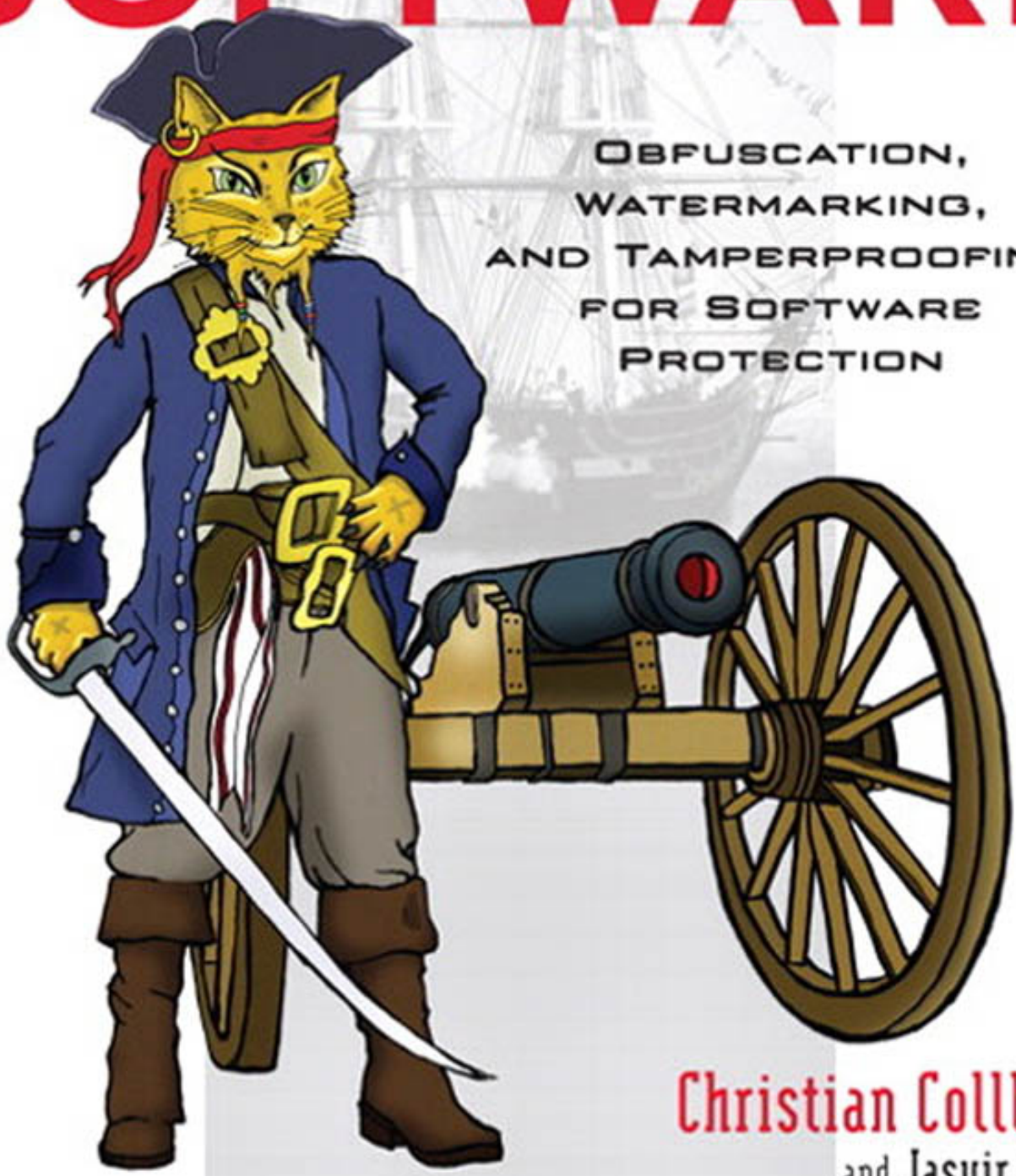
SURREPTITIOUS SOFTWARE

Obfuscation, Watermarking, and Tamperproofing for Software Protection



Christian Collberg
and Jasvir Nagra

Library of Congress Cataloging-in-Publication Data

Collberg, Christian.
Surreptitious software : obfuscation, watermarking, and tamperproofing for software protection / Christian Collberg, Jasvir Nagra. – 1st ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-54925-2 (pbk. : alk. paper)
1. Computer security. 2. Cryptography. 3. Data protection. 4. Copyright and electronic data processing–United States. I. Nagra, Jasvir. II. Title.
QA76.9.A25C6165 2009
005.8–dc22 2009015520

ISBN-13: 978-0-321-54925-9
ISBN-10: 0-321-54925-2

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, July 2009

Preface

Surreptitious software is the term we have chosen to describe a new branch of computer security research that has emerged over the last decade. It's a field that borrows techniques not only from computer security, but also from many other areas of computer science, such as cryptography, steganography, media watermarking, software metrics, reverse engineering, and compiler optimization. Surreptitious software applies these techniques in order to solve very different problems: It is concerned with protecting the secrets contained within computer programs. We use the word *secrets* loosely, but the techniques we present in this book (code obfuscation, software watermarking and fingerprinting, tamperproofing, and birthmarking) are typically used to prevent others from exploiting the intellectual effort invested in producing a piece of software. For example, software fingerprinting can be used to trace software pirates, code obfuscation can be used to make it more difficult to reverse engineer a program, and tamperproofing can make it harder for a hacker to remove a license check.

So let's look at *why* someone should read this book, *who* they might be, and *what* material the book will cover.

Why Should You Read This Book?

Unlike traditional security research, surreptitious software is not concerned with how to protect your computer from viruses, but rather how virus writers protect their code from you! Similarly, we're not interested in how to make your code free from security bugs, but rather how to riddle your program with buggy code that gets run only when someone tries to tamper with the program. And unlike cryptography research that protects the confidentiality of data, assuming that a secret key remains hidden, we're interested in how to hide that key. While software engineering research has devised a multitude of software metrics in order to be able to make programs well structured, we will use the same techniques to make your programs more

convoluted! Many of the techniques that we will describe in this book are based on algorithms developed by compiler optimization researchers, but unlike them, we're not interested in making your program faster or smaller. Rather, after you apply the algorithms in this book to your own code, your programs will be *larger* and *slower*! Finally, unlike traditional media watermarking and steganography that hides secrets in images, audio, video, or English text, surreptitious software hides secrets inside *computer code*.

So why, then, should you be interested in this book? Why learn about a branch of computer security research that doesn't teach how to protect yourself against viruses and worms? Why study optimizing compiler transformations that make your code slower and larger? Why bother with a branch of cryptography that breaks the most basic assumption of that field, namely, that the secret key must be kept hidden?

The answer is that there are real-world problems that don't fit neatly into traditional computer security and cryptography research but that are interesting nonetheless. For example, in this book we will show you how to use *software watermarking* to fight piracy. A software watermark is a unique identifier (such as someone's credit card number or your copyright notice) that you embed into your program to uniquely bind it to you (the author) or your customer. If you find illegal copies of your program being sold at a market in Singapore, you can extract the watermark and trace the illegal copy back to the original purchaser. You can use the same technique when you distribute beta copies of your new computer game to your partner companies—should any of them leak the code, you can trace the culprit and sue for damages.

Or if the new version of your program contains a novel algorithm that you don't want your competitors to get their hands on and incorporate into their competing product, you can *obfuscate* your program and make it as convoluted and hard to understand as possible in order to slow down your competitors' reverse engineering efforts. If you do suspect someone has stolen your code, we'll show you how to use *software birthmarking* techniques to identify the offending sections.

Or, say that you have included a secret in your program and you want to make sure that without this secret remaining intact, no one can execute the program. For example, you don't want a hacker to be able to modify your license checking code or the cryptographic key that unlocks an mp3 file in your digital rights management system. Our chapter on *tamperproofing* will discuss various techniques for ensuring that programs that have been tampered with will cease to function properly.

Now, we hear you object that including a crypto key in an executable is a *Really Bad Idea*! Surely experience has shown us that *security by obscurity* never works,

and whatever secret we try to hide in our program will eventually be discovered by a sufficiently determined hacker. And yes—we have to concede—you're right. None of the techniques we're advocating in this book are foolproof. They won't hide a secret forever. They won't make your program safe from tampering forever. They won't identify software theft all of the time. All we can hope for, barring major advances in the field, is that we can slow down our adversaries. Our goal is to slow them down *enough* so that either they give up on cracking our code because they figure it's too painful and not worth the trouble, or so that by the time they've come up with a successful attack we've already made enough of a profit or have moved on to the next version of the code.

For example, say that you're running a pay-TV channel that your customers access through a set-top box. Each box is personalized—hidden somewhere in the code is their unique identifier—so that you can grant and revoke viewing privileges depending on whether they've paid their bill or not. Now, an enterprising hacker extracts and disassembles the code, finds their unique identifier, and starts selling it over the Web at a fraction of the price you charge, along with instructions on how to implant it in the box. How do you counter this attack? Well, you may use tamperproof smartcards, which are not that hard to tamper with, as you will see in our chapter on hardware protection techniques. Or maybe you'll obfuscate your code to make it harder to analyze. Or maybe you'll use software tamperproofing to stop the code from executing once it is mucked with. More likely, you'll use a combination of all these techniques in order to protect your code. Even when employing all these techniques, you must know, and accept, that your code will eventually be broken and your secret will be out of the bag (or box, in this case). So why should you bother? If security through obscurity is a fundamentally flawed idea, and if none of the techniques you will read about in this book will give you "perfect and long-term security," why go through the trouble? *Why should you buy this book?* You will bother for the very simple reason that the longer you can keep your code safe, the more subscriptions you'll sell and the longer between set-top box upgrades, and hence, the more money you'll make.

It's as simple as that.

Who Uses Surreptitious Software?

Many well-known companies have shown an interest in surreptitious software. It's difficult to get a grip on the extent to which these techniques are actually being used in practice (most companies are notoriously tight-lipped about how they protect their code), but we can gauge their level of interest from their patents and current

patent applications. Microsoft owns several software watermarking [104,354], obfuscation [62,62,69,69,70,70,180,378], and birthmarking [364] patents. Intertrust holds a huge patent portfolio related to digital rights management, including patents on obfuscation [91,169] and tamperproofing [168]. In 2004, to the tune of $440 million [176], Microsoft settled a long-running lawsuit with Intertrust by licensing their entire patent portfolio. That same year Microsoft also partnered [250] with PreEmptive Solutions in order to include PreEmptive's identifier obfuscator (which is based on their patent [351]) in Visual Studio. Arxan, a spin-off from Purdue University researchers, has made a successful business from their tamperproofing algorithm [24,305]. Apple holds a patent on code obfuscation [197], perhaps intended to protect their iTunes software. Intel spun off a company, Convera, to explore their tamperproofing algorithm [27,268–270] for digital rights management. The Canadian telecom Northern Telecom spun off what to date has been the most successful company in this area, Cloakware, which holds patents [67,68,182] on what they call *whitebox cryptography*, how to hide cryptographic algorithms and keys in computer code. In December 2007 Cloakware was sold for $72.5 million to Irdeto, a Dutch company in the pay-TV business [162]. A relative latecomer, Sun Microsystems, has also recently filed several patent applications on code obfuscation [105–110].

Skype's VoIP client is highly obfuscated and tamperproofed by techniques similar to those of Arxan [24], Intel [27], and ourselves [89]. Protecting the integrity of their client is undoubtedly of the highest importance for Skype, since, if cracked, their protocol could be easily hijacked by cheaper competitors. Keeping their protocol secret allowed them to build an impressive user base, and this might have been one of the reasons eBay decided to acquire them for $2.6 billion in 2005. In essence, the protection afforded Skype by surreptitious software techniques bought them enough time to become VoIP market leaders. Even if, at this point, their protocol has been cracked (which it has; see Section 7.2.4), it will be difficult for a competitor to threaten this position.

Academic researchers have approached surreptitious software from a variety of angles. Some, like us, come from a compiler and programming languages background. This is natural since most algorithms involve code transformations that require static analysis, with which compiler optimization researchers are intimately familiar. In spite of the disdain cryptography researchers have in the past had for *security through obscurity*, some have recently applied their techniques to software watermarking and to discovering the limits of obfuscation. Researchers from media watermarking, computer security, and software engineering have also published in surreptitious software. Unfortunately, progress in the area has been hampered

by the lack of natural publication venues. Instead, researchers have struggled, and continue to struggle, to get their works accepted in traditional conferences and journals. Papers have appeared in venues such as the ACM Symposium on Principles of Programming Languages (POPL), the Information Hiding Workshop, IEEE Transactions on Software Engineering, Advances in Cryptology (CRYPTO), and Information Security Conference (ISC), as well as in various digital rights management conferences. As the field becomes more mainstream, we can expect journals, workshops, and conferences dedicated exclusively to surreptitious software, but this has yet to happen.

The military has also spent much effort (and taxpayer money) on surreptitious software research. For example, the patent [96] held on Cousot's software watermarking algorithm [95] is assigned to the French *Thales group*, the ninth-largest defense contractor in the world. Here's a quote from a recent (2006) U.S. Army solicitation [303] for what they term AT, anti-tamper research:

> All U.S. Army Project Executive Offices (PEOs) and Project Managers (PMs) are now charged with executing Army and Department of Defense (DoD) AT policies in the design and implementation of their systems. Embedded software is at the core of modern weapon systems and is one of the most critical technologies to be protected. AT provides protection of U.S. technologies against exploitation via reverse engineering. Standard compiled code with no AT is easy to reverse engineer, so the goal of employed AT techniques will be to make that effort more difficult. In attacking software, reverse engineers have a wide array of tools available to them, including debuggers, decompilers, disassemblers, as well as static and dynamic analysis techniques. AT techniques are being developed to combat the loss of the U.S. technological advantage, but further advances are necessary to provide useful, effective and varied toolsets to U.S. Army PEOs and PMs. . . . The goal of software AT technologies/techniques developed is to provide a substantial layer of protection against reverse engineering, allowing for maximum delay in an adversary compromising the protected code. This capability will allow the U.S. time to advance its own technology or otherwise mitigate any losses of weapons technologies. As a result, the U.S. Army can continue to maintain a technological edge in support of its warfighters.

This particular solicitation comes from the Army's Missile and Space program and focuses on providing protection in real-time embedded systems. So it's reasonable

to assume that the Army is worried about one of their missiles not exploding on impact, allowing the enemy access to the embedded targeting flight software.

The DoD's current interest in protecting their software goes back to the infamous incident in 2001 when a—what CNN [359] calls a US "Navy reconnaissance plane" but the rest of us know as a "spy plane"—had a run-in with a Chinese fighter jet and had to make an emergency landing on the Chinese island of Hainan. In spite of George W. Bush's plea [116] for the "prompt return 'without further damage or tampering' of the crew and plane," the Chinese stalled, presumably so that they could gather as much information as possible from the plane's hardware and software. There is a claim [116] that all the software was erased:

> A former Pentagon intelligence official told United Press International the crew would have "zeroed" out the crypto analytic equipment and other software on landing, essentially wiping their memories clean. Although the Chinese might have access to the hardware, the software that runs it would be almost impossible to penetrate.

Regardless of whether this is accurate[1] or a carefully orchestrated leak to reassure the U.S. public (why was the software just *almost* impossible to penetrate if it was in fact successfully wiped?), the DoD got significantly spooked and initiated a program to investigate technologies such as obfuscation and tamperproofing in order to protect sensitive weapons systems software. As one DoD official put it, "[T]he next time a plane goes down, we don't want to end up on the cover of the *New York Times* again [291]."

Here's another quote from the DoD [115]:

> The Software Protection Initiative (SPI) is an undertaking of the Department of Defense (DoD) to develop and deploy technologies to secure special-purpose computer programs containing information critical to DoD weapon programs. SPI offers a novel approach to protecting high value computer programs. It doesn't secure the computer or the network. Instead it empowers a single computer program to secure itself. This approach promises to significantly improve DoD's Information Assurance posture. SPI protection technology is effective on systems ranging from desktop

1. During a particularly lively dinner in Beijing in January 2008 with Chinese security researchers, we (Collberg) asked about this incident. One person proudly announced, "Yes, some of the information was recovered!" and then, realizing his mistake, immediately clammed up. This was later corroborated by a different researcher at a second dinner: "The Americans didn't have time to destroy everything!"

> computers to supercomputers. It is an integral layer of the defense-in-depth security paradigm. SPI technologies complement, but do no[t] rely upon, network firewalls, or physical security. These SPI products are currently being deployed to selected HPC centers and are in use at over 150 DoD government and contractor sites. Broader deployment will play a significant role in protecting the DoD's and the nation's critical application software.

What does this mean? The DoD is worried not only about dud missiles falling into the wrong hands, but also about software in use at hardened high-performance computer centers. In fact, theft of *any* software developed for the defense and intelligence communities may have negative consequences. Anecdotally, when the software on a fighter jet is to be updated, a service technician simply walks out to the plane with a laptop, hooks it up, and uploads the new code. What would happen if that laptop got lost or liberated by a technician in the service of a foreign government, as seems to happen [375] every so often? The code could be reverse engineered to be used in an adversary's planes, or even modified with a Trojan horse that would make the avionics fail at a particular time. While there is no substitute for good security practices, surreptitious software techniques can be used as a last line of defense. For example, the avionics software could be fingerprinted with the identifiers of those who have had it in their possession. If one day we find copies of the code in the onboard computers of a North Korean plane we've shot down and are now reverse engineering, we could trace it back to the person who last had authorized access to it.

