 points calculated from the equation $(2 + \frac{x}{64}) \sin(4 \times 2\pi x/256)$.

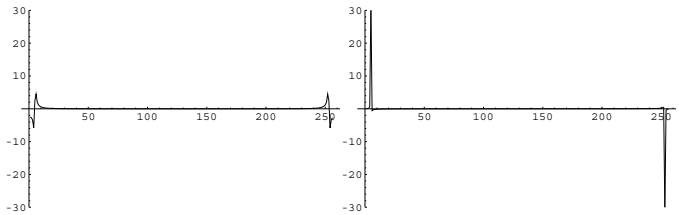


Figure 14.8: A graph of the real and imaginary parts of the Fourier transform computed from the data in Figure 14.7.

Figure 14.7 shows a graph of the function $(2+\frac{x}{64}) \sin(4\times 2\pi x/256)$. If the 256 points from this example are fed into a Fourier transform, the result has its largest values at the y_4 and the y_{251} position. The second spike is caused by aliasing. The values in the first $\frac{n}{2}$ elements report the Fourier transform of the function computed from left to right and the second $\frac{n}{2}$ elements carry the result of computing it from right to left. The results are mirrors.

Figure 14.8 shows the real and imaginary parts of the Fourier transform applied to the data in Figure 14.7. Many physicists and electrical engineers who use these algorithms to analyze radio phenomena like to say that most of the "energy" can be found in the imaginary part at y_4 and y_{251}. Some mathematicians talk about how Figure 14.8 shows the "frequency space", while Figure 14.7 shows the "function space". In both cases, the graphs are measuring the amount that the data can be modeled by each element.

This basic solution from the Fourier transform includes both real and imaginary values, something that can be confusing and unnecessary in many situations. For this reason, many also use the *discrete cosine transform* and its cousin the *discrete sine transform*. Both have their uses, but sine transforms are less common because they do a poor job of modeling the data near the point $x = 0$ when $f(0) \neq 0$. In fact, most users choose their modeling functions based on their performance near the endpoints.

Figure 14.5 shows the first four basis functions from one common set used for the discrete cosine transform:

$$\sqrt{\frac{2}{n}}\cos(\frac{\pi}{n}(k+\frac{1}{2})x), k = 0, 1, 2, 3, \ldots$$

Another version uses the similar version with a different value at the endpoint:

$$\sqrt{\frac{2}{n}}\cos(\frac{k\pi}{n}x), k = 0, 1, 2, 3, \ldots$$

Each of these transforms also has an inverse operation. This is useful because many mathematical operations are easier to do "in the frequency space". That is, the amount of energy in each frequency in the data is computed by constructing the transforms. Then some basic operations are done on the frequency coefficients, and the data is then restored with the inverse FFT or DCT.

Figure 14.9 shows the 64 different two-dimensional cosine functions used as the basis functions to model 8×8 blocks of pixels for JPEG and MPEG compression.

Smoothing data is one operation that is particularly easy to do with the FFT and DCT— if the fundamental signal is repetitive. Figure 14.10 shows the four steps in smoothing data with the Fourier transform. The first graph shows the noisy data. The second shows the absolute value of the Fourier coefficients. Both y_4 and y_{251} are large despite the effect of the noise. The third graph shows the coefficients after all of the small ones are set to zero. The fourth shows the reconstructed data after taking the inverse Fourier transform. Naturally, this solution works very well when the signal to be cleaned can be modeled well by sine and cosine functions. If the data doesn't fit this format, then there are usually smaller distinctions between the big and little frequencies making it difficult to remove the small frequencies.

14.7 Hiding Information with FFTs and DCTs

Fourier transforms provide ideal ways to mix signals and hide information by changing the coefficients. A signal that looks like $cos(4\pi x)$

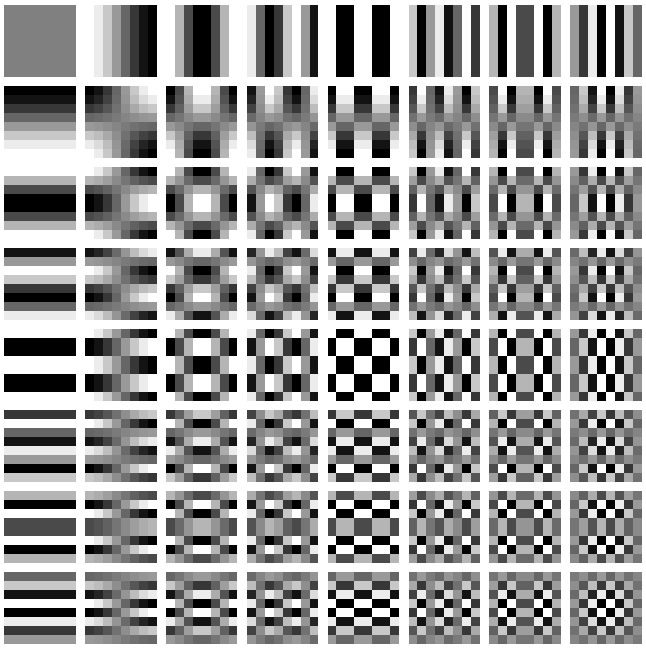


Figure 14.9: The 64 two-dimensional basis functions used in the two-dimensional discrete cosine transform of an 8×8 grid of pixels. The intensity at each particular point indicates the size of the function.

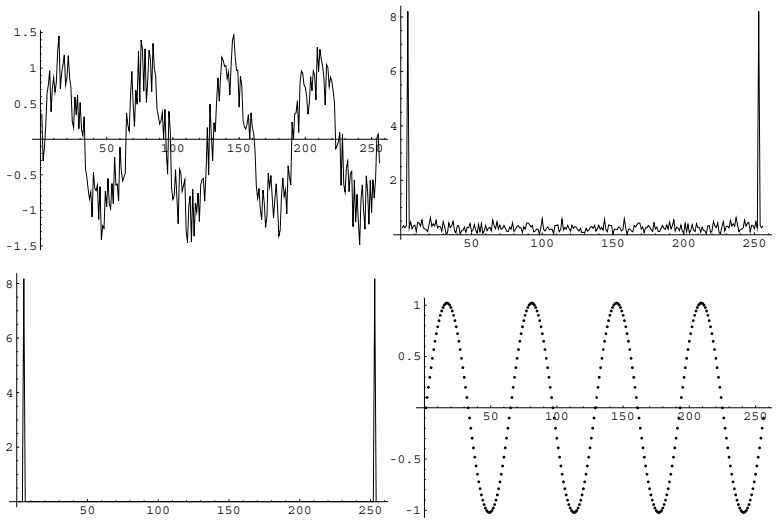


Figure 14.10: The four steps in smoothing noisy data with the FFT. The first graph shows the data; the second, the result of computing the FFT; the third, data after small values are set to zero; and fourth, the final result.

can be added by just increasing the values of y_4 and y_{251}. The more difficult challenge is doing this in a way that is hard to detect and resistant to changes to the file that are either malicious or incidental.

Here's an example. Figure 14.11 shows the absolute value of the first 600 coefficients from the Fourier transform of the voice signal shown in Figure 14.3. The major frequencies are easy to identify and change if necessary.

A simple watermark or signal can be inserted by changing setting $y_{300} = 100000$. The result after taking the inverse transform looks identical to Figure 14.3 at this level of detail. The numbers still range from $-30,000$ to $30,000$. The difference, though small, can be seen by subtracting the original signal from the watermarked one. Figure 14.12 shows that the difference oscilliates between 750 and -750 with the correct frequency.

14.7.1 Tweaking a Number of Coefficients

Ingemar Cox, Joe Kilian, Tom Leighton and Talal Shamoon [CKLS96] offer a novel way to hide information in an image or sound file by tweaking the k largest coefficients of an FFT or a DCT of the data. Call these $\{y_1, y_2, \ldots y_{k-1}\}$. The largest coefficients correspond to the

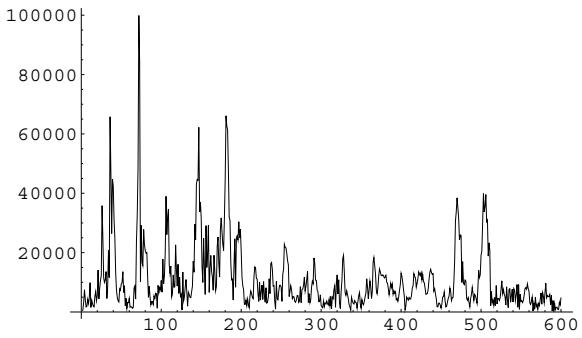


Figure 14.11: The first 600 coefficients from the Fourier transform of Figure 14.3.

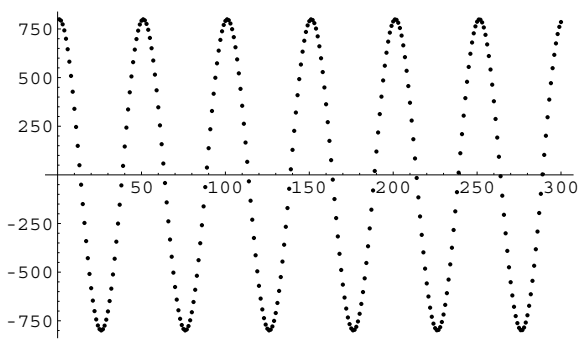


Figure 14.12: The result of subtracting the original signal shown in Figure 14.3 from the signal with an inflated value of y_{300}.

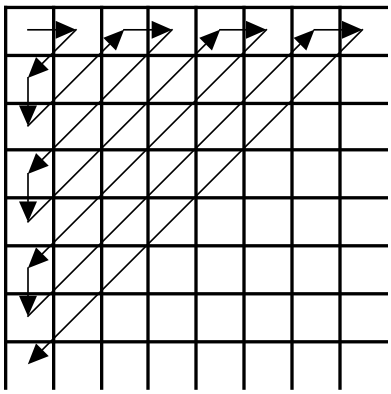


Figure 14.13: Another solution is to order the coefficients in significance. This figure shows the popular zig-zag method to order coefficients from a two-dimensional transform. The ones in the upper lefthand corner correspond to the lowest frequencies and are thus the most significant for the eye.

most significant parts of the data stream. They are the frequencies that have the most "energy" or that do the most for carrying the information about the final image.

Cox and colleagues suggest that hiding the information in the largest coefficients may sound counterintuitive, but it is the only choice. At first glance, the most logical place to hide the data is in the noise— that is, the smallest coefficients. But this noise is also the most likely to be modified by compression, printing, or using a less than perfect conversion process. The most significant parts of the signal, on the other hand, are unlikely to be damaged without damaging the entire signal.

This philosophy has many advantages. The data is spread out over numerous data elements. Even if several are changed or deleted, the information can be recovered. Cox and colleagues demonstrate that the images carrying this watermark can survive even after being printed and scanned in again. Of course, the bandwidth is also significantly smaller than other solutions like tweaking the least signficant bit.

Their algorithm uses these steps:

1. Use a DCT or FFT to analyze the data.

2. Choose the k largest coefficients and label them $\{y_0, y_1, y_2, \ldots y_{k-1}\}$ for simplicity. The smaller coefficients are ignored. The first co-

efficients representing the smallest frequencies will often be in this set, but it isn't guaranteed.

Another solution is to order the coefficients according to their visual significance. Figure 14.13 shows a zig-zag ordering used to choose the coefficients with the lowest frequencies from a two-dimensional transform. The JPEG and MPEG algorithms use this approach to eliminate unnecessary coefficients. Some authors suggest skipping the first l coefficients in this ordering because they have such a big influence on the image. [PBBC97] Choosing the next k coefficients produces candidates that are important to the description of the image but not too important.

3. Create a k-element vector, $\{b_1, b_1, b_2, \ldots b_{k-1}\}$, to be hidden. These can be either simple bits or more information rich real numbers. This information will not be recovered intact in all cases, so it should be thought of more as an identification number, not a vector of crucial bits.

4. Choose α, a coefficient that measures the strength of the embedding process. This decision will probably be made via trial and error. Larger values are more resistant to error, but they also introduce more distortion.

5. Encode the bit vector in the data by modifying the coefficients with one of these functions they suggest:

   - $y_i' = y_i + \alpha b_i$
   - $y_i' = y_i(1 + \alpha b_i)$
   - $y_i' = y_i e^{\alpha b_i}$

6. Compute the inverse DCT or FFT to produce the final image or sound file.

The existence of the embedded data can be tested by reversing the steps. This algorithm requires the presence of the original image, a problem that severely restricts its usefulness in many situations. The steps are:

1. Compute the DCT or FFT of the image.

2. Compute the DCT or FFT of the original image without embedded data.

3. Compute the top k coefficients.

4. Use the appropriate formula to extract the values of αb_i.

5. Use the knowledge of the distribution of the random elements, b_i, to normalize this vector. That is, if the values of b_i are real numbers chosen from a normal distribution around .5, then determine which value of α moves the average to $.5$. Remove α from the vector through division.

6. Compare the vector of $\{b_0, b_1, b_2, \ldots b_{k-1}\}$ to the other known vectors and choose the best match.

The last step for identifying the "watermark" is one of the most limiting for this particular algorithm. Anyone searching for it must have a database of all watermarks in existence. The algorithm usually doesn't identify a perfect match because roundoff errors add imprecision even when the image file is not distorted. The process of computing the DCTs and FFTs introduces some roundoff errors and encapsulating the image in a standard 8-bit or 24-bit format adds some more. For this reason, the best we get is the most probable match.

This makes the algorithm good for some kinds of watermarks but less than perfect for hidden communication. The sender and receiver must agree on both the cover image and some code book of messages or watermarks that will be embedded in the data.

If these restrictions don't affect your needs, the algorithm does offer a number of desirable features. Cox and colleagues tested the algorithm with a number of experiments that proved its robustness. They began with several 256×256 pixel images, distorted the images and then tested for the correct watermark. They tried shrinking the size by a factor of $\frac{1}{2}$, using heavy JPEG compression, deleting a region around the outside border, and dithering it. They even printed the image, photocopied it, and scanned it back in without removing the watermark. In all of their reported experiments, the algorithm identified the correct watermark, although the distortions reduced the strength.

The group also tested several good attacks that might be mounted by an attacker determined to erase the information. First, they tried watermarking the image with four new watermarks. The final test pulled out all five, although it could not be determined which were the first and the last. Second, they tried to average together five images created with different watermarks and found that all five could still be identified. They indicate, however, that the algorithm may not be as robust if the attacker were to push this a bit further, say by using 100 or 1000 different watermarks.

14.7.2 Removing the Original from the Detection Process

Keeping the original unaltered data on hand is often unacceptable for applications like watermarking. Ideally, an average user will be able to extract the information from the file without having the original available. Many of the watermarking applications assume that the average person can't be trusted with unwatermarked data because they'll just pirate it.

A variant of the previous algorithm from Cox et al. does not require the original data to reveal the watermark. A. Piva, M. Barni, F. Bartolini, and V. Cappellini produced a similar algorithm that sorts the coefficients to the transform in a predictable way. Figure 14.13, for instance, shows a zig-zag pattern for ordering the coefficients from a two dimensional transform according to their rough frequency. If this solution is used, there is no need to keep the original data on hand to look for the k most significant coefficients. Many other variants are emerging.

14.7.3 Tempering the Wake

Inserting information by tweaking the coefficients can sometimes have a significant effect on the final image. The most fragile sections of the image are the smooth, constant patches like pictures of a clear, blue, cloudless summer sky. Listeners can often hear changes in audio files with pure tones or long quiet segments. Data with rapidly changing values mean there is plenty of texture to hide information. Smooth, slowly changing data means there's little room. In this most abstract sense, this follows from information theory. High entropy data is a high bandwidth channel. Low entropy data is a low bandwidth channel.

Some algorithms try to adapt the strength of a watermark to the underlying data by adjusting the value of α according to the camouflaging data. This means the strength of the watermark becomes a function of location ($\alpha_{i,j}$) in images and of time (α_t) in audio files.

There are numerous ways to calculate this value of α, but the simplest usually suffice. Taking a window around the point in space or time and averaging the deviation from the mean is easy enough. [PBBC97] More sophisticated studies of the human perception system may be able to provide a deeper understanding of how our eyes and ears react to different frequencies.

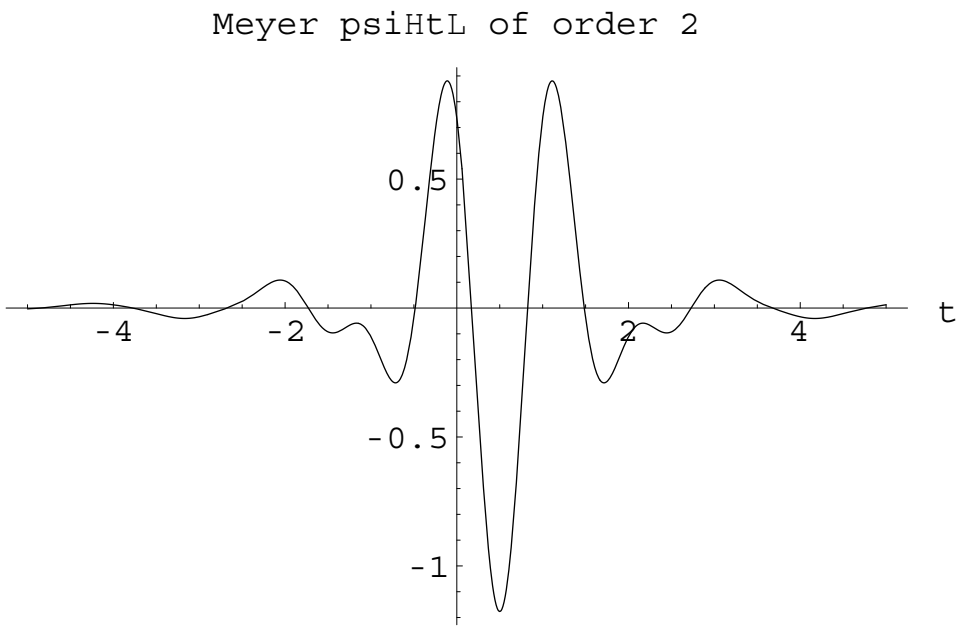


Figure 14.14: The Meyer ψ function.

14.8 Wavelets

Many of the algorithms use sines and cosines as the basis for constructing models of data, but there is no reason why the process should be limited to them alone. In recent years, researchers began devoting new energy to exploring how strange and different functions can make better models of data– a field the researchers call *wavelets*. Figure 14.14 shows one popular wavelet function, the Meyer ψ function.

Some of the steganography detection algorithms examine the statistics of wavelet decompositions. See Section 17.6.1.

Wavelet transforms construct models of data in much the same way Fourier transforms or Cosine transforms do — they compute coefficients that measure how much a particular function behaves like the underlying data. That is, the computation finds the correlation. Most wavelet analysis, however, adds an additional parameter to the mix by changing both the frequency of the function and the location or window where the function is non-zero. Fourier transforms, for instance, use sines and cosines that are defined from $-\infty$ to $+\infty$. Wavelet transforms restrict the influence of each function by sending the function to zero outside a particular window from a to b.

Using these localized functions can help resolve problems that occur when signals change over time or location. The frequencies in audio files containing music or voice change with time and one of the popular wavelet techniques is to analyze small portions of the file. A wavelet transform of an audio file might first use wavelets defined between 0 and 2 seconds, then use wavelets defined between 2 and 4 seconds, etc. If the result finds some frequencies in the first window but not in the second, then some researchers say that the wavelet transform has "localized" the signal.

Roz Chast's collection, Theories of Everything *lists books "not by the same author" including* Nothing But Cosines.

The simplest wavelet transforms are just the DCT and FFT computed on small windows of the data. Splitting the data into smaller windows is just a natural extension of these algorithms.

More sophisticated windows use multiple functions defined at multiple sizes in a process called *multi-resolution analysis*. The easiest way to illustrate the process is with an example. Imagine a sound file that is 16 seconds long. In the first pass, the wavelet transform might be computed on the entire block. In the second pass, the wavelet transform would be computed on two blocks between 0 and 7 seconds and between 8 and 15 seconds. In the third pass, the transform would be applied to the blocks 0 to 3,4 to 7, 8 to 11, and 12 to 15. This is three stage, multi-resolution analysis. Clearly, it is easier to simply divide each window or block by two after each stage, but there is no reason why extremely complicated schemes with multiple windows overlapping at multiple sizes can't be dreamed up.

Multiple resolution analysis can be quite useful for compression, a topic that is closely related to steganography. Some good wavelet-based compression functions use this basic recursive approach:

1. Use a wavelet transform to model the data on a window.
2. Find the largest and most significant coefficients.
3. Construct the inverse wavelet transform for these large coefficients.
4. Subtract this version from the original. What is left over is the smaller details that couldn't be predicted well by the wavelet transform. Sometimes this is significant and sometimes it isn't.
5. If the differences are small enough to be perceptually insignificant, then stop. Otherwise, split the window into a number of smaller windows and recursively apply this same procedure to the leftover noise.

This recursive, multi-resolution analysis does a good job of compressing many images and sound files. The wider range of choices

in functions means that the compression can be further tuned to extract the best performance. There are many different wavelets available and some are better at compressing some files than others. Choosing the best one is often as much an art as a science.

In some cases, steganographers suggest that the choice of the function can also act like a key if only the sender and the receiver know the particular wavelet.

Are highly tuned wavelets more or less stable for information encoding? That is, can a small change in a coefficient be reliably reassembled later, even after printing and scanning? In other words, how large must the α term be?

It is not possible to go into the wavelet field in much depth here because it is more complex and not much of this complexity affects the ability to hide information.

Most of the same techniques for hiding information with DCTs and DFTs work well with DWTs. In some cases, they outperform the basic solutions. It is not uncommon to find that information hidden with DWTs does a better job of surviving wavelet-based compression algorithms than information hidden with DCTs or DFTs. [XBA97] Using the same model for compression and information hiding works well. Of course, this means that an attacker can just choose a different compression scheme or compress the file with a number of schemes in the hope of foiling one.

14.9 Modifications

The basic approach to hiding information with sines, cosines or other wavelets is to transform the underlying data, tweak the coefficients, and then invert the transformation. If the choice of coefficients is good and the size of the change is manageable, then the result is pretty close to the original.

There are a number of variations on the way to choose the coefficients and encode some data in the ones that are selected. Here are some of the more notable:

Identify the Best Areas Many algorithms attempt to break up an image or sound file and identify the best parts for hiding the information. Smooth, stable regions turn mottled or noisy if coefficients are changed even a small amount.

Multi-resolution wavelet transforms are a good tool for identifying these regions because they recursively break up an image until a good enough model is found. Smooth, stable sections are often modeled on a large scale, while noisy, detailed sections get broken up multiple times. The natural solution is to hide the information in the coefficients that model the smallest, most detailed regions. This confines the changes to the

edges of the objects in images or the transitions in audio files, making it more difficult for the human perception system to identify them. [AKSK00]

Quantize the Coefficients to Hide Information Many of the transform hiding methods hide information by adding in a watermark vector. The information is extracted by comparing the coefficients with all possible vectors and choosing the best match. This may be practical for small numbers of watermarks, but it doesn't work well for arbitrary blocks of information.

A more flexible solution is to tweak the coefficients to hide individual bits. Let Q be some quantization factor. An arbitrary coefficient, y_i, is going to fall between $aQ \leq y_i \leq (a+1)Q$ for some integer, a. To encode a bit, round off y_i to the value where the least significant bit of a is that bit. For example, if the bit to be encoded is zero and $a = 3$, then set $y_i = (a+1)Q = 4Q$. [KH98]

Any recipient would extract a bit from y_i by finding the closest value of aQ. If the transform process is completely accurate, then there will be some integer where $aQ = y_i$. If the transform and inverse transform introduce some rounding errors, as they often do, then y_i should still be close enough to some value of aQ— if Q is large enough.

The value of Q should be chosen with some care. If it is too large, then it will lead to larger changes in the value of y_i. If it is too small, then it may be difficult to recover the message in some cases when error intrudes.

Deepa Kundur and Dimitrios Hatzinakos describe a quantization-based watermark that also offers tamper detection. [KH99]

This mechanism also offers some ability to detect tampering with the image or sound file. If the coefficients are close to some value of aQ but not exactly equal to aQ, then this might be the result of some minor changes in the underlying file. If the changes are small, then the hidden information can still be extracted. In some cases, the tamper detection can be useful. A stereo or television may balk at playing back files with imperfect watermarks because they would be evidence that someone was trying to destroy the watermark. Of course it could also be the result of some imperfect copying process.

Hide the Information in the Phase The Discrete Fourier Transform produces coefficients with a real and an imaginary value. These complex values can also be imagined in polar coordinates as having a magnitude and an angle. (If $y_i = a + bi$, then $a =$

$mcos(\theta)$ and $b = msin(\theta)$, where m is the magnitude and θ is the angle.) Many users of the DFT feel that the transform is more sensitive to changes made in the angles of the coefficient than changes made in the magnitude of the coefficients. [RDB96, LJ00]

This method is naturally adaptive to the size of the coefficients. Small values are tweaked a small amount if they're rotated $\theta + \psi$ degrees. Large values are tweaked a large amount.

Changing the angle this way requires a bit of attention to symmetry. When the input to a DFT are real values, as they almost are in steganographic examples, then the angles are symmetric. This symmetry must be preserved to guarantee real values will emerge from the inverse transform.

Let $\theta_{i,j}$ stand for the angle of the coefficient (i, j). If ψ is added to $\theta_{i,j}$, then $-\psi$ must be added to $\theta_{m-i,n-j}$, where (m, n) are the dimensions of the image.

14.10 Summary

Spreading the information over a number of pixels or units in a sound file adds more security and intractability. Splitting each bit of information into a number of pieces and distributing these pieces throughout a file reduces the chance of detection and increases resistance to damage. The more the information is spread throughout the file, the more redundancy blocks attacks.

Many solutions use well-understood software algorithms like the Fourier transform. These tools are usually quite popular in techniques for adding watermarks because many compression algorithms use the same tools. The watermarks are usually preserved by compression in these cases because the algorithms use the same transforms.

The Disguise Information is spread throughout a file by adding small changes to a number of data elements in the file. When all of the small changes are added up, the information emerges.

How Secure Is It? The changes are small and distributed so they can be more secure than other solutions. Very distributed information is resistant to attack and change because an attacker must destroy enough of the signal to change it. The more places the information is hidden, the harder it is for the attacker to locate it and destroy it.

