Preventing Piracy, Reverse Engineering, and Tampering

With the advent of networked appliances, mobile code, and pervasive access to the Internet, software protection has gained increasing importance. The authors survey current and promising new techniques designed to reliably preserve and protect software data vital to our privacy and security.

Gleb Naumovich
Nasir Memon
Polytechnic University

Personal privacy, national security, and other fundamental values hinge on the ability to protect data from unauthorized access. As computing becomes pervasive, concerns about data protection have taken on new urgency. For example, obtaining unauthorized access to someone's medical history once would have required physically breaking into a doctor's office, searching through one or more filing cabinets, and extracting the patient's folder. Today, it is often possible to obtain such records by breaking into the doctor's computer from a remote location.

What makes securing digital data difficult is that it is rarely static—rather, data is manipulated by software, often in a networked environment. For example, parts of a hospital's medical database may be accessible to queries by the police department. In many cases, to steal or manipulate data an attacker need not take over the host computer but instead only has to defeat the software programs responsible for protecting the data.

Software itself is a form of data and as such is vulnerable to theft and misuse. Given the enormous investment of time, money, and intellectual capital in software development, piracy has long been—and continues to be—a major threat to the software industry. In its most recent study, which tracked 26 popular business applications in 85 countries, the Business Software Alliance (www.bsa.org) reported that the global economic impact of pirated software totaled nearly US$11 billion in 2001.[1]

The problem, however, extends well beyond that of software piracy. Software is increasingly being distributed as mobile code in architecture-independent formats. Most such formats are essentially equivalent to source code, which makes the software susceptible to decompilation and reverse engineering. Malicious parties can steal the intellectual property associated with such code with relative ease.

Clearly, there is a strong need for developing more efficient and effective mechanisms to protect software. Unfortunately, none of the major approaches currently used by software developers and vendors provide adequate protection, especially on today's open computing platforms. However, three promising techniques under development—tamper proofing, obfuscation, and watermarking—offer hope for the future.

PIRACY PREVENTION

As Figure 1 shows, from a software vendor's point of view, a program executes in either a trusted or untrusted environment. We use the term "program" for simplicity—software protection mechanisms can and should be applied not only to complete programs, but also to program components, such as reusable class libraries. The typical end user's computer is inherently untrustworthy because anyone with access to the machine, either direct or remote, can potentially tamper with the software running on it.

To help prevent illegal distribution of software by pirates, the developer places some form of protection on the program. A pirate seeks to remove this protection—more precisely, to find and remove the code that implements protection, while leaving intact the code that performs the core functionality.