But, we hear you say, why should I be interested in how evil government agencies or companies with monopolistic tendencies are protecting their secrets? If crackers rip them off, maybe they're just getting what they deserve. Well, maybe so, but technical means of protecting against software theft may ultimately benefit the little guy more than big government or big industry. The reason is that legal forms of protection (such as patents, trademarks, and copyrights) are available to you only if you have the financial means to defend them in a court of law. In other words, even if you think you have a pretty good case against Microsoft for ripping off your code, you will still not prevail in court unless you have the financial backing to outlast them through the appeals process.[2] The technical means of protection that we will discuss in this book such as obfuscation and tamperproofing, on the other hand, can be cheap and easy to apply for the little guy as well as the big guy. And, if you find yourself up against Microsoft in a court of law, techniques such as

2. You don't.

watermarking and birthmarking can bolster your case by allowing you to present evidence of theft.

There is one final category of people that we have yet to touch upon that makes extensive use of surreptitious software: bad guys. Virus writers have been spectacularly successful in obfuscating their code to prevent it from being intercepted by virus scanners. It is interesting to note that while the techniques the good guys use (for example, to protect DVDs, games, or cable TV) seem to be regularly broken by the bad guys, the techniques the bad guys use (to build malware) seem much harder for the good guys to protect against.

What's the Goal of This Book?

The goal of surreptitious software research is to invent algorithms that slow down our adversaries as much as possible while adding as little computational overhead as possible. We also need to devise evaluation techniques that allow us to say, "After applying algorithm A to your program, it will take an average hacker T extra time to crack it compared to the original code, while adding O amount of overhead," or, failing that, at least be able to say that "compared to algorithm B, algorithm A produces code that is harder to crack." It's important to emphasize that research into surreptitious software is still in its infancy, and that the algorithms and evaluation techniques that we'll present in this book, while representing the state of the art, are nowhere near perfect.

In this book we attempt to organize and systematize all that is currently known about surreptitious software research. Each chapter covers a particular technique and describes application areas and available algorithms. In Chapter 1 (What Is *Surreptitious Software?*), we give an overview of the area, and in Chapter 2 (Methods of Attack and Defense), we discuss our adversarial model, i.e., what hacker tools and techniques we should try to protect ourselves against and what ideas are available to us as defenders. In Chapter 3 (Program Analysis), we detail the techniques that both attackers and defenders can use to analyze programs. Chapter 4 (Code Obfuscation), Chapter 5 (Obfuscation Theory), and Chapter 6 (Dynamic Obfuscation) give algorithms for code obfuscation. Chapter 7 (Software Tamperproofing) gives tamperproofing algorithms, Chapter 8 (Software Watermarking) and Chapter 9 (Dynamic Watermarking) give watermarking algorithms, and Chapter 10 (Software Similarity Analysis) gives birthmarking algorithms. Chapter 11 (Hardware for Protecting Software) presents hardware-based protection techniques.

If you're a manager interested in learning about the state of the art in surreptitious software research and how it can be applied in your organization, you'll want to

read Chapter 1 and Chapter 2. If you're a researcher with a background in compiler design, you can skip Chapter 3. It's advantageous to read the algorithm chapters in order, since, for example, the watermarking chapter relies on ideas you've learned in the obfuscation chapter. Still, we've tried to make each chapter as self-contained as possible, so skipping around should be possible. If you're an engineer charged with adding protection to your company's product line, you should read Chapter 3 carefully and maybe complement that by reading up on static analysis in a good compiler text. You can then move on to the algorithm chapter relevant to you. If you're a graduate student reading this book for a class, read the entire thing from cover to cover, and don't forget to review for the final!

We hope this book will do two things. First, we want to convince you, dear reader, that code obfuscation, software watermarking, birthmarking, and tamper-proofing are interesting ideas well worth studying, and that they are viable alternatives to protecting the intellectual property you have invested in your software. Second, we want this book to bring together all available information on the subject so that it can serve as a starting point for further research in the field.

Christian and Jasvir

Tucson and Mountain View
Groundhog Day, 2009

P.S. There is a third reason for writing this book. If, while reading this book, you are struck by the cleverness of an idea and, as a result, you become inspired to make your own contributions to the field, well, then, dear reader, our goal with this book has really been met. And when you've come up with your new clever algorithm, please remember to let us know so we can include it in the next edition!

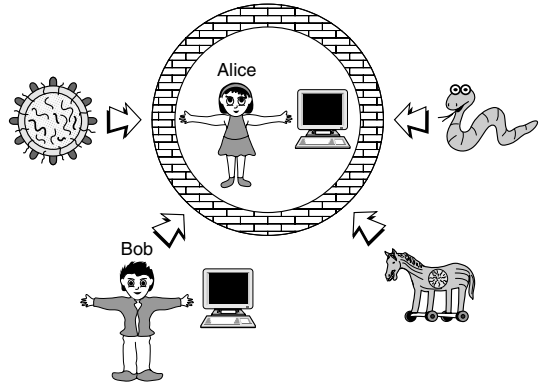
1 What Is *Surreptitious Software*?

In this first chapter we will talk about the basic techniques used to protect secrets stored in software, namely *obfuscation*, *watermarking*, *tamperproofing*, and *birthmarking*. These techniques have many interesting applications, such as the use of obfuscation and tamperproofing to protect media in *digital rights management* systems. What we think you will find particularly interesting is that obfuscation and the three other techniques "solve" problems that traditional computer security and cryptography can't touch. We put "solve" in quotation marks because there are no known algorithms that provide complete security for an indefinite amount of time. At the present time, the best we can hope for is to be able to extend the time it takes a hacker to crack our schemes. You might think that this seems highly unsatisfactory—and you'd be right—but the bottom line is that there are interesting applications for which no better techniques are known.

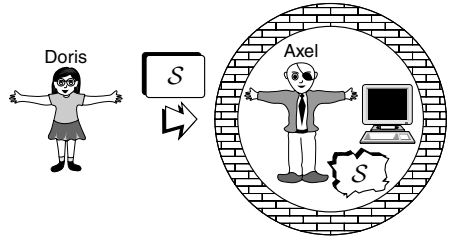
1.1 Setting the Scene

When you hear the term *computer security*, you probably imagine a scenario where a computer (owned by a benign user we'll call Alice) is under attack from an evil hacker (we'll call him Bob), or from the viruses, worms, Trojan horses, rootkits, and keyloggers that he's created. The goal of computer security research is to devise techniques for building systems that prevent Bob from taking over Alice's computer or that alert her when he does. The basic idea behind such techniques is to restrict

what Bob can do on Alice's computer without unduly restricting what she can do herself. For example, a *network firewall* allows Alice to access other computers on the network but restricts the ways in which Bob can access hers. An *intrusion detection system* analyzes the network access patterns on Alice's computer and alerts her if Bob appears to be doing something unusual or suspicious. A *virus scanner* refuses to run Bob's program unless it can convince itself that the program contains no harmful code. In other words, Alice adds protective layers around her computer to prevent someone from entering, to detect that someone has entered, or to stop someone from doing harm once they've entered:



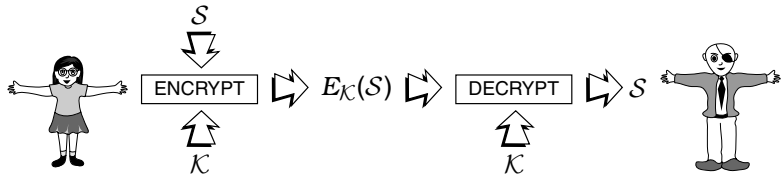
Now what happens if we invert the situation? What if, instead of Bob sending an evil program to penetrate the defenses around Alice's computer, we have a software developer, Doris, who sends or sells Axel[1] a benign program to run? To make this interesting, let's assume that Doris's program contains some secret $\mathcal{S}$ and that Axel can gain some economic advantage over Doris by extracting or altering $\mathcal{S}$:



1. In this book, Axel is the prototypical bad guy (hence the shaved head). For some variety in prose, we'll switch between calling him the *adversary*, the *attacker*, the *cracker*, the *reverse engineer*, or simply *he*. Doris is the person or institution who produces a piece of software with a secret she needs to protect. We'll call her the *author*, the *defender*, the *software developer*, *she*, or, most frequently, *you*.

The secret could be anything: a new super-duper algorithm that makes Doris program much faster than Axel's that he would love to get his hands on; the overall architecture of her program, which would be useful to Axel as he starts building his own; a cryptographic key that is used to unlock some media in a digital rights management system; or a license check that prevents Axel from running the program after a certain period of time. What can Doris do to protect this secret?

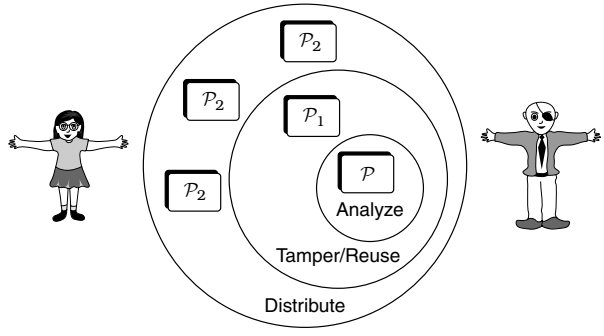
At first blush, you might think that cryptography would solve the problem, since, after all, cryptography is concerned with protecting the confidentiality of data. Specifically, a cryptographic system scrambles a cleartext $\mathcal{S}$ into a cryptotext $E_{\mathcal{K}}(\mathcal{S})$ so that it can't be read without access to a secret key $\mathcal{K}$:



So why doesn't Doris just protect the secret she has stored in her program by encrypting the program before selling it to Axel? Unfortunately, this won't work, since Axel needs to be able to execute the program and hence, at some point, it—and Doris' secret—must exist in cleartext!

What makes software protection so different from cryptography and standard computer security is that once Axel has access to Doris' program, there is no limit to what he can do to it: He can study its code (maybe first disassembling or decompiling it); he can execute the program to study its behavior (perhaps using a debugger); or he can alter the code to make it do something different than what the original author intended (such as bypassing a license check).

There are three components to a typical attack in a software protection scenario against Doris' program $\mathcal{P}$, namely, analysis, tampering, and distribution:

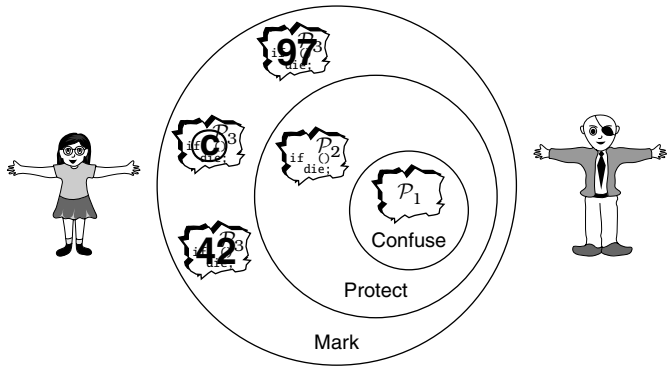


Axel starts by analyzing $\mathcal{P}$, extracting algorithms, design, and other secrets such as cryptographic keys or the location of license-checking code. Next, he modifies Doris' code (he may, for example, remove the license check) or incorporates pieces of it into his own program. Finally, Axel distributes the resulting program, thereby violating Doris' intellectual property rights.

There are many variants of this scenario, of course. Axel could remove a license check without redistributing the hacked program and just enjoy it for his own pleasure. He could resell the program along with a known license password, without ever having to tamper with the code. Finally, he could decompile and analyze the program to verify its safety (for example, that it doesn't contain damaging viruses or spyware, or, in the case of voting software, that it correctly counts every vote), without using this information to improve on his own programs. While these attacks occur in a variety of guises, they're all based on the following observation: Once a program leaves the hands of its author, any secrets it contains become open to attack.

In the scenarios we study, there is usually some financial motive for Axel to extract or alter information in the program. There is also typically a certain period of time during which Doris wants to protect this information. It is the goal of software protection to provide technical means for keeping the valuable information safe from attack for this period of time. A computer game developer, for example, may be happy if software protection prevents his program from being pirated for a few extra weeks, since most of the revenue is generated during a short time period after the release.

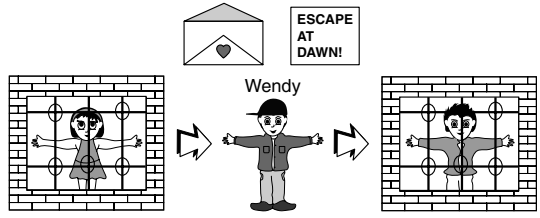
In a typical defense scenario, Doris adds *confusion* to her code to make it more difficult for Axel to analyze, *tamper-protection* to prevent him from modifying it, and finally *marks* the code (for example, with her copyright notice or Axel's unique identifier) to assert her intellectual property rights:



In this book we will consider five methods for Doris to protect her program: *Code obfuscation* for preventing analysis; *software watermarking*, *fingerprinting*, and *birthmarking* for detecting and tracing illegal distribution; and software- and hardware-based protection against tampering.

Although the primary motivation for the techniques developed in software protection has been protecting the secrets contained within computer programs, they also have applications to protecting the distribution chain of digital media (*digital rights management*), protecting against computer viruses, steganographic transfer of secret messages, and protecting against cheating in online computer games. We will also show how these techniques can be used maliciously to create stealthy computer viruses and to cheat in computer-based voting.

Software protection is related both to computer security and cryptography, but it has most in common with *steganography*, the branch of cryptography that studies how to transfer a secret *stealthily*. This is often illustrated by the so-called *prisoners' problem*. Here, Alice and Bob are planning a prison break by passing notes through their warden, Wendy:



Of course, if Wendy finds that a purported love note mentions a prison break, she will immediately stop any further messages and put Alice and Bob in solitary confinement. So what can the two conspirators do? They can't use cryptography, since as soon as Wendy sees a garbled message she will become suspicious and put an end to further communication. Instead, they must communicate surreptitiously, by sending their secrets hidden inside innocuous-looking messages. For example, Alice and Bob could agree on a scheme where the hidden message (the *payload*) is hidden in the first letter of each sentence in the *cover message*:

```
Easter is soon, dear! So many flowers! Can you smell
them? Are you cold at night? Prison food stinks! Eat
well, still! Are you lonely? The prison cat is cute!
Don't worry! All is well! Wendy is nice! Need you! ):
```

This is called a *null cipher*. There are many other possible types of cover messages. For example, Alice could send Bob a picture of the prison cat in which she has manipulated the low-order bits to encode the payload. Or she could send him an mp3-file of their favorite love song in which she has added inaudible echoes—a short one for every 0-bit of the payload, a longer one for every 1-bit. Or she could subtly manipulate the line spacing in a pdf-file, 12.0 points representing a 0, 12.1 points representing a 1. Or she could be even sneakier and ask Wendy to pass along a Tetris program she's written to help Bob while away the long hours in solitary. However, unbeknownst to Wendy, the program not only plays Tetris, but inside its control or data structures Alice has hidden the secret payload detailing their escape plan. In this book we will consider exactly this scenario and many like it. We call a program that contains both a secret and any technique for preventing an attack against this secret *surreptitious software*.

1.2 Attack and Defense

You cannot do computer security research, or computer security practice, without carefully examining your *attack model*, your assumptions about the adversary's abilities, and the strategies that he'll use to attack your system. In cryptography research, for example, you might assume that "the adversary cannot find the secret key" or "the adversary isn't better at factoring than I am" or "the adversary won't successfully tamper with the tamperproof smartcard." Once you have an adversarial model, you can go ahead and design a system that is secure against these attack scenarios. In the real world, adversaries will then immediately try to find scenarios you didn't think about in order to get past the defenses you've put up! The cheapest way to break a cryptosystem isn't to spend $100,000 on specialized hardware to factor a key—it's to spend $50,000 to bribe someone to *give* you the key. The easiest way to get secrets out of a smartcard isn't to pry the card open (having to bypass the security features that the designers put in place to defend against exactly this attack), but to induce faults in the card by subjecting it to radiation, modifying its power supply voltage, and so on, attacks the designers *didn't* have in mind.

In surreptitious software research, the situation is no different. Researchers have often made assumptions about the behavior of the adversary that have no basis in reality. In our own research, we (the authors of this book) have often made the assumption that "the adversary will employ static analysis techniques to attack the system," because coming from a compiler background, that's exactly what *we* would do! Or, others have speculated that "the adversary will build up a

complete graph of the system and then look for subgraphs that are weakly connected, under the assumption that these represent surreptitious code that does not belong to the original program." One might assume that those researchers came from a graph-theoretic background. Some work has endowed the adversary with not enough power ("the adversary will not run the program"—of course he will!) and some has endowed him with *too much* power: "The adversary has access to, or can construct, a comprehensive test input set for the program he's attacking, giving him the confidence to make wholesale alterations to a program he did not write and for which he does not have documentation or source code."