The system can be made much more secure by using a key to create a pseudo-random bit stream that is added into the data as it is mixed into the mix. Only the person with the key can remove this extra encryption.

How to Use It In the most abstract sense, just choose a number of locations in the file with a pseudo-random bit stream, break the data into little parts, and add these parts into all of the locations. Recover the message by finding the correct parts and adding them together.

This process is very efficient if it is done with a fast Fourier transform. In these cases, the data can be hidden by tweaking the coefficients after the transform. This can add or subtract different frequency data.

Further Reading

Using wavelets, cosines or other functions to model sound and image data is one of the most common strategies for watermarking images today. There are too many good papers and books to list all of them. Some suggestions are:

- *Digital Watermarking* by Ingemar Cox, Matthew L. Miller and Jeffrey A. Bloom is a great introduction to using wavelet techniques to embed watermarks and hide information. The 2007 edition is broader and it includes a good treatment of steganography and steganalysis. [CMB02, CMB07]
- The proceedings of the International Workshop on Digital Watermarking are invaluable sources that track the development of steganography using wavelets and other functional decomposition. [PK03, KCR04, CKL05, SJ06]
- A good conference focused on watermarking digital content is the Security, Steganography, and Watermarking of Multimedia Contents. [DW04, DW05]
- Jessica J. Fridrich, Miroslav Goljan, Petr Lisonek and David Soukal discuss the use of Michael Luby's LT Codes as a foundation for building better versions of perturbed quantization. These graph-based codes can be faster to compute than the matrix-based solutions.[Lub02, FGLS05]

Chapter 15

Synthetic Worlds

15.1 Slam Dunks

The Play By Play Man and the Color Man work through a way of encoding messages in an athletic contest.

PBPM: Things are looking a bit difficult for the Montana Shot Shooters. They had a lead of 14 points at the halftime, but now the Idaho Passmakers have hit two three-point shots in a row. Whammo! They're back in the game. The Shot Shooters have called a time out to regroup.

CM: Putting six points on the board that way really sends a message. They made it look easy.

PBPM: Those two swishes announced, "We're still here. You can't beat us that easily. We've got pride, intestinal fortitude and pluck." The emphasis is on pluck.

CM: And composure too. They whipped the ball around the key. They signaled, "We can move the rock and then send it home. We can pass the pill and force you to swallow it whole. We know basketball. This is our game too."

PBPM: There was even a startling subtext to the message. I believe the Passmakers were telling the Shot Shooters that this game was different from the last. Yes, the Shot Shooters beat them at home by 22 points, but that was two months ago. Now, Jimmy D's leg is better. He's quicker. The injury he sustained while drinking too much in the viscinity of a slippery pool deck is behind him. The nasty, golddigging girlfriend is history. The Passmakers are reminding the Shot Shooters that a bit of cortisone is a time proven solution for knee problems, but moving on with

your life and putting bad relationships behind you is an even better cure for the human heart. That's the message I think that is encoded in those three-point shots.

CM: We're back from the time out now. Let's see what the Shot Shooters can do.

PBPM: The Shot Shooters put the ball in play. Carter pauses and then passes the ball over halfcourt to Martin. He fakes left, goes right. It's a wide open lane. He's up and bam, bam, bam. That's quite a dunk. The Passmakers didn't even have a defense.

CM: Whoa. That sends a message right there. A dunk like that just screams, "You think three-point shots scare me? You think I care about your prissy little passing and your bouncy jump shots? There was no question where this ball was going. Nobody in the stands held their breath to see if it would go in. There was no pregnant pause, no hush sweeping the crowd, and no dramatic tension. This ball's destiny was the net and there was no question about it." He's not being steganographic at all.

15.2 Created Worlds

Many of the algorithms for sound and image files revolve around hiding information in the noise. Digitized versions of the real world often have some extra entropy waiting for a signal. But advances in computer graphics and synthesis mean that the images and sound often began life in the computer itself. They were not born of the real world and all of the natural entropy constantly oozing from the plants, the light, the animals, the decay, the growth, the erosion, the wind, the rain and who knows what else. Synthetic worlds are, by definition, perfect.

At first glance, perfection is not good for hiding information. Purely synthetic images began as mathematics and this means that a mathematician can find equations to model that world. A synthetic image of a ball in the light has a perfect gradient with none of the distortions that might be found in an image of an imperfect ball made by worn machinery and lit by a mass-produced bulb powered by an overtaxed electrical system.

These regularities make it easy for steganalysis to identify images with extra, hidden information. Even slight changes to the least significant bit become detectable. The only advantage is that the increasing complexity of the models means that any detection process must also become increasingly complex too. This does pro-

vide plenty of practical cover. Better computer graphics technology is evolving faster than any algorithm for detecting the flaws. More complicated models are coming faster than we can suss them out.

If the practical limitations aren't good enough, the models for synthesizing worlds can be deputized to carry additional information. Instead of hiding the extra information of the final image or sound file, the information can be encoded during the synthesis.

There are many opportunities to hide information. Many computer graphics algorithms use random number generators to add a few bits of imperfection and the realism that comes along with them. Any of these random number streams can be hijacked to carry data.

Another source can be found in tweaking the data used to drive the synthesis, perhaps by changing the least significant bits of the data. One version of an image may put the ball at coordinates $(1414, 221)$ and another version may put it at $(1413, 220)$. A watermarked version of a movie may encode the true owner's name in the position of one of the characters and the moment they start talking. Each version of the film will have slightly different values for these items. The rightful owner could be extracted from these subtle changes.

The program MandelSteg, developed by Henry Hastur, hides information in the least significant bit of an image of the Mandelbrot Set. This synthetic image is computed to seven bits of accuracy and then the message is hidden in the eighth. *See page 319.*

Wolfgang Funk worked through some of the details for perturbing three-dimensional models described with polynomial curves in the NURBS standard. [Fun06] The polynomials describing the objects are specified by points along the surface known as *control points*. New control points can be added without significantly changing the description of the surface if they're placed in the right spots. Old control points can usually be moved a slight amount so the shape changes a bit. Hao-Tian Wu and Yiu-ming Cheung suggest using a secret key to traverse the points describing the object and then modifying their position relative to the centroid of other points. Moving one direction by a small amount encodes a 0 and moving in the other direction encodes a 1.[WmC06]

An even more complicated location to hide the information can be found by changing the physics of the model. The acoustical characteristics of the room are easy to change slightly. The music may sound exactly the same. The musicians may start playing at exactly the same time. But the size and characteristics of the echos may change just a bit.

There are many ways that the parameters used to model the physics can be changed throughout a file. The most important challenge is guaranteeing that the changes will be detectable in the image or sound file. This is not as much of a problem as it can be for other

approaches. Many of the compression algorithms are tuned to save space by removing extraneous information. The locations of objects and the physical qualities of the room, however, are not extraneous. The timbre of the instruments and the acoustical character of the recording studio are also not extraneous. Compression algorithms that blurred these distinctions would be avoided, at least by serious users of music and image files.

Markus Kuhn and Ross Anderson suggest that tweaking the video signal sent to the monitor can send messages because the electron guns in the monitor emit so much electro-magnetic "noise".[KA98]

Designing steganographic algorithms that use these techniques can be something of an art. There are so many places to hide extra bits that the challenge is arranging for them to be found. The changes should be large enough to be detected by an algorithm but small enough to escape casual detection by a human.

15.3 Text Position Encoding and OCR

Chapters 6 and 8 show how to create synthetic textual descriptions and hide information in the process. A simpler technique for hiding information in text documents is to fiddle with the letters themselves.

Matthew Kwan developed a program called Snow which hides three bits at the end of each text line by adding between 0 *and* 7 *bits.*

One of the easiest solutions is to encode a signal by switching between characters that appear to be close to each other, if not identical. The number zero,'0', and the capital O are close to each other in many basic fonts. The number '1' and the lower-case l are also often indistinguishable. The front cover of the Pre-proceedings of the 4th Information Hiding Workshop carried a message from John McHugh. [McH01]

If the fonts are similar, information can be encoded by swapping the two versions. Detecting the difference in printed versions can be complicated because OCR programs often use context to distinguish between the two. If the number one is found in the middle of a word made up of alphanumeric characters, the programs often will fix the perceived mistake.

If the fonts are identical, the swap can still be useful for hiding information when the data is kept in electronic form. Anyone reading the file will not notice the difference, but the data will still be extractable.

This is often taken to extremes by some members of the hacker subculture who deliberately swap vaguely similar characters. The number four ('4') bears some resemblance to the capital A, the number three ('3') looks like a reversed capital E. This technique can elude keyword searches and automated text analysis programs, at least until the spelling becomes standardized and well known. Then a document with the phrase "3L33t h4XOR5" starts to look suspicious.

Humans also provide some error correction for basic spelling and grammatical errors. A hidden message can be encoded by introducing seemingly random misspellings from time to time. [1]

15.3.1 Positioning

Another possible solution is to simply adjust the positions of letters, words, lines and paragraphs. Typesetting is as much of an art as a job and much care can be devoted to algorithms for arranging these letters on a page. Information can always be hidden when making this decision.

The LaTeX and TeX typesetting systems used in creating this book justify lines by inserting more white space after a punctuation mark than after a word. American typesetters usually put three times as much space after punctuation. The French, on the other hand, avoid this distinction and set both the same. This mechanism is easy to customize and it is possible to change the "stretchability" of white space following any character. This paragraph was typeset so the whitespace after words ending in 'e' or 'r' received three times as much space with the`\sfcode` macro.

Changing these values throughout a document to smaller values is relatively simple to do. In many cases, detecting these changes is also relatively simple. Many ommercial OCR programs continue to make minor errors on a page, but steganographic systems using white space can often be more accurate. Detecting the size of the whitespace is often easier than sussing out the differences between the ink marks.

Jack Brassil, Steve Low, Nicholas Maxemchuk and Larry O'Gorman experimented with many techniques for introducing small shifts to the typesetting algorithms. [BO96, LMBO95, BLMO95, BLMO94] This can be easy to do with open source tools like TeX that also include many hooks for modifying the algorithms.

Dima Pröfrock, Mathias Schlauweg and Erika Müller suggest that actual objects in digitized video can be moved slightly to encode watermarks.[PSM06]

One of their most successful techniques is moving the individual lines of text. They successfully show that entire lines can be moved up or down one or two six-hundredths of an inch. Moving a line is easy to detect if any skew can be eliminated from the image. As long as the documents are close to horizontal alignment, the distance between the individual lines can be measured with enough precision to identify the shifted lines.

The simplest mechanism for measuring a line is to "flatten" it into one dimension. That is, count the number of pixels with ink

[1] Some might be tempted to blame me and the proofreader for any errors that crept into the text. But perhaps I was sending a secret message.

card into the smartcard programmer, it would just sit there acting like an ice scraper.

The basic ECM from DirecTV checks the software

Figure 15.1: Three lines of printed text scanned in at 400 pixels per inch.

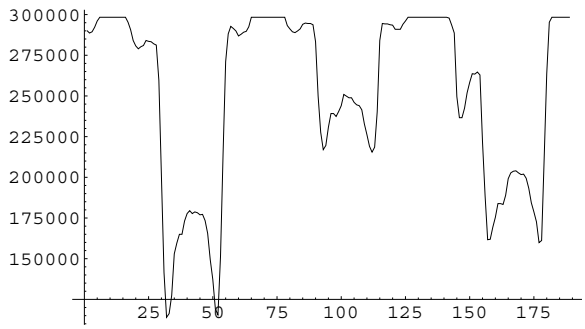


Figure 15.2: A graph of the sums of the rows in Figure 15.1. White is usually assigned 255, so the short line in the middle is less pronounced.

in each row. Figure 15.2 shows the result of summing the intensity of each pixel in the row. The maximum value of each pixel is 255 if it is completely white. For this reason, the second row has a much smaller valley because it is shorter and made up of more white space.

The structure of the peaks and valleys also varies with the words. The third row has more capital letters, so there is a more pronounced valley at the beginning corresponding to the horizontal lines in the capital letters. The choice of the font also changes the shape of this graph and can even be used to identify the font. [WH94] Generally, fonts with serifs make it easier to identify the baselines than sans-serif fonts, but this is not guaranteed.

More bits can be packed into each line by shifting individual words up or down a three hundredth of an inch. The detection process becomes more complicated because there is less information to use to measure the horizontal shift of the baseline. Short words are not as desireable as longer ones. In practice, it may make sense to group multiple small words together and shift them in a block.

Experiments by Jack Brassil and Larry O'Gorman show that the location of the baseline and any information encoded in it can be regularly extracted from text images even after repeated photocopying.

Moving the individual lines up or down by one or two six-hundredths of an inch is usually sufficient to be detected. They do note that their results require a well-oriented document where the baselines of the text are aligned closely with the raster lines. Presumably, a more sophisticated algorithm could compensate for the error by modeling the antialiasing, but it is probably simpler to just line up the paper correctly in the first place.

15.3.2 MandelSteg and Secrets

Any image is a candidate for hiding information, but some are better than others. Ordinarily, images with plenty of variation seem perfect. If the neighboring pixels are different colors, then the eye doesn't detect subtle changes in the individual pixels. This led Henry Hastur to create a program that flips the least significant bits of a Mandelbrot set. These images are quite popular and well-known throughout the mathematics community. This program, known as MandelSteg, is available with source code from the Cypherpunks archive (`ftp://ftp.csua.berkeley.edu/pub/cypherpunks/steganography/`).

The manual notes that there are several weaknesses in the system. First, someone can simply run the data recovery program, GifExtract, to remove the bits. Although there are several different settings, one will work. For this reason, the author suggests using Stealth, a program that will strip away the framing text from a PGP message, leaving only noise.

There are other weaknesses. The Mandelbrot image acts as a one-time pad for the data. As with any encoding method, the information can be extracted if someone can find a pattern in the key data. The Mandelbrot set might look very random and chaotic, but there is still plenty of structure. Each pixel represents the number of iterations before a simple equation ($f(z) = z^2 + c$) converges. Adjacent pixels often take a different number of pixels, but they are still linked by their common generating equation. For this reason, I think it may be quite possible to study the most significant bits of a fractal image and determine the location from where it came. This would allow someone to recalculate the least significant bits and extract the answer.[2]

[2]David Joyce offers a Mandelbrot image generator on the Web (`http://aleph0.clarku.edu/djoyce/julia/explorer.html`).

15.4 Echo Hiding

Hiding information in the noise of sound files is a good solution, but the information may be erased by good compression algorithms. Daniel Gruhl, Anthony Lu, and Walter Bender suggest tweaking the basic acousitics of the room to hide information. While this can still be obscured by sufficiently strong compression, it is often more likely to withstand standard algorithms. Echoes are part of recordings and sophisticated listeners with trained ears can detect small changes in them. Good recording engineers and their compressionists try to avoid eliminating the echos in an effort to provide as much verisimilitude as possible.[GLB96]

Many recording software programs already include the ability to add (or subtract) echoes from a recording. They can also change the character of the echo by twiddling with strength of the echo and the speed at which it vanishes.

Information can be included by changing either the strength or the length of the decay. Gruhl, Lu and Bender report success with encoding a single bit by changing the length of time before the echo begins. A one gets a short wait (about .001 seconds) and a zero gets a slightly longer wait (about .0013 seconds). More than one bit is encoded by splitting up the signal and encoding one bit in each segment.

The signal is detected by autocorrelation. If the audio signal is represented by $f(t)$, then the bit is extracted by computing $f(t+.001)$ and $f(t + .0013)$. Segments carrying a signal of one will generally produce a higher value of $f(t + .001)$ and segments carrying a signal of zero will produce a higher value of $f(t + .0013)$.

The bandwidth available depends on the sampling rate and a lesser amount on the audio file itself. Higher-frequency sounds and higher sampling rates can provide accurate results with shorter segments, both alone and in combination. Gruhl, Lu and Bender report success with segments lasting one-sixteenth of a second.

The success of this algorithm depends, to a large extent, on the ears listening to it. Some humans are born with good hearing, some train their ears to hear better, and some do both. The music industry continues to experiment with techniques like echo hiding to add a watermark to recordings. The results are often quite good, but still problematic. In many cases, the average human can't detect the additional echo. Many of those who do detect it think the sound is richer. Still, some of the best artists in the business often reject any change to their perfect sound. At times, this debate can ring with irony. Garage bands devoted to making noisy, feedback-rich music

sometimes complain about an tiny added bit of echo added as a watermark. This process still continues to require an artist's touch.

15.5 Summary

There is no reason to stop with just moving lines of text or adding echos. Any synthetic file can be tweaked during the construction. The real challenge is creating detection algorithms that will detect and extract the changes from the files. In some cases, the data is readily available. An animated presentation developed in Macromedia's Flash format, for instance, could encode information in the position and timing of the items. This data is easy to extract from the files using the publicly distributed information about the file format.

If the data can't be extracted from the file many, many of the techniques developed by the artificial intelligentisia for image and audio analysis can be quite useful. Machine vision algorithms, for instance, can extract the position and orientation of animated characters in a movie. Echo detection and elimination tools used by audio engineers can also help located echoes carring hidden information.

The Disguise Any synthetic file, be it text, sound, light, or maybe one day smell, can carry information by tweaking the parameters during synthesis.

How Secure Is It The security depends, to a large extent, on the nature of the files and the skill of the artist. High-quality graphics have many places where a slight twist of the head could carry several bits of information without anyone noticing.

How to Use It Any of the steganographic techniques for tweaking the least significant bits or adding in signals can be used on the raw data used to guide the synthesis. To a large extent, all of the techniques for steganography are just applied at an earlier step in the process. For instance, an animator may choose the position of a character's limbs and then feed this data to a rendering engine. The steganography is done just after the animator chooses the position but before the rendering.

Further Reading

- Siwei Lyu and Hany Farid analyzed the statistical structure of the wavelet decomposition of images and found subtle

but useful differences between the decomposition of natural scenes and the scenes produced by computer graphics. They were able to train a statistical quantifier to distinguish between the two. [Lyu05, LF05]

Chapter 16

Watermarks

16.1 A Patent for Watermarking Humans

Why should watermarks be limited to digital files and pieces of paper? Can any way of leaving a trace act like a watermark? We offer the claims to an unfiled patent application for "implanting a memetic watermark through humorous banter".

Claims:

1. A method for implanting and detecting a watermark in a human subject by

   identifying a funny sentence selected from a group of witticisms, whimsical retorts, riddles, puns, limericks, jokes, droll patter, parodistic remarks, and satirical levity;

   offering said funny sentence to said human subject in a manner designed to attract their attention and implant said funny sentence in their brain and create an instance of a memetic watermark;

   detecting the presence of said memetic watermark by repeating said funny sentence in order to analyze the response of said human subject, who will either laugh or announce that the joke was not new to them.

2. The method of claim (1) where said funny sentence is repeated and repeated until it is firmly implanted in said human subject's neural patterns.

3. The method of claim (2) where said funny sentence is repeated again and again in order to increase the reaction time and vol-

ume of the response of said human subject in order to increase the detection of said watermark.

4. The method of claim (3) where said funny sentence is repeated several more times, increasing the sensitivity in said human subject enough to prompt them to throttle the neck of the next person to repeat said funny sentence increasing further the ability to detect said watermark.

16.2 Tagging Digital Documents

One of the most demanding applications for the algorithms that hide information is protecting copyrighted information. The job requires the hidden information to somehow identify the rightful owner of the file in question and, after identifying it, prevent it from being used in unauthorized ways. This is a tall order because the content industry has great dreams for digital books, music, movies and other multimedia presentations. Putting a computer in the loop means that content producers can experiment with as many odd mechanisms for making money as they can imagine. Some suggest giving away the first $n-1$ chapters of a murder mystery and charging only for the last one with the identity of the murderers. Others propose giving people a cut when they recommend a movie to a friend and the friend buys a copy. All of these schemes depend on some form of secure copy protection and many of the dreams include hidden information and steganography.

Hiding information to protect text, music, movies, and art is usually called *watermarking*, a reference to the light image of the manufacturer's logo pressed into the paper when it was made. The term is apt because steganography can hide information about the creator of a document as well as information spelling out who can use it and when. Ideally, the computer displaying the document will interpret the hidden information correctly and do the right thing by the creators.

Treating the document as the creators demand is not an easy challenge. All of the algorithms in this book can hide arbitrarly complex instructions for what can and can't be done with the document carrying the hidden information. Some copy protection schemes use as few as 70 bits, a number that can fit comfortably in almost any document.

Just inserting the information is not good enough because watermarks face different threats. Most standard steganographic algorithms fight against discovery by blending in as well as possible to

avoid detection. Watermarks also try to hide— but usually to stay out of the way, not to avoid being discovered. Most consumers and pirates will know the watermark is there soon after they try to make a copy. The real challenge is keeping the consumer or pirate from making a copy and removing the watermark.

Digital Watermarking *by Ingemar J. Cox, Matthew L. Miller, and Jeffrey A. Bloom is a good survey of a quickly growing field.[CMB01]*

This too is not easy. The ideal watermark will stick with a document even after editing, cropping, compression, rotation, or any of the basic forms of distortion. Alas, there are no ideal watermarks out there to date, although many offer some form of resistance to basic distortions.

Defending against basic copying is easy. A digital copy of a document will be exact and carry any watermark along with it. But not all copies are exact. Artists often crop or rotate an image. Compression algorithms for sound or image files add subtle distortions by reproducing only the most significant parts of the information stream. Pirates seek to reproduce all of the salient information while leaving the hidden information behind. Defending against all of the possible threats is practically impossible.

This shouldn't come as a surprise. Making a copy of a document means duplicating all of the sensations detectable by a human. If the sky is a clear, bright blue in the document, then it should be a clear, bright blue in the copy as well. If a bell rings in the document, then it should ring with close to the same timbre in the copy. But if some part of a document can't be perceived, then there's no reason to make a copy of that part.

The watermark creator faces a tough conundrum. Very-well-hidden information is imperceptable to humans and thus easy to leave behind during copying. The best techniques for general steganography are often untenable for watermarks. Compression algorithms and nonexact copying solutions will strip the watermarks away.

But information that's readily apparent to human eyes and ears isn't artistically desirable. Distortions to the music and the images can ruin them or scar them, especially in an industry that often pushes the quality of the reproduction in marketing.

If the ideal watermark can't be created, there's no reason why a practical one can't solve some of the problems. Adobe Photoshop, for instance, comes with a tool for embedding a watermark designed by Digimarc. The software can insert a numerical tag into a photo that can then be used to find the rightful owner in a database. The solution that uses some of the wavelet-encoding techniques from Chapter 14 can resist many basic distortions and changes introduced, perhaps ironically, by Photoshop. The technique is not perfect, however,

and preliminary tests show that rotating an image by 45 degrees before blurring and sharpening the image will destroy the watermark.

All of the watermarking solutions have some weakness to tricks like that. Measuring the amount of resistance is hard to do. The *StirMark* suite is a collection of basic distortions that bend, fold and mutilate an image while testing to see if the watermark survives. This collection is a good beginning, but the range of distortion is almost infinite and difficult to model or define.

16.2.1 A Watermarking Taxonomy

At this point, watermark creators are still exploring the limits of the science and trying to define what can and can't be done to resist the real and somewhat imagined threats. Toward this end, they've created a kind of taxonomy of watermarks that describes the different kinds and their usefulness. Here is a list of the different ways to evaluate them:

Fragility Some watermarks disappear if one bit of the image is changed. Hiding information in the least significant bit (Chapter 9) is usually not a robust watermark because one flipped bit can make it impossible to recover all of the information. Even error correction and redundancy can add only so much strength.

Fragile watermarks, though, are not always useless. Some propose inserting watermarks that break immediately as a technique to detect any kind of tampering. If the watermark includes some digital signature of the document, then it offers assurance that the file is unaltered.

Continuity Some watermarks resist a wide range of distortions by disappearing gradually as the changes grow larger. Larger and larger distortions produce weaker and weaker indications that a watermark is present.

This continuity is often found in some of the wavelet-encoding solutions described in Chapter 14. The watermark itself is a vector of coefficients describing the image. Small changes in the image produce small changes of the coefficients. A vector matching algorithm finds the watermark by finding the best match.

In many cases, the strength of the watermark is a trade off with the amount of information in the watermark itself. A large

number of distinct watermarks requires a small distance between different watermarks. A small distance means that only a small distortion could convert one watermark into another.

Watermark Size How many "bits" of information are available? Some watermarks simply hide bits of information. Counting the number of bits stored in the document is easy.

Other watermarking schemes don't hide bits per se. They add distortions in such a way that the shape and location of the distortions indicate who owns the document. Hiding lots of information means having many different and distinct patterns of distortions. In some cases, packing many different patterns is not easy because the size, shape and interaction with the cover document are not easy to model or describe.

Blind Detection Some watermarks require providing some extra data to the detector. This might be the original unwatermarked image or sound file, or it could be a key. The best solutions offer *blind detection*, which provides as little information as possible to the algorithm that looks for a watermark. The ideal detector will examine the document, check for a watermark and then enforce the restrictions carried by the watermark.

Blind detection is a requirement for many schemes for content protection. Providing a clean, unwatermarked copy to the computers of the users defeats the purpose. But this doesn't mean that nonblind schemes are worthless. Some imagine situations where the watermark is only extracted after the fact, perhaps as evidence. One solution is to embedded the Id number of the rightful owner of a document in a watermark. If the document later appears in open circulation, perhaps on the Internet, the owners could use a nonblind scheme to extract the watermark and track down the source of the file. They could still hold the original clean copy without releasing it.

Resistance to Multiple Watermarks Storing more hidden information is one of the easiest attacks to launch against a document with hidden information. Using the same algorithm often guarantees that the same hidden spots will be altered to carry the new message.

An ideal watermark will carry multiple messages from multiple parties, who can insert their data and retrieve it without any coordination. Some of these least significant bit schemes from Chapter 9 offer this kind of resistance by using a key to

choose where to hide the data. Many can carry multiple messages without any problem, especially if error correction handles occasional collisions.

Unfortunately, these ideal solutions are often fragile and thus undesirable for other reasons. This is another trade off. Localizing the information in the watermark reduces the chance that another random watermark will alter or destroy it, but it also increases the chance that a small change will ruin it.