Published by the IEEE Computer Society
0018-9162/03/$17.00 © 2003 IEEE

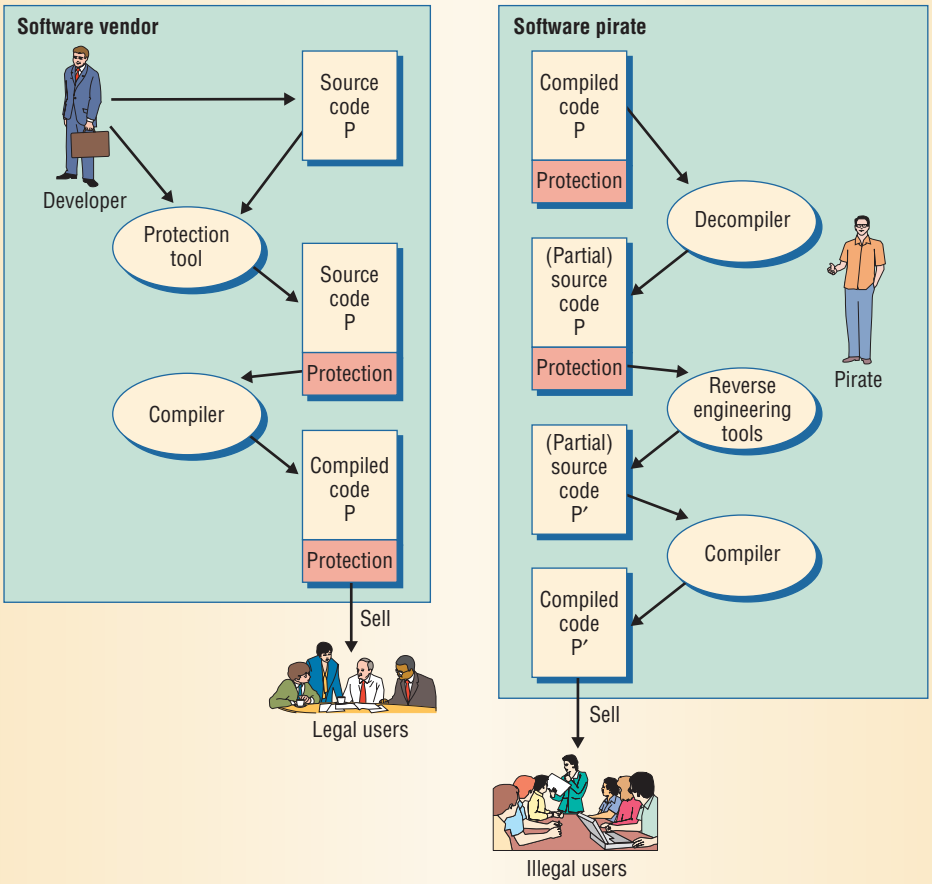

Figure 1. Software protection. To prevent illegal distribution of software by pirates, the developer places some form of protection on the program. A pirate seeks to remove this protection.

Consider, for example, home appliances that are increasingly being networked to one another and the Internet. Personal video recorders, set-top boxes, home surveillance systems, game consoles, Web picture frames, and other such devices contain software that enables them to function intelligently and also governs their usage policy. Tampering with this software may result in serious security problems, including piracy of content. A malicious user could tamper with the underlying software of, say, a set-top box to download a movie and view it without paying for it, watch it multiple times, or, even worse, make a perfect digital copy.

Tampering poses a more serious threat than just digital content piracy. Access to a networked appliance by a malicious party could lead to a loss of privacy on the owner's part, including a loss of digital identity. Although a vendor may provide a connected appliance with reliable, trusted, and secure software, that appliance is not trustworthy after installation in a user's home because it can be tampered with physically or via the network.

Application server

A radical solution to the problem of piracy does away with the concept of a software program as a stand-alone product. In the popular application server model, a server, presumably owned by a trusted entity or the software vendor, performs most computations, and clients receive dynamically generated content—for example, as HTML and XML documents. Although executable code may be embedded in Web documents as scripts, normally these scripts are simply used to collect input from clients and therefore do not represent significant intellectual property or a security risk.

Although widely used for Web content delivery, the application server model presents inherent scalability and performance problems. Users may not accept it in all cases, especially for desktop applications. In addition, the approach's success depends on maintaining server impenetrability; any security breach can therefore result in theft of the software as well as users' identities and data.

License files

The software industry has expended much effort—as revealed in patent literature—to develop mechanisms for protecting software from misuse and theft. Most of this work has concentrated on piracy prevention. License files or identification keys embedded in software programs are a common protection mechanism in commercial applications. The program contains code that checks for the presence of a license file and whether specific privileged information, or "the key," is present in

A tamper-proofing mechanism that detects any modification will disable some or all of the program functionality.

this file. In addition, the license file identifies the user for whom the file was issued. This information can be used in a legal action against any user who distributes copies of the software illegally. Attacks on this protection scheme involve reverse engineering of the program to enable detecting and either removing or bypassing key-checking code.

Hardware-based solutions

More elaborate piracy control techniques rely on trusted hardware components on end users' machines. A long-used scheme prevents programs from running unless a specific floppy, CD, dongle, or secure digital memory card is present. To enable partial or full functionality, the program must retrieve information stored on the hardware component that validates use of the program on the machine. However, this approach, which is technically similar to using license files, provides little protection because an attacker only has to disable the part of the software that retrieves and checks information from the hardware component.