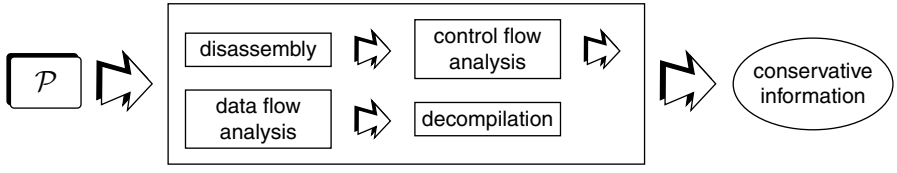
Unfortunately, much of the research published on surreptitious software has not clarified the attack model that was used. One of our stated goals with this book is to change that. Thus, for each algorithm, we present attacks that are possible now and that may be possible in the future.

In Chapter 2 we will also look at a *defense* model, ideas of how we good guys can protect ourselves against attacks from the bad guys. We will propose a model that tries to apply ideas taken from the way plants, animals, and human societies have used surreptition to protect themselves against attackers to the way we can protect software from attack. We will be using this model in the rest of the book to classify software protection schemes that have been proposed in the literature.

1.3 Program Analysis

An attack against a program typically will go through two stages: an *analysis stage* that gathers information about the program, and a *transformation stage* that makes modifications to the program based on the information that was collected. There are two basic ways of analyzing the program: You can just look at the code itself (this is called *static analysis*), or you can collect information by looking at the execution of the code (*dynamic analysis*).

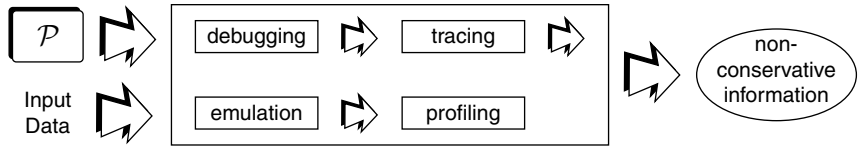
Static analyses takes only one input, the program P itself:



There are a huge number of different kinds of static analysis that have been developed over the years. The chief designers have been software engineering researchers

who want to analyze programs for defects and compiler researchers who want to analyze programs to optimize them, but there are also crackers who want to analyze programs to remove protection codes. Static analysis gathers information that we call *conservative*, that is, it may be imprecise but it will always err on the conservative side. So for example, if a static analysis tells you that "on line 45, variable `x` is always 42," you can be *sure* that this is the case. Sometimes conservative analyses will fail to gather a piece of information about the code that is in fact true, but at the very least, it will never lie and say that something is true when it isn't.

Dynamic analyses collect information about a program by executing it on a sample input data set:



The accuracy of the generated information depends on the completeness of the input data. Because of this, dynamic analysis can only make predictions such as, "On line 45, variable `x` is always 42, well, OK, at least for the set of inputs I've tried." Code transformations that make use of information only from static analyses are *safe* in that they won't turn a working program into a buggy one (assuming, of course, that the transformation itself is *semantics-preserving*, i.e., it doesn't change the meaning of the program). Transformations that use dynamic analysis results, on the other hand, will typically not be safe: They can fail if they are based on information gathered from an insufficient input data set.

Depending on what an attacker is trying to accomplish, he will choose different types of analyses and transformations. If all he wants to do is to disable a license check, the simplest of static and dynamic analyses may be all he needs: He can just run the program under a debugger until the "license expired" alert comes up, find the approximate location in the code, disassemble the code at that location, read until he finds something that looks like `if today's date > license date then...`, fire up a binary editor on the code, and edit out the offending lines. If, on the other hand, he wants to extract a complex algorithm from a huge program, being able to decompile it all the way to source code would be very helpful.

1.3.1 A Simple Reverse Engineering Example

To make this a little more concrete, let's look at an example. Assume that your boss has given you the following string of bytes and asked you to reverse engineer it:

```
06 3b 03 3c 1b 07 a2 00 15 1a 06 a3 00 0b
06 3b 84 01 01 a7 ff f1 06 3b a7 ff ec b1
```

He tells you that these bytes correspond to the bytecode for a competitor's Java program that contains a very secret and important algorithm. The code actually corresponds to the following program, but of course as a reverse engineer you don't know this yet:

```
public static void P() {
    int x = 3;
    int i = 0;
    while (i<4) {
        if (x <= 3)
            x = 3;
            i++;
        else {
            x = 3;
        }
    }
}
```

Since your goal is to find and understand the super-duper-secret algorithm, it would be great if you could turn this bytecode mess into some really clean and easy-to-read Java source code. If the owner of the code inserted some obfuscations or other trickery to confuse us, then it would be great if we were able to remove that too, of course.

As your first step, you'll want to disassemble the bytes. This means to convert the raw bytes into a symbolic, assembly code form. For Java bytecode this is essentially trivial, and there are many tools (such as `jasmin` or `javap`) that will do the job:

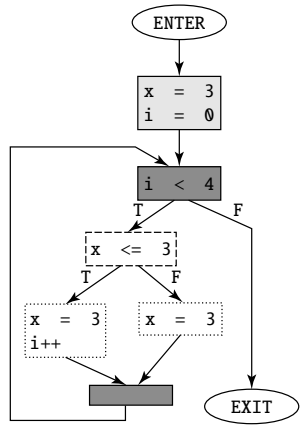
```
0:   [06]          iconst_3
1:   [3b]          istore_0
2:   [03]          iconst_0
3:   [3c]          istore_1

4:   [1b]          iload_1
5:   [07]          iconst_4
6:   [a2,00,15]    if_icmpge   27

9:   [1a]          iload_0
10:  [06]          iconst_3
11:  [a3,00,0b]    if_icmpgt   22

14:  [06]          iconst_3
15:  [3b]          istore_0
16:  [84,01,01]    iinc     1, 1

19:  [a7,ff,f1]    goto   4

22:  [06]          iconst_3
23:  [3b]          istore_0

24:  [a7,ff,ec]    goto   4

27:  [b1]          return
```

We've shaded the source code and instructions to make it easy for you to identify which part of the source code corresponds to which bytecode instructions. We've put the codebytes themselves in brackets.

Java bytecode was designed to make disassembly easy. The bytecode contains enough information to allow for the recovery of types and control flow. This is not true of other machine codes, such as those for x86 and other processors. For these binary codes, it is easy to insert code that will confuse disassembly. We will talk more about this in Chapter 3.

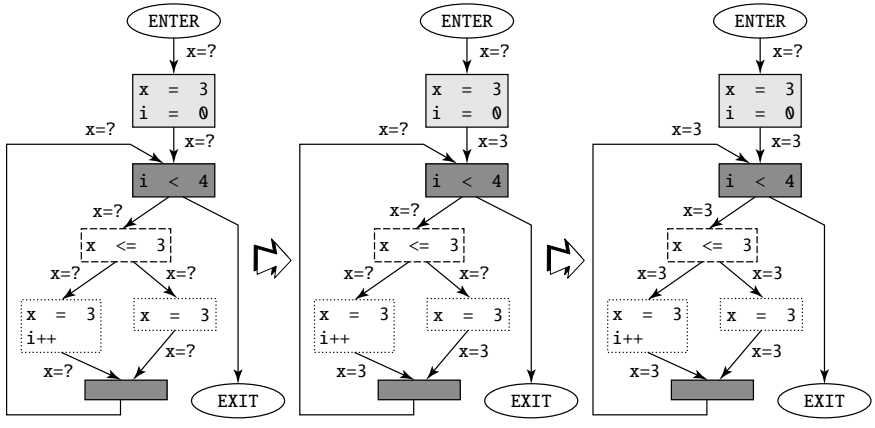
Now that you have the Java bytecode in an assembly code format, your next step is to perform *control flow analysis*, which will recover the order in which the code can be executed. The result of this analysis is a *control flow graph* (CFG). A node of this graph consists of straight-line code, except that the last statement can be a jump. There is an edge from one node to another if it is possible for us take

this path through the code during execution:

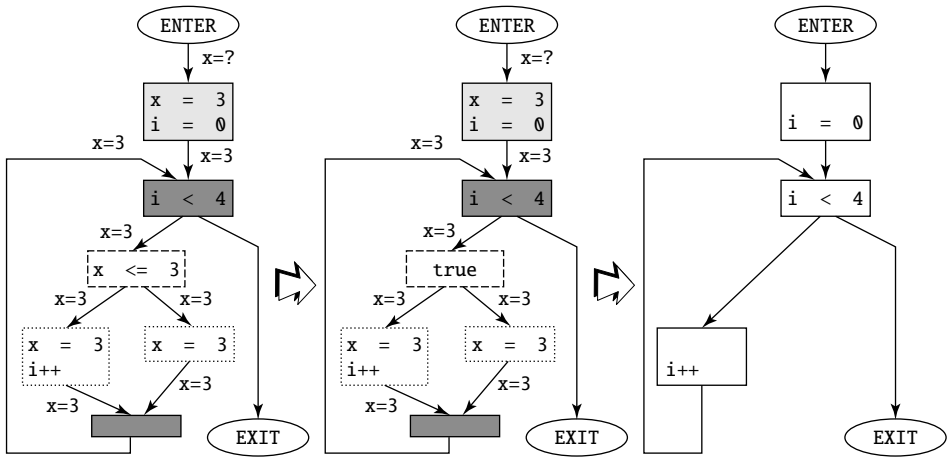


The nodes in the CFG are called *basic blocks*. CFGs are the central data structure around which many compilers and reverse engineering tools are built. We've again used shading to make it easy to see which basic blocks correspond to which bytecode instruction sequences.

Next, you will want to perform a variety of analysis tasks to gather as much information as you can about the code. This information may allow you to perform transformations that will make the code simpler and easier to understand, or even to remove some of the obfuscations that may have been added. One family of analyses common in optimizing compilers and reverse engineering tools is called *data flow analysis*. You'll learn more about this in Section 3.1.2 ▸ 127. In our example, an analysis called *Constant Propagation* can be used to track the value of variable `x` in order to see at which points in the code it will always have the same (constant) value:



In the leftmost flow graph, we've indicated that at first we know nothing about the value of `x` at the entry and exit of each basic block. In the second step, we've considered each basic block once, and have gathered some information. After a basic block has executed the statement `x = 3`, for example, it is clear that `x` must have the value 3. Also, if a basic block doesn't change the value of `x`, then `x` must have the same value after the block executes as it did before. When control can flow into a basic block from two different directions and we've computed that `x` has the same value on both paths, then we can safely assume that it will always have that value at the entry to the block. After considering all the basic blocks one more time, you're left with the annotated control flow graph to the left:



Given this CFG, you can now start to perform transformations. First, wherever `x` is used and you've determined that its value is constant, you can go ahead and replace `x` with the computed value. So, for example, `x<=3` can be replaced by `true`, since `x=3` at the entrance to this basic block. Given this transformation, you can now perform a *Dead Code Elimination*, getting rid of any basic block that can never be executed, and you can also get rid of any *redundant* statements. The result is the last CFG discussed earlier.

Now, eyeballing this code, it's clear that further transformations are possible. In fact, the loop reduces to a single statement, `i=4`, and since the procedure returns nothing and has no side effects, it can be removed altogether! In this particular case, this is easy to see, but in general, the result will depend on the power of the analyses and the complexity of the programs.

You can't be sure whether the "extra" code that you've been able to eliminate was inserted by a code obfuscator with the purpose of sowing confusion or was just

the result of a really broken compiler. All you know is that you've been able to turn the raw sequence of bytes your boss gave you into a much simpler structured form and to get rid of some irrelevant code in the process.

The final step of your reverse engineering effort should be to take the transformed control flow graph and turn it into source code. The graph is so simple that this *decompilation* step is trivial. You get:

```
public static void P() {
   int i = 0;
   while (i<4)
      i++;
}
```

Obviously, other source forms are possible, for example, using a `for` loop.

Now, turning raw bytes into readable source code is very helpful for a reverse engineer, but the process typically won't stop when source code is generated. The final stage, extracting a deep understanding of the algorithms employed or modifying the code to bypass license checks, and so on, will often have to be done by hand.

In Chapter 2 (Methods of Attack and Defense), we'll show you the general strategies that reverse engineers go through when they attack code in order to reveal its secrets or to make it perform tasks it wasn't designed to do. In Chapter 3 (Program Analysis), we'll delve into even more detail and discuss the many kinds of tools and analysis techniques available to your adversary. We will not simply present the off-the-shelf tools that happen to be available now, but will discuss those that *could be* built given currently known techniques. It's much too easy to say, "Our software protection algorithm is secure because we've implemented it for dynamically linked x86 executables, and current decompilers only handle statically linked code." A much more interesting statement would be, "This algorithm employs protection techniques that can cause code explosion in *any* decompiler, current and future."

1.4 Code Obfuscation

The first protection technique we're going to look at is code obfuscation. What's particularly interesting about obfuscation is that it's a double-edged sword: Bad guys use it to protect their malware from discovery (you will see this in the next section), good guys use it to protect their programs from reverse engineering, and bad guys can also use it to destroy secrets (such as watermarks) stored in the good guys' programs.

In the most general sense, to obfuscate a program means to transform it into a form that is more difficult for an adversary to understand or change than the original code. We are deliberately vague about defining "difficult," but we typically take it to mean that the obfuscated program requires more human time, more money, or more computing power to analyze than the original program. Under this definition, to distribute a program in a compiled form rather than as source code is a form of obfuscation, since analyzing binary machine code is more demanding than reading source. Similarly, we would consider a program that has been optimized to be more obfuscated than one that has not, since many code optimizations make analysis (both by humans and tools such as disassemblers and decompilers) more onerous.

However, the tools and techniques we present in this book go further than compilation and optimization in order to make a program hard to understand. In contrast to an optimizer that rearranges code for the purposes of efficiency, a *code obfuscator* transforms code for the sole purpose of making it difficult to analyze. A negative by-product of obfuscating transformations is that the resulting code often becomes larger, slower, or both. The author of the code has to decide whether the protection that the transformations afford is worth this overhead.

Obfuscation is often confused with *security through obscurity*, a term (used contemptuously) for the "branch" of cryptography or security where the algorithms used are expected to remain secret. This is in contrast to mainstream research that teaches that you must assume that all algorithms are public, and the only secrets you may keep are the cryptographic keys, and so on, that are the inputs to the algorithms. The idea is that many eyes examining the same algorithm or piece of code will likely be able to find flaws, and the more eyes that have failed to find a flaw, the more confident you can be that the algorithm is, in fact, secure. This principle is frequently violated, and you'll often see unscrupulous web-sites advertise "military-strength, proprietary" cryptographic algorithms, arguing that "since no one knows what our algorithm does, this will make it that much harder to break." The same argument is sometimes made in reverse by software vendors like Microsoft: "Open-source software is inherently more vulnerable to attacks than our closed-source code since you can easily read the source and find bugs to exploit." We know from experience that both claims are false. Hackers have no problem finding holes in closed-source software, and once a proprietary algorithm is leaked (which, inevitably, happens) it is often found to have serious and exploitable flaws.

As we define it in this book, obfuscation isn't security through obscurity. As with research in cryptography, we generally expect that the obfuscating code transformation algorithms are known to the attacker and that the only thing the defender can

Listing 1.1 Obfuscation example. The original unobfuscated version of the code can be found on the book's Web site.

```
public class C {
 static Object get0(Object[] I) {
  Integer I7, I6, I4, I3; int t9, t8;
  I7=new Integer(9);
  for (;;) {
   if (((Integer)I[0]).intValue()%((Integer)I[1]).intValue()==0)
        {t9=1; t8=0;} else {t9=0; t8=0;}
   I4=new Integer(t8);
   I6=new Integer(t9);
   if ((I4.intValue()^I6.intValue())!=0)
     return new Integer(((Integer)I[1]).intValue());
   else {
     if ((((I7.intValue()+ I7.intValue()*I7.intValue())%2!=0)?0:1)!=1)
        return new Integer(0);
     I3=new Integer(((Integer)I[0]).intValue()%((Integer)I[1]).intValue());
     I[0]=new Integer(((Integer)I[1]).intValue());
     I[1]=new Integer(I3.intValue());
    }
  }
}
```

assume is kept secret are the seeds that determine how and where these algorithms are applied.

Before we look at some applications of code obfuscation, let's have a look at what obfuscated code might actually look like. Check out Listing 1.1 for a very simple example generated by the SandMark Java code obfuscator. Without peeking (the answer is in footnote 2), time yourself to see how long it takes you to analyze this 20-line program and figure out what it does. Now imagine that rather than being 20 lines long, it's of a "normal" size for a program today: hundreds of thousands of lines to a few million lines. *Then* how long would it take you? What does your intuition tell you? Does the time to understanding grow linearly with the size of the code and the number of obfuscations applied? Do you think some obfuscations would add more confusion than others? Might some obfuscations be harder to undo than others? If so, how much harder? Are some impossible to undo? Unfortunately, the answers to these questions are largely unknown. As of now, we don't have any models that can tell us how much longer it would take to reverse engineer a program that's

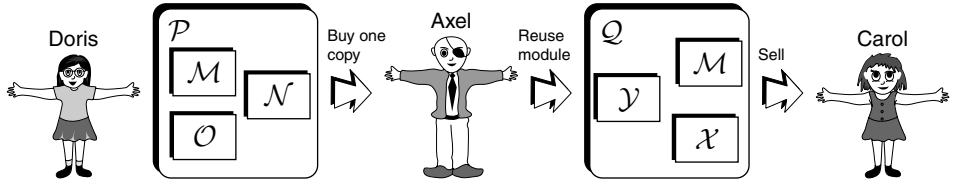
2. The program computes the Greatest Common Denominator of its arguments.

been obfuscated by a particular transformation or sequence of transformations, nor do we know what overhead these transformations will entail (although this is certainly easier to measure). Much current obfuscation research tries to devise such models [289], but we don't yet have any that are developed enough to be used by practitioners.