Accuracy Many watermarking schemes achieve some robustness to distortion by sacrificing accuracy. Many rely on finding the best possible match and thus risk finding the wrong match if the distortion is large enough. These algorithms sacrifice accuracy in a strange way. Small changes still produce the right answer, but large enough changes can produce a dramatically wrong answer.

Fidelity One of the hardest effects of watermarks to measure is the amount of distortion introduced by the watermarking process itself. Invisible or inaudible distortions may be desirable, but they're usually easy to defeat by compression algorithms that strip away all of the unnecessary data.

The best schemes introduce distortions that are small enough to be missed by most casual observers. These often succeed by changing the relative strength or position of important details. One classic solution is to alter the acoustics of the recording room by subtly changing the echos. The ear usually doesn't care if the echoes indicate a 8×8 room or a 20×20 room. At some point, this approach fails and the art is finding the right way to modulate the accoustics without disturbing the greatest number of listeners.

Many of the wavelet-encoding techniques from Chapter 14 succeed by changing the relative strength of the largest coefficients assembled to describe the image. The smallest coefficients are easy to ignore or strip away, but the largest can't be removed without distorting the image beyond recognition. The solution is to change the relative strengths until they conform to some pattern.

Resistance to Framing One potential use for watermarks is to identify the rightful owner of each distinct copy. Someone who might want to leak or pirate the document will try to remove that name. One of the easiest techniques is to buy multiple copies and then average them together. If one pixel comes from

one version and another pixel comes from another, then there's a good chance that neither watermark will survive. Some schemes deliberately try to avoid this kind of attack by embedding multiple copies of the signature and creating identification codes that can survive this averaging.

Keying Is a key required to read the watermark? Is a key required to insert it? Some algorithms use keys to control who inserts the data to prevent unauthorized people from faking documents or creating faked watermarks. Others use keys to ensure that only the right people can extract the watermark and glean the information. Chapter 12 describes some of the approaches.

No algorithm offers the ideal combination of these features, in part because there's often no way to have one feature without sacrificing the other. The good news is that often watermarks that fail one task can find use in another form.

16.3 A Basic Watermark

Here is a basic technique for watermarking that blends together many of the different solutions proposed in recent years. This description is a bit abstract, which obscures the challenges of actually producing a working version that implements the technique. Here's the algorithm:

1. Begin with a document in a standard form.

2. Choose a mechanism for decomposing the document into important components. One solution is to use the discrete cosine transform to model the signal as the sum of a collection of cosine functions multiplied by some coefficients. Let $\{c_0, \ldots, c_n\}$ be the set of coefficients.

3. The coefficients measure the size of the different components. Ideally, the model will guarantee that large coefficients have a large effect on the document and small coefficients have a small effect. If this is the case, find a way to exclude the small coefficients. They're not important and likely to be changed dramatically by small changes in the document itself.

4. Quantize the coefficents by finding the closest replacement from a small set of values. One of the simplest quantization schemes for a value, c_i, is to find the integer k_i such that $k_i Q$ is

closest to x. The value of Q is often called the *quanta*. Exponential or logarithmic schemes may be appropriate for some cases.

5. Let $\{b_0, \ldots, b_n\}$ be a watermark. Insert a watermark by tweaking each coefficient. Each value of c_i lies between an odd and an even integer multiple of Q. That is, $k_i Q \leq c_i \leq (k_i + 1)Q$. To encode $b_i = 0$ at coefficient c_i, set c_i to the even multiple of Q. To encode $b_i = 1$, set c_i to be the odd multiple of Q.

6. Use a reverse transform to reconstruct the original document from the new values of $\{c_0, \ldots, c_n\}$. If the understanding of the decomposition process is correct, the changes will not alter the image dramatically. Of course, some experimentation with the value of Q may be necessary.

This scheme for encoding a watermark can be used with many models for deconstructing images and sound files. The greatest challenge is setting the value of Q correctly. A large Q adds robustness at the cost of introducing greater distortion. The cost of a large Q should be apparent by this point. The value can be seen by examining the algorithm for extracting the watermark:

1. To extract a watermark, begin by applying the same deconstructive technique that models the document as a series of coefficients: $\{c'_0, \ldots, c'_n\}$.

2. Find the integer k_i such that $|k_i Q - c'_i|$ is minimized. If there's been no distortion in the image, then $c'_i = k_i Q$. If our model of the document is good, small changes in the document should correspond to small changes in the coefficients. Small changes should still result in the same values of k_i.

3. If k_i is odd, then $b_i = 1$. If k_i is even, then $b_i = 0$. This is the watermark.

Many watermarking schemes add an additional layer of protection by including some error correction bits to the watermark bits, $\{b_0, \ldots, b_n\}$. (See Chapter 3.) Another solution is to compare the watermarking bits to a known set of watermarks and find the best match. To some extent, these are the same techniques. Choosing the size of Q and the amount of error correction lets you determine the amount of robustness available.

This solution is a very good approach that relies heavily on the decomposition algorithm. The discrete cosine transform is a good solution, but it has weaknesses. Even slight rotations can introduce

big changes in the coefficients produced by the transform. Some researchers combat this with a polar transform that produces the same coefficients in all orientations of the document. This solution, though, often breaks if the document is cropped thus changing the center. Every model has strengths and weaknesses.

16.3.1 Choosing the Coefficients

Another challenge is choosing the coefficients to change. Some suggest changing only the largest and most salient. In one of the first papers to propose a watermark scheme like this, Ingemar Cox, Joe Kilian, Tom Leighton, and Talal Shamoon suggested choosing the largest coefficients from a discrete cosine transform of the image. The size guaranteed that these coefficients contributed more to the final image than the small ones. Concentrating the message in this part of the image made it more likely that the message would survive compression or change.[CKLS96]

Others suggest concentrating in a particular range for perceptual reasons. Choosing the right range of discrete cosine coefficients can introduce some resistance to cropping. The function $\cos(2\pi x)$, for instance, repeats every unit while $\frac{\cos(2\pi}{1000x)}$ repeats every 1000 units. A watermark that uses smaller, shorter waves is more likely to resist cropping than one that relies on larger ones. These shorter waves also introduce smaller, more localized distortions during the creation of the watermark.

16.4 An Averaging Watermark

Cropping is one of the problems confronting image watermark creators. Artists frequently borrow photographs and crop them as needed. A watermark designed to corral these artists must withstand cropping.

An easy solution is to repeat the watermark a number of times throughout the image. The first solution in this chapter accomplishes this to some extent by using decomposition techniques like the discrete cosine transform. Many of the same coefficents usually emerge even after cropping.

This example is a more ordinary approach to repeating the watermark. It is not elegant or mathematically sophisticated, but the simplicity has some advantages.

A watermark consists of an $m \times n$ block of small integers. For the sake of simplicity, let's assume the block is 4×4 and constructed of

values from the set $\{-2,-1,0,1,2\}$. There are $5^{16} = 152587890625$ possible watermarks in this example, although it will not always be practical to tell the difference between them all. Let these values be represented by $w_{i,j|}$, where $0 \leq i < m$ and $0 \leq j < n$. In this case, $m = n = 4$.

The watermark is inserted by breaking the image into 4×4 blocks and adding the values into the pixels. Pixel $p_{i,j}$ is replaced with $p_{i,j} + w_{i \bmod m, j \bmod n}$. If the value is too large or too small, it is replaced with either the maximum value or zero, usually 255 and 0.

How is the watermark recovered? By averaging pixels. Let $w'_{a,b}$ be the average intensity of all pixels, $p_{i,j}$, such that $i \bmod m = a$ and $j \bmod n = b$. In the 4×4 example, $w'_{0,1}$ is the average of pixels like $p_{0,1}, p_{4,1}, p_{8,1}, p_{0,5}, p_{4,5}, p_{8,9}$, etc.

The success of this step assumes that the patterns of the image do not fall into the same $m \times n$ rhythm as the watermark. That is, the average value of all of the pixels will be the same. A light picture may have a high average value while a dark picture may have a low average value. Ideally, these numbers balance out. If this average value is S, then the goal is to find the best watermark that matches $w' - S$.

This is easy if the image has not been cropped or changed. $w' - S$ should be close to, if not the same as, the inserted watermark. The only inaccuracy occurs when the watermark value is added to a pixel and the pixel value overflows.

Recovering the watermark is still simple if the image is cropped in the right way. If the new boundaries are an integer multiple of m and n pixels away from the original boundaries then the values of w' and w will still line up exactly.

This magical event is not likely and any of the mn possible orientations could occur. One solution is to compare the values recovered from the image to a database of known watermarks. This takes kmn steps, where k is the number of known watermarks. This may not be a problem if the system uses only a small number of watermarks, but it could become unwieldy if k grows large.

Another solution is to create a canonical order for the watermark matrix. Let p and q be the canonical offsets. The goal is to find one pair of values for p and q so that we always find the same order for the watermark, no matter what the scheme. This is a bit cumbersome and there are other solutions.

1. For this example, let $z_{i,j} = 5^{4i+j}$. Five is the number of possible values of the watermark.

2. Let $F(w,p,q) = \sum z_{i,j}(2 + w_{(i+p) \bmod 4 (j+q) \bmod 4})$.

3. Try all possible values of $F(w, p, q)$ and choose the maximum (or minimum). This is the canonical order of w.

If mn is a reasonable value, then it might make more sense to store mn versions of each watermark in a database. If it is large, then a canonical order might make more sense.

It should become clear at this point that all possible 5^{16} watermarks can't be used. Some of them have identical canonical values. Every watermark has 15 other shifty cousins.

16.4.1 Effects of Distortion

Much of the success of this watermark depends on the way that the averaging balances out any potential changes. If the noise or distortion in the image is uniformly distributed, then the changes should balance out. The averages will cancel out the changes.

Not all distortions are equal. Several of the StirMark changes introduce or delete rows or columns of pixels from the middle of the image. This can throw off the averaging completely and destroy this kind of watermark. While it may be possible to recover it by sampling sections of the image, the process is neither easy nor guaranteed.

16.4.2 Birthday Marks

Here is a short, more basic example loosely based on a number of systems. The phrase *birthday marks* is borrowed from the *birthday paradox*, the fact that the odds of any two people having the same birthday in a group of n people increases quadratically, $O(n^2)$, as the group grows. In a group of 23 people, there's a 50-50 chance that one pair will share the same birthday and it becomes almost a sure thing, ($> 99\%$) when there are 57 people. Clearly at 367 people, it's guaranteed.

The birthday paradox was also an inspiration for the birthday attacks using hash collisions. [Cop85, GCC88]

Imagine that you can put $m_{i,j}$ marks in the content by tweaking positions in the file. While the notation suggests a grid because there are two parameters, the marks don't need to be arranged in a grid. Indeed, there shouldn't be any detectable connection between the parameters, i and j, and the locations of the marks. If a file has k locations, you might store the marks at a position computed by encrypting i and j.

Although there won't be any connection between the locations and the parameters i and j, let's write these marks in a matrix for clarity. For simplicity, let's imagine there are 16 users so we'll assign 16 unique ids. When user a buys a document, let's put a mark on spots $m_{a,i}$ and $m_{i,a}$ for all $0 \leq i < 16$. For the sake of example,

assume $a = 3$ spots with marks are drawn as ones in this matrix and the unmarked spots are left as zeros:

0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Now assume that there's another user out there with an Id number of 7. The matrix of marks made in 7's copy of the content will look like this:

0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

Let's assume that 3 and 7 decide to collude to strip out their marks. They don't understand the watermarking algorithm. They don't know how the data is inserted into the files. They just have two versions of the content and they can compare them with each other. If they do this, they'll find a number of places where there are differ-

ences: every $m_{i,j}$, where i or j is either 3 or 7 *except* for $m_{3,3}, m_{3,7}, m_{7,3}$ and $m_{7,7}$.

If 3 and 7 are able to blur away the marks at every place where there's a difference, they'll still leave four marks that identify the pair. The content owners will be able to track the document to these two people.

If 3 and 7 decide to use an attack and flip a coin at every difference they find in their document and keep one or the other version, they'll still leave plenty of marks with their unique identity.

This naive solution does not have the best performance as the number of unique Ids grows larger because it requires $O(n^2)$ marks for n unique Ids. This can be avoided to some extent by repeating the process a number of times. Imagine, for instance, that a unique Id number consists of six hexadecimal digits. The previous matrix with $16^2 = 256$ marks could be repeated six times, once to encode each digit. Instead of requiring $2^{24^2} = 2^{48}$ marks, it would only require 6×256, a much smaller number.

16.5 Summary

Watermarking is not an easy challenge for steganography. Many of the researchers exploring it have high ideals and want their watermarks to remain intact even after a fairly daunting array of distortions and changes.

This chapter offers two basic algorithms that resist some basic distortions. The discussion avoids much of the complexity involved in determining when a watermark may or may not be present. Finding it is easy if the image is not changed. Finding it afterwards becomes an exercise in informed guessing. Just when is one vector of numbers close enough to another? Which is the best match for a watermark? These are questions of engineering and design not answered in this chapter. Finding the best solution requires building a system and testing it with a collection of sample images and sound files. Much of the real challenge is tweaking the coefficents.

The Disguise Watermarks are steganographic solutions designed to withstand more general attacks from a savvy set of users and potential pirates. The ideal watermark will embed information in such a way that the only way to destroy it is to introduce so much noise and distortion that the original image is unusable and perhaps unrecognizable.

How Secure Is It? None of the watermarks come close to this ideal, but some are quite useful. Systems like Digimarc's are in wide

use thanks to the company's partnership with Adobe. They can be circumvented but they can withstand many casual distortions. Much of their success depends on the individual algorithm and the sample files.

How to Use It . Find an image, introduce some changes, and then hope to track down the person violating your copyright. In practice, the legal and logistical headaches may be greater than the problems of making the watermark work. If you get really frustrated, just start suing people as some companies have done.

Further Reading

While any of the algorithms in this book can be used to embed a watermark, most of the common algorithms use mechanisms based on either the discrete cosine transform or the discrete wavelet decomposition. Chapter 14 digs deeply into this area. Some other suggestions are:

- *Digital Watermarking* by Ingemar Cox, Matthew L. Miller and Jeffrey A. Bloom is a great introduction to using wavelet techniques to embed watermarks and hide information. The 2007 edition is broader and it includes a good treatment of steganography and steganalysis. [CMB02, CMB07]
- The proceedings of the International Workshop on Digital Watermarking are invaluable sources that track the development of steganography using wavelets and other functional decomposition. [PK03, KCR04, CKL05, SJ06]
- A good conference focused on watermarking digital content is the Security, Steganography, and Watermarking of Multimedia Contents. [DW04, DW05]

Chapter 17

Steganalysis

17.1 Code Words

Authors of recommendation letters often add a secret layer of meaning hidden beneath the surface of a letter that is ostensibly positive.

- All of his colleagues are astounded by his scrupulous attention to detail *and* the zeal with which he shares this information.
- Let us just say that his strength is his weakness.
- Bold words and sweeping statements are his friend.
- We continue to be amazed by his ability to whip off articles, journal papers, and conference talks with such a minimal time in the lab.
- This candidate is a master of the art of not saying what he means. He slips quickly between two-faced disinformation and carefully worded calumny.

17.2 Finding Hidden Messages

Many of the techniques in this book are far from perfect. A wise attacker can identify files with hidden information by looking carefully for some slight artifacts created by the process of hiding information. In some cases, the tests are easy enough to be automated with a high degree of reliability. This field is often called *steganalysis*, a term that mimics the word *cryptanalysis*, the study of breaking codes.

The field of steganalysis is usually more concerned with simply identifying the existence of a message instead of actually extracting it. This is only natural because the field of steganography aims to conceal the existence of a message, not scramble it. Many of the basic tests in steganalysis will often just identify the possible existence of a message. Recovering the hidden data is usually beyond the capabilities of the tests because many algorithms use cryptographically secure random number generators to scramble the message as it is inserted. In some cases, the hidden bits are spread throughout the file. Some of these algorithms can't tell you where they are, but they can tell that the hidden bits are probably there.

Identifying the existence of a hidden message can often be enough for an attacker. The messages are often fragile and an attacker can destroy the message without actually reading it. Some data can be wiped out by storing another message in its place. Other data can be nullified by flipping a random number of the least significant bits. Many small distortions can wipe out the information, which after all is stored in the form of small distortions. While the attacker may not read the message, the recipient won't either. It may even be argued that adding small, random permutations is more effective than trying to detect the existence of the message in the first case. This is a corollary to Blaise Pascal's idea that one might as well believe in a God because there's no downside if you're wrong.

All of these attacks depend on identifying some characteristic part of an audio or image file that is altered by the hidden data. That is, finding a way where the steganography failed to imitate or camoflage enough. In many cases, the hidden data is more random than the data it replaces and this extra "perfection" often stands out. The least significant bits of many images, for instance, are not random. In some cases the camera sensor is not perfect and in others the lack of randomness is introduced by some file compression. Replacing the least significant bits with a more random (i.e. higher entropy) hidden message removes this artifact.

"When a thing is funny, search it carefully for a hidden truth." –George Bernard Shaw, unsourced

There are limitations. Many of these techniques must be tuned to attack the output from particular software programs. They can be highly effective in the hands of a skilled operator searching for hidden information created by a known algorithms, but they can begin to fail when they encounter the results from even slightly different algorithms. There is no guarantee that the steganalysis algorithms can be automatically extended to each version of the software. There is no magic anti-steganography bullet.

But there are also no guarantees that any steganographic algorithm can withstand clever steganalysis. None of the algorithms in

this book offer mathematical guarantees that they are free from statistical or computational artifacts. Many of the automatic steganographic programs introduce errors if they are used without care. Many are very easy to detect if used without caution.

17.3 Typical Approaches

The basic approaches for steganalysis can be divided into these categories:

Visual or Aural Attacks Some attacks strip away the significant parts of the image in a way that lets a human try to search for visual anomalies. One common test displays the least significant bits of an image. Completely random noise often reveals the existence of a hidden message because imperfect cameras, scanners, and other digitizers leave echoes of the large structure in the least significant bits. (See Figures 9.1 and 9.2 for instance.)

The brain is also capable of picking up very subtle differences in sound. Many audio watermark creators are thwarted by very sensitive ears able to pick up differences. Preprocessing the file to enhance parts of the signal makes their job easier.

Structural Attacks The format of the data file often changes as hidden information is included. Often these changes can be boiled down to an easily detectable pattern in the structure of the data. In some cases, steganographic programs use slightly different versions of the file format and this gives them away. In others, they pad files or add extra information in another way.

Statistical Attacks The patterns of pixels and their least significant bits can often reveal the existence of a hidden message in the statistical profile. The new data doesn't have the same statistical profile as the standard data is expected to have.

Of course, there is no reason to limit the approaches. Every steganographic solution uses some pattern to encode information. Complex computational schemes can be created for every algorithm to match the structure. If the algorithm hides the information in the relative levels of pairs of pixels, then an attack might compute the statistical profile of pairs. If an algorithm hides data in the order of certain elements, then one attack may check the statistical profile of the orders.

There is no magic theoretical model for either steganalysis or antisteganalysis. This is largely because theoretical models are always

limited in their power. The mimic functions described in Chapters 7 and 8 are theoretically hard to break. The classical work from Kurt Goedel, Alan Turing, and others firmly establishes that there can't be computer programs that analyze other computer programs.

Such theoretical guarantees are comforting, but they are rarely as strong as they sound. To paraphrase Abraham Lincoln, "You can fool all of the computer programs some of the time, and some of the computer programs all of the time, but you can't fool all of the computer programs all of the time." Even if no program can be created to crack mimic functions all of the time, there's no reason why something might not detect imperfections that happen most of the time. For instance, the software for hiding information in the voice-over of baseball games is going to become repetitive after some time. Some form of statistical analysis may reveal something. It won't work all of the time, but it will work some of the time.

The best the steganographer can do is constantly change the parameters and the locations used to hide information. The best the steganalyst can do is constantly probe for subtle patterns left by mistake.

17.4 Visual and Aural Attacks

The simplest form of steganalysis is to examine the picture or sound file with human eyes or ears. Our senses are often capable of complex, intuitive analysis that can, in many ways, outstrip the power of a computer. If the steganographic algorithm is any good, the changes should not be apparent at first.

17.4.1 Visual Attacks

Hiding the information from human eyes is the first challenge. Some basic algorithms will make mistakes and make large color changes in the process of hiding the information, but most should produce a functionally identical image or sound file.

But a bit of computer enhancement can quickly make a hidden message apparent to our eyes. If the most important parts of the image are stripped away, the eye can often spot encoded information without any trouble. Figures 17.1 and 17.2 show the least significant bits of an image before and after information is hidden with the EzStego program.

The figures illustrate how the least significant bits in an image are often far from random. Notice how the saturated areas of the image

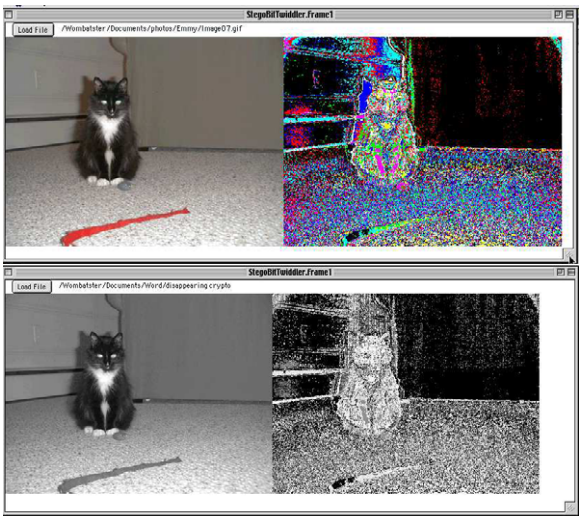


Figure 17.1: The top line shows a picture and its least significant bits before approximately 6000 bytes were hidden in the image with EzStego. The image had a capacity of 15000 bytes. The bottom line shows the image and the least significant bits *afterwards*.

can still be seen in the least significant bits. When the image is either all white or all black, the least significant bit is far from random. It's usually pegged at either a zero or a one. Even the nonsaturated sections are far from random. The positions of the objects in an image and the lights that illuminate them guarantees that there will often be gradual changes in colors. These gradients are also far from random. There's also just pure imperfection. Digital cameras do not always have a full 24 bits of sensitivity. The software may pump out 24-bit images, but only after padding the results and adding extra detail. The least significant bits are not always assigned randomly when the sensors do not have sufficient resolution.

"In the long run, there are no secrets. in science. The universe will not cooperate in a cover-up." – Arthur C. Clarke and Michael P. Kube-Mcdowell in The Trigger.

After information is hidden in the least significant bits, though, all of these regions become much more random. The eye is often the fastest tool for identifying these changes. It's very easy to see the effects in Figures 17.1 and 17.2.

There is no reason why more complicated visual presentations can't be created. Information does not need to be hidden in the least significant bits, in part because it is often very fragile there. [SY98] More complicated presentations might combine several bit planes and allow the attacker to try to identify where the extra information may be hidden.

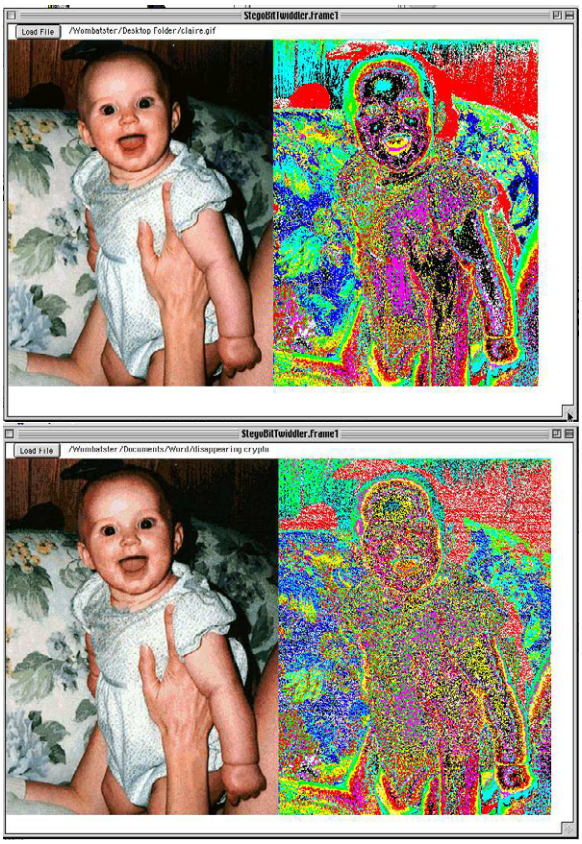


Figure 17.2: The top line shows a picture and its least significant bits before approximately 17,000 bytes were hidden in the image with EzStego. The image had a capacity of 20,000 bytes. The bottom line shows the image and the least significant bits *afterwards*.

Figure 17.3: The JPEG compression function often leaves strong visual artifacts around hard edges. This image was created by compressing text with a very low quality, a process that encourages the artifacts by limiting the number of coefficients kept.

In some cases, file compression algorithms can leave large areas with the same least significant bits. The JPEG algorithm, for instance, stores each image as the weighted sum of some cosine functions. If the 64 pixels in the 8×8 block are sufficiently similar, the algorithm will simply store the average value. The GIF algorithm will also replace similar colors with the same value in the interest of saving space. Both of these effects conspire to prevent the least significant bits from being truly random.

The JPEG algorithm can also leave artifacts, a factor illustrated by Figure 17.3. In this figure, the compression quality was deliberately kept extremely low to amplify the artifacts that seem to echo around the hard edges. If the coefficients of the JPEG image are tweaked to encode the message, the echos and artifacts will be inconsistent and this inconsistency may even be detectable by the eye. In most cases, statistical techniques will be more powerful and many of the algorithms described later in this chapter are sensitive to the existence of these artifacts. The algorithms often become less adept at detecting steganography when the JPEG algorithm is used with higher quality.

17.4.2 Aural Attacks

Skilled audio technicians and musicians can often hear changes ignored by the average pair of ears. Many creators of watermarks for music systems find it simple to inject extra echos in places that are never heard by most people. The trained ears, however, can detect them immediately.

Average ears can pick up information if the data is normalized. Most techniques depend on the way that the human brain picks up

the most significant frequency while ignoring softer versions of similar frequencies.

17.5 Structural Attacks

In many cases, steganographic algorithms leave behind a characteristic structure to the data. This is often as distinctive as a person's handwriting or an artist's brushstroke. If you know what to look for, you can often spot the effects of some algorithms very quickly.