Adding a hardware component can provide stronger protection if, along with validating information, it supplies some information necessary for normal functioning of the program—for example, code for part of the program. Nevertheless, such hardware-assisted protection schemes can be attacked. For example, a removable disk can be easily copied, and the costs of hardware duplication have fallen rapidly in recent years. Even if a hardware component cannot be tampered with or duplicated directly, an attacker can intercept and analyze information from the component while the program uses it.

In addition, all hardware-based protection mechanisms suffer the general drawback of lacking user-friendliness, which relegates the use of such mechanisms to protecting high-end, low-volume software.

TAMPERPROOFING

Protection schemes implemented in client software are designed to merely hinder reverse engineering. In contrast, *tamperproofing* seeks to prevent modification of software code. Unwanted modifications include both manual code changes by end users and automated changes by a virus that, for example, attaches itself to the program and activates every time the program runs. A tamper-proofing mechanism that detects any such modification will disable some or all of the program functionality. Tamperproofing can be either built into the program or external to it—for example, provided as a service by operating systems. A realistic approach would exploit several of the following techniques to provide defense in depth.

Checksums

A straightforward tamperproofing technique is to examine the software program right before it runs and compare it to the original program in some way, usually using checksums. However, it is difficult to conceal the nature of checks, and, once they are detected, an attacker can easily remove or patch around them.

Guards

Hoi Chang and Mikhail Atallah[2] argue that tamperproofing techniques should not rely on a single module whose only purpose is to tamperproof the entire program. Instead, a number of small program units, called *guards,* should work together in a network to protect one another. Even if an attacker identifies a number of guards and removes or patches around them, the remaining guards will detect this tampering and prevent the program from running. Only by removing all guards can an attacker gain the program's full functionality.

Assertion checking

Another category of tamperproofing techniques, related to assertion checking for software quality assurance, is based on checking intermediate program results. For example, if program variable i must be positive at a specific point in the program but has been assigned the value –3 prior to this point, the program fails.

This approach has several drawbacks. First, an unexpected variable value can result from a program bug, not tampering. Programs can recover from many bugs—for example, by displaying an error message or disabling the operation that exhibits the error—but this type of tamperproofing technique would terminate the program instead, thereby decreasing fault tolerance and program usability. Second, a large number of intermediate checks can slow down the program. Third, application of this technique can be labor intensive because it is hard to automate: In the absence of formal specification, a programmer would need to review the program and decide what constitutes intermediate results and what their legal values are. Finally, there is no guarantee that, after tampering, the program will always produce invalid intermediate results.

Software aging

One intriguing defense against software piracy, based on the concept of software aging,[3] relies on periodic program updates. Updates offer bug fixes and new features to users as well as keep programs in sync with other software on which they depend. By providing frequent updates that lessen the utility of older versions of software, vendors could force those who illegally resell a program to provide their own updates, thereby taxing pirates' resources and increasing the likelihood of detection.

However, this method is useful only for document-centric applications such as Microsoft Word that rely on data being in specific formats. The software aging technique modifies document formats from one version to the next. Cryptographic techniques ensure that later versions of the software can use documents created by older versions, but not vice versa. Frequent updates should not inconvenience legitimate users of the program. Markus Jakobsson and Michael Reiter argue that adding functionality to the program that automatically downloads and installs updates can overcome this difficulty.[3]

Cryptographic techniques

The surest way to prevent someone from modifying software is to prevent that person from seeing the code. Cryptography is commonly used to protect data from prying eyes, and some researchers have suggested that it could also be used to encrypt code for a software program. Although intuitively appealing, this approach must overcome several technical obstacles:

- distributing cryptographic keys can present problems unless a trusted infrastructure is in place;
- the runtime system must decrypt the code before running the program, which can adversely affect performance unless dedicated coprocessors perform the decryption; and
- most importantly, any tamperproof architecture based on cryptography must not let an attacker intercept decrypted code—for example, using a debugger—and save it in an unencrypted form.

A number of recently proposed schemes rely on hardware protection. For example, David Lie and colleagues[4] advocate adding a feature called Execute Only Memory, which contains code that the machine's users can execute but not view. Cryptographic techniques ensure that the software program does not decrypt and execute such code until it is loaded in the XOM.