1.4.1 Applications of Code Obfuscation

Now let's look at a few scenarios where you can use code obfuscation to protect your code.

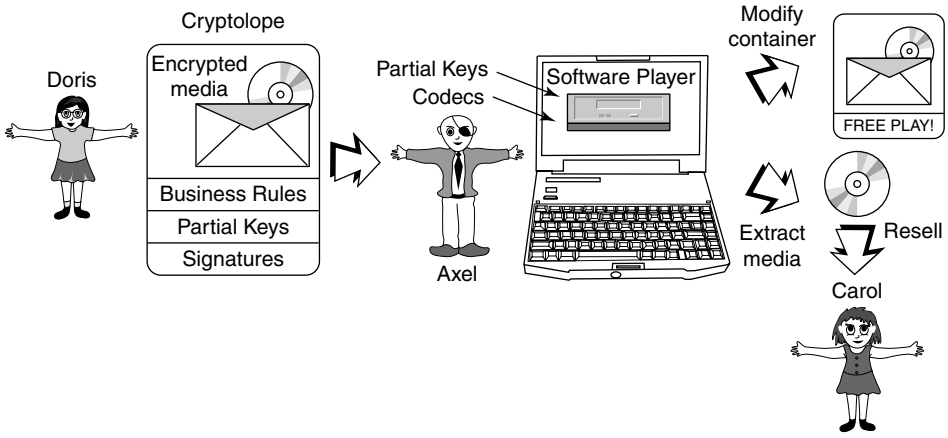
1.4.1.1 Malicious Reverse Engineering In the first scenario, *malicious reverse engineering*, Doris builds a program that contains a valuable trade secret (a clever algorithm or design), which Axel, a rival developer, extracts and incorporates into his own program and sells to his customer, Carol:



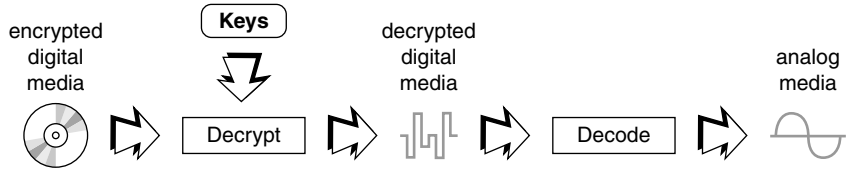
This scenario is what most people have in mind when they think of code obfuscation. As we'll soon see, it's far from the only one. The assumption (although there's no formal proof of this proposition) is that given enough time and resources, Axel will be able to reverse engineer any program. In other words, no secret hidden in a program will remain a secret forever. Doris' goal, instead, has to be to use obfuscation to slow Axel down as much as possible, while at the same time adding as little overhead as possible. Ideally, the code is convoluted enough that Axel gives up trying to understand it and says "OK, fine, then! I'll just reinvent this darned algorithm myself from scratch." Ideally, Doris is able to choose just the right set of obfuscating transformations and apply them in just the right places to not make her program so slow and bloated that her customers will no longer buy it.

1.4.1.2 Digital Rights Management In a *digital rights management* scenario, Doris is in the business of building a software media player. The player will only play music, images, or video that is distributed encrypted in a special file format known as a

cryptolope. The player contains cryptographic keys that are necessary to unlock and play the media:



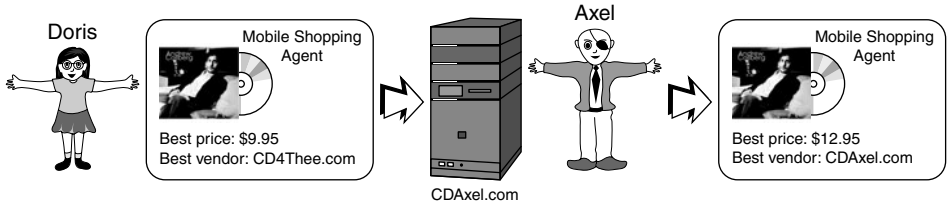
Since you want to be able to enjoy the encrypted media that you've bought in an "untethered environment," say, watching a movie on your laptop on a plane where there's no network connectivity, Doris is forced to store the decryption keys somewhere on your computer, probably inside your player's code. Along with the keys, of course, the code will also contain the decryption algorithm and a decoder that turns the decrypted media into analog signals that you can hear or watch. In a typical player, the decoding chain looks like this:



You should notice that there are three targets for Axel to attack here. He could steal the keys (and if they're universal keys he can now decode *any* media designed for this player, and, if they're not tied specifically to him, he can sell them on the Web), he could steal the digital media, or he could steal the less desirable analog output. The possible weak points of such a system are many. First of all, it's probably unreasonable to believe that the cryptographic algorithm used by the system will not be well known to an attacker. So unless the decryptor is obfuscated, a simple pattern-matching attack may be all that is necessary in order to locate the decryptor

and the keys it uses. Dynamic attacks are also possible. For example, cryptographic algorithms have very specific execution patterns (think tight loops with lots of `xors`) and if they're not heavily obfuscated, they'd be easy to find using a dynamic trace of the program. The keys themselves are a weak point. They're long strings of bits with a high degree of randomness, and as such, unusual beasts in most programs. So Axel could simply scan through the player code looking for, say, a 512-bit long string that's more random than expected. Any code that uses this string is likely to be the decryptor. Once Axel has found the location of the decryptor, he should have little problem finding where the decrypted media is generated and sent to the decoder. He can then simply add some code that writes the decrypted content to a file, and he's done. What we learn from this is that Doris needs to obfuscate her code so that a simple pattern-match against it won't reveal the location of the decryptor or decoder, or the interfaces between them. She needs to tamperproof the code so that Axel can't insert new code, she needs to obfuscate not only the static code but also the dynamic behavior of the player, and she needs to obfuscate static data (the keys) in the code as well. And, still, she has to assume that these defense measures are only temporary. Given enough time, Axel will bypass them all, and so she needs to have a plan for what to do when the system is broken.

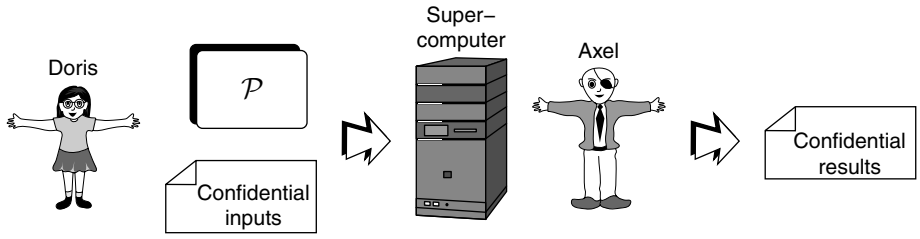
1.4.1.3 Mobile Agent Computing In our next scenario, Doris sends out a mobile shopping agent, which visits online stores in order to find the best deal on a particular CD. The agent traverses the Web and asks every store it encounters if they have the CD and how much it costs, records the best price so far, and eventually, returns to Doris with the site where she can get the best deal. Of course, if evil Axel runs a store there's no reason why he wouldn't cheat. First of all, he can just erase the information that the agent has collected so far and substitute his own price:



This strategy will only help him if the agent returns directly to Doris when it's done with Axel's site. Much better (for Axel) would be to manipulate the code so that regardless of which stores it visits after his, it will still record his (higher) price as the best one.

One defense that has been proposed (there are many others) is for Doris to obfuscate the agent [165], thereby slowing down an attack. Ideally, this way Axel won't have enough resources (he's servicing many simultaneous requests, after all) to reverse engineer and modify the agent. Also, Doris might be able to detect that the agent spends a suspicious amount of time at Axel's site. She can further complicate Axel's attack by differently obfuscating every agent she sends out. This way, he won't be able to speed up his analyses over time as he gathers more information about the agents and their defenses.

1.4.1.4 Grid Computing In the *grid computing* scenario, Doris wants to run her program P but lacks the computational resources to do so. So she buys cycles from Axel to run P on his supercomputer, sends Axel P and the input data, and receives the output data in return. The problem arises when one or more of P, the inputs, or the outputs are confidential:



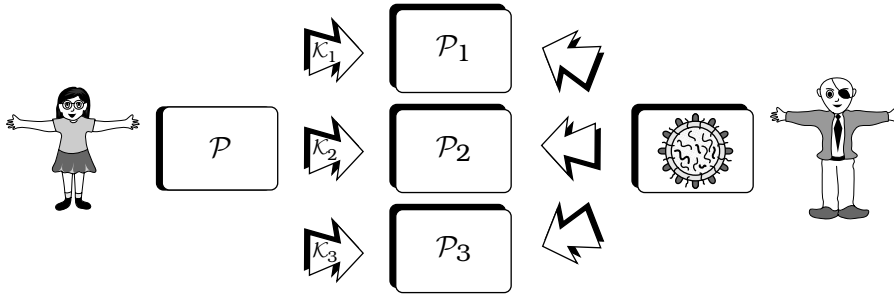
Doris must worry not only about Axel snooping on her algorithms or her inputs and outputs but also about his tampering with her program. If she can't trust that P maintains its integrity on Axel's site, she can't trust the validity of the output data that Axel returns to her.

One way to defend the confidentiality of the inputs and outputs is to encrypt them and transform P into a program that operates directly on encrypted inputs and produces encrypted results. There is considerable research on such *homomorphic encryption* schemes, but the ones invented so far are inefficient and not applicable to real programs.

Alternatively, Doris can obfuscate P to help maintain confidentiality of its algorithms or tamperproof it to help maintain its integrity by using the techniques in this book. To preserve the confidentiality of the data, something similar to a DRM scheme can be used, where obfuscation and tamperproofing are used to hide and protect the encryption code.

Grid computing is a harder scenario to protect than many others. The reason is that you care about the confidentiality of algorithms and data, integrity of code, and on top of that, *performance*. The reason that Doris sent her code to Axel, after all, was so that it would execute faster on his superior hardware! She would be very unhappy, indeed, if the protection techniques she applied negated the performance boost she was paying for.

1.4.1.5 Artificial Diversity Code obfuscation techniques have also been applied to operating systems to protect them against attacks by malware such as viruses and worms [74,75]. The idea is for Doris to randomize her code so that a malicious agent will not be able to locate or take advantage of a known vulnerability. Just like in the mobile agent scenario, we can take advantage of multi-versioning: If every distributed version of Doris' code is obfuscated differently, Axel's virus will need to be very clever to infect all of them:

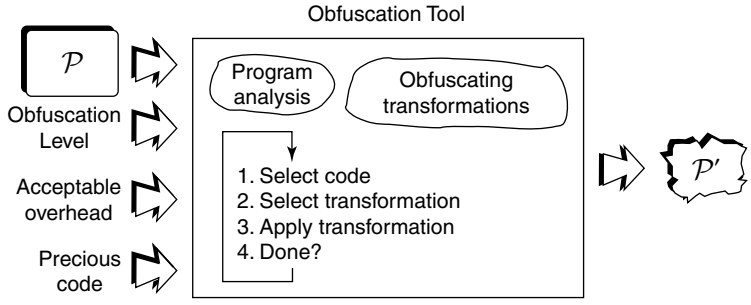


This is known as *artificial diversity*. Of course, viruses themselves make use of obfuscation techniques to avoid detection by virus scanners, and with spectacular success. We will talk about this in the next section.

1.4.2 Obfuscating Transformations

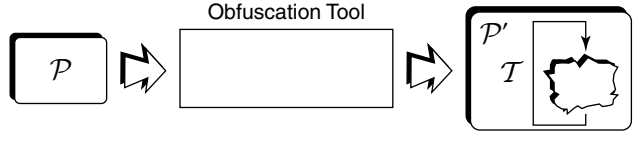
It's of course possible to take your program with its precious cargo and manually transform it into a mess that's hard for your adversary to understand and manipulate. In practice, though, that's too tedious and error-prone. A better idea is to build an obfuscation tool that translates your well-designed, easy-to-comprehend, easy-to-modify program into an incomprehensible mess of spaghetti code that's near-impossible to alter. Such an obfuscator is similar to a compiler, except that instead of generating efficient and compact code, it generates code that's hard for your adversary to comprehend.

Conceptually, an obfuscation tool takes four inputs: the program $\mathcal{P}$ you want to transform, the amount of obfuscation you would like to add, the amount of overhead you can accept, and a list of the code locations that are particularly precious to you that you would like to protect the most:



Internally, the obfuscator has a set of obfuscating code transformations, a set of program analyses needed to implement those transformations, and a loop that iteratively applies the transformations to $\mathcal{P}$. The analyses will be similar to those used by compilers and reverse engineering tools. The process continues until the amount of obfuscation you desire has been reached or the maximum amount of overhead you can accept has been exceeded. The output is a program $\mathcal{P}'$ that behaves the same as $\mathcal{P}$ but whose internal structure is very different. Practical code obfuscators may have a simpler structure than this. It's common, for example, to have just a small number of transformations and to apply them in a fixed order.

There are four broad classes of obfuscating code transformations. *Abstraction transformations* break down the overall structure of the program, i.e., they obfuscate the way the programmer has organized the program into classes, modules, and functions. *Data transformations* replace the data structures the programmer has selected with others that reveal less information. *Control transformations* modify the control structures (if- and while-statements) in the program to hide the paths it may take at runtime. *Dynamic transformations*, finally, insert a transformer $\mathcal{T}$ into the program so that, at runtime, $\mathcal{T}$ causes the program to continuously transform itself. At runtime, the program therefore looks like this:



We've spread our discussion of code obfuscation over three chapters. In Chapter 4 (Code Obfuscation), you will see many control, data, and abstraction transformations. We'll discuss the amount of confusion they add, how hard they are to defeat, and the amount of overhead they incur. In Chapter 6 (Dynamic Obfuscation), we'll do the same for dynamic obfuscation. In Chapter 5 (Obfuscation Theory), we will look at the theoretical underpinnings of obfuscation. In particular, we'll be interested in finding out what can be obfuscated, and what can't.

To give you some idea of what obfuscating code transformations do, let's go through a really trivial example. We'll start with a little C program in its original form and show how it changes as you apply, first, an abstraction transformation, then a data transformation, next a control transformation, and finally, a dynamic transformation. Here's the original program:

```
int main() {
   int y = 6;
   y = foo(y);
   bar(y,42);
}
```

```
int foo(int x) {
   return x*7;
}
```

```
void bar(int x, int z) {
   if (x==z)
      printf("%i\n",x);
}
```

The first thing we're going to do is to hide the fact that the program consists of two functions. The programmer had something in mind when he decided to break the program into three parts, `main`, `foo`, and `bar`; presumably, this matched the mental model he had of his program. So let's break this abstraction by merging `foo` and `bar` into one function, `foobar`. This new function takes three parameters. Two of them, `x` and `z`, are necessary to accommodate `bar`'s arguments, and the third, `s`, we'll use to distinguish calls that should execute `foo`'s and `bar`'s bodies. Here's `foobar` and the transformed version of `main`:

```
int main() {
   int y = 6;
   y = foobar(y,99,1);
   foobar(y,42,2);
}
```

```
int foobar(int x, int z, int s) {
   if (s==1)
      return x*7;
   else if (s==2)
      if (x==z)
         printf("%i\n",x);
}
```

Notice how it appears as if `main` calls the same function twice when, in fact, it's really calling two different functions.

Now, in many programs the precious thing that you want to protect is *data* rather than *code* or *design*. This is, for example, the case in a digital rights management system where you want to prevent the adversary from getting their hands on the cleartext media. Ideally, in a system like that, the data is *never* in cleartext. Rather, it is always encoded in some incomprehensible (to the attacker) format and always operated on in this format. Let's assume that, in our little example program, we want to protect all the integer values from the prying eyes of an attacker, who, for example, might be examining the program by running it under a debugger.

As it turns out, we're lucky. The program only performs three operations on the data, namely, assignment, multiplication, and comparison for equality. Why is this lucky? Well, there's a very simple encoding on integers that supports exactly these operations, namely, RSA encryption! We'll leave the details of this encoding to later in the book. For right now, you'll just have to take our word that setting

$$\begin{aligned} p &= 3 \\ q &= 17 \\ N &= pq = 51 \\ E(x) &= x^3 \bmod 51 \\ D(x) &= x^{11} \bmod 51 \end{aligned}$$

leads to a program where no integer values are ever in cleartext:

```
int main() {
    // E(6) = 12
   int y = 12;
   y = foobar3(y,99,1);
    // E(42) = 36
   foobar3(y,36,2);
}
```

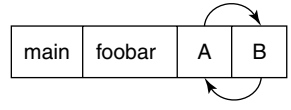
```
int foobar(int x, int z, int s) {
   if (s==1)
      return (x*37)%51;     // E(7) = 37
   else if (s==2)
      if (x==z) {           // x11 = D(x)
         int x2=x*x % 51,    x3=x2*x % 51;
         int x4=x2*x2 % 51,  x8=x4*x4 % 51;
         int x11=x8*x3 % 51; printf("%i\n",x11);
      }
}
```

In particular, you can see how 6 is encoded as 12, 42 as 36, and 7 as 37! Not until the program absolutely *has to* have a value in cleartext (when it needs to pass it to `printf` to print it out) is it finally decoded. Note also that the multiplication `x*7` takes place in the encoded domain; again, no values are in cleartext until necessary.