Many of the basic steganographic solutions that hide information in the least significant bits of images are hampered by the data formats used to represent images. Hiding information in the least significant bit is easy when each pixel is represented by 24 bits, with 8 bits allocated for the amount of red, green and blue. Scanners and cameras often leave enough randomness in the three least significant bits assigned to each pixel to make it feasible to store the data.

Unfortunately, most images don't allocate 24 bits for each image. File formats like the GIF or PNG allocate 8 bits or fewer by building a *palette* of selected colors. An i bit palette means 2^i possible colors. These can significantly reduce the size of the images, especially when they're combined with run-length encoding to compress long stretches of identical pixels.

In some cases, the structure used by the software hiding the data is different from the standard. Many GIF files are written with entries for 256 colors, even if only a small number are needed. The software SysCop only writes out the colors in the image. The extra efficiency may shrink the file, but it can distinguish the output.

There are glitches and minor incompatibilities in many of the current packages. Version 4.1 of Hide and Seek, for instance, requires all images to be 320×480 arrays of pixels. StegoDos uses 320×200 pixel images. None of these limitations are difficult to fix, but they show how dangerous quick and dirty steganography software can be. [JJ98a]

Hiding information in the compressed GIF or PNG format by tweaking the least significant bit can fail dramatically because the palette entries are often not close enough to each other. Entry 01001001 may be a deep blue while entry 01001000 may be a hot pink. Twiddling the last bit will distort the image.

Many of the steganographic schemes in Chapter 9 try to avoid this problem by constructing a special palette. This process can leave a damning mark.

A good technique is to choose a smaller palette and then add duplicate colors that can be used to hide information. S-Tools,

for instance, first creates an optimal 32-color palette for the image. Then it creates seven near duplicate colors that differ by one pixel in either the red, green or blue component. If one of the 32 chosen colors has the RGB profile of $(64, 250, 120)$, then 7 additional colors will be added with the RGB values of $(64, 250, 121)$, $(64, 251, 120)$, $(64, 251, 121)$, $(65, 250, 120)$, $(65, 250, 121)$, $(65, 251, 120)$ and $(65, 251, 121)$. Less dramatic versions of this approach are also common. MandelSteg, for instance, reduce the size of the palette to 128 different colors and then creates only one near duplicate of each color.

This approach can hide up to three extra bits of information at every pixel— a significant payload but one that comes with a cost because palettes like these are also easy to detect. When clusters of colors appear in the palette, they are often indicators of bit-twiddling schemes like this. Natural palette creation algorithms try to choose colors as widely dispersed as possible in order to minimize the error of reducing the number of colors.

In other cases, the algorithms try to order the elements of the palette to place similar colors next to each other. Flipping the least significant bit should not distort the image too much because a similar color should be near by. EzStego, a program written by Romana Machado, uses this technique with some success.

Section 13.7 describes how to hide information in the order *of the colors, a technique used in GifShuffle.*

The attacker may be able to intercept the bits if the sorting method is publicly known or easy to figure out. If the data is protected by an additional layer of encryption, then the message will be indistinguishable from random noise.

An attacker may still detect the presence of a message by examining the statistical profile of the bits. An encrypted hidden message should come with an equal probability of ones and zeros. If the numbers of zeros and ones are equal, then the odds point toward a hidden message.

There are two ways to thwart these attacks. The first is to use some keyed version of the sorting routine in order to prevent an eavesdropper from assembling the sorted palette. The palette sorting algorithm is not deterministic because it tries to arrange points in a three-dimensional space so that the distance between adjacent points is minimized. This is a discrete version of the Traveling Salesman problem, a known difficult problem. EzStego uses one reasonable approximation that makes guesses. Instead of using a random source to make decisions, a cryptographically secure keyed random number generator can take the place and create a keyed sorting algorithm.

The second is to use the statistical mimic functions described in

Chapter 6 to create a statistically equivalent version of least significant bits. The patterns from the least significant bits of the original image can be extracted and then used to encode the data. This approach may fool some statistical tests, but not all of them. It may also fail some of the visual tests described in section 17.4.

17.5.1 Interpolated Images

One of the secrets of modern digital photography is that an n-megapixel camera doesn't really have n million sensors that detect the intensity of the red, green and blue light falling on that point. Many cameras have only one sensor at each pixel and this detects either red, green or blue. The other two values are guessed at by averaging their neighbors. This guessing is often called either *color filter array* or *demosaicking*

Here's one of the typical CFA arrangements commonly known as the *Bayer array* in tribute to its creator, Bryce Bayer. There are twice as many green pixels in the grid because the human eye is more sensitive to green:

b	g	b	g	b
g	r	g	r	g
b	g	b	g	b
g	r	g	r	g
b	g	b	g	b
g	r	g	r	g

Let $(3,4)$ be a g, or green sensor pixel, in the grid just shown. The red value for $(3,4)$ is calculated with a weighted average of the red sensors to the left and right, $(2,4)$ and $(4,4)$, the red sensors above, $(2,2)$ and $(4,2)$, and the red sensors below, $(2,6)$ and $(4,6)$. Some cameras may use an even larger collection, but this can increase errors along the edges or at other places where the intensity changes dramatically. The weights are usually chosen to give more weight to the closest neighbors, although the systems usually include some functionality to reduce errors caused along edges.

The CFA algorithms often differ from model to model, something that Sevinc Bayram, Husrev T. Sencar, Nasir D. Memon, and Ismail Avcibas discovered can be used to identify the model from a picture.[BSMA05]

Alin Popescu and Hany Farid looked at this fact and recognized that doctored images will have values that don't satisfy these averages. A red pixel at one location won't be a weighted average of its neighbors. The computations will be most significant along the edges. [PF05, Pop05]

They devised an algorithm that inverts the averages by estimating the original sensor values and then determines where the pixels were generated in a way that doesn't correspond to the standard averaging

Figure 17.4: The upper image shows a photo of Ocean City, NJ modified by pasting a boat in the sky. The lower image is the output of Popescu and Farid's algorithm. The darkest spots are the ones where there's the most disagreement with the neighbors.

done by digital cameras. That is, it looks for an estimate of the original sensor and measures how much the image in question deviates from it. This solution will also detect basic doctoring or photoshopping of images as well as steganography.

Their algorithm is necessary because the images are modified once again by the compression process. This means that the value at a particular pixel is not the exact value produced by the CFA interpolation. The lossy compression algorithm has added more averaging.

Figure 17.4 shows a picture of a beach with a lifeboat pasted on the sky. The second image highlights the most suspicious pixels with darker values. There are a number of grey pixels in the image, but the darkest are found around the edges of the boat itself. Some parts of the boat's interior are not colored because the cutting and pasting didn't move the entire boat. The other objects in the image aren't changed, but they still suggest that there's some incongruity, an effect caused by the compression process.

There are some limits to this approach. A smart steganographer might use the Bayer table to control how information is mixed in the grid. The first pixel gets additional information added to the blue component, the second pixel gets it added to the green channel, etc. Then the results for the other pixels are guessed or interpolated using the same algorithm that a camera might use. The result looks as if it came directly from the camera because it is processed with exactly the same algorithms.

17.6 Statistical Attacks

Much of the study of mathematical statistics is devoted to determining whether some phenomenon occurs at random. Scientists use these tools to determine whether their theory does a good job of explaining the phenomenon. Many of these statistical tools can also be used to identify images and music with hidden messages because a hidden message is often more random than the information it replaces. Encrypted information is usually close to random unless it has been reprocessed to add statistical irregularities.

The simplest statistical test for detecting randomness is the χ^2 (*Chi-Squared*) test which sums the square of the discrepancies. Let $\{e_0, e_1, \ldots\}$ be the number of times that a sequence of events occurs. In this case, it may be the number of times that a least significant bit is one or zero. Let $E(e_i)$ be the expected number of times the event should occur if the sample was truly random. The amount of

randomness in the sample is measured with this equation:

$$\chi^2 = \sum \frac{(e_i - E(e_i))^2}{E(e_i)}.$$

High scores indicate an nonrandom condition— one that was probably part of an original picture or sound file created by an imperfect set of sensors. Low scores indicate a high degree of randomness— something that is often connected with encrypted hidden information.

The χ^2 test can be applied to any part of the file. The least significant bits can be analyzed by looking at two events, e_0, when the least significant bit is zero, and e_1, when the bit is one. A low score means the bits occur with close to equal probability, while a higher one means that one bit outnumbers the other. In this case, $E(e_0) = E(e_1) = .5$.

A better solution is to create four events that look at the pattern of neighboring least significant bits. Natural images often leave neighboring bits set to the same value. Files with hidden information have neighbors that are often different.

Event	bit	neighbor bit
e_0	0	0
e_1	0	1
e_2	1	0
e_3	1	1

Files with a high amount of hidden information will usually have low scores in this χ^2 test. More natural, undoctored images often have higher scores, as Figures 17.1 and 17.2 indicate.

Neil Johnson, Sushil Jajodia, Jessica J. Fridrich, Rui Du, and Meng Long report that measuring the number of close colors is a good statistical test for detecting images with data hidden in the least significant bits. A pair of close colors differs by no more than one unit in each of the red, green and blue components. Naturally constructed files have fewer close pairs than ones with extra inserted data. This is especially true if the image was stored at one time by a lossy compression mechanism like JPEG. Testing for the number of close pairs is an excellent indicator. [JJ98a, JJ98b, FDL00, Mae98]

These tests will often do a good job of identifying basic least-significant bit stegangraphy. More complicated mechanisms for hiding data, however, would avoid this test and require one tuned to the algorithm at hand. Imagine the sender was hiding information by choosing pairs of pixels and occasionally swapping them to encode either a zero or a one. The overall distribution of colors and their least significant bits would not be changed in the process. Swapping doesn't change the statistical profile of the least significant bits.

Section 13.7 describes how to hide information in the sorted order of pixels.

An enhanced version of the test can identify a hidden collection of bits in some cases if the attacker can identify the pairs. The order of pairs in an image with hidden information should occur with equal frequency, while that in a natural image should probably come with some imperfection.

More sophisticated tests can be tuned to different applications. The program JSteg hides information by changing the least significant bit of the integer coefficients used in the JPEG algorithm. In normal pictures, smaller coefficients are more common than larger ones. The value of 1 is more than twice as common as the value of 2, a value that is in turn about twice as common as 3. [Wes01] When the least significant bits of these values are tweaked to hide information, the occurrences equalize. The number of 1s and 2s become equal, the occurrences of 3s and 4s become equal, and so forth. If two coefficients differ by only the least significant bits, then their occurrences becomes equal as information is hidden.

The χ^2 test can help identify JPEG photos where the coefficients occur with too much similarity.

17.6.1 Wavelet Statistics

Another solution is to examine the statistics produced by applying a set of functions to the image. Hany Farid noted that many of the wavelet functions used to model images often produced distinctive statistical profiles. [RC95, Sha93, BS99] He applied one set, the *quadrature mirror filters* (QMF) , at multiple scales and found that the basic statistical profile of the coefficients generated by these wavelet decompositions could predict the presence or absence of a message in some cases. That is, the mean, variance, skewness and kurtosis were different enough to be distinctive.

Basic programs like Jsteg and EzStego could be detected with accuracy rates approach 98% while more careful programs like Outguess could be found as often as 77% of the time. Of course, the success depended heavily on the size of the message encoded in the images. The high success rate came when the hidden message was about 5% of the carrier image's size. If the image size dropped, the success rate dropped to next to nothing (2%).

Some of this success is no doubt due to the fact that a program like Outguess only tries to balance the first-order statistics. Multiscale decompositions with more complicated statistics are still affected by even these balanced tweaks. A more sophisticated version of Outguess designed to keep the QMF statistics in balance could probably defeat it. Of course, keeping the message small is one of

the easiest solutions.

17.6.2 Re-alignment

One of the simplest yet most efficient approaches for detecting steganographic tweaks is to break up the blocks used during compression. These are usually 8×8 pixel blocks in the JPEG algorithm, but they may be larger and more varied in more sophisticated wavelet compression schemes.

The blocks can be broken up by trimming parts of the image. Jessica J. Fridrich suggests cropping away a margin of 4 pixels on all sides of a JPEG image. Each block is split into four parts and glued together with the blocks of the neighbors.

The original and cropped images are compared by looking at the statistical profiles of both versions of the image. If the original was only compressed with a pure JPEG algorithm without any steganographic tweaking, the statistical profile should be the same. The alignment of the grid with the image shouldn't change anything.

If the original image does have some steganographic tweaking in the coefficients, then the changes will distort some but not all of the blocks. When these changes are parceled out according to the new, redrawn grid, they will be averaged out and diminished. The cropped image should have a statistical profile that's closer to an unaltered image than the original.

In comparing the images, Fridrich constructs first- and second-order histograms of the low-order frequencies. The middle and higher frequencies are often much less common and so they become less significant. "Most steganographic techniques in some sense add entropy to the array of quantized [JPEG] coefficients and thus are more likely to increase the variation V than decrease [it]", concludes Fridrich.[Fri04]

17.7 Summary

For every steganographic algorithm, there may be a form of steganalysis that detects it. Many of the early algorithms were relatively easy to detect because they left statistical anomalies. The newest algorithms are much more secure, but only time will tell whether they are able to withstand sophisticated analysis.

A good open question is whether there can be any solid description of which tweaks change which class of statistics. It would be nice to report that simply swapping the least significant bits will change

a particular set of stats and leave unchanged another particular set. This work, however, needs a deeper understanding of how digital cameras, cameras, and microphones convert our world into numbers.

In the meantime, the easiest way to slip by steganographic attacks is to minimize the size of the embedded message. The smaller the message, the fewer changes in the file and the slighter the distortion in the statistics. In this realm, as in many others, the guiding rule is "Don't get greedy."

Mark Ettinger uses game theory to model the cat and mouse game between hider and attacker. [Ett98]

The Disguise Data mimicry often fails to completely hide the existence of a message because efforts to blend the data often leave other marks. Steganalysis often detects these patterns and reveals the existence of the message.

How Secure Is It? Many of the early software packages for hiding information in the noise of an image are often easy to detect. Images have more structure than we can easily describe mathematically. In many cases, the cameras or scanners don't generate files with true noise in the least significant bit and this means that any efforts to hide information there will be defeated.

Mehdi Kharrazi, Husrev T. Sencar and Nasir Memon built a large collection of test images and tested three detection schemes against several embedding algorithms.[KSM06] They found that, in general,

- Comparing the correlation between the least significant and the next-to-least significant bit planes is the most effective detector of pure or RAW images that haven't been compressed with JPEG or another DCT-based compression scheme.
- Artifacts from the cosine transform help many statistical tools because they add a regularity that is distorted when the message and all of its additional entropy are added to the file. Steganography is easier to detect in files that have been heavily compressed.
- The cropping or realignment from Section 17.6.2 is generally the most accurate of the three on files that have some JPEG compression in their history.
- Recompressing JPEG files, especially after cropping, can confound the detectors.

How to Use It The best steganalysis is aimed at individual algorithms and the particular statistical anomalies they leave behind. The best solution is to learn which software is being used and analyze it for characteristic marks. More general solutions are usually far from accurate. The best general solution is to check the randomness of the least significant bits. Too much randomness may be a sign of steganography— or a sign of a very good camera.

Further Reading

- The *Investigator's Guide to Steganography* by Gregory Kipper describes many of the major programs floating around the Internet and explains some techniques for detecting when they've been used. [Kip03]
- Nicholas Zhong-Yang Ho and Ee-Chien Chang show how it is possible to identify the information in redacted documents by looking at the edge effects like the ones in Figure 17.3. If the redaction does not black out the entire 8×8 block, some residual effects from JPG compression will be left behind. This may be enough to recover the information removed during redaction. [HC08]

Chapter 18

Obfuscation

18.1 Regulation

From DEPARTMENT OF THE TREASURY
Internal Revenue Service
26 CFR Part 1
[TD 9115]
RIN 1545-BC27

Temporary regulations issued under 1.168(k)-1T and 1.14-00L(b)-1T (TD 9091, 68 FR 52986 (September 8, 2003)) provide that the exchanged basis (referred to as the carryover basis in such regulations) and the excess basis, if any, of the replacement MACRS property (referred to as the acquired MACRS property in such regulations) is eligible for the additional first year depreciation deduction provided under section 168(k) or 1400L(b) if the replacement MACRS property is qualified property under section 168(k)(2), 50-percent bonus depreciation property under section 168(k)(4), or qualified New York Liberty Zone property under section 1400L(b)(2). However, if qualified property, 50-percent bonus depreciation property, or qualified New York Liberty Zone property is placed in service by the taxpayer and then disposed of by that taxpayer in a like-kind exchange or involuntary conversion in the same taxable year, the relinquished MACRS property (referred to as the exchanged or involuntarily converted MACRS property in such regulations) is not eligible for the additional first year depreciation deduction under section 168(k) or 1400L(b), as applicable. However,

> the exchanged basis (and excess basis, if any) of the replacement MACRS property may be eligible for the additional first year depreciation deduction under section 168(k) or 1400L(b), as applicable, subject to the requirements of section 168(k) or 1400L(b), as applicable. The rules provided under 1.168(k)-1T and 1.1400L(b)-1T apply even if the taxpayer elects not to apply these temporary regulations.

18.2 Code Rearrangement

Some call it shrouded code.

In the beginning, everyone was happy if the code just ran. Anyone who lived with punch cards or paper tape, or any system without persistent memory, spent too much time loading the software.

Now software is everywhere and it is much easier to analyze code on a meta-level. Every compiler reads our code and tweaks it to make it run faster. Most good programming editors and integrated development environments have meta-tools that make suggestions for rearranging the code.

These tools are an opportunity for sneaking in hidden messages on two different levels. First, it is possible to tweak the code to hide a message that is independent of the code— a message might hold a watermark with a user's Id or something entirely different. Second, the code itself can be twisted up in a way that makes it harder to read or understand. The first is generally called *watermarking* or *steganography*, while the second is often called *obfuscation*.

The SandMark tool by Christian Collberg will embed information while obfuscating Java byte code.

The two goals depend on the same basic observation: software code consists of a number of different steps but not all of the steps need to be in the same order. If steps $A, B,$ and C can be done in any order, we can hide a message in this order by giving one person software that does them in order $C, B,$ and A while giving another a package that executes the steps as $B, C,$ and then A. There are six different messages ($3! = 3 \times 2 \times 1$) that can be hidden here.

The process of obfuscation can be helped along by scrambling the order of the steps if we assume that some of the orders are easier to comprehend than others. Programmers, of course, will always argue about the cleanest way to write their software, but even after accounting for natural differences, there are some orders that are harder to understand.

Reordering the instructions is just the beginning. Savvy tools can rewrite the code by replacing one set of instructions with another set that does exactly the same thing. A loop of code that keeps repeating as long as some counter is strictly less than, say, 10 can be replaced

by a loop that repeats as long as the counter is less than *or equal to* 9. Both end at the same time. Both do the same thing. But this flexibility can hide one bit of information. String together 30 of them in the code and there's a watermark.

Obfuscation offers opportunities for more fun. If the goal is to keep a human from understanding the code, it can help to rename the variables with odd names— even words that themselves hide steganographic messages.

In the end, obfuscation may never be completely secure. If the software is to be understood by a computer, then it must be in some form that can be understood by a human— at least as long as the human is thinking like a computer. The general impression is that obfuscated code can provide a hurdle for any attacker but will never reach the security offered by standard encryption packages.

Turn to [BGI+ 01] for a theoretical discussion of the limits of obfuscation. The paper has seven authors and $\log 7!$ *bits can be hidden in the order of their names.*

18.3 Compiling Intelligence

Most of the techniques in this section were originally developed by compiler developers who wanted to rearrange the order of code so they could optimize the speed of the processors. Getting information from the main memory into the processor chip was often much slower than using information directly from the small internal memory, and so the code optimizer would delay the execution until all of the information was in the internal memory.

Rearranging the code to hide a message or the intent of the code itself isn't limited by such concerns. Any valid order is okay and so there are more opportunities to hide messages. If the code obfuscation is done at the *source code* level, the compiler will probably strip out many of the hidden bits by optimizing the compiled code. If it is done at the *binary* level, obfuscating the code may make it slower. [1]

Here is a list of techniques for re-arranging the code.

- **Worthless Instructions** The simplest way to hide bits and add some confusion to code is to just create new instructions and interleave them. As long as you don't use any real variables in your new code, there shouldn't be any weird side-effects, except perhaps for memory usage if you decide to consume huge blocks of memory.

  Many compilers will strip out the so-called *dead code* before generating the object code, an effect that can be useful if you

[1] Some languages like Java have binary code that is further refined at runtime by another layer of translation and compilation. This further optimization can reverse any of the inefficiencies added by the obfuscation.

want to avoid slowing down the software. These dead code detection schemes, though, can also be used to identify the worthless instructions and remove them.

Another solution is to ensure that the worthless code will always compute some value of zero and then add this result to a live variable. Here's an example:

```
int dummy=2*live;
dummy=dummy*10;
for (int i=0;i<20;i++){
 dummy=dummy-live;
}
live=live+dummy;
```

This code won't change the value of $live$, at least until it overflows a 32-bit architecture.

- **Renaming** Code rearrangement for optimization uses renaming to let two different blocks of code run simultaneously even if they happen to be using the same variable name. It's common, for instance, for many loops to use the variable i to count. In the source code, two independent loops can be given different names without affecting the results.

  For fun, some programmers run the International Obfuscated C Code Contest, the Obfuscated Perl Contest, the International Obfuscated Ruby Code Contest, and the Obfuscated PostScript Contest.

  If the goal is obfuscation, new variables can be created and information can be copied from one to the other. If two blocks of code use variable $foo5000$, then an additional variable, $bar422$, can be created to replace $foo5000$ in one of the blocks if the right copying statements are included. $foo5000 = bar422$ should be added to the end of one block and $bar422 = foo5000$ should be added to the other. If you can be certain that one block is always executed before the other, then you can remove the copying statement from the second.

- **Naming** The names of the actual variables can be used to encode information in the process of obfuscating the code— a technique that doesn't do anything for optimization. Any of the techniques from the text steganography Chapters such as Chapter 6 can create variable names that encode hidden bits. If you're programming in very prolix languages like Java that encourage people to give their variables names thatAreEntireSentencesSeparatedByCamelCase, then you might even use variable names built by grammar-based tools like the ones in Chapter 7 or 8.

At first glance, there may be little that you can hide in binary code because the variables are just registers and there aren't many of them, but there's still room for hidden bits. The order that the registers are used by the code is an ordered list and an ordered list of n items can hold $\log_2 n!$ bits of hidden information. See Chapter 13. This technique could also be used with source code-level hidden information.

- **Reordering** If any n blocks of instructions are independent from each other, then they can be reordered in arbitrary ways to hide $\log_2 n!$ bits of data.

  If the blocks have several instructions in them, they can often be shuffled together in arbitrary ways to increase the complexity:

  Before:

```
a1();
a2();
a3();
b1();
b2();
b3();
c1();
c2();
c3();
```

  After:

```
a1();
b1();
c1();
c2();
a2();
a3();
b2();
b3();
c3();
```

  The number of potential shuffles is quite high and depends on a number of assumptions about the underlying instruction set.

- Loop Unrolling Loop unrolling is a technique to interleave the instructions from different interations of a loop. That is, it starts the $i + 1^{th}$ iteration of loop before the i^{th} is finished.

  Here's a loop:

```
i=0;
while (i<3) {
 a(i);
 b(i);
 c(i);
}
```

Here's the loop after being unrolled:

```
a(0);
b(0);
c(0);
a(1);
b(1);
c(1);
a(2);
b(2);
c(2);
```

Here's the loop after being unrolled and reordered:

```
a(0);
a(1);
b(0);
a(2);
b(1);
 c(0);
c(1);
b(2);
c(2);
```

All of the usual rules about independence apply to the reordering. In many cases, the passes through the loops are independent and so can be used liberally.

This can be a powerful technique for obfuscation if a loop can be unrolled a number of times. If some variable renaming is introduced in the middle of the different blocks, it can be extremely confusing.

- **Branch Composition** A chain of simple if-then statements can be rewritten in a number of ways that both obscure the meaning of the underlying code while storing some hidden bits.

```
if (a<b) {
 if (c<d){
 //...
 } else {
  //...
 }
} else {
 if (c<d){
 //...
 } else {
  //...
 }
}
```

This could be written:

```
if ((a<b) and (c<d)) {
} else if ((a<b) and (c>=d)) {
} else if ((a>=b) and (c<d)) {
}else {
}
```

The branching can be composed in many ways that are beyond the scope of this book.

- **Inlining Routines** Many programmers like to break up their software into short functions, methods or subroutines in order to increase the readability of the code. This can be easily defeated by replacing the calls to the subroutines with the subroutine itself. This is often done by code optimizers to reduce the number of calls to the subroutines, an operation that can often slow down software, but can also obliterate the software author's attempts at breaking up the package into easily readable snippets.

  This technique can be quite useful when it is composed with loop unrolling and other transformations because it creates longer blocks with more instructions. In other words, more fertile grounds for more operations.

- **Innane Comments** All of the text steganography from Chapters 6, 7, and 8 can be used to create innane comments for source code.

- **Loop Endpoint Modification** This technique offers little to code optimization, but it's a cheap, quick way to hide one bit

of information. If the loop counter is an integer and it's incremented in a predictable way, it's possible to use one of these two equivalent pieces of code:

```
for (int i=0;i<10;i++){
for (int i=0;i<=9;i++){
```

- **Transformation** Most programming languages are, like C, Java and Perl, inherently declarative and filled with statements that are executed in some predefined order. Others are defined more by transformations. They are not much different from the grammars used in Chapters 7 and 8. These chapters offer a number of ways to transform the grammars, in essence, to change the encryption structure. These techniques, like the ones in Section 7.3.3, will also help obfuscate code.

These techniques themselves are pretty independent and it's usually possible to use all of them with some code.

In fact, I know one person who worked at a place that used automated tools to search for comments. He turned around and wrote an automated comment generator.

The techniques are also easy to apply in different contexts. Reordering independent blocks of instructions can usually be done easily with source code if the language doesn't require that the methods or variables be declared in a fixed order. The methods in a Java class, for instance, can appear in any order, and this is an easy way to add a watermark to Java code without bothering to do any complicated code analysis to determine whether two blocks of code are truly independent.

The technique can often be applied in binary packages too if the binary code doesn't require code to appear in a particular order. Various routines can be reordered in code as long as all references to the location are rewritten.