David Aucsmith[5] has suggested a software-based approach to prevent interception of decrypted code that breaks a program into parts and encrypts each one separately. When the program executes, it jumps to decrypted parts according to a sequence of pseudorandom values generated from a cryptographic key. At any time, only one part of the program is decrypted; upon execution of this part, it is encrypted again. Because the program's state always depends on the majority of its preceding states, any tampering with one part will likely be detected by a subsequently executed part. Unfortunately, this technique does not lend itself well to type-safe languages such as Java.

Stanley Chow and colleagues[6] have implemented a range of techniques that protect cryptographic keys hidden in software code from detection, even by sophisticated attacks using automated program analysis tools. The authors demonstrate how their mechanisms make it difficult to obtain a hidden Data Encryption Standard key even when the attacker can determine that the program is using a DES cipher.

The practical goal of obfuscation is to make reverse engineering uneconomical.

OBFUSCATION

In many cases, it is desirable to prevent end users from understanding a program's design and code. Software is a valuable form of data, representing significant intellectual property, and reverse engineering of software code by competitors may reveal important technological secrets. *Obfuscation* refers to the practive of disguising other software-based protection mechanisms. For example, an attacker should not be able to distinguish the program code that performs tamperproofing operations from other parts of the code through either manual inspection or tool-assisted analyses.

Completely obfuscating software code that physically runs on an end-user system and is not protected by hardware-assisted schemes is impossible.[7] Given enough time and resources, a determined attacker can reverse engineer even machine-level code. The practical goal of obfuscation is therefore to make reverse engineering uneconomical. It is sufficient that the program code be difficult to understand, requiring more effort from the attacker than writing a program with the same functionality from scratch. Although obfuscation results in less efficient code, it is relatively cheap to perform and has aroused increased interest in the past two years.

Software watermarking provides a mechanism software authors can use to prove their ownership of intellectual property in the event someone steals that property.

Types of obfuscation

Christian Collberg and Clark Thomborson[8] classify obfuscating transformations according to program information.

Lexical obfuscation. This involves renaming program identifiers to avoid giving away clues to their meaning. While useful, lexical obfuscations are not alone sufficient because a determined attacker can infer the meaning of program identifiers from the context.

Control obfuscation. This type of transformation makes it difficult to understand control flow in individual program functions. One example, *opaque predicates*, uses conditional statements whose predicates always evaluate to true or false. The branch of such a conditional that is always taken will contain meaningful code, while the other branch will contain arbitrary code.

Control obfuscations must be resistant to automated analyses, which are present in many compilers. If a compiler determines statically that a predicate always evaluates to true, it will remove the conditional as well as the code on the branch that will never execute. An approach by Chenxi Wang and colleagues[9] modifies jump instructions so that the target of each jump instruction is unknown at compile time.

Stanley Chow and colleagues[10] suggest a similar approach in which a special *dispatcher* code component determines targets of jump instructions at runtime. This component is modeled as a finite-state automaton whose transition function is deliberately complicated by embedding into it a hard combinatorial problem.

Data obfuscation. This approach obscures the purpose of program data fields. For example, it is possible to replace an integer variable in a program with two integer variables in such a way that the original variable's value at any point can be determined by adding the two new variables' values.

More recently, we have proposed more potent data-type techniques that aim to obscure high-level designs of object-oriented programs by breaking the abstractions of classes. One technique merges arbitrary classes into a single class, while another splits an arbitrary class into two classes related by inheritance. A third technique—specific to Java, with its concept of lightweight data structures—substitutes numerous interfaces for an arbitrary class, whenever possible, to obscure its purpose in the program design.

Experiments with our automated design tool indicate that runtime overhead of these obfuscations is not excessive. In fact, merging a small number of Java classes even somewhat reduced a test program's running times in a number of our experiments. This optimization is likely an artifact of Java's dynamic-class loading mechanism: Java runtime systems only locate and read bytecodes for a class if it is actually used during program execution. Therefore, loading one large class appears to take less time than loading two smaller classes.