Structured programming dictates that you organize your functions by properly nesting conditional and loop statements. This makes the code easy to understand and modify. One popular kind of obfuscating control transformation, *control flow flattening*, rewrites functions to turn structured statements into spaghetti code. Here's what the last version of the `foobar` function looks like after control structures have been replaced by plain old `goto` statements:

```
int foobar(int x, int z, int s) {
   char* next = &&cell0;
   int retVal = 0;

   cell0: next = (s==1)?&&cell1:&&cell2; goto *next;
   cell1: retVal=(x*37)%51; goto end;
   cell2: next = (s==2)?&&cell3:&&end; goto *next;
   cell3: next = (x==z)?&&cell4:&&end; goto *next;
   cell4: {
      int x2=x*x % 51,   x3=x2*x % 51;
      int x4=x2*x2 % 51, x8=x4*x4 % 51;
      int x11=x8*x3 % 51;
      printf("%i\n",x11); goto end;
   }
   end: return retVal;
}
```

Have a look at Listing 1.2▸25. Here, we've broken the body of `foobar` into two functions, `A` and `B`. This, by itself, isn't a very effective transformation, but what's interesting here is what happens to `A` and `B` at *runtime*. Every time `foobar` is called, it makes `A` and `B` trade places:



From an attacker's point of view this code is hard to understand in two different ways. If he looks at our program *statically*, i.e., without running it, the abstraction transformation will have removed the way we chose to organize the program, the data transformation the way we chose our data structures, and the control transformations the way we structured our flow of control. If, instead, he decides to learn

Listing 1.2 The result of a dynamic obfuscating transformation. The functions `A` and `B` will continuously trade places at runtime. The `swap` function is in Listing 6.3 ▸377.

```
int start = 0;

typedef int (*printfT) (char const *str,...);
typedef int (*FuncPtr)(int,int,int,uint32,int,printfT,void  *  funcs);
typedef FuncPtr FuncPtrArr[];
static FuncPtrArr funcs ={&A,&B};

int A(int x, int z, int s, uint32 begin,
      int start, printfT printf,void  *  funcs) {
    int next = 0;
    int retVal = 0;
    char* cells[]={&&cell0-(uint32)&A,&&cell1-(uint32)&A,
                   &&cell2-(uint32)&A,&&cell3-(uint32)&A,
                   &&cell4-(uint32)&A,&&end-(uint32)&A};
    goto *(cells[next]+begin);
    cell0:  next = (s==1)?1:2;  goto  *(cells[next]+begin);
    cell1:  retVal=(x*37)%51;  next=5; goto  *(cells[next]+begin);
    cell2:  next = (s==2)?3:5;  goto  *(cells[next]+begin);
    cell3:  next = (x==z)?4:5;  goto  *(cells[next]+begin);
    cell4:  FuncPtr f = ((FuncPtr*)  funcs)[(1+start)%2];
            f(x,z,s,(uint32)f,start,printf,funcs);
            next=5; goto *(cells[next]+begin);
    end:  return retVal;
}

int B(int x, int z, int s, uint32 begin,
      int start,printfT printf,void  *  funcs) {
    int x2=x*x % 51; int x3=x2*x % 51; int x4=x2*x2 % 51;
    int x8=x4*x4 % 51; int x11=x8*x3 % 51; printf("%i\n",x11);
}

int foobar(int x, int z, int s) {
    int retVal = funcs[0+start](x,z,s,(uint32)funcs[0+start],
                                start,printf,funcs);
    swap((caddr_t)funcs[0],(caddr_t)funcs[1],1024);
    start = (start+1) % 2;
    return retVal;
}
```

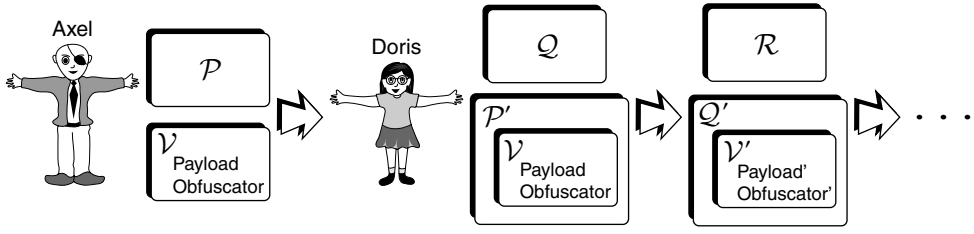
about the program by executing it, he will find that the dynamic obfuscation has violated a very basic assumption about programs, namely, that every time control reaches a particular point, the same code is executed.

1.4.3 Black Hat Code Obfuscation

For many years obfuscation was considered nothing more than a curiosity, something that no serious researcher would touch. The *International Obfuscated C Code Contest* (*IOCCC*) [177,227], for example, is an annual event that (humorously) tries to write the worst programs possible, C being a particularly suitable language for this task. It was generally assumed that there could be no real value to any code obfuscation technique and that anyone using one was just foolish for not using "real" security algorithms. Only fairly recently has it been acknowledged that there are legitimate applications where obfuscation and related techniques are the only available methods of protection.

Unfortunately, however, it's turned out that the applications where obfuscation has had its most spectacular success are in what we like to call black hat scenarios. This should probably not come as much of a surprise. It's not uncommon for the bad guys to adopt techniques first designed by the good guys. Cryptography, for example, can be used to protect the communication between criminals as well as between law enforcement agents. Steganography can be used by freedom fighters to avoid detection by an oppressive regime as well as by terrorists to avoid detection by national security forces. TV set-top box hackers have been known to first break through the smartcard-based defenses of the box and then turn around and use smartcards themselves to protect these hacks from counterattacks by the set-top box manufacturers!

One black hat scenario is when Axel uses obfuscation to protect his virus V from detection by Doris:



The virus comes in two parts, the *payload*, which is the code that's designed to cause harm, and the obfuscator, which the virus uses to protect itself from discovery. In the first step, Axel infects a program $\mathcal{P}$ with $\mathcal{V}$ and sends it out into the wild. If

Doris installs the infected $\mathcal{P}'$ on her site, the virus may be able to infect another program, Q. Before infection, however, the virus uses the obfuscator to generate a different version of itself. The idea is that if every version of the virus is different, it will be difficult for Doris' virus scanner to detect it. This is similar to the artificial diversity scenario you saw earlier, only this time the good guy and the bad guy have traded places!

1.4.3.1 Misdirection—Stealthy Obfuscation If you look at a few of the programs submitted to the IOCCC, it should be clear that the code looks far from natural. While machine-generated code, obfuscated code, or optimized code can often look this bad, code written by humans typically doesn't. For example, you can tell by looking at the obfuscator-generated code in Listing 1.1 ▸15 that it wasn't written by a typical human programmer. So if Axel was looking for a secret in Doris' program, a section that looked like this would likely raise his suspicion—could the secret be hidden behind this obviously obfuscated mess? You'll see many cases in this book where the *stealthiness* of a protection technique is important; the attacker mustn't be given a clue as to where in the code the technique was applied or the order in which a sequence of techniques were applied.

Misdirection is a particularly nasty black hat obfuscation technique that is based on the extreme stealthiness of an inserted bug. Look at Listing 1.3 ▸28, which shows a program to collect and tally the votes for *American Idol*. The program reads votes from standard input, and after the contest displays a summary of the votes cast. Here is a sample run of the program:

```
> cat votes-cast.txt
alice
alice
bob
alice
dmitri
bob
zebra
> java Voting < votes-cast.txt
Total: 7
Invalid: 1
alice: 3
bob: 2
charles: 0
dmitri: 1
```

Listing 1.3 Obfuscated voting code.

```
public class Voting {
  final int INVALID_VOTE = -1;
  int invalidVotes, totalVotes = 0;
  String[] candidates = {"alice", "bob", "charles", "dmitri"};
  int[] tally = new int [candidates.length];
  BufferedReader in = null; BufferedWriter log = null;
  public Voting () {
    in = new BufferedReader (new InputStreamReader (System.in));
  }
  public String readVote () {
    try {return in.readLine();}
    catch (Exception e) {return null;}
  }
  public boolean isValidTime (Date today) {
    SimpleDateFormat time = new SimpleDateFormat ("HH");
    int hour24 = Integer.decode (time.format(today)).intValue();
    return !(hour24 < 9 || hour24 > 21);
  }
  public int decodeVote (String input) {
    for (int i=0; i < candidates.length; i++)
      if (candidates[i].equals (input)) return i;
    return INVALID_VOTE;
  }
  public void logVote (Date date, int vote) throws Exception {
    if (log == null)
      log = new BufferedWriter (new FileWriter ("log.txt"));
    log.write ("TIME: " + date + " VOTE: " + vote);
  }
  public void printSummary () {
    System.out.println ("Total:"+totalVotes +
                                "\nInvalid:"+invalidVotes);
    for (int i=0; i < candidates.length; i++)
      System.out.println ("Votes for " + candidates[i] +": "+tally[i]);
  }
  public void go () {
    while (true) {
      String input = readVote();
      int vote = 0;
      if (input == null)break;
      try {
        Date today = new Date();
        if (isValidTime (today)) vote = decodeVote (input);
        else                     vote = INVALID_VOTE;
```

Listing 1.3 Obfuscated voting code. (*Continued*)

```
        logVote (today, vote);
      } catch (Exception e) {}
      totalVotes++;
      if (vote == INVALID_VOTE) invalidVotes++;
      else                      tally[vote]++;
    }
    printSummary();
  }
  public static void main (String args[]) {
    Voting voting = new Voting (); voting.go();
  }
}
```

Can you tell whether the program produces a fair count or not, or has it, in fact, been deliberately manipulated to favor a particular candidate? Before reading the answer in footnote 3, time yourself to see how long it took to analyze this 58-line program. Now, how long would it take for you to find a potential problem in a real-world voting system comprising hundreds of thousands of lines of code? What if the techniques used in Listing 1.3 were combined with those in Listing 1.1 ▶15? Might it be possible to provide enough confusion so that by the time the obfuscation has been penetrated and any irregularities identified, the next American Idol has already been selected (or, in the case of the United States' presidential election, the Electoral College has convened and cast their votes)?

1.4.3.2 Obfuscating Viruses As you will see in this book, there are important real-world problems that, to date, only obfuscation is able to tackle. Unfortunately, however, it's in black hat code scenarios where obfuscation has had its most spectacular successes: Obfuscation is used by malware writers to protect their viruses, worms, trojans, and rootkits from detection. It is entertaining to examine techniques that hackers have invented and successfully used to foil security researchers, and to see whether the good guys can make use of the same techniques. Virus writers and virus scanner writers are engaged in a cat-and-mouse game: When a new virus detection technique is invented, the virus writers counter with a more sophisticated

3. Java's integer decoding routine interprets numbers that start with zero as octal. As a result, this routine throws an unexpected number-format exception between 8 a.m. and 9:59 a.m., which in turn results in the incorrect value of vote being counted.

code obfuscation technique, which is then countered by a more powerful detection technique, and so on. So far, the hackers seem to be winning. Recent reports claim that 25% of the world's computers have been penetrated and taken over by botnets [368]. Obviously, this can be attributed not only to the success of obfuscated malware but also to the fact that many people run unpatched operating systems, outdated virus scanners, and so on.

Virus writers typically beat virus scanners by making their virus code "invisible" to the scanners. Scanners have only a limited amount of resources and cannot fully analyze a file to decide whether or not it's malicious. Even if they could, there are theoretical results [74] that state that it may still fail. The scanner therefore identifies a virus by its *signature*, some aspect of it that is easy enough to extract and that does not change from one infection to the next. This is similar to *birthmarks*, which you will see in Chapter 10 (Software Similarity Analysis).

What sort of tricks does a virus writer use to make the viruses stay "invisible" to virus scanners? Look at the self-reproducing viral Java program in Listing 1.4▸31. Real viruses are rarely, if ever, written in Java, but this simplified example will help you understand how a virus can use obfuscation to protect itself.

There are several things that are noteworthy in this Java virus. The first thing to note is that the program seems to have its entire source code encoded as a string within itself. In order to manage this in a finite space, a program takes advantage of the duplicate structure of the program. This trick is used in a common geek amusement to devise *quines*—programs that when run produce their own source code as their only output [1].

The source code is built in the variable `self`. It's eventually written to a file and compiled using Sun's internal Java compiler. The recent Slapper worm [320] and its variants also used a compiler to make the virus less platform-dependent. Using a compiler in this way has another side effect that is advantageous for a virus writer. Different compilers can produce slightly different (though semantically equivalent) programs from the same source code. This makes it a bit harder to find good signatures that scanners can use to reliably detect a virus.

In Listing 1.4▸31, the example goes one step further. Before the new copy of the virus is written to a file, it is passed to the function `morph`. The `morph` function adds a `i++` statement between every line of the main program. This instruction has no effect on the output or behavior of the program. However, this trivial obfuscation ensures that every new version of the program will be different from its predecessor! This means that a virus scanner that uses a trivial checksum-based signature detector will not be powerful enough to identify the virus.

Listing 1.4 A metamorphic virus in Java.

```
import java.io.*;

public class m {
  private static int i = 0;
  private static com.sun.tools.javac.Main javac=
                                new com.sun.tools.javac.Main();
  public static String morph (String text) {
    return text.substring(0,360) +
           text.substring(360).replaceAll (
             new String (new char[] { ';' }),
             new String (new char[] { ';','i','+','+',';' }));
  }
  public static void main (String[] a) throws IOException {
    String self="import java.io.*;   public class m { private
      static int i = 0; private static com.sun.tools.javac.Main
      javac=new com.sun.tools.javac.Main();public static String
      morph (String text) { return text.substring(0,360) +
      text.substring(360).replaceAll (new String (new char[]
      { ';' }),new String (new char[] { ';','i','+','+',';' })
      );} public static void main (String[] a) throws IOException
      { String self=@;char q=34;char t=64;String text=
      self.replaceAll(String.valueOf(t),q+morph(self)+q);String
      filename = new String (new char[] { 'm','.','j','a','v',
      'a' });File file=new File(filename); file.deleteOnExit();
      PrintWriter out=new PrintWriter(new FileOutputStream(file
      )); out.println(text);out.close(); javac.compile(new
      String[]{filename});}}";
    char q=34; char t=64;
    String text=self.replaceAll(String.valueOf((char)64),
                                q+morph(self)+q);
    text=text.replaceAll(String.valueOf((char)35),
                         String.valueOf((char)34));
    String filename =
      new String (new char[] { 'm','.','j','a','v','a' });
    File file = new File(new String (filename));
    file.deleteOnExit();
    PrintWriter out =
      new PrintWriter(new FileOutputStream(file));
    out.println(text);
    out.close();
    javac.compile(new String[] { filename });
  }
}
```

Imagine if, instead of our very trivial `morph` function, a virus writer included and used one of the more sophisticated obfuscations we discussed in the last section. What sort of analysis would a scanner need to perform to detect such a virus?

Real viruses use obfuscation to counter detection by ensuring that as many properties as possible change between infections. *Metamorphic viruses* use obfuscating transformations to change the entire body of the program from one generation to the next. *Polymorphic viruses* are a simpler variation of metamorphic viruses. Instead of morphing the entire program, each generation of the virus encrypts itself with a different key. Of course, for the virus to still be functional it must also contain a decryption and execution routine. Polymorphic viruses only morph this decryption and execution routine. In this way, morphing protects the decryption and execution routine from the scanner, while encryption protects the rest of the program.

1.5 Tamperproofing

One of the uses of obfuscation is to add so much confusion to a program that an adversary will give up trying to understand or modify it. But what if Axel is able to break through Doris' obfuscation defenses, then what? Well, in addition to obfuscating her code, Doris can also *tamperproof* it. This means that if Axel tries to make modifications to Doris' program, he will be left with a program with unintended side effects: The cracked program may simply refuse to run, it could crash randomly, it could destroy files on Axel's computer, or it could phone home and tell Doris about Axel's attack, and so on.

In general, a tamperproofing algorithm performs two duties. First, it has to detect that the program has been modified. A common strategy is to compute a checksum over the code and compare it to an expected value. An alternative strategy is check that the program is in an acceptable execution state by examining the values of variables.

Once tampering has been detected, the second duty of a tamperproofing algorithm is to execute the tamper response mechanism, for example causing the program to fail. This is actually fairly tricky: You don't want to alert the attacker to the location of the tamperproofing code since that will make it easier for him to disable it. For example, tamperproofing code like

```
if (tampering-detected()) abort()
```

is much too weak because it's easy for the attacker to trace back from the location where the program failed (the call to `abort()`) to the location where tampering

was detected. Once he has that information, the tamperproofing is easy to disable. A good tamperproofing system separates the tamper detection from the tamper response in both space and time.