18.4 Real Tools

There are too many tools available for obfuscation to list them here. This short list is provided only as a starting point:

- **JODE** A nice package for Java from Jochen Hoenicke that can be downloaded from `jode.sourceforge.net`. It includes a decompiler, an obfuscator that works by renaming, and a few other tools like a dead code eliminator. This tool can be used to introduce watermarks or remove them.

- **yGuard** Another Java-based package from yWorks (`www.yworks.com`) will rename variables, methods, classes and what-not while eliminating other dead code. This can also shrink the size of the binaries dramatically.

- **SWF Encrypt** A commercial package from Amayeta (`www.amayeta.com`) will scramble Flash (SWF) files while encrypting them.

There are also steganographic tools for adding watermarks or data to source code, obfuscating the behavior a bit in the process. Hydan, a tool written by Rakan El-Khalil and Angelos D. Keromytis, will swap equivalent operations in x86 binary code. For instance, both `add %eax, $20` and `sub %eax, $-20` will add 50 to the register `eax`, providing the opportunity to hide one bit. They report that they can embed about 1 bit per 110 bits of x86 binary code on average, an amount they determined empirically by embedding watermarks in x86 code compiled for Linux, Windows and BSD systems. The software can be downloaded from `http://www.crazyboy.com/hydan/`. [EKK04]

18.5 Summary

Obfuscation is a technique that is normally used to hide the meaning of some software by rearranging the operations, but it can also be used to add weak watermarks to the code. In both cases, the algorithms rely on a collection of transformations that change the apparent operation of the software without changing the results. An obfuscated program should produce exactly the same results as an unobfuscated one.

The Disguise The results of scrambling these programs can be quite useful in many simple scenarios. Removing the variable names and inlining a few instructions will make it difficult for any reader to follow the simple flow of the software.

How Secure Is It? There are theoretical proofs that suggest that this technique will never produce completely inscrutible code— an understandable result given that any useful software must be understood by the computer. Still, software written by someone else is often hard enough to read. Strip away the comments, the variable names, and some of the structure, and it could require a lot of work to reassemble.

This technique may be more useful as a watermarking tool than as an obfuscator. Small changes in the code can mark individual copies of the software.

Still, there are simple techniques that can be used to strip away some of the watermarking tools. Decompiling the binary software and then recompiling it is one simple solution that will remove many of the effects described here. Or simply adding another watermark on top of the software can remove the old one.

How to Use It? There are a number of packages that will do simple obfuscation. See Section 18.4. Most of these will only perform basic obfuscation by renaming variables and doing some basic reordering. More sophisticated commercial tools are also widely available.

Further Reading

- There are a number of code obfuscation tools on the market, but they are not designed to hide information while obfuscating the code. RetroGuard from `retrologic.com` and CodeShield from `codingart.com` work with Java code.
- Christian S. Collberg and Clark Thomborson survey many of the techniques for obfuscation and watermarking, including dynamic watermarking that morphs as the program executes, making it harder to identify the watermark. [CT02]
- Bertrand Anckaert, Matias Madou and Koen De Bosschere offer a general way to think about self-modifying code, something that makes it much simpler to build dynamic watermarks.

Chapter 19

Synchronization

19.1 Stealing Baseball's Signs

To: Newly Signed Recruits
From: Third-Base Coach

Here are the signs for tonight's game. As usual, I will string together a sequence of bogus signs in front of the two trigger signals. After the two trigger signals, *alpha* and *beta* follow in immediate sequence. I will deliver the one sign that isn't fake. Pay attention to that one. Then I'll add in another three to seven fake signals to close out the set. Remember: Ignore all signs that don't follow the *alpha* and *beta* sign in sequence. If you see another sign between the *alpha* and *beta*, ignore them all. If you see the *beta* before the *alpha*, ignore it. Use the *alpha* and*beta* to synchronize yourself on the sign that matters.

Sign	**Meaning**
Touch Nose	Drug tester wants to check your steroid level. Drink lots of fluids.
Grab Left Ear	Someone dinged your new Escalade in the parking lot.
Grab Right Ear	*alpha*
Grab Left Ear	Your broker called with bad news about that investment in pork bellies.
Cover Eyes	*beta*
Slap Left Hip	Watch your mouth. We could lipread that last epithet.
Kick Left Foot	The owner is calling your bluff. He said "No" to your salary demands.
Muss Hair	We should dock your pay for that last one.
Hitch up Pants	The team doctor says I should cut back on the steak dinners.

19.2 Getting In Sync

Most of the algorithms in this book depend, in one form or other, on getting the complete file. The message may be hidden in a small fraction of the file, but everything is available for extraction. If the hidden bits are supposed to be located 14 pixels from the left and 212 from the bottom, that spot can be found because the entire frame is available.

There are many cases when the entire file isn't around. Photos are routinely cropped before being reused. Only a short clip of a sound file or a movie may be available. Anyone turning on a radio at a particular moment may start receiving the signal at any arbitrary place in the stream.

Unfortunately, some of the more extreme algorithms in this book rely heavily on absolute synchronization to function correctly. Most effective encryption algorithms change their process throughout the file, encrypting the n^{th} byte differently from the $n + 1^{th}$ byte. Some of the algorithms for hiding information use a subset of the data and rely heavily on the boundaries to define those pixels.

Synchronizing a bit stream between a sender and a receiver is not hard to do, but it is a bit tricky to do in an efficient manner. The simplest trick is to introduce a special synchronization character and only use it at the beginning or end of a word. The easiest example is *unary* or base one encoding, where a 1 is used as the boundary and the number of 0s encodes the value. So a message of $2, 3, 2$ would look like 1001000100.

Morse code uses two different-sized pauses in the signal stream. Short pauses break up the dits and the dashes, while pauses that are three times as long break up the letters. The gap between the three dots in an S (...) should be one third the size of the gap between the last dot and the beginning of the next letter. The factor of 3 is just a convention and many people probably deviate.

Dedicating an entire character like 1 to synchronization in an algorithm is expensive. Can you imagine transmitting the yearly budget of the United States government (currently about $3.1 trillion) in unary? While the approach is basic, it's not easy to extend in the digital world, where there are only two real characters. Other characters are just composed of blocks of 0s or 1s and it's not simple to find the beginning and end of the block.

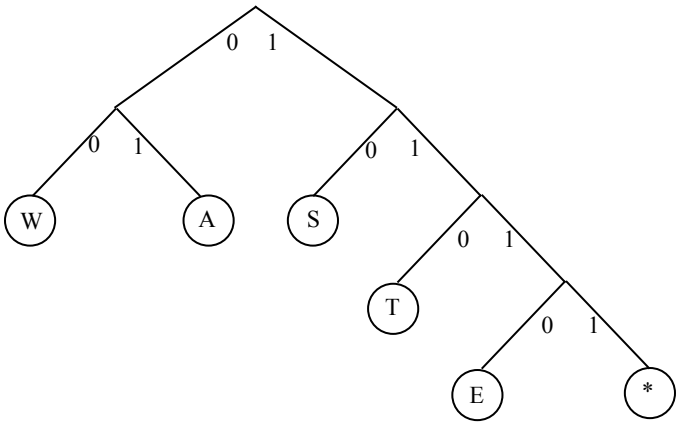


Figure 19.1: An imperfect modification of the Huffman tree from Figure 5.1. The extra synchronization character, '*', is not unique.

19.3 Extending Other Tools

Section 5.2.1 describes *Huffman coding*, a compression algorithm that will produce variable-length codes for characters and compress the bit stream by giving shorter codes to more common letters and longer codes to the rarer ones. The standard algorithm may be very good at squeezing out the extra entropy from a file, but it is extremely fragile. If only one bit is lost, the rest of the stream can be compromised.

Adding a synchronization character to a Huffman code is not as easy as creating another character and adding it the mix. Figure 19.1 is a modified version of Figure 5.1 created by adding another character, '*', to the alphabet to act as a synchronizer. This seems to work. Adding the asterisk at the beginning of a word or a paragraph would be a signal. The word 'ate' would be encoded as '1111011101110'. Anyone who needs to synchronize a stream of bits could look for the four 1s in a row and then start forward from that point.

This would seem to work. There are four 1s in the asterisk's code block, '1111', and because of the shape of the tree, four 1s only occur in this asterisk. But there are three problems:

- The rest of the tree is not guaranteed to be as short as it is in this example. This can be guaranteed by assigning the synchronization character the lowest probability, something that ensures it will be the longest code in the tree.
- Two letters together can produce the synchronization code.

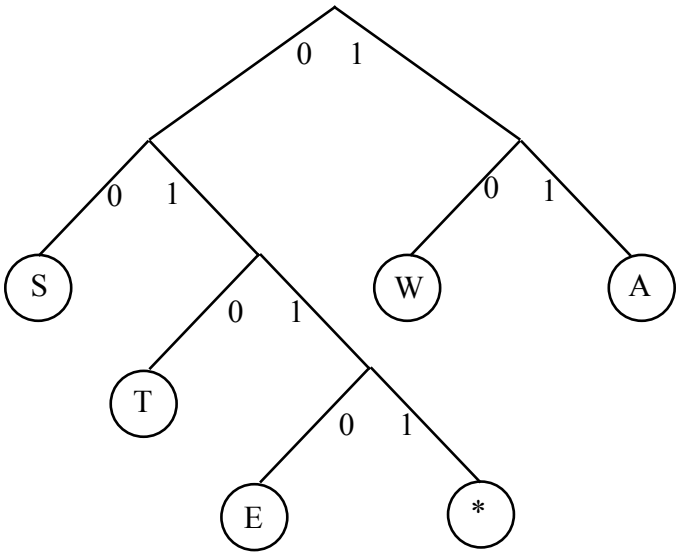


Figure 19.2: Figure 19.1 after the tree is rearranged to make the synchronization code be 0111. These techniques are also used in Section 7.3.3 to scramble the tree to add more confusion to the code.

The code for 'at' is '011110'. Is this the place where we should start decoding?

- It's not clear where the code itself begins. The word '*tea' would be encoded: '1111110111001'. Where do the four 1s end and the real message begin?

The last two problems can be solved by choosing the synchronization code carefully and tweaking the tree by adding extra values or nodes. This will decrease the efficiency but not as much as converting to unary.

The key to avoiding the third problem is choosing a synchronization code that is not susceptible to *prefix inversion*. That is, it's not possible to break the code block into two separate parts, A and B, such that $A + B = B + A$ where $+$ stands for concatenation. The block 1111 is one of the worst candidates for this because there are three different ways to break it into A and B that satisfy the equation. Setting $A = 111$ and $B = 1$ is just one of them. A better choice is something like 0111. Figure 19.2 shows a tree redrawn to avoid this problem.

Avoiding the second problem is a bit trickier. If two adjacent characters can produce the synchronization code, then it would intro-

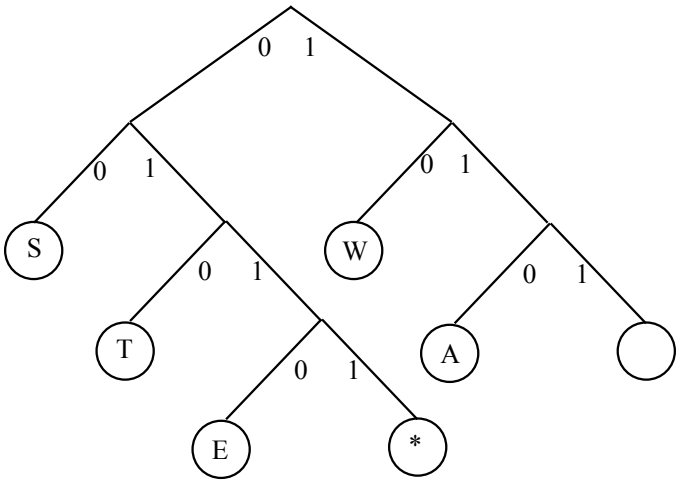


Figure 19.3: Figure 19.2 after the code for 'A' is changed from 11 to 110. Notice that there's a blank node for the code 111. This shouldn't be used.

duce bad synchronization characters. The solution is to add extra bits to the codes to eliminate this possibility— a process that will hurt the code's efficiency. The price, however, may be fairly low if the codes are long. Adding an extra bit to a 12-bit code isn't as bad as adding an extra one to a 3-bit code.

The trick is to ensure that the code for one character can't be broken into parts A and B such that B is a prefix for the synchronization code. So if the synchronization code is 0111, then a code that ends with 01 would be bad because it is a prefix of the synchronization code.

The codes produced by the Huffman algorithm can be made longer by adding extra characters to the end. It's not easy to add them to the beginning or the middle, but it is to add them to the end. These are wasted bits that serve no purpose except to prevent the synchronization code from appearing.

Figure 19.3 shows Figure 19.2 after the code for A has been rewritten to use an extra bit, 110. This creates a code, 111, that is never used when compressing the data. This inefficency, though, ensures that none of the other letters will combine to form the synchronization code. There is no code that will produce three 1s in a row except the synchronization code.

The construction in Figure 19.3 is more of an accident produced by the fact that the tree is small and there is only one letter that ends

in a 1. There are a number of similar constructions that can produce fairly efficient codes.

Imagine, for instance, that the tree will have height $2n$, meaning that the longest codes will be $2n$ bits long. Let one of these long codes be the synchronization code created from n 0s followed by n 1s. So if $n = 4$, then the synchronization code would be 00001111. Let's represent this as $0^4 1^4$ or $0^n 1^1$. Using a long code for synchronization may be a bit inefficient if you intend to include it often, but it simplifies the process of ensuring that no two letters can combine to form the synchronization character.

We can fix the codes for the remaining characters by adding the right suffixes:

- If a code for a character ends with more than n 0s, add 101 to the code.
- If a code for a character ends with fewer than n 0s, add 1 to the code.
- If a code for a character ends with more than n 1s, add nothing.
- If a code for a character ends with fewer than n 1s, add 01 to the code.

This construction may not be optimal in many cases, but it should suffice to prevent n 0s from appearing before n 1s except in the synchronization character.

There are other techniques that rearrange the tree and provide more efficient codes.

19.4 Summary

Being able to pick up a steganographic trail in the middle of a stream is a useful technique for adding watermarks and other messages when the cover file may be cropped or truncated along the way. When users cut and paste snippets of songs or parts of an image, the watermark can still survive if it's possible to synchronize the solution.

The algorithms given here are just a small selection. Synchronization is a common problem in data communications because noise can corrupt signals. Many radio devices like cell phones must be able to synchronize themselves with a network when they turn on.

The Disguise The algorithms given here don't add any cover to the process themselves. They just make it simpler to recover the

hidden message when the disguise is chopped or truncated. A snippet can still reveal enough information.

How Secure Is It? This construction doesn't add any security itself, but it does offer it indirectly by making it possible to recover information from partial fragments.

How to Use It? If you have a short message and a long cover file, repeat the message throughout the cover file numerous. This only works on methods that don't base the encoding on the relative position in the file. Random walks that leave the steganographic method in places won't work. The algorithms should use some regular hiding algorithm that will be possible to pick up even after cropping.

Further Reading

- S. Manoharan describes how self-synchronizing codes can solve issues created by cropping and sampling. [Man03]

Chapter 20

Translucent Databases

20.1 Missed Connections

- To the kind person who took my new iPod from the gym on 49th and Broadway, enjoy the ear mites.
- To the person who left their coat behind at a bar in the East Village, you can recover it at the coin lockers at Grand Central Station. It's in locker 421 and the pass code is the first three letters in the name of the bar followed by the two digits of the date you left it behind.
- To the redhaired stunner in a blue Nike running jacket on the F train on Monday morning around 9 am. You took my heart yesterday, a major felony, btw, that should be worth 5 to 7 years and a fine not exceeding $10,000. LOL. I spoke with my parole officer, who told me it was okay to date on the first weekend out of the joint. Call me, please.
- To the two snickering high school students who looked at my outfit, caught my eye and then covered their mouths while laughing at me on the Number 2 train last Thursday. You can stop looking for your black leather gloves. You left them behind on the train and I was too busy being hurt to tell you when you left.

20.2 Hiding In Databases

Much of the work in this book is aimed at hiding information in the cracks of other pieces of information, in essence turning it into

an extra passenger on some bigger file traveling throughout the net. The tricks in this book are also useful when the information remains sitting in one central location like a database. The math remains the same, although there are some new twists from the database world that can be used to hide signals in images or other traditional homes for steganography.

Traditional centralized servers will collect gigabytes and gigabytes of information about users and then use simple access control mechanisms like passwords to control who gets to see which piece of information. The database becomes an all-seeing oracle with the responsibility of guarding the data. It becomes a big target for attackers, who can read all of the information inside the database if they can find a weakness in any of the layers in the system. Some thieves simply steal the hard disk to circumvent the locks placed by the operating system.

The term *translucent databases* is meant to describe any effort to lock up a central pile of data so only the rightful owners, the clients, can use it. A typical translucent system requires the user or client to encrypt the bits before storing them in the database, stripping the central server from having any responsibility for protecting the information. If the encryption is done in the correct way, the database can still search through records and retrieve the information for the client without knowing much about that information.

There are a number of different way to build a translucent mechanism and this chapter will only describe two: *steganographic functions* and *one-way functions*. The first approach is a natural descendent of the algorithms found in this book and the various tricks for steganographically encoding a message let the client encrypt it and store it in the database. The one-way functions are more thorough tools that completely scramble data; they also can be used for steganography.

A more complete exploration of this topic is found in my book, Translucent Databases *[Way03].*

Imagine someone wants to add image-tagging features to an online auction site but the complexity of the legacy code makes it difficult to add them in a traditional way. The users could simply begin using some of the image tools described in this book to add the descriptive words to the images without disrupting the normal software. No changes would be necessary to the core auction database. Anyone who wanted to search these image tags could run a separate crawler that scans for them without disturbing the regular workflow of the site.

The power this vests in the client means that the auction site wouldn't even need to grant permission or even know that the action was going on. The users would have the power to add features of

their own accord and on their own terms.

20.2.1 One-way Functions

One of the simplest ways to build a translucent database is to obscure some of the columns of the database by passing them through a one-way function, a mathematical convenience that is easy to compute but practically impossible to invert. (Use $h(x)$ to denote the one way functions.) So-called cryptographically secure hash functions like SHA256 are some of the most commonly used one-way functions today, but almost any good cryptographic algorithm can be converted into acting the same way by disposing of the keys.

A translucent version of a store's database holding the customer's records might look like this:

$h(name)$	item number	size	color
$ab9938c\ldots$	4000	XL	green
$ab9938c\ldots$	4000	XL	blue
$2c42d45\ldots$	4002	L	rose
$99ab993\ldots$	4003	M	yellow
$99ab993\ldots$	4000	M	green

This table does not store the name of the customer directly. It stores $h(name)$, which is the result of passing the name through a one-way function like SHA256. The other three columns are left unscrambled, a feature that lets the marketing and inventory departments study past sales.

Storing the customer's name in this way lets the store track a customer's purchases while giving the customer some control over how the information is used. The store can ask its database to answer a question like "Which colors for item 4000 are popular?" but it can't poke around and ask, "Who is buying item 4000?" If the customer returns to the store, though, the customer can calculate $h(name)$ and look up the past item numbers and sizes. Once the customer provides $h(name)$, the store can provide all of the intrusive services that it might want to provide.

There are limitations to this approach. The central database administrator can't invert the values of $h(x)$ and so the administrator is often unable to fix entries in the table. Attackers who are targeting particular individuals and know their names can defeat simple systems like this by computing $h(x)$ on their own— a problem that can be fixed by asking the users to add a password to the mix and store $h(name, password)$ instead.

This solution, though, is still often much cheaper and simpler than building sophisticated access mechanisms and finding a secure operating system to host the database.

20.3 Using Strong One-Way Functions

Using steganography with a centralized database produces databases with two levels of access that can be controlled by the users alone. The solutions make it simple to add new functions to legacy applications without disturbing them. If $f(x, C)$ is some function that hides the secret bits of x in the cover C in a way that can't be detected, then a client stores $f(x, C)$ in the database instead of C. Only the people who know about f and have the right keys to reverse f can extract x. Everyone else sees plain old values of C.

Here are a few quick examples of how this can be useful:

- A database that tracks ships can blur the actual position of a ship by adding a vector, $(h_1(x_i), h_2(y_i))$, to each ship's position, (x_i, y_i). If the values of h_1 and h_2 can only be computed by some users, then only those users can find the position of the ship with the highest level of precision. One possible solution is to use $h_1(x) = SHA256(name, password, x)$, where $name$ is the name of the ship and $password$ is some secret password.-
  A technique like this might be useful if a shipping company wants to dispense positions with rough accuracy to the general public so they can track the movement of their goods, but keep the greatest accuracy for internal use.

The HMAC is a mechanism for hashing a longer document with a key. It is the best technique for mixing a key into a generic hash function. [Pro, NN06, BCK96b, BCK96a]

- An image royalty company might want to piggyback on the wide-open, public databases like `flickr.com` by embedding sales information inside an image. They might add prices and copyright information inside the image so anyone browsing the site could get immediate royalty information.

20.3.1 One-Way Functions and Steganography

The paradigm of encrypting columns in a database can be converted into a tool for hiding information in plain site. Instead of storing the table in some distant database, why not distribute the entire thing and let the user find the right rows?

Imagine a table with n rows much like the table like the one in Section 20.2.1 where $n - 1$ rows are made up of random information and one row contains the true signal. A legitimate user can find

the right row by knowing either $h(x)$ or just x and the structure of h. That is, anyone searching for the secret data can find it in the sea of noise by knowing either the right key, $2c42d45\ldots$, or the right way to calculate the key (Use your $name$, and compute $h(name +' swordfish')$).

There are a number of ways that this technique can be extended. One simple mechanism is to put random numbers in one column, call them x_i, and digital signatures in another, $y_i = s(x_i)$. Only the legitimate rows have digital signatures that check out. These signatures could be as simple as a *message authentication code* or MAC like $s(x) = SHA256(x + password)$. Still, using fast authentication computations makes this an expensive mechanism because someone looking for data must look at every row and decide whether y_i matches $s(x_i)$.

20.4 Summary

Databases don't need to store information in the clear and rely on elaborate security mechanisms and good operating systems to protect the information. The client can encrypt or hash the data before storing it. Only the client can fetch it later.

This increases the security of the database and reduces the responsibility on the shoulders of the database administrator at the cost of eliminating their ability to help. Removing the danger of a superuser also eliminates the superuser as a source of assistance.

The Disguise Hide information in a database by either encrypting it completely or storing it steganographically inside other information.

How Secure Is It? It depends on the quality of the encryption or steganographic algorithm. In most cases, the best hash functions will suffice, although some mathematical research is chipping away at their strength.

How to Use It? Use $SHA256(data)$ instead of $data$ for a column. Only the true owner of $data$ can then use that column as a key to locate the row.

Further Reading

- My book, *Translucent Databases*, describes a number of techniques for hiding information in databases.[Way01]

- The proper way to use synthetic one-way functions like SHA256 is a fertile topic because of all the recent research identifying weaknesses in some of the standard designs like MD5 and SHA-1. While many of the results won't have much immediate practical effect on some of the techniques used in translucent databases, there's little doubt that the research could lead to a consistent and useful attack capable of producing useful collisions. Even some of the more sophisticated tools like the HMAC are susceptible. [RR08, FLN07, MTMM07]

Chapter 21

Plain Sight

21.1 Laughtracks

To: Production Staff and Set Design
From: Jerry Brown, Asst. to the Executive Writing Arc Supervisor
Re: Meta-textual clues for audience.

Recent audience surveys show that the larger broadcast audience fails to grasp the humorous possibilities of the narrative arc designed by the office of the Writing Arc. To facilitate the absorbtion of our humor, we are asking that we construct a new set of neon signs that will cue the studio audience to inject the kind of meta-narrative instructions that will allow our broadcast audience to, for lack of a bigger term, laugh at the right places.

All future scripts from the Office of the Writing Arc Supervisor will include cues for when to activate these lighted signs.

Snicker For use when a mild signal should be injected into the story stream.

Snort We hope to limit this to the jokes told by our particularly curmugeonly characters.

Please Groan Best used for puns and other cheap forms of humor. This will indicate that we're not stooping to cheap tricks to get laughs but approaching them with an obviously ironic pose.

Knee Slapper For older jokes that we've borrowed from the old school of comedy.

Through the Nose For physical comedy.

Atomic Funny This signals super-duper funniness. The atomic bomb of humor. Not to be confused with humor that just bombs. To be used sparingly, no more than twice per episode.

21.2 Hiding in the Open

In the middle of Michael Crichton's *Jurassic Park*, when the characters are coming to realize the depth of their predicament, the mathematician in the bunch, Ian Malcolm, asks the dinosaur curators to reprogram their computers. The original software began with an expected count of dinosaurs and then scanned the park looking to account for all of the dinosaurs on the list. If one was missing, it raised an alarm and triggered a search. It all seemed bullet-proof.

But when Malcolm asked them to raise the expected count, the computer came back and found even more dinosaurs than it did in the original count.

"Now you see the flaw in your procedures," Malcolm said. "You only tracked the expected number of dinosaurs. You were worried about losing animals, and your procedures were designed to advise you instantly if you had less than the expected number. But that wasn't the problem. The problem was, you had more than the expected number."[Cri90]

The dinosaurs were breeding and the elaborate control system couldn't account for them. Malcolm saw this weakness in the system as an example of a larger truth about the universe.

"[S]traight linearity, which we have come to take for granted in everything from physics to fiction, simply does not exist. Linearity is an artificial way of viewing the world. Real life isn't a series of interconnected events occurring one after another like beads strung on a necklace. Life is actually a series of encounters in which one event may change those that follow in a wholly unpredictable, even devastating way." he explained.

All data formats for computers begin with expectations. While the formats may not be linear in the strictest sense of the word, they are still well defined and the strength of this definition leads to its weakness.

Much of this book involves tweaking the actual data stored in a file by introducing small changes like adding a bit more red to a pixel. These solutions are useful, but they can introduce distortions that can lead to detection. As Chapter 17 shows, many of the simplest algorithms distort the statistical profile of the files in subtle but often detectable ways. To make matters worse, the approach is in constant competition with compression algorithms that try to squeeze out all

extraneous information and noise to save space. This is why many researchers suggest that the only long-term solution is to hide information in the salient features, the actual visible or audible parts of a file.