Layout obfuscation. This involves obscuring the logic inherent in breaking a program into procedures. One approach is to in-line code in a procedure in all places from which the procedure is called. Optimizing compilers routinely in-line small, frequently called procedures, thereby providing a measure of obfuscation. Similarly, an arbitrary part of code can be out-lined to become a separate procedure.

Tool-assisted attacks

Software obfuscation must be resistant to attacks using debuggers, decompilers, and other automated tools as well as manual methods. Several low-level obfuscation techniques have been proposed to foil attempts at decompilation of class files by the popular Java decompilers Mocha and Jad. Similarly, obfuscations attempt to confuse debuggers that may be used in the reverse-engineering process. One well-known antidebugging feature involves making important operations dependent on a specific pattern of interactions among threads in a multithreaded program. In such cases, using an intrusive tool such as a debugger likely will change the pattern of interactions and disrupt program operations.

An attacker can also use various program analysis tools, including points-to or alias analysis, dependency analysis, and program slicing. Unlike debuggers, these techniques analyze code statically, instead of at runtime. Adding highly dynamic behaviors to the program—for example, using the reflection mechanism in Java or pointer arithmetic in C++—decreases static analysis precision and may render such analyses ineffective.

WATERMARKING

It is not always possible to prevent attackers from illegally reverse engineering programs. A mechanism is therefore necessary to let software authors prove their ownership of intellectual property in the event someone steals that property. *Watermarking* is widely used to authenticate physical objects, including digital media and hardware designs, and could likewise be used to embed ownership marks in software. However, because soft-

ware is malleable, such watermarks have to withstand commonly occurring transformations—for example, by optimizing compilers, which alter a program's structure while preserving its functionality—as well as those expressly designed to destroy watermarks.

Watermark properties

Collberg and Thomborson[8] describe three important properties of software watermarks. *Resilience* represents the ability to withstand several forms of "dewatermarking" attacks. *Stealth* quantifies the difference between the types of instructions used to embed a watermark and those used for other program computations. No specific metrics are yet available for either property—resilience is highly subjective, and the instruction mix even for identical watermarking techniques is likely to vary widely across different programming languages and platforms. Finally, the *data rate* measures overheads imposed by the watermark, including increases in code size, runtime, and memory usage.

Static watermarks

Collberg and Thomborson propose a software watermarking taxonomy that, at a general level, classifies all schemes as either static or dynamic. *Static* watermarks are stored in the program executable, object, or source code and can be either data or code watermarks. An example of a static data watermark is a program variable that contains a copyright string. Static code watermarks use redundancy in the code to represent a watermark. For example, if two adjacent statements in a program are not dependent on each other, they can appear in an arbitrary order. The program's author could leave a static watermark by following some rule, such as lexicographic ordering, to arrange such pairs of statements.

Unfortunately, both kinds of static watermarks are generally easy to attack. For example, a malicious party could destroy a static code watermark by statically analyzing the code of a program and then randomly rearranging the order of independent adjacent statements. In addition, it is doubtful whether such a watermark meets the strict legal requirement of unambiguously identifying the author.

Julien P. Stern and colleagues[11] propose a method for watermarking assembly and machine code that exploits how frequently selected instruction groups occur in the code. The watermark is encoded using additional instructions in such a way that the instruction group occurrence frequency changes in a random but controlled fashion. This technique is resistant to small-scale code modifications by an attacker, but because decompiling and then compiling the program is likely to result in significantly different instruction occurrence frequencies, it is not applicable to scripting or bytecode languages such as Java.

Akito Monden and colleagues[12] suggest a method for embedding a watermark in a single procedure in a Java program. It is used only for watermarking purposes and thus never executed by the program, though, to ensure stealth, obfuscation or other steps must be taken to make it appear that the program can call it. The method includes standard Java bytecodes for Java virtual machine (JVM) verification that do nothing useful apart from encoding a watermark. The authors report on an experiment in which 20 of 23 watermarks embedded in a program using this technique survived attacks directed at their removal.

Software watermarks have to withstand commonly occurring transformations as well as those expressly designed to destroy watermarks.