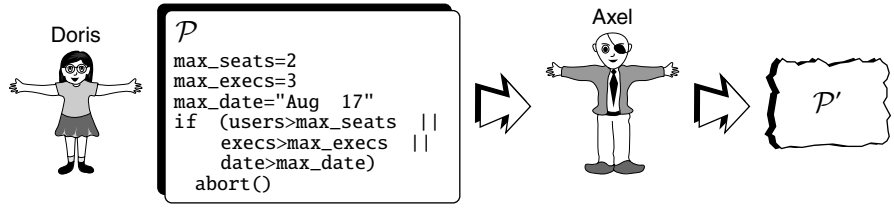
1.5.1 Applications of Tamperproofing

Tamperproofing is important in digital rights management systems where any change to the code could allow Axel to enjoy media for free. Here's a top-level view of a DRM system:

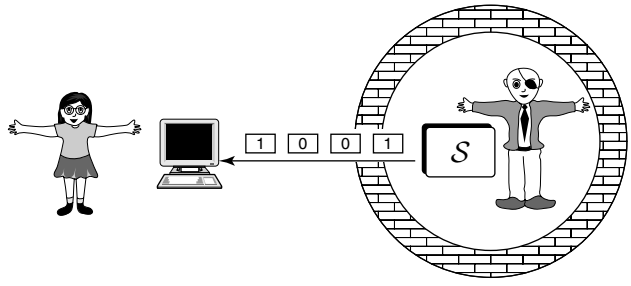
```
static final long key = 0xb0b5b0b5;
void play(byte[] media) {
   // if (!hasPaidMoney("Bob")) return;
   System.out.print(key);
   byte[] clear = decrypt(key,media);
   System.out.print(clear);
   float[] analog = decode(clear);
   System.out.print(analog);
   device.write(analog);
}
```

Notice how Axel has been able to delete a check (light gray) that prevented him from playing the media if he had no money, and to insert extra code (dark gray) to dump the secret cryptographic key and the decrypted (digital and analog) media. In general, tampering with a program can involve deleting code, inserting new code, or modifying original code. All can be equally devastating.

Another common use of tamperproofing is to protect a license check. Doris inserts a check in the code that stops Axel from running the program after a specific date or after a certain number of executions, or to let no more than a given number of Axel's employees run the program at any one time. Axel, being an unscrupulous user, locates and disables the license check to give him unlimited use of the program. Doris can obfuscate her program (to make it hard for Axel to find the license code), but she can also tamperproof the license check so that if Axel finds and alters it, the program will no longer function properly:



There are situations where simply failing when tampering is detected isn't enough; you also need to alert someone that an attack has been attempted. For example, consider a multi-player online computer game where, for performance reasons, the client program caches information (such as local maps) that it won't let the player see. A player who can hack around such limitations will get an unfair advantage over his competitors. This is a serious problem for the game manufacturer, since players who become disillusioned with the game because of rampant cheating may turn to another one. It's therefore essential for the game administrators to detect any attempt at tampering so that anyone who tries to cheat can immediately be ejected from the game. The term we use for this problem is *remote tamperproofing*. The goal is to make sure that a program running on a remote untrusted host is the correct, unadulterated version, and that it is running in a safe environment. "Safe" here can mean that the program is running on the correct hardware (and not under a hacked emulator), that the operating system is at the correct patch-level, that all environment variables have reasonable values, that the process has no debugger attached, and so on. As long as the client code determines that it is running safely and hasn't been tampered with, it sends back a stream of surreptitious "I'm-OK" signals to the server:



In this figure, the program is using steganographic techniques to embed this secret bitstream in TCP/IP headers.

Even if you don't intend to cause harm to a cheating game player, monitoring your protected programs for evidence of an ongoing attack can still be useful. We know of at least one major software protection company that includes a phone home facility in their code that allows them to watch, in real time, an attack in progress. Being able to see which strategies the attacker is using, see which strategies are successful, measure how long an attack takes to complete and thus, ultimately, get a feel for which protection algorithms are effective and which are not is definitely useful. However, we know that other software

protection companies eschew such eavesdropping because of its problematic privacy issues.

1.5.2 An Example

If you have ever run a signed Java applet or installed a program signed by Microsoft, you've already had experience with programs designed to detect tampering. A signed program carries a certificate issued by a known authority that can be used to detect if some part of the program has been changed. Usually such detection is extrinsic to the program itself. For example, if you try to install a signed Microsoft program that has been tampered with, the OS will check that the signature is one that Microsoft trusts and that the program has not been altered since it was signed. Unfortunately, the obvious way of extending such a scheme to allow a program to check *itself* for tampering is not effective. To see this, have a look at the program in Listing 1.5▸36, which converts temperatures expressed in Fahrenheit into Celsius. The function `bad_check_for_tampering` protects the program from tampering by comparing a checksum of the program file to a checksum computed when the program was compiled. If they're not the same, the program simply quits.

Can you think of any difficulties in writing such a program? One problem is that checksum is embedded in the program and as a result affects the checksum itself! To successfully construct such a program requires searching for a value that, when inserted into the program as a checksum, results in the program having that value as a checksum.

A further problem is that in spite of its clever construction, once a function has been identified by the attacker as a tamper-detection function, it's easy to remove any calls to it.

The function `better_check_for_tampering` is slightly better. Instead of merely checking for tampering, it incorporates the output of the checksum into the code itself. If the program is altered, the value returned by this function changes, which in turn makes the program subtly incorrect.

In spite of the improvements, `better_check_for_tampering` remains vulnerable to the same attack as `bad_check_for_tampering`. Both tamper-checking functions are easy to identify, because programs rarely attempt to read themselves. Once a function is identified as a tamper-checking function, it can be replaced by a simple function that returns the original checksum as a constant. Thereafter, an attacker can go ahead and modify the program arbitrarily.

These weaknesses notwithstanding, checksumming code forms the basis of many tamperproof detection techniques used in practice. You'll see some of them in Chapter 7 (Software Tamperproofing).

Listing 1.5 A self-checking program in Java.

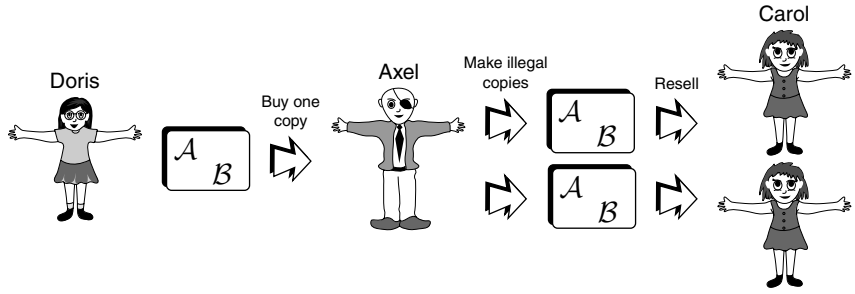
```
import java.io.*;
import java.security.*;

public class tamper {

   public static int checksum_self () {
      File file = new File("tamper.class");
      FileInputStream fr = new FileInputStream (file);
      DigestInputStream sha = new DigestInputStream (fr,
                              MessageDigest.getInstance ("SHA"));
      while (fr.read () != -1) {}
      byte[] digest = sha.getMessageDigest().digest ();
      int result = 12;
      for (int i=0; i <  digest.length; i++) {
         result = (result + digest[i]) % 16;
      }
   }
   public static boolean bad_check_for_tampering () throws Exception {
      return checksum_self() != 9;
   }
   public static int better_check_for_tampering () throws Exception {
      return checksum_self();
   }
   public static void main (String args[]) throws Exception {
      if (bad_check_for_tampering()) System.exit (-1);
      float celsius=Integer.parseInt (rgs[0]);
      float fahrenheit=better_check_for_tampering() * celsius / 5 + 32;
      System.out.println (celsius + "C = " + fahrenheit + "F");
    }
}
```

1.6 Software Watermarking

There are various scenarios where you would like to *mark* an object to indicate that you claim certain rights to it. Most famously, the government marks all their paper currency with a *watermark* that is deeply embedded in the paper and is therefore difficult to destroy or reproduce. For example, if you found a torn or damaged part of a note, you could hold it up to the light and use the watermark to identify the original value of the note. Also, if a forger attempts to pay you using photocopied

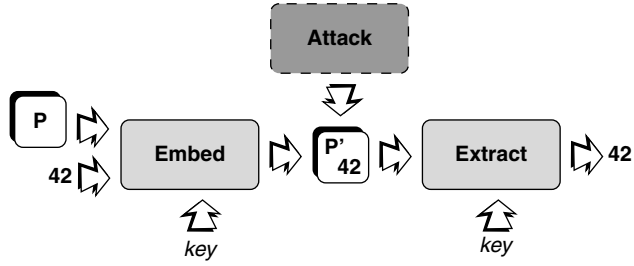
currency, the missing watermark will alert you to the fact that the currency is not genuine.

There has been much work in the area of *media watermarking*, where the goal is to embed unique identifiers in digital media such as images, text, video, and audio. In a typical scenario, Doris has an online store that sells digital music. When Axel buys a copy of a song, she embeds two marks in it: a copyright notice $\mathcal{A}$ (the same for every object she sells) that asserts her rights to the music, and a customer identification number $\mathcal{B}$ (unique to Axel) that she can use to track any illegal copies he might make and sell to Carol:



If Doris gets ahold of an illegal copy of a song, she can extract the customer mark $\mathcal{B}$ ($\mathcal{B}$ is often referred to as a *fingerprint*), trace the copy back to Axel as the original purchaser, and then take legal action against him. If Axel were to claim "Well, I wrote and recorded this song in the first place," Doris can extract her copyright notice $\mathcal{A}$ to prove him wrong.

Media watermarking algorithms typically take advantage of limitations in the human sensory systems. For example, to embed a watermark in a piece of music, you can add short—and to the human ear, imperceptible—echos. For every 0-bit of the mark, you'd add a really short echo and for a 1-bit, you would add a slightly longer echo. To mark a PDF file, you'd slightly alter the line spacing: 12 points for a 0-bit, and 12.1 points for a 1-bit. To mark an image, you'd slightly increase or decrease the brightness of (a group of) pixels. In all these cases you also need to decide *where* in the digital file you will make the changes. This is often done by means of a random number generator that traces out a sequence of locations in the file. The seed to the generator is the *key* without which you cannot extract the watermark. So a typical watermarking system consists of two functions, *embed* and *extract*:



Both functions take the secret key as input. The *embed* function also takes the original object (known as the *cover object*) and the watermark (the *payload*) as input, and produces a watermarked copy (the *stego object*) as output. The *extract* function, as the name implies, extracts the watermark from the stego object, given the correct key. This is just one basic watermarking system, and we'll discuss others in Chapter 8 (Software Watermarking).

As you see from the figures above, we also have to take the adversary into account. Axel will want to make sure that before he resells Doris' object he's destroyed every watermark she's embedded. More precisely, he needs to somehow disturb the watermark extraction process so that Doris can no longer extract the mark, even given the right key. In a typical attack, Axel inserts small disturbances into the watermarked object, small enough that they don't diminish its value (so he can still resell it), but large enough that Doris can no longer extract the mark. For example, he might randomly adjust the line spacing in a PDF file, spread a large number of imperceptible echoes all over an audio file, or reset all low-order bits in an image file to 0. Media watermarking research is a game between the good guys who build marking algorithms that produce robust marks and the bad guys who build algorithms that attempt to destroy these marks. In both cases, they want the algorithms to have as little impact on a viewer's/listener's perception as possible.

Of course, our interest in this book is watermarking *software*, not media. But many of the principles are the same. Given a program $\mathcal{P}$, a watermark w, and a key k, a software watermark embedder produces a new program $\mathcal{P}_w$. We want $\mathcal{P}_w$ to be semantically equivalent to $\mathcal{P}$ (have the same input/output behavior), be only marginally larger and slower than $\mathcal{P}$, and of course, contain the watermark w. The extractor takes $\mathcal{P}_w$ and the key k as input and returns w.

1.6.1 An Example

Take a look at the example in Listing 1.6▸39. How many fingerprints with the value `"Bob"` can you find? Actually, that's not a fair question! As we've noted, we must

Listing 1.6 Watermarking example.

```
import java.awt.*;
public class WMExample extends Frame {
   static String code (int e) {
      switch (e) {
         case 0 : return "voided";
         case 6 : return "check";
         case 5 : return "balance";
         case 4 : return "overdraft";
         case 2 : return "transfer";
         case 1 : return "countersign";
         case 3 : return "billing";
         default: return "Bogus!";
      }
   }

   public void init(String name) {
      Panel panel = new Panel();
      setLayout(new FlowLayout(FlowLayout.CENTER, 10, 10));
      add(new Label(name));
      add ("Center", panel);
      pack();
      show();
   }

   public static void main(String args[]) {
      String fingerprint = "Bob";

      if (args[0].equals("42"))
         new WMExample().init(code(7).substring(0,2) + code(5).charAt(0));

      int x = 100;
      x = 1 - (3 % (1 - x));
   }
}
```

assume that the algorithms Doris is using are public, and that the only thing she's able to keep secret are the inputs to these algorithms. But nevertheless, have a look and see what you can find. One fingerprint stands out, of course, the string variable `"fingerprint"`! Not a very clever embedding, one might think, but easy to insert and extract, and if nothing else it could serve as a decoy, drawing Axel's attention away from more subtle marks.

What else? Take a look at the `code` method. What Doris did here was to encode the string `"Bob"` in base 26 as $bob_{26} = 1 \cdot 26^2 + 14 \cdot 16^1 + 1 = 1041_{10}$, using $a = 0$, $b = 1, \ldots, o = 14, \ldots, z = 25$. She then turned 1041 into a permutation of the integers $\langle 0, 1, 2, 3, 4, 5, 6 \rangle$, getting

$$\langle 0 \to 0, 1 \to 6, 2 \to 5, 3 \to 4, 4 \to 2, 5 \to 1, 6 \to 3 \rangle$$

This permutation, in turn, she used to reorder the cases of the switch statement in the `code` method. To extract the mark, she would have to do the process in reverse. First, she would need to find the method into which the mark is embedded (the secret key would point out the `code` method), extract the permutation from the switch statement, turn the permutation into 1041, and finally, decode that as the string `"bob"`. There are many algorithms that, like this one, are based on permuting aspects of a program to embed a mark. The very first published watermarking algorithm [104,263] (a Microsoft patent), for example, permutes the basic blocks of a function's control flow graph. In Section 8.4▶486, we will discuss this further.

Now what about the statement `x=1-(3%(1-x))`? Here, Doris created a translation table from letters to binary operators:

	+	–	*	/	%
1	a	b	c	d	e
2	f	g	h	i	j
3	k	l	m	n	o
4	p	q	r	s	t
5	u	v	w	x	y
6	z				

Thus, the three letters of the string `"bob"` turn into the operand/operator-pairs `1-`, `3%`, `1-`, which when stitched together become `x=1-(3%(1-x))`. This is similar in flavor to an algorithm by Monden [252,263] et al., which we will talk about in Section 8.7.1▶505.

The three marks we've seen so far are all *static*, i.e., they're embedded directly into the code of the program. In our example we've embedded into source code, but we could have used any convenient program representation, including binary code, Java bytecode, or any of the many intermediate code forms used in compilers. We will discuss static algorithms further in Chapter 8.

There is one final mark in the program, however, and this is a *dynamic* fingerprint. What this means is that the fingerprint only appears at runtime, and only for

a particular input to the program. When Doris starts the example program with the secret input key 42, the statement

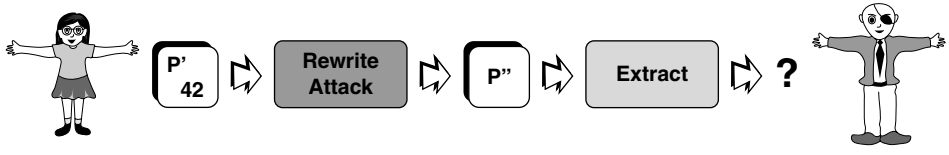
```
new  WMExample().init(code(7).substring(0,2)  +  code(5).charAt(0));
```

executes and displays the embedded fingerprint:

In Chapter 9 (Dynamic Watermarking), we will discuss these types of watermarks. In practice, of course, they are never as obvious as this: It's too easy for Axel to find the code that would pop up a window with a string in it. Rather, the watermark is hidden somewhere in the dynamic state of the program, and this state gets built only for the special, secret input. A debugger or a special-purpose recognizer can then examine the state (registers, the stack, the heap, and so on) to find the fingerprint.