Wojciech Mazurczyk and Krzysztof Szczypiorski found that Voice Over IP calls could hide information because the algorithms work around missing packets— or packets replaced with hidden messages.[MS08]

Here's a counter approach: Instead of hiding information in the data itself, hide information in the gaps in the data structures, in the places where the software won't look for it.

Consider how to join these two facts: (1) GIF files, like most files, begin at the beginning of the data with a few bytes that describe the size of the data, while (2) ZIP files begin at the end with a table that describes the location of data inside. This makes it possible to concatenate a GIF file and a ZIP file so that both are still decodable, at least least in theory. Just type this on a Mac or UNIX box:

"K is for Keeler, As fresh as green paint, The fastest and mostest To hit where they ain't."–Ogden Nash[Nas49]

```
cat somefile.zip >> somefile.gif
```

This will append `somefile.zip` to the end of `somefile.gif`. It essentially hides a GIF file at the beginning of a ZIP file or a ZIP file at the end of the GIF file. A program looking for a GIF file will start at the beginning, decode the header information describing the size of the image, and then unpack it beginning at the very beginning. A ZIP file decoder will do the same, but from the end. I assume that the ZIP file was designed this way to make it easier to add more files to a ZIP file by just appending them to the end.

Neither software package will notice the other— unless there are some unspoken assumptions made by the programmers. It's entirely possible that a clever programmer will mix up the two different values: (1) the length of the file as described by the header and (2) the length of the file as described by the operating system. This will lead the results to crash nonstandard implementations.

21.3 Other Formats

Many formats make it easy to add freeloading data to a file. In fact, good programmers have been pushing this as a design feature for software because it makes it simpler to improve software without crashing older versions. A good file format will include a mechanism to add more data later in case the need becomes necessary.

Many of the modern tagged languages like XML (Extensible Markup Language) or its cousins like SGML (Standard Generalized Markup Language) or HTML (Hypertext Markup Language) are designed to let the programmer toss in additional information or create additional data as necessary.

Here's a common example:

```
<recipe name="brownies" >
  <title>Chocolate Brownies</title>
   <ingredient amount=".75" unit="cup">
     melted butter</ingredient>
   <ingredient amount="1.5" unit="cup">
     sugar</ingredient>
   <ingredient amount="1.5" unit="teaspoon">
     vanilla</ingredient>
   <ingredient amount="3" unit="item">
     eggs</ingredient>
   <ingredient amount=".5" unit="cup">
     cocoa powder</ingredient>
   <ingredient amount=".75" unit="cups">
     flour</ingredient>
   <ingredient amount="1" unit="teaspoon">
     salt</ingredient>
   <ingredient amount="1" unit="cup">
     nuts</ingredient>
   <ingredient amount="1" unit="cup">
     semi-sweet chocolate chips</ingredient>
  <instructions>
   <step>Mix together.</step>
   <step>Pour into pan.</step>
   <step>Bake in the oven at 350(degrees)F.</step>
  </instructions>
 </recipe>
```

Here's how easily it can be extended with some new tags and attributes:

```
<recipe name="brownies" >
  <title>Chocolate Brownies</title>
  <creator email="unknown@unknown.com"> Mrs. Quinn </creator>
   <ingredient amount=".75" unit="cup" type="liquid">
     melted butter</ingredient>
   <ingredient amount="1.5" unit="cup" type="level">
     sugar</ingredient>
   <ingredient amount="1.5" unit="teaspoon" type="liquid">
     vanilla</ingredient>
   <ingredient amount="3" unit="item">
     eggs</ingredient>
   <ingredient amount=".5" unit="cup" type="level">
     cocoa powder</ingredient>
   <ingredient amount=".75" unit="cup" type="level">
```

```
    flour</ingredient>
  <ingredient amount="1" unit="teaspoon" type="heaping">
    salt</ingredient>
  <ingredient amount="1" unit="cup" type="heaping">
    nuts</ingredient>
  <ingredient amount="1" unit="cup" type="heaping">
    semi-sweet chocolate chips</ingredient>
 <instructions>
  <step>Mix together.</step>
  <step>Pour into pan.</step>
  <step>Bake in the oven at 350(degrees)F.</step>
 </instructions>
</recipe>
```

The new version includes attributes describing the type of measurement used for the ingredients and a new tag giving credit to the creator. Most software tuned to the original package will ignore these extra tags and find only the data it expects to find.

This is usually the case with XML, but it is not always true. The specification includes a mechanism for defining the legitimate pattern for the tags and this specification can block the adding of extra tags. The *Document Type Definition*, or DTD, includes definitions for which tags should be found and where they can be found.

Many programmers report a fair amount of frustration with rigid DTDs because they cause incompatibilities between versions of the software. This is especially true if the software relies on a feature that allows a DTD to be downloaded from a web site. I know one open-source project that started crashing after some of the keepers updated the DTD on a distant web site.

21.3.1 Microformats

One of the more established attempts at building a regular mechanism for revising and extending HTML lives at the `www.microformats.org` web site. The Microformats project includes a number of additions to HTML that add more semantic meaning to the text. The extra meaning identifies the context or meaning of the text on a web page by identifying its role. One type of tag wraps around a zip or postal code. Nother identifies a telephone number.

Here's an example of an address spelled out in the the *vcard* format:

```
<div class="vcard">
<a class="fn org url" href="http://www.munster.com/">
```

```
   Herman Munster</a>
 <div class="adr">
  <span class="type">Work</span>:
  <div class="street-address">1313 Mockingbird Lane</div>
  <span class="locality">Mockingbird Heights</span>,
  <abbr class="region" title="California">CA</abbr>
  <span class="postal-code">94301</span>
  <div class="country-name">USA</div>
 </div>
 <div class="tel">
  <span class="type">Work</span> +1-650-556-1234
 </div>
 <div class="tel">
  <span class="type">Fax</span> +1-650-556-1235
 </div>
 <div>Email:
  <span class="email">info@munster.com</span>
 </div>
</div>
```

There are similar formats for calendars (*hCalendar*), opinions (*hReview*), social networks (*XFN*), geography (*geo*) and a few others. All are designed to add the information in a way that will be ignored by the browser. It will slip by the web browser like a dinosaur because the web browser isn't looking for it.

21.3.2 Rice's Theorem

One of the more interesting theorems from computer science theory suggests that it may be theoretically impossible for anyone to examine a program and determine whether it is packing extra information in a data file. This theorem is worth mentioning even if it may not have much practical use.

A casual version of the theorem, due to Henry Gordon Rice, states that a software program can't be counted on to detect whether another program is conforming to some standard for a file format. The theorem itself says that the problem is *undecidable*, a term that means that the program is guaranteed to halt and give a definitive answer. If it doesn't halt, it could go on checking, rechecking or looking for some complex answer.

This theorem may help establish a theoretical limit to checking for secret messages in file formats, but it may not be of practical value because the theoretical result is based on asking a computer program to examine itself, a sort of logical tongue-twister that can have odd

logical consequences. A less rigorous piece of software may be able to do a good job of testing for errant code.

The parsers for XML, for instance, can test to see whether XML conforms to a well-defined model. Extra tags and attributes are flagged and reported. Even if most software can't rely on these rules, they exist and do a good job of checking the data flowing along the wires. The XML standard, though, isn't Turing-complete and so it's possible to build a fairly straightforward testing tool.

21.4 Summary

Almost every data format has plenty of loopholes that can be used to add extra data. If the code reads the first n items on a line, you can stick more information after the n^{th} item. If there's a special end of file marker, say a zero, then you can add more after the zero. This technique makes it easy to add information in many cases.

A neat trick is mixing together two files with head-first and tail-first ordering of data like the GIF format and the ZIP formats. If these two parts are glued together, decoding algorithms will frequently fail to notice the other half. This lets a GIF hitch a ride on ZIP file and a ZIP file hitch a ride on a GIF.

The Disguise Information is stored in the spare corners of data files, a surprisingly easy process.

How Secure Is It? It may be theoretically impossible to detect that a piece of software is capable of reading or hiding extra data in a file. This theoretical barrier, though, may not have much practical weight.

How to Use It? The simplest solution may be to glue together a ZIP and a GIF file. Or just add extra nodes to an XML file.

Further Reading

There are a many data format books out there. It's impossible to list them all.

Chapter 22

Coda

As I've been writing this book, I've been haunted by the possibility that there may be something inherently evil or wrong in these algorithms. If criminals are able to hide information so effectively, justice becomes more elusive. There is less the police can do before a crime is committed and there is less evidence after the fact. All of the ideas in this book, no matter how philosophical or embellished with allegory or cute jokes, carry this implicit threat.

This threat became a bit more obvious after the destruction of the World Trade Center on September 11th, 2001. Some news reports offered the supposition that the attackers may have coordinated their efforts by hiding information in images. While there is no evidence that this occurred as I rewrite this chapter, there's no doubt that it could have occurred.

The U.S. Federal Bureau of Investigation, or at least its senior officers, are clearly of the opinion that they need ready access to all communications. If someone is saying it, writing it, mailing it, or faxing it, the Bureau would like to be able to listen in so they can solve crimes. This is a sensible attitude. More information can only help make sure that justice is fair and honest. People are convicted on the basis of their own words—not the testimony of stool pigeons who often point the finger in order to receive a lighter sentence.

The arguments against giving the FBI and the police such power are more abstract and anecdotal. Certainly, the power can be tamed if everyone follows proper procedures. If warrants are filed and chains of evidence are kept intact, the power of abuse is minimized. But even if the police are 100 times more honest than the average citizen, there will still be rogue cops on the force with access to the communications of everyone in the country. This is a very powerful

tool and the corruption brought by power is one of the oldest themes.

Both of these scenarios are embodied in one case that came along in the year before I began writing the first edition of the book. The story began when a New Orleans woman looked out the window and saw a police officer beating her son's friend. She called the Internal Affairs department to report the officer. By lunchtime, the officer in question knew the name of the person making the accusation, her address, and even what she was wearing. He allegedly ordered a hit and by the end of the day, she was dead.

How do we know this happened? How do we know it wasn't a random case of street violence? Federal authorities were in New Orleans following the officer and bugging his phone. He was a suspect in a ring of corrupt cops who helped the drug trade remain secure. They audiotaped the order for the hit.

There is little doubt that secure communications could have made this case unsolvable. If no one had heard the execution order except the killer, there would be no case and no justice.

There is also little doubt that a secure Internal Affairs office could have prevented the murder. That leak probably came from a colleague, but the corrupt cops could have monitored the phones of the Internal Affairs division. That scenario is quite conceivable. At the very least, the murder would have been delayed until a case was made against the officer. The right to confront our accusers in court means that it would have been impossible to keep her identity secret forever.

Which way is the right way? Total openness stops many crimes, but it encourages others forms of fraud and deceit. Total secrecy protects many people, but it gives criminals a cover.

One solution is to encrypt the session key with a special public key. Only the government has access to the private key.

In the past, the FBI and other parts of the law enforcement community suggested a system known as "key escrow" as a viable compromise. The escrow systems broadcast a copy of the session key in an encrypted packet that can only be read by designated people. Although Department of Justice officials have described extensive controls on the keys and on access to them, I remain unconvinced that there will not be abuse. If the tool is going to be useful to the police on the streets, they'll need fast access to keys. The audit log will only reveal a problem if someone complains that their phone was tapped illegally. But how do you know your phone was tapped? Only if you discover the tapes someone's hands.

There really is no way for technology to provide any ultimate solution to this problem. At some point, law enforcement authorities must be given the authority to listen in to solve a crime. The more this ability is concentrated in a small number of hands, the more

powerful it becomes and the more alluring the corruption associated with breaking the rules. Even the most dangerous secret owned by the United States, the technology for building nuclear weapons, was compromised by an insider. Is there any doubt that small-time criminals with needs won't be able to pull off small-time corruption across the country?

The depth and complexity of this corruption can lead to ironic situations. Robert Hanssen, an FBI agent who worked on counterespionage, turned out to be spying for the Russians at the same time. In September 2001, the head of the Cuban desk at the Defense Intelligence Agency was arrested and charged with being a spy for, of all places, Cuba. If the goal is to protect US information from Russian or Cuban ears, we must realize that putting eavesdropping technology in the hands of the FBI or the DIA may also be putting this technology at the disposal of foreign powers.

If giving into widespread eavesdropping is not a cure, then allowing unlimited steganography and cryptography is not one either. The technology described in this book offers a number of ways for information to elude the police dragnets. Encrypted files may look like secrets and secrets can look damning. Although the Fifth Amendment to the U.S. Constitution gives each person the right to refuse to incriminate themselves, there is little doubt that invoking that right can look suspicious.

But in the end it doesn't matter what they see or think they see. The terminals are equipped with holographic scanners, which decode the binary secret of every item, infallibly. This is the language of the waves and radiation, or how the dead speak to the living. And this is where we wait together, regardless of age, our carts stocked with brightly colored goods.
—Don DeLillo in White Noise

Mimic functions, anonymous remailers, and photographic steganography allow people to create files and hide them from sight. If no one can find them, no one can demand the encryption key. This may offer a powerful tool for the criminal element.

There are some consolations. Random violence on the street can't be stopped by phone taps. Muggers, rapists, robbers, and many other criminals don't rely on communications to do their job. These are arguably some of the most important problems for everyone and something that requires more diligent police work. None of the tools described in this book should affect the balance of power on the street.

Nor will banning cryptography or steganography stop terrorism. The hijackers used knives and guile, not cutting epigrams floating around in the netherworld of stegospace. The pen is not always mightier than the box cutter.

In reality, very few crimes can be readily solved through wiretaps because very little crime depends on communication and the exchange of information. Bribery of officials, for instance, is only committed when two people sit down in private and make an agreement. If the money can't be traced (as it so often can't be), then the only

way to prove the crime happened is to record the conversation or get someone to testify. Obviously, the recorded conversation is much more convincing evidence. State and local police in some states are not allowed to use wiretaps at all. Some police officers suggest that this isn't an accident. The politicians recognized that the only real targets for wiretaps were politicians. They were the primary ones who broke laws in conversation. Almost all other lawbreaking involved some physical act that might leave other evidence.

The power to create secret deals that this technology offers is numbing. The only consolation is that these deals have been made in the past and only a fool would believe that they won't be made in the future. No wiretap laws stopped them before cryptography became cheap. Meeting people face to face to conduct illegal business also has other benefits. You can judge people better in person. Also, locations like bars where such deals may be made offer drinks and often food. You can't get that in cyberspace.

The fact that crooks found ways to elude wiretaps in the past can be easily extended to parallel a popular argument made by the National Rifle Association. If cryptography is outlawed, only outlaws will have cryptography. People who murder, smuggle, or steal will probably not feel much hesitation to violate a law that simply governs how they send bits back and forth. Honest people who obey any law regulating cryptography will find themselves easy marks for those who want to steal their secrets.

I mulled over all of these thoughts while I was writing this book. I've almost begun to feel that the dispute is not really about technology. If criminals can always avoid the law, this won't change with more technology. The police have always been forced to adapt to new technology and they will have to do the same here.

In the end, I began concentrating on how to balance the power relationships. If power can be dispersed successfully, then the results of abuse can be limited. If individuals can control their affairs, then they are less likely to be dominated by others. If they're forced to work in the open, then they're more likely to be controlled.

The dishonest will never yield to the law that tells them not to use any form of steganography. The cliché of gangsters announcing the arrival of a shipment of 10,000 bananas will be with us forever. The question is whether the honest should have access to the tools to protect their privacy. Cryptography and steganography give individuals this power and this is, for better or for worse, the best place that it should be.

Appendix A

Software

Many programs can also be found in numerous archives. Some of the better ones include:

- http://www.stegoarchive.com/
- http://crypto.radiusnet.net/archive/
- http://www.cl.cam.ac.uk/f̃app2/steganography/
- http://www.geocities.com/Paris/9955/priv.html
- http://www.student.seas.gwu.edu/s̃owers/digwat.html
- http://www.jjtc.com/Steganography/tools.html
- http://www.watermarkingworld.org/
- http://www.funet.fi/pub/crypt/steganography/
- http://glu.freeservers.com/stegano.htm

A.1 Commercial Packages

http://www.bluespike.com/ This company led by Scott Moskowitz holds a number of significant patents for watermarking and embedding information in files The Giovanni watermarking system offers point-and-click watermarking.

http://www.neobytesolutions.com/ Invisible Secrets will encrypt and hide information in your file system.

http://www.datamark-tech.com DataMark offers digital watermarking technology for content management.

http://pc-magic.com/ Magic Folders from PC-Magic will hide and encrypt files in your file system.

http://www.steganos.com The Internet Privacy Suite from Steganos includes a version of Hide and Seek.

http://www.signumtech.com/ Signum builds watermarking software.

http://www.mediasec.com Mediasec offers MediaSign and MediaTrust tools for embedding digital signatures in files.

http://web.clicknet.ro/xidie/stegano.html The Xidie Security Suite, by Laic Aurelian, compresses, encrypts, and embeds information in a wide variety of files, including most image and sound formats. It is free for noncommercial use and a commerical license is available.

A.2 Open Packages

These packages are either open source, freeware or shareware:

http://www.wbailer.com/wbstego wbStego is a tool written by Werner Bailer for inserting information into the least significant bits of BMP files which also includes options for embedding information in the PDF, HTML and ASCII files. The core tools include a variety of different encryption algorithms as well as the ability to spread out the hidden information to different locations inside the file.

http://skyjuicesoftware.com Data Stash from Lim Chooi Guan of SkyJuice Software will hide data in a wide variety of files.

http://www.pariahware.com Pict Encrypt from Pariahware is a basic tool for embedding some encrypted text in an image. It runs on the Macintosh and does not include much documentation about the algorithm.

http://wwwrn.inf.tu-dresden.de/w̃estfeld/f5.html The F5 package encodes information in JPEG images using a technique to thwart visual and statistical attacks. [Wes01]

http://diit.sourceforge.net/ The Digital Invisible Ink Toolkit written by Kathryn Hempstalk includes several algorithms for encoding information and several algorithms for analyzing the results to look for steganography. The hiding algorithms all work on the least significant bits. The more sophisticated algorithms use Sobel filters or Laplace filters to flag the best pixels for hiding information. These tend to flag the edges between objects where the intensity of the image changes quickly.

http://web.clicknet.ro/xidie/stegano.html The Xidie Security Suite by Laic Aurelian compresses, encrypts and embeds information in a wide variety of files including most image and sound formats. It is free for non-commercial use and a commerical license is available.

http://www.hermetic.ch/hst/hst.htm Hermetic Stego will hide information in BMP files. For Windows.

http://www.heinz-repp.onlinehome.de/Hide4PGP.htm Hide4PGP by Heinz Repp will hide information in the least significant bits of BMP and WAV files. The source and versions for Windows and Linux are available.

http://www.fourmilab.ch/stego Steganosaurus, from John Walker, is a C-based program that uses a dictionary to turn bits into gibberish. [Wal94]

ftp://ftp.hacktic.nl/pub/crypto/macintosh/ Paranoid, by Nathan Mariels, will encrypt information and hide it in sound files.

http://cypherspace.org/adam/stealth/ PGP Stealth, from Adam Back, will strip off all of the headers from PGP files, producing something that should be random.

http://www.nic.funet.fi/pub/crypt/steganography/ Texto, by Kevin Maher, is an text steganography program that uses some basic grammars. [Mah95]

http://www.stego.com/ Romana Machado distributed the Java version of her Stego and EzStego software from here. This cross-platform tool hides information in the least significant bit of an image *after* the colors in the image are sorted. This usually works quite well, but there can be some inconsistencies. The software was distributed with the GNU Public License.

http://wwwrn.inf.tu-dresden.de/w̃estfeld/f5.html The F5 software used for hiding information in JPEG images includes a number

of enhancements designed to avoid steganalytic techniques discovered by the creator, Andreas Westfeld. [Wes01, WP99]

http://www.mcdonald.org.uk/StegFS/ This is the source for the Steganographic File System described in Section 4.5. This software works well with Linux file systems and can probably be extended to any other file systems with some work. It is released under the GNU GPL.

http://www.smalleranimals.com/stash.htm The StashIt software hides data in the least significant bits of images with five different techniques. There is no charge for it.

http://www.darkside.com.au/snow/ The Snow software, developed by Matthew Kwan, will insert extra spaces at the end of each line. Three bits are encoded in each line by adding between 0 and 7 spaces that are ignored by most display programs, including web browsers.

http://www.infonex.com/m̃ark/pgp/m-readme.html The MandelSteg software hides information in the least significant bit of an image of the Mandelbrot set. The set can be synthesized for any set of coordinates in the plane with seven bits of accuracy. The last bit is the message.

http://www.stella-steganography.de/ The Stella (Steganography Exploration Lab) software is both a tool for hiding information in bitmaps and a lab for exploring how hidden the information may be. The software includes a number of different tools for taking apart the images to see the effects.

http://www.darkside.com.au/gifshuffle/ The Gifshuffle program written by Matthew Kwan hides information in the ordering of the palette of an image. If there are $n!$ different ways to arrange n objects, then $\log_2(n!)$ bits can be hidden in the choice of which sorting to choose. GifShuffle hides 209 bytes in the way that it selects 256 colors.

http://glu.freeservers.com/sgpo.htm David Glaude and Didier Barzin created this program (SteganoGifPaletteOrder) that hides information in the permutation of the colors in the GIF palette in the same manner as GifShuffle.

http://www.spammimic.com David McKellar created one grammar that encodes message in spam-like phrases removed from his collection of spam messages. It is based on the algorithms described in Chapter 7.

http://www.steganos.com/ Steganos sells a suite of security products that includes The Safe, a "hard drive that disappears at the click of a button. "

http://www.tiac.net/users/korejwa/jsteg.htm This is the JSteg software enhanced with a Windows shell.

http://linux01.gwdg.de/ãlatham/stego.html The JPHide and JPSeek programs, written by Allan Latham, hide information in the JPEG coefficients using classical algorithms. The software keeps track of the change in the statistical profile of the coefficients to help you avoid steganalysis. (See Chapter 17.)

http://www.compris.com/subitext/ Compris sells TextHide and TextSign,a software programs that hide information by changing the structure of sentences. The text should, in theory, say the same thing after the extra information is inserted.

http://www.ctgi.net/nicetext/ Mark Chapman created NiceText as his master's thesis project during his time at the University of Wisconsin studying with George Davida. The software assembles a dictionary and classifies words to make it possible to approximate styles while also hiding information in text. [CD97] See also `http://www.nicetext.com`.

http://www.datamark-tech.com/ DataMark Technologies sells four programs using steganography. One offers watermarking, one embeds raw information, one adds a digital signature to an image, and one builds a "safe".

http://www.stealthencrypt.com/ Stealth Encrypt bundles a steganography wizard with its security suite.

http://www.heinz-repp.onlinehome.de/Hide4PGP.htm Hide4PGP stores data in the least significant bit of either BMP or WAV files. It's a small, free program.

http://www.blindside.co.uk/ BlindSide hides information in bitmapped images after using a proprietary encryption algorithm for extra protection.

http://steghide.sourceforge.net/ The Steghide software is a GPL-protected package started by Stefan Hetzl for hiding information in the least significant bits of images (BMPs) or sound files (WAV or AU).

http://www.brasil.terravista.pt/Jenipabu/2571/e_hip.htm Hide In Picture stores information in the least significant bits of image files.

http://www.intar.com/ITP/itpinfo.htm In The Picture hides information in 4-bit, 8-bit and 24-bit images. The software can also store multiple files protected with different passwords.

http://sourceforge.net/projects/mixmaster/ MixMaster is an excellent set of tools for running and using anonymous remailers.

http://members.nbci.com/fredc/encryptpic.html EncryptPic hides information in 24-bit BMP images after scrambling them with the Cast algorithm.

ftp://idea.sec.dsi.unimi.it/pub/security/crypt/code/s-tools4.zip Andrew Brown wrote S-Tools, one of the first programs for hiding information in image and sound files.

http://www.neobytesolutions.com/invsecr/index.htm Invisible Secrets is a shareware program for storing information in the usual places. It is a well-designed and highly polished program. A version supported by banner ads is also available.

http://www.neobytesolutions.com/invsecr/index.htm S-Mail hides information in x86 executable files (.exe or .dll) The programs still work after the information is inserted.

http://www.camouflagesoftware.co.uk/ Camouflage is a basic tool for compressing, encrypting and then appending the information to the end of a file. The information isn't inserted steganographically into the actual data, it's just stuck at the end. This is often good enough an is guaranteed not to leave any distortion to the cover file.

http://wbstego.wbailer.com/ wbStego is a polished, professional tool for hiding information in sound, image and text formats. The latest version can also store them in Adobe PDF files in order to help establish ownership.

http://www.scramdisk.clara.net/ If you want to hide information in a scrambled directory on your hard drive, Scramdisk provides the mechanism.

http://www.petitcolas.net/fabien/steganography/mp3stego/index.html Fabien A. P. Petitcolas created MP3Stego for hiding information

in the very popular MP3 files. The mechanism tweaks the parity of some of the quantized coefficients chosen using a random number generator. [AP98]

http://www.outguess.org/ Niels Provos built the Outguess system to hide information in JPEG files without distorting the statistical profile. He also distributes the StegDetect program, which will detect distortions in other steganographic systems.

http://www.psionic.com/papers/covert/ Psionic Software created this package for hiding information in the redundant or optional bits of the TCP/IP headers. (The IP packet identification field, the TCP initial sequence number field and the TCP acknowledged sequence number field.)

http://sandmark.cs.arizona.edu/ The SandMark tool, by Christian Collberg, will embed information while obfuscating Java byte code.

ftp://ftp.funet.fi/pub/crypt/steganography/PGM.stealth.c.gz PGM-Stealth hides data in the least significant bits of PGM files on UNIX boxes.

ftp://ftp.funet.fi/pub/crypt/steganography/piilo061195.tar.gz Piilo hides data in the least significant bits of PGM files on UNIX boxes.

http://www.cl.cam.ac.uk/f̃app2/watermarking/stirmark/ The StirMark software helps test watermarking or image steganographic methods by scrambling the images in subtle ways. The software treats the image like a rubber sheet by stretching some parts, blurring other parts, destroying some parts, and even duplicating small parts. The meddling is controlled with parameters so watermark creators can make claims like, "This software resists StirMark at settings up to 1.5."