Dynamic watermarks

In contrast to static watermarks, *dynamic* watermarks are stored in the program's execution state. For example, on one or more rarely occurring inputs, the program may generate and print a copyright message without storing the whole message statically in the executable. There are three general kinds of dynamic watermarks:

- *Easter egg* watermarks use some very unusual input to produce an identification of authorship, such as a copyright string;
- *data structure* watermarks embed a message in the dynamic state of a program; and
- *execution trace* watermarks are embedded, usually as a statistical property, such as some constraints on the use of registers, in the trace of the program when it executes a special input.

Of these three techniques, dynamic data structure watermarking is most resilient. Because the input that produces the Easter egg watermark must be very unusual, in most cases it is easy to determine by either manual or automated code analysis what this input is. Execution trace watermarks often can be distorted by obfuscating program transformations, including some optimizing transformations such as constant propagation.

Collberg and Thomborson[8] have proposed a dynamic-data-structure watermarking scheme that creates and encodes a watermark in some form

Digital fingerprints can be used to identify individual copies of pirated software programs.

when a specific input is supplied to the program. To reveal the watermark—for example, in a court of law—the program owners must disclose this input and demonstrate that the program uses it to generate a data structure that can be decoded to represent a specific message. They can do so using a debugger or a special tool that can examine the program's runtime state—for example, that of a JVM's heap memory space. Because watermarks are usually encoded to avoid detection by reverse engineers, the program owners can use statistical techniques to show how extremely unlikely it would be for a memory data structure to have its form only by chance.

An important detail of this technique is identifying the start point of the data structure encoding the watermark. For example, if a general graph-like structure is created to represent the watermark, the graph node that represents the beginning of the watermark must be distinguishable in some way—for example, by a predetermined value of one of this node's fields. To conceal the watermarking code's purpose from an attacker, this technique allows trying several of the watermarking structure's nodes as the start node and using a statistical argument to justify the number of false tries.

Despite its high resilience against functionality-preserving program transformations, this approach suffers from the major drawback that the watermark structure stands apart from other dynamic structures constructed by the program. For example, using a points-to analysis tool, an attacker could determine the existence of several disjoint dynamic structures constructed by the program. By removing the instructions used to construct each of the data structures in turn and running the program on sample inputs, the attacker could detect the instructions added to the program to create the watermark and then remove the watermark.

We have developed an improved technique that encodes watermarks using existing program data structures. Specifically, it creates additional reference instance fields in program data structure classes and uses one of many possible encodings to assign values to these fields to represent the watermark. For example, a simple encoding for a binary form represents each 0 in the watermark string with a null value of the watermarking field and each 1 with a non-null value of this field. The above-mentioned automated attack using a points-to analysis tool does not succeed against this technique.

Tamper protection

Software watermarking could theoretically be used in a tamper-protection subsystem based on periodically retrieving the watermark from the running software; if such retrieval fails, the subsystem would shut down the program. However, we are not aware of any such implementation. Special-purpose hardware such as XOM[4] can be used to protect the tamper-protection subsystem against reverse engineering, but other software-based techniques offer an easier way to disguise the system.

Fingerprinting

A technique related to watermarking, *fingerprinting*, creates a watermark that varies from one copy of software to another. While a general watermark identifies only the software authors, a fingerprint can identify the untrusted system on which the trusted program was run. Digital fingerprints can be used to identify individual copies of pirated software programs. From a technical standpoint, the differences between watermarking and fingerprinting, which both aim to embed a secret message in a software program, are minor.

TCPA

The Trusted Computing Platform Alliance (www.trustedcomputing.org) is a collaborative effort initiated by various hardware, software, and technology vendors to define specifications for a hardware-assisted, OS-based, trusted subsystem that would become an integral part of standard computing platforms. The TCPA subsystem would use public-key cryptography and an enabling public-key infrastructure to assign a reliable identity to each PC or computing device.

This capability, combined with secure storage, trusted paths within the system, and a secure coprocessor for performing cryptographic primitives and random number generation, would let software vendors verify the trustworthiness of the environment within which their software is running. The subsystem could be used to grant access to sensitive data only to signed and trusted applications and could also protect applications from tampering.