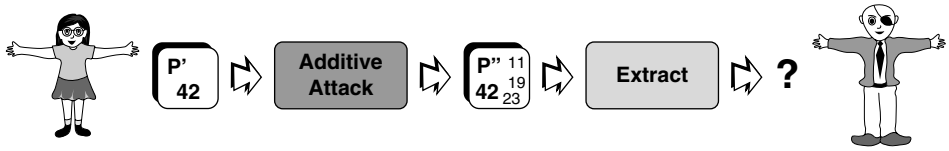
1.6.2 Attacks on Watermarking Systems

As in every security scenario, you need to consider possible attacks against the watermark. Doris has to assume, of course, that Axel will try to destroy her marks before trying to resell the program. And, unfortunately, there's one attack that will *always* succeed, that will always manage to destroy the watermark. Can you think of what it is? To be absolutely sure that the program he's distributing doesn't contain a watermark, well, Axel can just rewrite the program from scratch, sans the mark![4] We call this a *rewrite attack*:

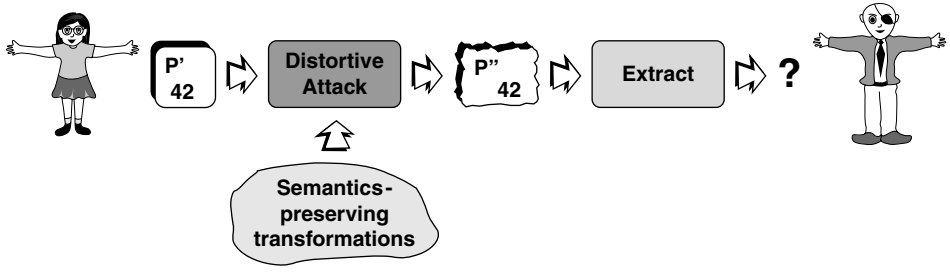


Axel can also add his own watermarks to the program. We call this an *additive attack*:

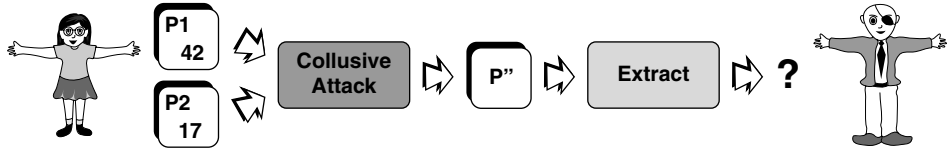
4. Yeah, yeah, so what if it was a trick question?



An additive attack might confuse Doris' recognizer, but more important, it may help Axel to cast doubt in court as to whose watermark is the original one. A *distortive attack* applies semantics-preserving transformations (such as code optimizations, code obfuscations, and so on) to try to disturb Doris' recognizer:



Finally, Axel can launch a *collusive attack* against a fingerprinted program by buying two differently marked copies and comparing them to discover the location of the fingerprint:



To prevent such an attack, Doris should apply a different set of obfuscations to each distributed copy, ensuring that comparing two copies of the same program will yield little information.

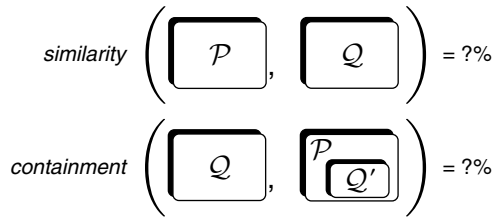
One clever attack that Axel may try to use is not an attack on the watermark itself. Rather, Axel could try to bring into question the validity of Doris' watermark by pretending that the software contains his own watermark. Axel simply writes his own recognizer that "recognizes" this program as containing his mark. If he is successful, we could not tell which was the true recognizer and Doris would not be able to present a legally convincing claim on her own program.

In Chapter 8 and Chapter 9 we will describe many software watermarking algorithms. Some will be useful for watermarking entire applications, others are

good for parts of applications. Some will work for binary code, others are for typed bytecode. Some will embed stealthy marks, some will embed large marks, some will embed marks that are hard to remove, and some will have low overhead. However, we know of no algorithm that satisfies *all* these requirements. This is exactly the challenge facing the software watermarking researcher.

1.7 Software Similarity

There's a class of software protection problems that are not amenable to algorithms based on code transformations, and we lump them together under the term *software similarity analysis*. They have in common that, conceptually, they rely on your being able to determine if two programs are very similar or if one program is (partially) contained in another. We capture this in the two functions *similarity* and *containment*:

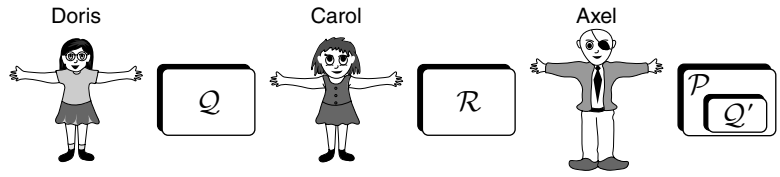


1.7.1 Plagiarism

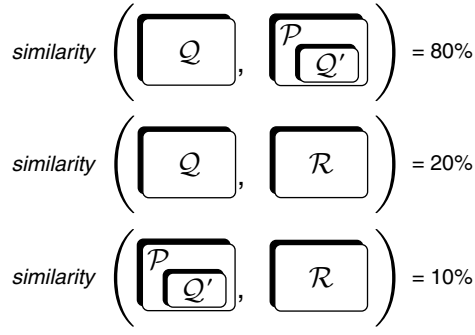
We think of plagiarism as chiefly occurring in academic settings where students copy each other's assignments or researchers copy each other's work, but it's really found anywhere that some humans create and others try making a shortcut to profit by "borrowing" these creations. Ideas from works of art are copied, as are pieces of music, furniture, or fashion designs and so on. Over the years, many famous authors, artists, and musicians have found themselves in court for incorporating too much of a colleague's work into their own. Exactly what "too much" means is ultimately left up to the courts to define. Famous cases include John Fogerty, who was sued for plagiarizing himself (known as *self-plagiarism*) when his new songs sounded too much like his old ones that were under copyright to a previous publisher.

In this book, we're interested in plagiarism of *code*. This occurs frequently in computer science classes where students find it easier and faster to borrow their classmates' assignments than to write them from scratch themselves. Here, Axel

has copied a piece of Doris' program Q, inserted it into his own program P, and submitted it as his own:



With large classes, it becomes impossible for computer science professors to manually look for code copying in every pair of submitted programming assignments. Instead (being programmers and used to automating *everything*), they build tools that perform the comparisons automatically. For the example above, the output might look something like this, listing all the pairs of programs in order, from most likely to least likely to have been enhanced by copying:

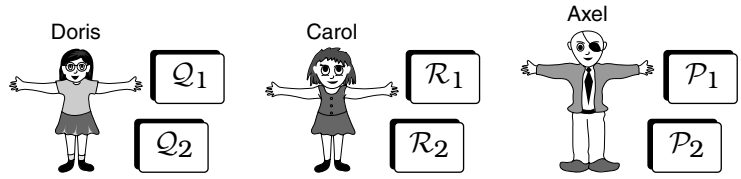


Automatic methods are best used to weed out from consideration programs highly unlikely to be the result of plagiarism, leaving a few serious suspects for the professor to examine by hand.

Students who are aware that instructors are using tools to look for copying will naturally try to fool them. Simple code transformations such as renaming identifiers and reordering functions are common, and it's therefore important that plagiarism detectors are immune to such transforms.

1.7.2 Software Forensics

Software forensics answers the question, "Who, out of a group of suspected programmers, wrote program S?" To answer the question, you need to start out with code samples from all the programmers you think might have written S:



From these samples, you extract features that you believe are likely to uniquely identify each programmer and then compare them to the same features extracted from S. From the example above, the output might look something like this, indicating that Axel is much more likely to have written S than either Doris or Carol:

$$similarity\left(f(Doris), f\left(\boxed{\mathcal{S}}\right)\right) = 20\%$$

$$similarity\left(f(Axel), f\left(\boxed{\mathcal{S}}\right)\right) = 80\%$$

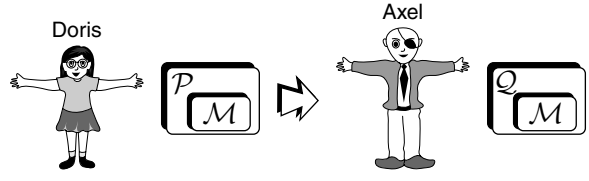
$$similarity\left(f(Carol), f\left(\boxed{\mathcal{S}}\right)\right) = 10\%$$

Here, f is the function that extracts a feature set from a program. Exactly which set of features will best indicate authorship is hotly debated, but the feature sets often include measures related to line length, function size, or the placement of curly braces.

Most work on software forensics has been done on source code, but binary code forensics is just as interesting. For example, once the FBI catches a suspected malware author, they could compare his programming style to those of all known viruses. Being able to bring a large collection of malware to court as evidence against him might strengthen the government's case.

1.7.3 Birthmarking

You've already encountered the idea of code lifting, a competitor copying a module M from your program P into his own program Q:

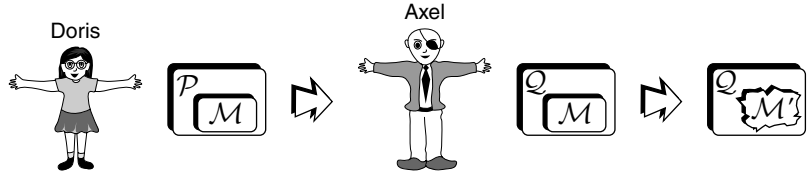


Both obfuscation and watermarking can make this attack harder to mount successfully. By obfuscating P you make it more difficult to find M, or more difficult to extract it cleanly from P. For example, you could mix M with code from other modules so that along with M's code any automatic extraction tool would produce a whole lot of irrelevant code.

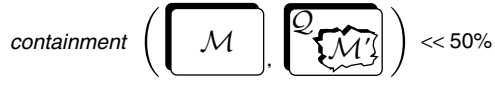
You could also embed a watermark or fingerprint in M. Say, for example, that M is a graphics module produced by a third party (you) that Doris licenses to use in her own game. If Doris' fingerprint shows up in a program sold by Axel, you could use that as evidence that he's lifted it, and even evidence that he's lifted it from Doris' program. You could take legal action against Axel for code theft or against Doris if she's not lived up to her license agreement to properly protect the module from theft.

For a variety of reasons, you may choose not to use obfuscation and watermarking. For example, both come with a performance penalty, and obfuscation can make debugging and quality assurance difficult. Also, you may want to detect theft of legacy code that was never protected against intellectual property attacks. Instead of using either one, you could simply search for your module M inside the attacker's program Q:

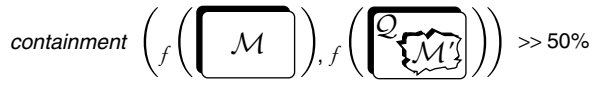
This works fine, unless the adversary has anticipated this and applied some code transformations, such as obfuscation, to M (or possibly all of Q) to make it hard to find M:



Depending on the severity of the code transformations, this could make a straightforward search for M difficult:



This is where the concept of *birthmark* comes in. The idea is to extract "signals" from Q and from M, and then look for M's signal within Q's signal rather than looking for M directly within Q:



Here, f is a function that extracts the signal, which we call a birthmark, from a program or module. The idea is to select the birthmark so that it's invariant under common code transformations such as obfuscation and optimization.

We know of at least one case where birthmarking was successfully used to argue code theft. In a court case in the early 1980s [128], IBM sued a rival for theft of their PC-AT ROM. They argued that the defendant's programmers pushed and popped registers in the same order as in the original code, which was essentially a birthmark. They also argued that it would be highly unlikely for two programs to both say `push` R_1; `push` R_2; `add` when `push` R_2; `push` R_1; `add` is semantically equivalent.

1.7.4 A Birthmarking Example

To be effective, a birthmarking algorithm must extract the mark from a language feature that is hard for an attacker to alter. One idea that has been reinvented several times and that we'll explore further in Chapter 10 (Software Similarity Analysis) is to compute the birthmark from the calls the program makes to standard library functions or system calls. Some of these functions are difficult for the adversary to replace with his own functions. For example, the only way to write to a file on Unix is to issue the `write` system call. A birthmark extracted from the way the program uses `write` system calls should therefore be reasonably robust against semantics-preserving transformations.

Consider this C function that reads two strings from a file, converts them to integers, and returns their product:

```
int x() {
  char str[100];
  FILE *fp = fopen("myfile", "rb");
  fscanf(fp,"%s",str);
  int v1 = atoi(str);
  fscanf(fp,"%s",str);
  int v2 = atoi(str);
  fclose(fp);
  return v1*v2;
}
```

Several birthmark-extracting functions are possible. You could, for example, make the birthmark be the sequence of calls to standard library functions:

$$bm_1(\texttt{x}) = \langle \texttt{fopen}, \texttt{fscanf}, \texttt{atoi}, \texttt{fscanf}, \texttt{atoi}, \texttt{fclose} \rangle$$

Or you could ignore the actual sequence, since some calls are independent and could easily be reordered. The resulting birthmark is now a set of the calls the function makes:

$$bm_2(\texttt{x}) = \{\texttt{atoi}, \texttt{fclose}, \texttt{fopen}, \texttt{fscanf}\}$$

Or you could take into account the number of times the function calls each library function. The birthmark becomes a set of system calls and their frequency:

$$bm_3(\texttt{x}) = \{\texttt{atoi} \mapsto 2, \texttt{fclose} \mapsto 1, \texttt{fopen} \mapsto 1, \texttt{fscanf} \mapsto 2\}$$

An attacker would get a copy of `x`, include it in his own program, `P`, and perform a variety of transformations to confuse our birthmark extractor. Here, he's renamed the function, added calls to `gettimeofday` and `getpagesize` (functions he knows have no dangerous side effects), reordered the calls to `fclose` and `atoi`, and added further bogus calls to `fopen` and `fclose`:

```
void y() {...}

int f() {
  FILE *fp = fopen("myfile", "rb");
  char str[100];
  struct timeval tv;
  gettimeofday(&tv, NULL);
  fscanf(fp,"%s",str);
  int v1 = atoi(str);
  fscanf(fp,"%s",str);
  fclose(fp);
  int v2 = atoi(str);
  int p = getpagesize() * getpagesize();
  fp = fopen("myfile", "rb");
  fclose(fp);
  return v1*v2;
}

void z() {...}
```

Bogus calls are shaded in dark gray. Assuming that the rest of P (functions y and z) don't contain any standard library calls, you get these possible birthmarks for P:

$$bm_1(\texttt{P}) = \langle \texttt{fopen}, \texttt{gettimeofday}, \texttt{fscanf}, \texttt{atoi}, \texttt{fscanf}, \texttt{fclose}, \texttt{atoi}, \texttt{getpagesize}, \texttt{getpagesize} \rangle$$

$$bm_2(\texttt{P}) = \{\texttt{atoi}, \texttt{fclose}, \texttt{fopen}, \texttt{fscanf}, \texttt{getpagesize}, \texttt{gettimeofday}\}$$

$$bm_3(\texttt{P}) = \{\texttt{atoi} \mapsto 2, \texttt{fclose} \mapsto 2, \texttt{fopen} \mapsto 2, \texttt{fscanf} \mapsto 2, \texttt{getpagesize} \mapsto 2, \texttt{gettimeofday} \mapsto 1\}$$

To determine whether the attacker has included your function x in his program P, you compute $containment(bm_i(\texttt{x}), bm_i(\texttt{P}))$, where *containment* returns a value between 0.0 and 1.0 representing the fraction of x that's contained in P. In this case, it would seem like the attacker has done a pretty good job of covering his tracks and altering the sequence of calls, the set of functions being called, and the frequency of calls to the different functions.

1.8 Hardware-Based Protection Techniques

What makes it so difficult to design unassailable software protection techniques is that you never have any solid ground to stand on. The difference between software security and cryptography is that in cryptography you're allowed to assume that there is one secret (the key) that your adversary will never be able to get his hands on. All security rises and falls with that assumption. In software protection, you can assume no such safe zone. The code of your obfuscated, watermarked, and tamperproofed program will always be available to the attacker for analysis because the attacker (who may also be your customer!) needs the code in order to run it.

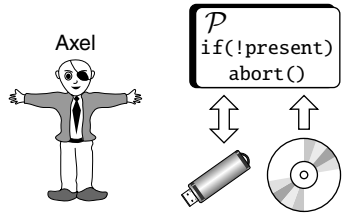
Hardware-based protection techniques try to change that by providing a safe haven for data, code, and/or execution. The applications are the same as you've already seen: to protect the code from reverse engineering, to prevent the code from being tampered with, and to prevent illegal copying of the code. The difference is that now you've got one piece of hardware whose integrity you can trust and on which you can build protection schemes.

1.8.1 Distribution with Physical Token

The root cause of software piracy is that digital objects are trivial to clone. Cloning is not unheard of in the physical world, of course: To see that, just visit one of the many clothing and antique markets in China where near-identical copies of name-brand

clothing lines and replicas of ancient artifacts are sold at cut-rate prices. Still, physical cloning requires considerable skill and sophisticated tools, whereas cloning in the virtual world only requires the attacker to learn how to use his computer's `copy` command.