A.3 Steganalysis Software

http://www.spy-hunter.com/stegspydownload.htm StegSpy, by Bill Englehardt, identifies some of the more common signatures used in steganography programs.

http://www.outguess.org/ Niels Provos built the StegDetect system to detect statistical differences in files with embedded messages. He also distributes the OutGuess program, which hides information in JPEG files.

http://www.sarc-wv.com/ The Steganography Analysis and Research Center (SARC), a division of BackBone Security, builds the Steganography Analyzer Signature Scanner (StegAlyzerSS) and the Steganography Analyzer Artifact Scanner (StegAlyzerAS), two tools that detect unique statistical signatures or artifacts added by some common steganography programs.

Bibliographic Notes

This book is quite incomplete because it offers the reader an introduction to many of the topics. Some topics are simply left out because of time and space constraints. Others are only touched briefly. This preface to the bibliography is intended to offer some suggestions for further reading and exploration.

A good place to begin is with history. David Kahn's *Codebreakers* is an excellent survey of the history of cryptology [Kah67]. There are numerous descriptions of steganographic solutions like secret inks and microdots. More recent histories are published in *Cryptologia*.

There are a number of other good books on the subject. Stefan Katzenbeisser and Fabien A.P. Petitcolas edited a collection of essays from the leading researchers entitled *Information Hiding Techniques for Steganography and Digital Watermarking*. [SKE00] Neil Johnson, Zoran Duric, and Sushil Jajodia's recent addition, *Information Hiding: Steganography and Watermarking*, is the first part of a series. [JDJ01]. Ross Anderson's general survey, *Security Engineering*, also includes some information on steganography and watermarking.[And01]

Some of the best material can be found, in it's original form, in the Proceedings of the Information Hiding Workshop. There have been nine conferences and more are on their way.

Other more specific information can be found in these areas.

Error-Correcting Codes The chapter in this book can not do justice to this wide field. There are many different types of codes with different applications. Some of the better introductions are: [LJ83] and [Ara88].

Compression Algorithms Compression continues to be a hot topic and many of the latest books aren't current any longer. The best solution is to combine books like [Bar88, BS88] with papers from the the procedings from academic conferences like

[Kom95]. I also wrote an introductory book on compression last millenium. [Way99]

Subliminal Channels This idea is not covered in the book, but it may be of interest to many readers. Much of the work in the area was done by Gus Simmons, who discovered that many digital signature algorithms had a secret channel that could be exploited to send an extra message. [Sim84, Sim85, Sim86, Sim93, Sim94] This is pretty easy to understand in the abstract. Many of the algorithms, like the Elgamal signature scheme [ElG85] or the Digital Signature Algorithm [NCS93] create a new digital signature at random. Many valid signatures exist and the algorithm simply picks one at random. It is still virtually impossible for someone without the secret key to generate one, but the algorithms were intended to offer authentication without secrecy.

Imagine that you want to send a one bit message to someone. The only encryption software you can use is a DSA signature designed not to hide secrets. You could simply send along a happy message and keep recomputing the digital signature of this message until the last bit is the bit of your message. Eventually, you should find one because the algorithm chooses among signatures at random.

This abstract technique only shows how to send one bit. There are many extra bits available for use and the papers describe how to do the mathematics and exploit this channel.

The algorithms form an important basis for political discussions about cryptography. The U.S. Government would like to allow people to use authentication, but they would like to restrict the use of secrecy-preserving encryption. Algorithms like the DSA appear to be perfect compromises. The existence of subliminal channels, however, shows how the current algorithms are not a perfect compromise.[1]

Covert Channels This is, in many ways, just an older term for the same techniques used in this book. The classic example comes from operating system design: Imagine that you run a computer system that has an operating system that is supposed to be secure. That means the OS can keep information from traveling between two users. Obviously, you can implement such an OS by shutting down services like file copying or electronic

[1] They may be a perfectly adequate practical compromise because implementing the software to use this additional channel is time consuming.

mail. It is not clear, however, that you can completely eliminate every way of communicating.

The simplest example for sending a message is to tie up some shared resource like a printer. If you want to send a '1' to a friend, then you print a file at 12:05 and tie up the printer. If you want to send a '0', then you print the file at 12:30. The other person checks the availability of the printer. This may not be a fast method, but it could work. The speed of the channel depends on the shared system resources and the accuracy of detection. Obviously one way to defend against covert channels is to create timing errors, but then that just creates other problems.

Some beginning sources are [NCS93, PN93, MM92]

Digital Cash There are many different ways to exchange money over digital wires, but some of the most interesting systems offer complete anonymity. People are able to spend their money without fear of records being kept. This is a fairly neat trick because digital cash must be counterfeit-resistant. Paper cash achieves this goal when it is printed with a sophisticated press. Digital copies, on the other hand, are easy to make. If people can copy files of numbers meant to represent cash, then anonymity would seem to allow people the freedom to counterfeit without being caught.

The cleverest schemes involve a complicated spending system that forces the spender to reveal part of their identity. If the spender tries to use a bill twice, enough of the identity should be revealed to expose the criminal.

Anonymous Voting People often want to cast their votes anonymously because this can prevent coercision. Paper ballots are generally successful if no one checks the ballot before they enter the box. Providing the same accountability and security is no simple feat.

Interest in this topic is very high and there are enough good algorithms to justify a separate book. K. Sako and J. Kilian [SK95], for instance, modified the Mixmaster protocol described in Chapter 10 to provide a simple way for people to cast their vote. Each person can check the tally and compare their vote to the recorded vote to guarantee that the election was fair.

Many of the newer systems rely upon the homomorphic encryption systems that allow manipulation of encrypted data.

One notable examples includes the work of Martin Hirt, Kazue Sako and Joe Kilian [HS00, Hir01, SK94].

There are many features in the different Sensus system from Lorrie Faith Cranor and Ron K. Cytron, for instance, provides the user the ability to vote for one person but effectively hide and this fact from others.

Other systems include [BY86, Boy90, FOO93, CC96].

Finally, newer and better papers can be found through electronic paper archives like the CiteSeer system run by NEC (`http://citeseer.nj.nec.com/`). This is an invaluable source of knowledge.

Bibliography

[AFYZ04] James Aspnes, Joan Feigenbaum, Aleksandr Yampolskiy, and Sheng Zhong. Towards a theory of data entanglement. In *Proceedings of the 2004 European Symposium on Research in Computer Security*, 2004.

[Age95] National Security Agency. N.S.A. press release: Venona documents released. Technical report, National Security Agency, July 1995.

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[AK91] Dana Angluin and Michael Kharitonov. When won't membership queries help? In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 444–454. ACM Press, 1991. to appear in *JCSS*.

[AKSK00] S. Areepongsa, N. Kaewkamnerd, Y. F. Syed, and K.R.Rao. Exploring steganography for low bit rate wavelet based coder in image retrieval system. In *Proceedings of TENCOM '00*, volume 3, pages 250–255, Kuala Lumpur, Malaysia, September 2000.

[AMRN00] Mikhail J. Atallah, Craig J. McDonough, Victor Raskin, and Sergei Nirenburg. Natural language processing for information assurance and security: an overview and implementations. In *NSPW '00: Proceedings of the 2000 workshop on New security paradigms*, pages 51–65, New York, NY, USA, 2000. ACM.

[And96a] R. Anderson. The eternity service, 1996.

[And96b] Ross J. Anderson, editor. *Information Hiding, First International Workshop, Cambridge, U.K., May 30 - June 1,*

1996, Proceedings, volume 1174 of *Lecture Notes in Computer Science*. Springer, 1996.

[And96c] Ross J. Anderson. Stretching the limits of steganography. In *Information Hiding* [And96b], pages 39–48.

[And01] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 2001.

[And07] Nate Anderson. Steroid bust shows feds can still get at 'private' and 'secure' e-mail. *Ars Technica.com*, November 2007.

[ANS98] Ross Anderson, Roger Needham, and Adi Shamir. The steganographic file system. In *IWIH: International Workshop on Information Hiding*, 1998.

[AP98] Ross Anderson and Fabien Petitcolas. On the limits of steganography. *IEEE Journal on Selceted Areas in Communications*, pages 474–481, May 1998.

[Ara88] Benjamin Arazi. *A Commonsense Approach to the Theory of Error Correcting Codes*. MIT Press, Cambridge, MA, 1988.

[ARC+01] Mikhail J. Atallah, Victor Raskin, Michael Crogan, Christian Hempelmann, Florian Kerschbaum, Dina Mohamed, and Sanket Naik. Natural language watermarking: Design, analysis, and a proof-of-concept implementation. In Ira S. Moskowitz, editor, *Information Hiding: Fourth International Workshop*, volume 2137 of *Lecture Notes in Computer Science*, pages 185–199. Springer, April 2001.

[ARH+02] Mikhail J. Atallah, Victor Raskin, Christian F. Hempelmann, Mercan Karahan, Radu Sion, Umut Topkara, and Katrina E. Triezenberg. Natural language watermarking and tamperproofing. In Fabien A. P. Petitcolas, editor, *Information Hiding: Fifth International Workshop*, volume 2578 of *Lecture Notes in Computer Science*, pages 196–212. Springer, October 2002.

[ARH+03] Mikhail J. Atallah, Victor Raskin, Christian Hempelmann, Mercan Karahan, Radu Sion, Umut Topkara, and Katrina E. Triezenberg. Natural language watermarking and tamperproofing. In *IH '02: Revised Papers from*

the 5th International Workshop on Information Hiding, pages 196–212, London, UK, 2003. Springer-Verlag.

[Aur95] Tuomas Aura. Invisible communication. Technical report, Helsinki University of Technology, November 1995.

[Bar88] Michael F. Barnsley. Fractal modelling of real world images. In Heinz-Otto Peitgen and Dietmar Saupe, editors, *The Science of Fractal Images*, chapter 5, pages 219–239. Springer-Verlag, 1988.

[Bar93] Michael F. Barnsley. *Fractals Everywhere*. Academic Press, Cambridge, MA, USA, 2nd edition, 1993.

[BB00a] Mihir Bellare and Alexandra Boldyreva. The security of chaffing and winnowing. In *ASIACRYPT*, pages 517–530, 2000.

[BB00b] Mihir Bellare and Alexandra Boldyreva. The security of chaffing and winnowing. In *ASIACRYPT '00: Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security*, pages 517–530, London, UK, 2000. Springer-Verlag.

[BBWBG98] S. Blackburn, S. Blake-Wilson, M. Burmester, and S. Galbraith. Shared generation of shared rsa keys. Technical Report CORR98-19, Department of Combinatorics and Optimization, University of Waterloo, 1998.

[BC05] Michael Backes and Christian Cachin. Public-key steganography with active attacks. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 210–226. Springer, 2005.

[BCK96a] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. *Lecture Notes in Computer Science*, 1109:1–??, 1996.

[BCK96b] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message authentication using hash functions: the HMAC construction. *CryptoBytes*, 2(1):12–15, Spring 1996.

[Ben04] Krista Bennett. Linguistic steganography: Survey, analysis, and robustness concerns for hiding information in text. Research paper accepted in partial fulfillment of the Dept. of Linguistics preliminary examination requirement, 05 2004. CERIAS TR 2004-13.

[BF97] D. Boneh and M. Franklin. Efficient generation of shared RSA keys. *Lecture Notes in Computer Science, CRYPTO 97*, 1294:425–??, 1997.

[BFHMV84] I.F. Blake, R. Fuji-Hara, R.C. Mullin, and S.A. Vanstone. Computing logarithms in finite fields of characteristic two. *SIAM Journal on Algebraic Discrete Methods*, 5, 1984.

[BG07] Nikita Borisov and Philippe Golle, editors. *Privacy Enhancing Technologies, 7th International Symposium, PET 2007 Ottawa, Canada, June 20-22, 2007, Revised Selected Papers*, volume 4776 of *Lecture Notes in Computer Science*. Springer, 2007.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of Crypto 2001, Springer Verlag LNCS Volume 2139*, 2001.

[BGML96] Walter Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *IBM Systems Journal*, 35(3):313, 1996.

[BH92] M. Barnsley and L. Hurd. *Fractal Image Compression*. AK Peters, Ltd., Wellesley, Ma., 1992.

[BJN00] Dan Boneh, Antoine Joux, and Phong Q. Nguyen. Why textbook elgamal and RSA encryption are insecure. In *ASIACRYPT '00: Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security*, pages 30–43, London, UK, 2000. Springer-Verlag.

[BJNW57] F.P. Brooks, A.L Hopkins Jr., Peter G. Neumann, and W.V. Wright. An experiment in musical composition. EC-6(3), September 1957.

[BL85] Charles Bennett and Rolf Landauer. The fundamental physical limits of computation. *Scientific American*, pages 48–56, July 1985.

[BLMO94] J Brassil, S Low, N Maxemchuk, and L O'Garman. Electronic marking and identification techniques to discourage document copying. In *Proceedings of IEEE Infocom 94*, pages 1278 – 1287, 1994.

[BLMO95] Jack Brassil, Steve Low, Nicholas Maxemchuk, and Larry O'Gorman. Hiding information in document images. In *Proceedings of the 1995 Conference on Information Sciences and Systems*, March 1995.

[Blu82] M. Blum. Coin flipping by telephone: A protocol for solving impossible problems. *Proceedings of the 24th IEEE Computer Conference (CompCon)*, 1982.

[BMS01] Adam Back, Ulf Moeller, and Anton Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Fourth Information Hiding Workshop*, pages 257–269, 2001.

[BO96] Jack Brassil and Larry O'Gorman. Watermarking document images with bounding box expansion. In *Information Hiding, Lecture Notes of Computer Science (1174)*, New York, Heidelberg, 1996. Springer-Verlag.

[Boy90] C. Boyd. A new multiple key cipher and an improved voting scheme. In *Advances in Cryptology–EUROCRYPT '89 Proceedings*. Springer-Verlag, 1990.

[Bra93] S.A. Brands. An efficient off-line electronic cash system based on the representation problem. Technical Report CSR9323, Computer Science Department, CWI, Mar 1993.

[Bra95a] Stefan A. Brands. *Rethinking Publike Key Infrastructure and Digital Certificates– Building in Privacy*. PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[Bra95b] Stefan A. Brands. Secret-key certificates. Technical Report CS-R9510, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[Bri82] E.F. Brickell. A fast modular multiplication algorithm with applications to two key cryptography. In *Advances in Cryptology: Proceedings of Crypto 82*. Plenum Press, 1982.

[BS88] Michael F. Barnsley and Alan D. Sloan. A better way to compress images. *Byte Magazine*, pages 215–223, January 1988.

[BS95] D Boneh and J Shaw. Collusion-secure fingerprinting for digital data. In *15th Annual International Cryptology Conference*, number 963, pages 452–465, Santa Barbara, California, U.S.A., 27–31 1995.

[BS99] R.W. Buccigrossi and E.P. Simoncelli. Image compression via joint statistical characterization in the wavelet domain. *IEEE Transactions on Image Processing*, 8(12):1688–1701, 1999.

[BSMA05] Sevinc Bayram, Husrev T. Sencar, Nasir D. Memon, and Ismail Avcibas. Source camera identification based on cfa interpolation. In *ICIP (3)*, pages 69–72, 2005.

[BVR05] Nader Bagherzadeh, Mateo Valero, and Alex Ramírez, editors. *Proceedings of the Second Conference on Computing Frontiers, 2005, Ischia, Italy, May 4-6, 2005*. ACM, 2005.

[BY86] J.C. Benaloh and M. Yung. Distributing the power of government to enhance the privacy of voters. *Proceedings of the 5th ACM Symposium on the Principles in Distributed Computing*, 1986.

[CC96] Lorrie Cranor and R. Cytron. Design and implementation of a practical security-conscious electronic polling system. Technical Report WUCS-96-02, Washington University Department of Computer Science, St. Louis, 1996.

[CCJS07] Jan Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee, editors. *Information Hiding, 8th International Workshop, IH 2006, Alexandria, VA, USA, July 10-12, 2006. Revised Selected Papers*, volume 4437 of *Lecture Notes in Computer Science*. Springer, 2007.

[CD97] Mark Chapman and George Davida. Hiding the hidden: A software system for concealing ciphertext as innocuous text. In *International Conference on Information and Computer Security (ICICS'97)*, Beijing, P.R. China, November 1997.

[CFN93] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In *Proceedings of Crypto 88*, page ???, New York, Berlin, Heidelberg, London, Paris, Tokyo, 1993. Springer-Verlag.

[Cha81] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb 1981.

[Cha95a] David Charlap. The bmp file format, part i. *Dr. Dobbs Journal*, Mar 1995.

[Cha95b] David Charlap. The bmp file format, part ii. *Dr. Dobbs Journal*, Apr 1995.

[Che00] Brian Chen. *Design and Analysis of Digital Watermarking, Information Embedding, and Data Hiding Systems*. PhD thesis, MIT, Cambridge, MA, June 2000.

[CKL05] Ingemar J. Cox, Ton Kalker, and Heung-Kyu Lee, editors. *Digital Watermarking, Third InternationalWorkshop, IWDW 2004, Seoul, SouthKorea, October 30 - November 1, 2004, Revised Selected Papers*, volume 3304 of *Lecture Notes in Computer Science*. Springer, 2005.

[CKLS96] Ingemar Cox, Joe Kilian, Tom Leighton, and Talal Shamoon. A secure, robust watermark for multimedia. In *Information Hiding, Lecture Notes of Computer Science (1174)*, New York, Heidelberg, 1996. Springer-Verlag.

[CKM+06] R. C. Chakinala, Abishek Kumarasubramanian, R. Manokaran, G. Noubir, C. Pandu Rangan, and Ravi Sundaram. Steganographic communication in ordered channels. In Camenisch et al. [CCJS07], pages 42–57.

[Cla83] F. Clarke. *Optimization and Nonsmooth Analysis*. John Wiley, New York, 1983.

[Cla99] Ian Clarke. A distributed decentralised information storage and retrieval system. Technical report, Division of Informatics, University of Edinborough, 1999.

[CM58] Noam Chomsky and G.A. Miller. Finite state languages. *Information and Control*, 1:91–112, 1958.

[CM97] Ingemar J. Cox and Matt L. Miller. A review of watermarking and the importance of perceptual modeling. In *Proc. of Electronic Imaging '97*, February 1997.

[CM98] Camenisch and Michels. A group signature scheme with improved efficiency. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*. LNCS, Springer-Verlag, 1998.

[CMB01] Ingemar J. Cox, Matthew L. Miller, and Jeffrey A. Bloom. *Digital Watermarking*. Morgan Kaufman, San Fransisco, CA, 2001.

[CMB02] Ingemar Cox, Matthew L. Miller, and Jeffery A. Bloom. *Digital watermarking*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[CMB07] Ingemar Cox, Matthew L. Miller, and Jeffery A. Bloom. *Digital watermarking and Steganography*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[CMBF08] Ingemar J. Cox, Matthew Miller, Jeffrey Bloom, and Jessica Fridrich. *Digital Watermarking*. Morgan Kaufman, San Francisco, CA, 2008.

[Cop85] Don Coppersmith. Another birthday attack. In Hugh C. Williams, editor, *CRYPTO*, volume 218 of *Lecture Notes in Computer Science*, pages 14–17. Springer, 1985.

[Cri90] Michael Crichton. *Jurassic Park*. Knopf, New York, 1990.

[CSWH00] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000.

[CT02] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proffing, and obfuscation: tools for software protection. *IEEE Trans. Softw. Eng.*, 28(8):735–746, 2002.

[CW93] K.W. Campbell and M.J. Wiener. DES is not a group. In *Advances in Cryptology–CRYPTO '92 Proceedings*. Springer-Verlag, 1993.

[CW99] B. Chen and G. Wornell. Achievable performance of digital watermarking systems, 1999.

[CW00] Brian Chen and Greg Wornell. Quantization index modulation: A class of provably good methods for digital watermarking and information embedding (1 page). In *ISIT: Proceedings IEEE International Symposium on Information Theory, sponsored by The Information Theory Society of The Institute of Electrical and Electronic Engineers*, 2000.

[dAJ05] Rennie deGraaf, John Aycock, and Michael Jr. Jacobson. Improved port knocking with strong authentication. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 451–462, Washington, DC, USA, 2005. IEEE Computer Society.

[DDM03] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 2, Washington, DC, USA, 2003. IEEE Computer Society.

[deG07] Reinderd Gordon Nathan deGraaf. *Enhancing Firewalls: Conveying User and Application Identification to Network Firewalls*. PhD thesis, University of Calgary, 2007.

[DF00] D. M. Roger Dingledine and Michael J. Freedman. The Free Haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[DMS03] Roger Dingledine, Nick Mathewson, and Paul Syverson. Reputation in p2p anonymity systems, 2003.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[DR00] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *Smart Card Research and Applications, LNCS 1820*, pages 288–296. Springer-Verlag, 2000.

[DR01] Joan Daemen and Vincent Rijmen. Rijndael, the advanced encryption standard. *Dr. Dobb's Journal*, 26(3):137–139, March 2001.

[DS03] Claudia Díaz and Andrei Serjantov. Generalising mixes. In Roger Dingledine, editor, *Proceedings of Privacy Enhancing Technologies workshop (PET 2003)*, pages 18–31. Springer-Verlag, LNCS 2760, March 2003.

[DW04] Edward J. Delp and Ping Wah Wong, editors. *Security, Steganography, and Watermarking of Multimedia Contents VI, San Jose, California, USA, January 18-22, 2004, Proceedings*, volume 5306 of *Proceedings of SPIE*. SPIE, 2004.

[DW05] Edward J. Delp and Ping Wah Wong, editors. *Security, Steganography, and Watermarking of Multimedia Contents VII, San Jose, California, USA, January 17-20, 2005, Proceedings*, volume 5681 of *Proceedings of SPIE*. SPIE, 2005.

[ed.92] G.J. Simmons ed. *Contemporary Cryptology: The Science of Information Integrity*. IEEE Press, Piscataway, NJ, 1992.

[EKK04] Rakan El-Khalil and Angelos D. Keromytis. Hydan: Hiding Information in Program Binaries. In *Proceedings of the* 6^{th} *International Conference on Information and Communications Security (ICICS)*, pages 187–199, October 2004.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology: Proceedings of CRYPTO 84*. Springer-Verlag, 1985.

[ESG00a] Joachim J. Eggers, Jonathan K. Su, and Bernd Girod. Asymmetric watermarking schemes. In *Sicherheit in Mediendaten*, Berlin, Germany, September 2000.

[ESG00b] Joachim J. Eggers, Jonathan K. Su, and Bernd Girod. Public-key watermarking by eigenvectors of linear transforms. In *EUSIPCO 2000*, Tampere, Finland, September 2000.

[Ett98] J. Mark Ettinger. Steganalysis and game equilibria. In *Information Hiding Workshop, Lecture Notes of Computer Science (1525)*, New York, Heidelberg, 1998. Springer-Verlag.

[FBS96] J. Fridrich, Arnold Baldoza, and Richard Simard. Robust digital watermaking based on key-dependent basis functions. In *Information Hiding Workshop, Lecture Notes of Computer Science (1525)*, New York, Heidelberg, 1996. Springer-Verlag.

[FD99] J. Fridrich and Rui Dui. Secure steganographic methods for palette images. In *3rd Information Hiding Workshop, Lecture Notes of Computer Science (1768)*, New York, Heidelberg, 1999. Springer-Verlag.

[FDL00] J. Fridrich, Rui Du, and Meng Long. Steganalysis of LSBencoding in color images. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, August 2000.

[FG99] J. Fridrich and M. Goljan. Protection of digital images using self embedding. In *Proceedings of NJIT Symposium on Content Security and Data Hiding in Digital Media*, Newark, NJ, May 1999.

[FGLS05] Jessica J. Fridrich, Miroslav Goljan, Petr Lisonek, and David Soukal. Writing on wet paper. In Delp and Wong [DW05], pages 328–340.

[FGS04] Jessica Fridrich, Miroslav Goljan, and David Soukal. Perturbed quantization steganography with wet paper codes. In *MM&Sec '04: Proceedings of the 2004 workshop on Multimedia and security*, pages 4–15, New York, NY, USA, 2004. ACM.

[FLN07] Pierre-Alain Fouque, Gaëtan Leurent, and Phong Q. Nguyen. Full key-recovery attacks on HMAC/NMAC-MD4 and NMAC-MD5. In *Proc. CRYPTO '07*, volume 4622 of *Lecture Notes in Computer Science*, pages 13–30. Springer, 2007.

[FMY98] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed RSA-key generation. In *Symposium on Principles of Distributed Computing*, page 320, 1998.

[FOO93] A. Fujioka, T. Okamoto, and K. Ohta. A practical secret voting scheme for large scale elections. In *Advances in Cryptology–AUSCRYPT '92 Proceedings*. Springer-Verlag, 1993.

[Fou98] Electronic Frontier Foundation. *Cracking DES :Secrets of Encryption Research, Wiretap Politics and Chip Design*. O'Reilly, 1998.

[Fra05] Michael P. Frank. Introduction to reversible computing: motivation, progress, and challenges. In Bagherzadeh et al. [BVR05], pages 385–390.

[Fre82] Ed Fredkin. Conservative logic. *International Journal of Theoretical Physics*, 21, 1982.

[Fri99] J. Fridrich. A new steganographic method for palette-based images. In *Proceedings of the IS&T PICS Conference*, pages 285–289, Savannah, Georgia, April 1999.

[Fri04] Jessica J. Fridrich. Feature-based steganalysis for jpeg images and its implications for future design of steganographic schemes. In Jessica J. Fridrich, editor, *Information Hiding*, volume 3200 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2004.

[FT82] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[Fun06] Wolfgang Funk. Reversible watermarking of nurbs-based cad models. In Camenisch et al. [CCJS07], pages 172–187.

[GB98] Daniel Gruhl and Walter Bender. Information hiding to foil the casual counterfeiter. In *Information Hiding Workshop, Lecture Notes of Computer Science (1525)*, New York, Heidelberg, 1998. Springer-Verlag.

[GCC88] Marc Girault, Robert Cohen, and Mireille Campana. A generalized birthday attack. In *EUROCRYPT*, pages 129–156, 1988.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York City, New York, 1979.

[GJ04] Philippe Golle and Ari Juels. Dining cryptographers revisited. In *Proceedings of Eurocrypt 2004*, May 2004.

[GLB96] Daniel Gruhl, Anthony Lu, and Walter Bender. Echo hiding. In *Proceedings of the First International Workshop on Information Hiding*, pages 293–315, London, UK, 1996. Springer-Verlag.