Although the eventual standardization and widespread availability of TCPA-enabled systems would go a long way toward protecting data and software on computing devices, several potential drawbacks may prevent universal adoption. Some object that vendors could use it to unfairly prevent consumers from switching to alternate products, while others fear that governments could use such a powerful mechanism for political censorship.

Even if TCPA eventually succeeds, the resulting systems will not provide perfect protection of data and software. Currently, 80 percent of the processors in use are embedded, with no mechanisms available to protect software that runs on such platforms. It is unlikely that a single approach will meet the needs of any real-life application; rather, a combination of sound system design, cryptographic primitives, and various software- and hardware-based techniques will be needed. Although cryptography and systems engineering are mature subjects, tamperproofing, obfuscation, and watermarking are at a nascent stage and the future will see many new developments. ■

Acknowledgments

We are grateful to Mikhail Sosonkin for his work on obfuscation of designs of object-oriented programs and Heather Yu for many helpful discussions. We also thank Panasonic Research for funding our work on software watermarking and obfuscation.

References

1. Business Software Alliance, "Seventh Annual BSA Global Software Piracy Study," June 2002; www.bsaa.com.au/media/FINAL7thAnnualGlobalSoftwarePiracyStudyJune2002.pdf.
2. H. Chang and M.J. Atallah, "Protecting Software Code by Guards," *Proc. ACM Workshop Security and Privacy in Digital Rights Management*, ACM Press, 2001, pp. 160-175.
3. M. Jakobsson and M. Reiter, "Discouraging Software Piracy Using Software Aging," *Proc. ACM Workshop Security and Privacy in Digital Rights Management*, ACM Press, 2001, pp. 1-12.
4. D. Lie et al., "Architectural Support for Copy and Tamper Resistant Software," *Proc. 9th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2000, pp. 168-177.
5. D. Aucsmith, "Tamper Resistant Software: An Implementation," *Proc. 1st Int'l Workshop Information Hiding*, LNCS 1174, Springer-Verlag, 1996, pp. 317-333.
6. S. Chow et al., "A White-Box DES Implementation for DRM Applications," *Proc. 2nd ACM Workshop Digital Rights Management, 9th Ann. ACM Conf. Computer and Communications Security*, to appear in LNCS 2696, Springer-Verlag, 2003.
7. B. Barak et al., "On the (Im)possibility of Obfuscating Programs," *Advances in Cryptology—Crypto 2001, Proc. 21st Ann. Int'l Cryptology Conf.*, LNCS 2139, Springer-Verlag, 2001, pp. 1-18.
8. C.S. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection," *IEEE Trans. Software Eng.*, Aug. 2002, pp. 735-746.
9. C. Wang et al., "Software Tamper Resistance: Obstructing Static Analysis of Programs," tech. report CS-2000-12, Dept. Computer Science, Univ. Virginia, Charlottesville, 2000.
10. S. Chow et al., "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs," *Proc. 4th Int'l Conf. Information Security*, LNCS 2200, Springer-Verlag, 2001, pp. 144-155.
11. J.P. Stern et al., "Robust Object Watermarking: Application to Code," *Proc. 3rd Int'l Workshop Information Hiding*, LNCS 1768, Springer-Verlag, 1999, pp. 368-378.
12. A. Monden et al., "A Practical Method for Watermarking Java Programs," *Proc. 24th Ann. Int'l Computer Software and Applications Conf.*, IEEE CS Press, 2000, pp. 191-197.

Gleb Naumovich *is an assistant professor in the Department of Computer and Information Science at Polytechnic University. His research interests are in software engineering, programming languages, application security, and intellectual property protection. Naumovich received a PhD in computer science from the University of Massachusetts, Amherst. He is a member of the ACM SIGSOFT. Contact him at gleb@poly.edu.*

Nasir Memon *is an associate professor in the Department of Computer and Information Science at Polytechnic University. His research interests include data compression, image and video processing, computer security, and multimedia computation and communication. Memon received a PhD in computer science from the University of Nebraska. He is a member of the IEEE, the ACM, and the Society of Photo-Optical Instrumentation Engineers. Contact him at memon@poly.edu.*