Several software anti-piracy techniques are based on the idea that a program will refuse to run unless the user first shows possession of a physical object. Thus, you ship your program in two parts: the easily clonable binary code itself and a hard-to-clone physical object that's required for the binary to execute. The physical object needs to somehow be connected to the computer so that the program can occasionally check to see that the user actually has possession of it:



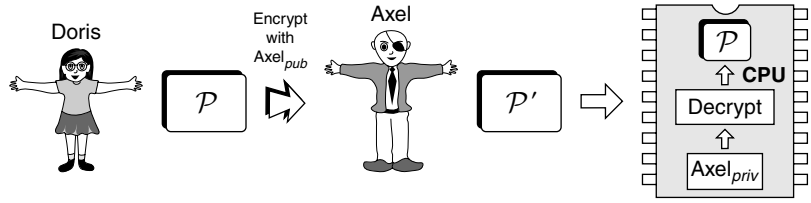
For the physical object to be effective, you need to manufacture it in such a way that it's difficult to clone without specialized and expensive tools. Two types of objects have been common: *dongles* and *program distribution disks* (floppy disks, CDs, and DVDs). Dongles are connected to a port on the computer, these days mostly the USB port. The CD containing the application must be inserted into the computer's CD drive in order for it to run. Often, the CD is manufactured so that ordinary copying software and hardware won't be able to make an identical copy.

This anti-piracy technique has fallen out of favor for all but the most expensive programs. There are many reasons. First, the technique has proven to be highly annoying to legitimate users who can't make backup copies of their legally purchased program, who lose the use of a port or a CD drive, and who can no longer use the program if they misplace the physical object. Second, it's also annoying to the manufacturer who loses revenue from making and distributing the CD or dongle, who needs to deal with users who misplace it, and who can no longer simply distribute the program over the Internet.

1.8.2 Tying the Program to the CPU

If every manufactured copy of a CPU has a unique identity, you can solve the piracy problem by distributing your program so that each copy is only executable by a

CPU with a particular identity. In practice, this is typically done by including in each CPU the private part of a unique public key cryptography key-pair along with a decryption unit. Here, Doris encrypts her program with Axel's public key and distributes the encrypted program to him:



As a result, even if Axel were to resell the program to his friends, their CPUs will have different private keys and won't be able to run the program.

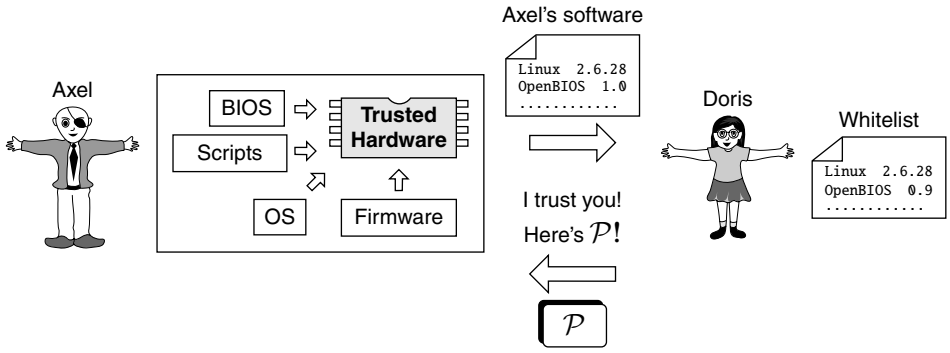
This scheme isn't without its own problems. First, manufacturing the CPU becomes more complicated since every copy will have to contain a different key. One solution is to squirt the key into the CPU after manufacturing and then disable any further modifications. Second, selling shrink-wrapped programs becomes difficult since every distributed copy needs to be differently encrypted. This may not be much of a problem these days, when many programs are sold and downloaded over the Internet. Finally, what happens when the user upgrades to a new computer with a faster CPU? Since it will have a different private key, he will need new versions of all the encrypted programs he ever bought. This means there must be some way for him to convince the program manufacturer that he's no longer using the old CPU and that they should issue him a version to run on his new one.

1.8.3 Ensuring Safe Execution Environment

In the scenarios we consider in this book, Doris assumes that Axel's computer, on which her program is running, is an *unsafe environment*; it could contain hostile programs (such as debuggers, binary editors, emulators, and disk copiers) that Axel can use to pirate, tamper with, or reverse engineer her program.

There has been considerable effort to build protection systems that would instead assure Doris that Axel's computer *is* trustworthy, and that she can safely allow him to run her program without fear of it being violated. This is difficult to do using only software: The code on Axel's computer that checks to see if he's running any unsafe programs could itself be hacked! The idea instead is to have one small trusted hardware unit on Axel's computer that helps Doris collect a reliable list of

software that he's running:



The protocol starts with Axel asking Doris for a copy of her program P to run, to which Doris responds, "Not so fast, prove to me that I can trust you first!" Axel's computer (with the help of the trusted hardware) then collects a list of all its security-sensitive software and firmware and sends it to Doris. She compares it to a list of software she trusts, and only when she's convinced herself that Axel's computer constitutes a safe environment for P does she send it to him.

We've omitted many details here that you'll learn about in Chapter 11. But it shouldn't be hard for you to spot right now some fundamental issues with this scheme. First, any user who wants to run Doris' program has to have a computer that's been fitted with extra hardware, and obviously this will come at a cost. Second, Doris has to maintain a whitelist of all versions of all programs that she trusts, or a blacklist of all versions of all programs that she *doesn't* trust. Given the vast number of programs available to run on many different hardware platforms and operating system configurations, this is likely to be a huge logistic nightmare. Finally, an actual protocol needs to take into account any privacy issues. For example, a user may not want to reveal to every site he's communicating with which software he's running.

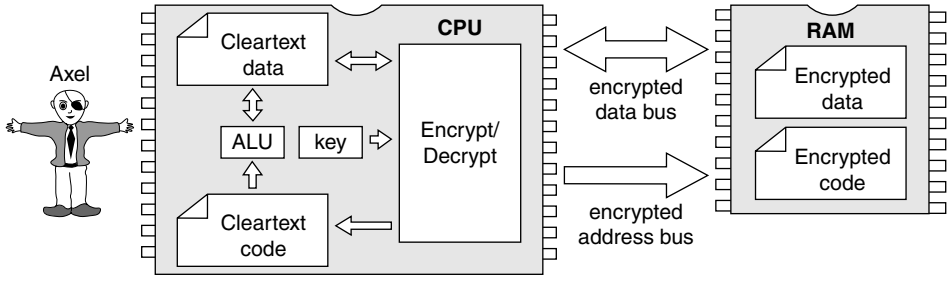
An additional issue that makes many people queasy is that this protocol would be very useful in a digital rights management scenario. P would be the DRM player and the protocol would assure Doris that Axel won't be able crack it to get to the decrypted media. This, in turn, would give Doris the confidence to allow Axel to download encrypted media to play.

1.8.4 Encrypted Execution

The ultimate solution to the software protection problem is to encrypt the program. As long as the program remains encrypted, the adversary can't easily tamper with it,

and can't learn anything about its internal algorithms, and pirating it makes no sense since it's not executable. However, eventually you'll want your user (and potential adversary) to run the program. This means they will need access to the decryption key, which will give them access to your program in cleartext, which means they can do with it what they want. Game over.

For cryptography to provide unassailable protection, the program must remain encrypted throughout its lifetime, until it's safely within the CPU itself. The CPU contains the key with which the program is encrypted. The key must never escape the CPU capsule or the adversary will gain access to the cleartext program. Schematically, crypto-processors are organized roughly like this:



The program (and any sensitive data) is stored encrypted in RAM but decrypted on the CPU itself. An encryption/decryption unit sits between the RAM and any on-chip caches and decrypts incoming code and data and encrypts any outgoing data.

As you will see in Chapter 11, to get this to work without revealing any information, *everything* must be protected once it leaves the safe haven of the CPU. The sequence of addresses that appear on the address bus can reveal information unless they're properly scrambled, the sequence of encrypted data values that appear on the data bus can reveal information if they don't change on every store, and the adversary might even get away with tampering with the encrypted program unless you check on every read that nothing has changed.

All this extra machinery obviously comes at a price, namely, performance. For example, modern CPUs are designed to take advantage of the *locality of reference* that programs typically display. If addresses are scrambled to hide access patterns, then the advantages of caches and locality-improving code optimizations are nullified. There's also the question of how you convince your ordinary users that they should "upgrade" to a slower crypto-processor only for the benefit of protecting *your* program from piracy and tampering by *themselves*. Even if it's unlikely that

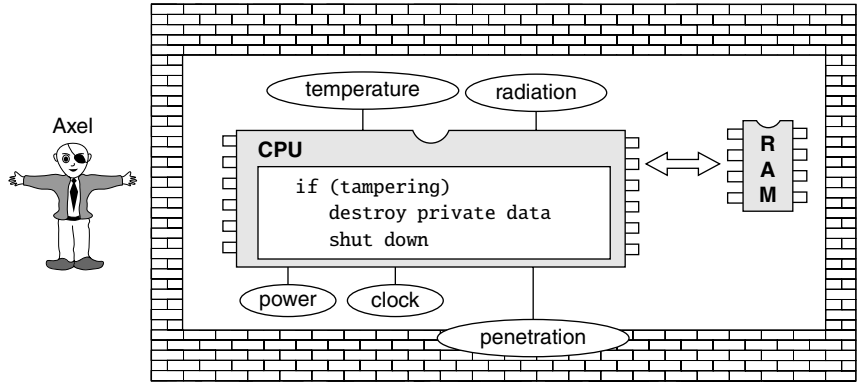
they'll make it into commodity PCs any time soon, crypto-processors have an important role to fill in financial systems such as ATMs.

1.8.5 Physical Barriers

Any software protection system that makes use of a crypto-processor assumes that the secret key hidden within the CPU, in fact, remains hidden. An adversary, of course, will try to violate that assumption! There have been many attacks devised against smartcards, for example, since they are used in pay-TV systems, as stored-value cards, and as tickets on mass-transit systems, and thus breaking their security can provide financial gains to the attacker.

Attacks against crypto-processors are either *invasive* or *non-invasive*. Invasive attacks, essentially, scrape off the top of the packaging to give direct physical access to the internal circuitry. This can then be probed to reveal secret data (such as keys) and algorithms encoded in hardware. A non-invasive attack doesn't physically abuse the CPU but rather feeds it bogus code or data, or forces it to operate under adverse environmental conditions, all with the intention of coaxing it to give up its secrets. For example, popular non-invasive attacks include delivering power spikes or irregular clock signals to the processor, or subjecting it to radiation. This can make it execute in a way its designers didn't intend and, possibly, force it to reveal secret code and data.

Crypto-processors include physical layers of protection to prevent these kinds of attacks:



For example, sensors will alert the CPU if the temperature, voltage, clock signal, or radiation level appears abnormal. If the CPU believes it's under attack, it will destroy any private data (such as keys), shut down, or even self-destruct. To prevent

invasive attacks, the processor will have layers of shielding that make it difficult to drill down to the internal circuitry. These shields can also have sensors that can alert the CPU that it's being probed.

Unfortunately, adding physical barriers will affect cost as well as performance. The higher the clock frequency, the more heat the CPU dissipates, and the thicker the layers of protection, the harder it will be for the processor to get rid of this excess heat! As a result, processors with thick physical protection barriers will have to be clocked at a lower frequency than equivalent unprotected processors.

1.9 Discussion

There are plenty of reasons to use the techniques you will read about in this book to protect your code, and plenty of reasons not to. In the end, you must weigh the pros and cons to see if using them makes sense in your particular case.

1.9.1 Reasons to Use Software Protection . . .

On the plus side, there are situations where even a short delay in cracking your code will be beneficial. For example, these days computer games often escape into the wild within days or a few weeks of release. Sometimes, as the result of insider attacks, they even escape *before* they're released! Since gamers are interested in playing the latest and greatest, the revenue from a new game is typically collected within the first few weeks of its release. Game manufacturers have therefore traditionally been eager to adopt software protection techniques, even those that are sure to be cracked within a very short time.

In situations where you're selling a particularly high-value program, software protection might also make sense. If every year you sell only a few tens of copies of a software system that costs $100,000 per seat, the loss of even one sale due to piracy could be devastating to your bottom line. In these situations, it's common to use a combination of hardware techniques (such as dongles) to prevent illegal copying and software techniques (such as obfuscation) to protect the interface between the hardware and the application.

In some situations, you may want to use software protection to prevent individuals from cracking your program, individuals you are unlikely to catch (because they're far off in another country) or who are legally untouchable (because they have no assets to forfeit). In other cases, you may want to use software protection as an aid in criminal prosecution. It is said that more money is lost to piracy performed by corporations than by individuals. The reason is that corporations often

will buy, say, 10 licenses to use a program but will let 50 of their employees use it. These corporations have huge assets, and if you can prove license infringement, you may actually have a chance to use the legal system to force them to abide by the license agreement. Individuals, on the other hand, often crack a program or download a pirated program that they would never have bought legally anyway. The amount of revenue lost to this type of piracy may therefore not be as high as often reported, and going after these types of adversaries is less likely to be profitable.

1.9.2 . . . and Reasons Not To

On the downside, adding software protection to your program can cause problems in terms of cost, performance, a more complex software development cycle, and last but not least, annoyance to your legitimate users.

Costs can increase both during development and distribution. During development, you have the option of purchasing a protection tool, developing one yourself, or applying the protection techniques by hand. Regardless, you will be adding a step and complexity to the development cycle. In several techniques you'll see in this book (such as fingerprinting and code encryption), every distributed copy of your program will be unique. This means additional headaches during distribution, quality assurance, and when fielding bug reports.

Many of the techniques you'll see come with significant performance overhead. And when we say "significant" we mean *significant*. Whether higher levels of protection necessarily incur a more severe performance hit is unknown, but it seems not an unreasonable possibility. In the end, this may require you to make difficult trade-offs: using techniques with low overhead and a low level of protection, using powerful techniques with high overhead but only for the security-sensitive parts of your program, or accepting the high overhead of a powerful protection technique but only selling to users with sufficiently powerful computers.

Many software protection techniques have been cracked not because the attacker necessarily wanted to pirate or reverse engineer the program, but because he wanted to perform what to him seemed like reasonable operations on it. Maybe he wanted to run the program on his laptop as well as his desktop computer, maybe he wanted to make a backup copy of it, maybe he wanted to transfer it from his old machine to his newly bought one, maybe he wanted to run the program without the annoyance of having to enter a password or insert a CD or USB token every time, and so on. Whether or not you, as the software developer, think these are legitimate concerns, and whether or not the software license agreement allows it, the user may well get ticked off enough to feel no qualms about cracking the program.

And that's not to mention bugs. Many of the techniques you will see in this book are based on code transformations. As such, they're very similar to the optimizing code transformations you'll find in sophisticated compilers. And as such, they're susceptible to the usual menagerie of bugs. As a result, your program that was perfectly functional before protection was added may exhibit erratic behavior afterward, either as a result of a bug in the protection tool itself, or because its code transformations revealed a latent bug in your own code.

1.9.3 So Which Algorithms Should I Use?

So you've written your fantastic new program and for whatever reasons you've decided that it makes sense to add some sort of software protection to it. You've bought this book[5] and read it cover to cover in search of the One True Algorithm to implement. Unfortunately, if this was your attitude, you will have been sorely disappointed.

The problems facing software protection research are multitudinous, but it boils down to one central issue: How do you evaluate the effectiveness of an algorithm? Unfortunately, the state of the art is sorely lacking. Ideally, this book would end with a single large table over all the algorithms that lists for each one *effort to implement*, *effort to defeat*, *parallelizability of attacks*, and *performance overhead*.[6] As you may have already guessed, there's no such table.

Without knowing how many more hours/days/weeks/months your program will remain uncracked if you protect it with the algorithms in this book, how can you know if software protection will make sense in your particular case? And without being able to compare two algorithms to say that one is better than another, how can you pick the right algorithm to use? And how does the field progress when a paper purporting to present a new, improved algorithm can't substantiate its claims?

Disturbing questions, indeed. In practice, software protection takes a belt-and-suspenders approach to security. You layer protection algorithms until you've convinced yourself that your program is secure "enough" while at the same time remaining within performance bounds. If you can afford to, maybe you employ a professional red-team to try to break through the defenses and give you some feel for how long the protection will survive in the wild. But in the end, you have to accept the fact that you're engaged in a cat-and-mouse game: If crackers deem your program sufficiently crack-worthy, eventually they'll succeed. This situation may

5. Thanks!
6. Effort is measured in person wall-clock hours. A parallelizable attack can be sped up by simply throwing more crackers at the problem.

seem depressing, but it is not much different from other areas of life. In the end, all it means is that you will have to monitor the efforts of your adversaries and be prepared to continuously update your defenses.

1.10 Notation

In this book we have tried to devise a uniform naming convention for surreptitious software algorithms. Every algorithm is referred to by a name that consists of a prefix (WM for watermarking, OBF for obfuscation, TP for tamperproofing, and SS for software similarity) followed by the authors' surname initials. For algorithms that *attack* programs we use the prefix RE (for reverse engineering). For a single author algorithm, we use the initial of the surname followed by the initial of the given name. When a list of authors have multiple algorithms, we add a subscript. If two different lists of authors share the same initials, we add given name initials and initials from the article title until names are disambiguated.

To facilitate navigating through the book, we've added page numbers to all cross-references, using the following notation: "In Section 3.2.3▸163 you will see the totally awesome Algorithm 3.1▸165 . . . ."

Index

P

X