[GRJK00] Rosario Gennaro, Tal Rabin, Stanislav Jarecki, and Hugo Krawczyk. Robust and efficient sharing of RSA functions. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 13(2):273–300, 2000.

[Gro95] Wendy Grossman. alt.scientology.war. *Wired*, 3(2), December 1995.

[Gun88] C.G. Gunther. A universal algorithm for homophonic coding. In *Advances in Cryptology–Eurocrypt '88 Lecture Notes in Computer Science*, number 330, pages 405–414, New York, 1988. Springer-Verlag.

[HC08] Nicholas Zhong-Yang Ho and Ee-Chien Chang. Residual information of redacted images. hidden in the compression artifacts. In *Information Hiding 2008, Lecture Notes of Computer Science*, New York, Heidelberg, 2008. Springer-Verlag.

[Hec82] Paul Heckbert. Color image quantization for frame buffer display. In *Proceedings of SIGGRAPH 82*, 1982.

[HG97] Frank Hartung and Bernd Girod. Fast public-key watermarking of compressed video. In *International Conference on Image Processing (ICIP'97)*, volume I, pages 528–531, Santa Barbara, California, U.S.A., 1997.

[Hil91a] D Hillman. The structural of reversible one-dimensional cellular automata. *Physica D*, 52(2 / 3):277, sep 1991.

[Hil91b] David Hillman. The structure of reversible one-dimensional cellular automata. *Physica D*, 54:277–292, 1991.

[Hir01] Martin Hirt. *Multi-Party Computation: Efficient Protocols, General Adversaries, and Voting*. PhD thesis, ETH Zurich, September 2001. Reprint as vol. 3 of *ETH Series in Information Security and Cryptography*, ISBN 3-89649-747-2, Hartung-Gorre Verlag, Konstanz, 2001.

[HS00] Martin Hirt and Kazue Sako. Efficient receipt-free voting based on homomorphic encryption. In *Eurocrypt 2000*, pages 539–556. Springer-Verlag, 2000.

[HSG99] Frank Hartung, Jonathan K. Su, and Bernd Girod. Spread spectrum watermarking: Malicious attacks and counterattacks. In *Security and Watermarking of Multimedia Contents, Proc. SPIE 3657, Jan. 1999*, pages 147–158, Jan 1999.

[HU79] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Huf59] D.A. Huffman. Canonical forms for information-lossless finite-state machines. *IRE Transactions on Circuit Theory (special supplement) and IRE Transactions on Information Theory (special supplement)*, CT-6 and IT-5:41–59, May 1959. A slightly revised version appeared in E.F. Moore, Editor, *Sequential Machines: Selected Papers*, Addison-Wesley, Reading, Massachusetts, 1964.

[JDJ01] Neil F. Johnson, Zoran Duric, and Sushil Jajodia. *Information Hiding : Steganography and Watermarking - Attacks and Countermeasures (Advances in Information Security, Volume 1).* Kluwer Academic Publishers, February 2001.

[Jea06] Sebastien Jeanquier. *An Analysis of Port Knocking and Single Packet Authorization.* PhD thesis, Royal Holloway College, University of London, September 2006.

[JJ98a] Neil F. Johnson and Sushil Jajodia. Steganalysis of images created using current steganography software. In *Information Hiding, Second International Workshop*, pages 273–289, 1998.

[JJ98b] Neil F. Johnson and Sushil Jajodia. Steganalysis: The investigation of hidden information, 1998.

[JKM90] H.N. Jendal, Y. J. B. Kuhn, and J. L. Massey. An information-theoretic treatment of homophonic substitution. In *Advances in Cryptology–Eurocrypt '89*, New York, 1990. Springer-Verlag, Lecture Notes in Computer Science.

[JKTS07] Peter C. Johnson, Apu Kapadia, Patrick P. Tsang, and Sean W. Smith. Nymble: Anonymous ip-address blocking. In Borisov and Golle [BG07], pages 113–133.

[KA98] Markus G. Kuhn and Ross J. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In *Information Hiding Workshop, Lecture Notes of Computer Science (1525)*, New York, Heidelberg, 1998. Springer-Verlag.

[Kah67] David Kahn. *The Codebreakers*. Macmillan, New York City, 1967.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[KCR04] Ton Kalker, Ingemar J. Cox, and Yong Man Ro, editors. *Digital Watermarking, Second International Workshop, IWDW 2003, Seoul, Korea, October 20-22, 2003, Revised Papers*, volume 2939 of *Lecture Notes in Computer Science*. Springer, 2004.

[Kea89] Michael Kearns. *The Computational Complexity of Machine Learning*. PhD thesis, Harvard University Center for Research in Computing Technology, May 1989.

[KH98] Deepa Kundur and Dimitrios Hatzinakos. Digital watermarking using multiresolution wavelet decomposition. In *International Conference on Acoustic, Speech and Signal Processing (ICASP)*, volume 5, pages 2969–2972, Seattle, Washington, U.S.A., 1998.

[KH99] Deepa Kundur and D. Hatzinakos. Digital watermarking for telltale tamper-proofing and authentication. *Proceedings of the IEEE Special Issue on Identification and Protection of Multimedia Information*, 87(7):1167–1180, July 1999.

[Kip03] Gregory Kipper. *Investigator's Guide to Steganography*. CRC Press, Inc., Boca Raton, FL, USA, 2003.

[Knu81] D. Knuth. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. 2nd edition, Addison-Wesley, Reading, MA, 1981.

[KO84] Hugh Kenner and Joseph O'Rourke. A travesty generator for micros. *BYTE*, page 129, November 1984.

[Kob87] Neal Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag, New York, Berlin, Heidelberg, London, Paris, Tokyo, 1987.

[Kom95] John Kominek. Convergence of fractal encoded images. In J. A. Storer and M. Cohn, editors, *Data Compression Conference 1995*, pages 242–251, Snowbird, UT, USA, March 1995.

[KQP01] Farinaz Koushanfar, Gang Qu, and Miodrag Potkonjak. Intellectual property metering. In *Fourth Information Hiding Workshop*, 2001.

[Kra94] Hugo Krawczyk. Secret sharing made short. In *CRYPTO '93: Proceedings of the 13th annual international cryptology conference on Advances in cryptology*, pages 136–146, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[Krz03a] Martin Krzywinski. Port knocking. *Linux Journal*, June 2003.

[Krz03b] Martin Krzywinski. Port knocking: Network authentication across closed ports. *SysAdmin Magazine*, 12:12–17, 2003.

[KSM06] Mehdi Kharrazi, Husrev T. Sencar, and Nasir Memon. Performance study of common image steganography and steganalysis techniques. *Journal of Electronic Imaging*, 15(4), 2006.

[KV89] Michael Kearns and Leslie Valient. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 433–444, Seattle, WA, May 1989.

[Lag84] J. C. Lagarias. Performance analysis of shamir's attack on the basic merkle-hellman knapsack cryptosystem. In *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, pages 312–323, London, UK, 1984. Springer-Verlag.

[LC04] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[Leh82] D.J. Lehmann. On primality tests. *SIAM Journal on Computing*, 11(2), May 1982. ((page numbers are wrong)).

[LF05] S. Lyu and H. Farid. How realistic is photorealistic? *IEEE Transactions on Signal Processing*, 53(2):845–850, 2005.

[Lia95] Wilson MacGyver Liaw. Reading gif files. *Dr. Dobbs Journal*, Feb 1995.

[LJ83] Shu Lin and Daniel J. Costello Jr. *Error Control Coding: Fundaments and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1983.

[LJ00] Vinicius Licks and R. Jordan. On digital image watermarking robust to geometric transformations. In *IEEE International Conference on Image Processing 2000, Vancouver, Canada*, 2000.

[LMBO95] Steve Low, Nicholas Maxemchuk, Jack Brassil, and Larry O'Gorman. Document marking and identification using both line and word shifting. In *Proceedings of the 1995 Conference on Infocom '95*, April 1995.

[LMSP98] John Lach, WIlliam H. Mangione-Smith, and Miodrag Potkonjak. Fingerprinting digital circuits on programmable hardware. In *Information Hiding Workshop, Lecture Notes of Computer Science (1525)*, New York, Heidelberg, 1998. Springer-Verlag.

[Lub02] Michael Luby. Lt codes. In *FOCS*, pages 271–. IEEE Computer Society, 2002.

[Lyu05] S. Lyu. *Natural Image Statistics for Digital Image Forensics*. PhD thesis, Department of Computer Science, Dartmouth College, Hanover, NH, 2005.

[Mae98] Maurise Maes. Twin peaks: The histogram attack to fixed depth image watermarks. In *Information Hiding Workshop, Lecture Notes of Computer Science (1525)*, New York, Heidelberg, 1998. Springer-Verlag.

[Mah95] Kevin Maher. Texto. circulating on the web, February 1995. `http://www.ecn.org/crypto/soft/texto.zip`, accessed 2005-03-22.

[Man03] S. Manoharan. Using self-synchronizing codes to improve steganography. In *Internet and Multimedia Systems and Applications IMSA 2003* , Honolulu, USA, 8 2003.

[Mar84] N. Margolus. Physics-like models of computation. *Physica D*, 10:81–95, 1984.
Discussion of reversible cellular automata illustrated by an implementation of Fredkin's Billiard-Ball model of computation.

[MBR99] L. Marvel, C. Boncelet, and C. Retter. Spread spectrum image steganography. *IEEE Transactions on Image Processing 8*, pages 1075–1083, August 1999.

[McH01] John McHugh. Cover image. In *Fourth Information Hiding Workshop*, page 0, 2001.

[MCLK99] M. Miller, I. Cox, J. Linnartz, and T. Kalker. A review of watermarking principles and practices, 1999.

[Mer93] Ralph Merkle. Reversible electronic logic using switches. *Nanotechnology*, 4:21–40, 1993.

[MF05] Daniel B. Miller and Edward Fredkin. Two-state, reversible, universal cellular automata in three dimensions. In Bagherzadeh et al. [BVR05], pages 45–51.

[ML05] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into tcp/ip. In Mauro Barni, Jordi Herrera-Joancomartí, Stefan Katzenbeisser, and Fernando Pérez-González, editors, *Information Hiding*, volume 3727 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2005.

[MM92] IS Moskowitz and AR Miller. The channel capacity of a certain noisy timing channel. *IEEE Trans. on Information Theory*, IT-38(4):1339 – 43, 1992.

[Mon85] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170), 1985.

[MS08] Wojciech Mazurczyk and Krzysztof Szczypiorski. Steganography of voip streams, 2008.

[MTMM07] Robert P. McEvoy, Michael Tunstall, Colin C. Murphy, and William P. Marnane. Differential power analysis

of hmac based on sha-2, and countermeasures. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *WISA*, volume 4867 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2007.

[Mur06] Steven J. Murdoch. Hot or not: revealing hidden services by their clock skew. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 27–36, New York, NY, USA, 2006. ACM.

[MvV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, New York, 1997.

[Nas49] Ogden Nash. Lineup for yesterday. *Sport Magazine*, January 1949.

[NCS93] NCSC. A guide to understanding covert channel analysis of trusted systems. Technical Report TG-030, NCSC, November 1993.

[Neu64] P.G. Neumann. Error-limiting coding using information-lossless sequential machines. *IEEE Transactions on Information Theory*, IT-10:108–115, April 1964.

[NN06] Mohannad Najjar and Firas Najjar. d-hmac dynamic hmac function. In *DepCoS-RELCOMEX*, pages 119–126. IEEE Computer Society, 2006.

[NW06] Arjun Nambiar and Matthew Wright. Salsa: a structured approach to large-scale anonymity. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 17–26, New York, NY, USA, 2006. ACM.

[NY89] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the 21nd Annual ACM Symposium on Theory of Computing*, pages 33–43. ACM, 1989.

[NY90] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 427–437. ACM, 1990.

[Odl84] A. M. Odlyzko. Cryptanalytic attacks on the multiplicative knapsack scheme and on Shamir's fast signature scheme. *IEEE Trans. Inform. Theory*, IT-30:594–601, 1984.

[ØS07a] Lasse Øverlier and Paul Syverson. Improving efficiency and simplicity of tor circuit establishment and hidden services. In Borisov and Golle [BG07], pages 134–152.

[ØS07b] Lasse Øverlier and Paul Syverson. Location hidden services and valet nodes. *Telektronikk*, pages 52–60, February 2007.

[PBBC97] A. Piva, M. Barni, F. Bartolini, and V. Cappellini. DCT-based watermark recovering without resorting to the uncorrupted original image. In *IEEE Signal Processing Society 1997 International Conference on Image Processing (ICIP'97)*, Santa Barbara, California, October 1997.

[PF05] Alin C. Popescu and Hany Farid. Exposing digital forgeries in color filter array interpolated images. *IEEE Transactions on Signal Processing*, 53(10):3948–3959, 2005.

[PK03] Fabien A. P. Petitcolas and Hyoung Joong Kim, editors. *Digital Watermarking, First International Workshop, IWDW 2002, Seoul, Korea, November 21-22, 2002, Revised Papers*, volume 2613 of *Lecture Notes in Computer Science*. Springer, 2003.

[PN93] N Proctor and P Neumann. Architectural implications of covert channels. In *Proceedings of the 15th National Computer Security Conference*, 1993.

[Poe44] Edgar Allan Poe. The purloined letter. *The Gift: A Christmas and New Year's Present for 1844*, 1844.

[Pop05] A.C. Popescu. *Statistical Tools for Digital Image Forensics*. PhD thesis, Department of Computer Science, Dartmouth College, Hanover, NH, 2005.

[PP90] B Pfitzmann and A Pfitzmann. How to break the direct rsa-implementation of mixes. In *Advances in Cryptology– Eurocrypt '89*, number 434. Springer-Verlag, 1990.

[Pro] Federal Information Processing. Fips pub 198: The keyed-hash message authentication code (hmac).

[Pro01a] Niels Provos. Defending against statistical steganalysis, August 2001.

[Pro01b] Niels Provos. Probabilistic methods for improving information hiding. Technical Report 01-1, University of Michigan, January 2001.

[PSM06] Dima Pröfrock, Mathias Schlauweg, and Erika Müller. Video watermarking by using geometric warping without visible artifacts. In Camenisch et al. [CCJS07], pages 78–92.

[Pub88] Publius. *Federalist: A Collection of Essays Written in Favor of the New Constitution.* J. and A. McClean, 1788.

[QC82] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for rsa public-key cryptosystem. *Electronic Letters*, 18, 1982.

[Qu01] Gang Qu. Keyless public watermarking for intellectual property authentication. In *Fourth Information Hiding Workshop*, 2001.

[Rab89a] Michael Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 38:335–348, 1989.

[Rab89b] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.

[RC95] R. Rinaldo and G. Calvagno. Image coding by block prediction of multiresolution submimages. *IEEE Transactions on Image Processing*, 4:909–920, 1995.

[RDB96] J. Ruanaidh, W. Dowling, and F. Boland. Phase watermarking of digital images. In *Proceedings of ICIP'96*, volume III, pages 239–242, Lausanne, Switzerland, September 1996.

[Riv98] Ron Rivest. Chaffing and winnowing: Confidentiality without encryption, summer 1998.

[Riv04] Ronald L. Rivest. Peppercoin micropayments. In Ari Juels, editor, *Financial Cryptography*, volume 3110 of *Lecture Notes in Computer Science*, pages 2–8. Springer, 2004.

[Rob62] L. G. Roberts. Picture coding using pseudo-random noise. *IRE Trans. on Information Theory*, IT-8, Feb 1962.

[RP04] M. Rennhard and B. Plattner. Practical anonymity for the masses with morphmix, 2004.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[RR08] Christian Rechberger and Vincent Rijmen. New results on nmac/hmac when instantiated with popular hash functions. *Journal of Universal Computer Science*, 14(3):347–376, feb 2008.

[RSG98] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

[Rut06] Joanna Rutkowska. Subverting vista kernel for fun and profit. In *Black Hat Briefings*, 2006.

[SA00] Frank Stajano and Ross J. Anderson. The cocaine auction protocol: On the power of anonymous broadcast. In *IH '99: Proceedings of the Third International Workshop on Information Hiding*, pages 434–447, London, UK, 2000. Springer-Verlag.

[Say00] Khalid Sayood. *Introduction to data compression (2nd ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[Sch94] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, New York, 1994.

[Sei08] Jacob Seidelin. Compression using canvas and png-embedded data. *nihilogic*, April 2008.

[Ser07] Andrei Serjantov. A fresh look at the generalised mix framework. In Borisov and Golle [BG07], pages 17–29.

[SGR97] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. Anonymous connections and onion routing. In IEEE, editor, *Proceedings / 1997 IEEE Symposium on Security and Privacy, May 4–7, 1997, Oakland, California*, pages 44–54, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. IEEE Computer Society Press.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 24(11), Nov 1979.

[Sha82] Adi Shamir. A polynomial time algorithm for breaking the basic merkle-hellman cryptosystem. In *CRYPTO*, pages 279–288, 1982.

[Sha93] J. Shapiro. Embedded image coding using zerotrees of wavelet coeffcients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, 1993.

[Sha01] Toby Sharp. An implementation of key-based digital signal steganography. In *Fourth Information Hiding Workshop*, 2001.

[Sha08] Adi Shamir. How to solve it: New techniques in algebraic cryptanalysis. In *Crypto 2008*, Santa Barbara, California, August 2008.

[Shi99] Natori Shin. One-time hash steganography. In *3rd Information Hiding Workshop, Lecture Notes of Computer Science (1768)*, New York, Heidelberg, 1999. Springer-Verlag.

[Sim84] G.J. Simmons. The prisoner's problem and the subliminal channel. In *Advances in Cryptology: Proceedings of CRYPTO '83*. Plenum Press, 1984.

[Sim85] G.J. Simmons. The subliminal channel and digital signatures. In *Advances in Cryptology: Proceedings of EUROCRYPT 84*. Springer-Verlag, 1985.

[Sim86] G.J. Simmons. A secure subliminal channel (?). In *Advances in Cryptology–CRYPTO '85 Proceedings*. Springer-Verlag, 1986.

[Sim93] G.J. Simmons. The subliminal channels of the U.S. Digital Signature Algorithm (DSA). In *Proceedings of the Third Symposium on: State and Progress of Research in Cryptography*, Fondazone Ugo Bordoni, Rome, 1993.

[Sim94] G.J. Simmons. Subliminal communication is easy using the DSA. In *Advances in Cryptology–EUROCRYPT '93 Proceedings*. Springer-Verlag, 1994.

[SJ06] Yun-Qing Shi and Byeungwoo Jeon, editors. *Digital Watermarking, 5th International Workshop, IWDW*

2006, Jeju Island, Korea, November 8-10, 2006, Proceedings, volume 4283 of *Lecture Notes in Computer Science*. Springer, 2006.

[SK94] Kazue Sako and Joe Kilian. Secure voting using partially compatible homomorphisms. In *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 411–424, London, UK, 1994. Springer-Verlag.

[SK95] K Sako and J Kilian. Receipt-free mix-type voting schemes. In *Advances in Cryptology– Eurocrypt '95*, pages 393–403. Springer-Verlag, 1995.

[SKE00] A.P. Petitcolas (Editor) Stefan Katzenbeisser (Editor), Fabien. *Information Hiding Techniques for Steganography and Digital Watermarking*. Artech House, January 2000.

[Spi05] Matthew D. Spisak. *An analysis of perturbed quantization steganography in the spatial domain*. PhD thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, April 2005.

[SRG00] Paul F. Syverson, Michael G. Reed, and David M. Goldschlag. Onion routing access configurations. In *DISCEX 2000: Proceedings of the DARPA Information Survivability Conference and Exposition*, volume I, pages 34–40, Hilton Head, SC, January 2000. IEEE CS Press.

[SSG97] Paul F. Syverson, Stuart G. Stubblebine, and David M. Goldschlag. Unlinkable serial transactions. In *Financial Cryptography*, pages 39–56, 1997.

[Sto88] James Storer. *Data Compression*. Computer Science Press, Rockville, MD, 1988.

[STRL00] Paul F. Syverson, Gene Tsudik, Michael G. Reed, and Carl E. Landwehr. Towards an analysis of onion routing security. In *Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.

[SW63] Claude E. Shannon and Warren Weaver. *A Mathematical Theory of Communication*. University of Illinois Press, Champaign, IL, USA, 1963.

[SY98] Sabrina Sowers and Abdou Youssef. Testing digital watermark resistance to destruction. In *Information Hiding Workshop, Lecture Notes of Computer Science (1525)*, New York, Heidelberg, 1998. Springer-Verlag.

[TA05] Eugene Tumoian and Maxim Anikeev. Network based detection of passive covert channels in tcp/ip. In *LCN '05: Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary*, pages 802–809, Washington, DC, USA, 2005. IEEE Computer Society.

[Tah92] H. Taha. *Operations Research An Introduction*. Macmillan Publishing Company., New York, 1992.

[TB06] Parisa Tabriz and Nikita Borisov. Breaking the collusion detection mechanism of morphmix. In George Danezis and Philippe Golle, editors, *Privacy Enhancing Technologies*, volume 4258 of *Lecture Notes in Computer Science*, pages 368–383. Springer, 2006.

[TL05] Tommaso Toffoli and Lev B. Levitin. Specific ergodicity: an informative indicator for invertible computational media. In Bagherzadeh et al. [BVR05], pages 52–58.

[TM87] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines*. MIT Press, London, 1987.

[Tof77a] T. Toffoli. *Cellular Automata Mechanics*. PhD thesis, The University of Michigan, 1977.
Toffoli's demonstration of reversible universal computation.

[Tof77b] Tommaso Toffoli. Computation and construction universality of reversible cellular automata. *Journal of Computer and System Sciences*, 15:213–231, 1977.

[TRHTA06] Mercan Topkara, Guiseppe Riccardi, Dilek Hakkani-Tur, and Mikhail J. Atallah. Natural language watermarking: Challenges in building a practical system. In *Proceedings of the SPIE International Conference on Security, Steganography, and Watermarking of Multimedia Contents*, January 2006.

[TTA06] Umut Topkara, Mercan Topkara, and Mikhail J. Atallah. The hiding virtues of ambiguity: quantifiably resilient watermarking of natural language text through synonym substitutions. In *MM&Sec '06: Proceedings*

of the 8th workshop on Multimedia and security, pages 164–174, New York, NY, USA, 2006. ACM.

[TTTD06] Cuneyt M. Taskiran, Umut Topkara, Mercan Topkara, and Edward J. Delp. Attacks on lexical natural language steganography systems. In *Proceedings of the SPIE International Conference on Security, Steganography, and Watermarking of Multimedia Contents*, January 2006.

[Tur36a] Alan Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Math Soceity*, 2(42):230–265, 1936.

[Tur36b] Alan Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Math Soceity*, 2(43):544–546, 1936.

[vAH04] Luis von Ahn and Nicholas J. Hopper. Public-key steganography. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 323–341. Springer, 2004.

[Val84] Leslie G. Valient. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[VR05] Alexis De Vos and Yvan Van Rentergem. Reversible computing: from mathematical group theory to electronical circuit experiment. In Bagherzadeh et al. [BVR05], pages 35–44.

[Wal94] John Walker. Steganosaurus. circulating on the web, December 1994. `http://www.fourmilab.ch/stego/stego.shar.gz`, accessed 2005-03-25.

[Wal95a] S. Walton. Information authentication for a slippery new age. *Dr. Dobbs Journal*, 20(4):18–26, April 1995.

[Wal95b] Steve Walton. Image authentification for a slippery new age. *Dr. Dobbs Journal*, Apr 1995.

[Way85] Peter Wayner. Building a travesty tree. *BYTE*, page 183, September 1985.

[Way92] Peter C. Wayner. Content-addressable search engines and DES-like systems. In *Advances in Cryptology: CRYPTO '92 Lecture Notes in Computer Science, volume 740*, pages 575–586, New York, 1992. Springer-Verlag.

[Way95a] Peter Wayner. Strong theoretical steganography. *Cryptologia*, 19(3):285–299, July 1995.

[Way95b] Peter C. Wayner. *Digital Cash: Commerce on the Net*. AP Professional, Boston, 1995.

[Way97a] Peter Wayner. *Digital Cash, 2nd Edition*. AP Professional, Chestnut Hill, MA, 1997.

[Way97b] Peter Wayner. *Digital Copyright Protection*. AP Professional, Chestnut Hill, MA, 1997.

[Way99] Peter Wayner. *Data Compression for Real Programmers*. AP Professional, Chesnutt, Hill, MA, 1999.

[Way00] Peter Wayner. *Compression algorithms for real programmers*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[Way01] Peter Wayner. *Translucent Databases*. Flyzone Press, Baltimore, MD, USA, 2001.

[Way03] Peter Wayner. *Translucent Databases*. Flyzone Press, Baltimore, Maryland, 2003.

[Way05] Peter Wayner. *Policing Online Games*. Flyzone Press, Baltimore, Maryland, 2005.

[Wei76] Joseph Weizenbaum. *Computer power and human reason : from judgment to calculation*. W.H. Freeman, San Fransisco, 1976.

[Wes01] Andreas Westfeld. High capacity depsite better steganalysis: F5- a steganographic algorithm. In *Fourth Information Hiding Workshop*, pages 301–315, 2001.

[WH94] Peter Wayner and Dan Huttenlocher. Image analysis to obtain typeface information. *U.S. Patent*, 5253307, 1994.

[WmC06] Hao-Tian Wu and Yiu ming Cheung. A high-capacity data hiding method for polygonal meshes. In Camenisch et al. [CCJS07], pages 188–200.

[Won98] P.W. Wong. A public key watermark for image verification and authentication. In *In Proc. of ICIP'98*, volume 1, pages 425– 429, Chicago, USA, October 1998.

[WP99] Andreas Westfeld and Andreas Pfitzmann. Attacks on steganographic systems. In *Information Hiding, Third International Workshop, IH'99*, volume 1768, pages 61–76, Dresden, Germany, 1999. Springer Verlag.

[WRC00] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[WS99] Wright and Spalding. Experimental performance of shared RSA modulus generation (short). In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[XBA97] X.-G. Xia, C. Boncelet, and G. Arce. A multiresolution watermark for digital images. In *IEEE Signal Processing Society 1997 International Conference on Image Processing (ICIP'97)*, Santa Barbara, California, October 1997.

Index