Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection

Christian S. Collberg, *Member*, *IEEE Computer Society*, and Clark Thomborson, *Senior Member*, *IEEE*

Abstract—We identify three types of attack on the intellectual property contained in software and three corresponding technical defenses. A defense against reverse engineering is obfuscation, a process that renders software unintelligible but still functional. A defense against software piracy is watermarking, a process that makes it possible to determine the origin of software. A defense against tampering is tamper-proofing, so that unauthorized modifications to software (for example, to remove a watermark) will result in nonfunctional code. We briefly survey the available technology for each type of defense.

Index Terms—Obfuscation, watermarking, tamper-proofing, intellectual property protection.

1 Background—Malicious Clients vs. Malicious Hosts

Until recently, most computer security research was concerned with protecting the integrity of a *benign host* and its data from attacks from *malicious client* programs (Fig. 1a). This assumption of a benign host is present in Neumann's influential taxonomy of "computer-related risks," in which the job of a security expert is to design and administer computer systems that will fulfill "certain stringent security requirements most of the time" [64, p. 4]. Other security experts express similar worldviews, sometimes by choosing a title such as "Principles and Practices for Securing IT Systems" [82]; sometimes by making explicit definitions that presuppose benign hosts "... a security flaw is a part of a program that can cause the system to violate its security properties" [48]; sometimes in an explanation "... vulnerabilities to the system ... could be exploited to provide unauthorized access or use." [42, p. 51]; and sometimes in a statement of purpose "[we take] the viewpoint of the system owner" [52].

The benign-host worldview is the basis of the Java security model, which is designed to protect a host from attacks by a potentially malicious downloaded applet or a virus-infested installed application. These attacks usually take the form of destroying or otherwise compromising local data on the host machine.

To defend itself and its data against a malicious client, a host will typically restrict the actions that the client is allowed to perform. In the Java security model, the host uses bytecode verification to ensure the type safety of the untrusted client. Additionally, untrusted code (such as applets) is prevented from performing certain operations, such as writing to the local file system. A similar technique is Software Fault Isolation [53], [90], [91], which modifies the client code so that it is unable to write outside its designated area (the "sandbox").

- *C.S. Collberg is with the Department of Computer Science, University of Arizona, Tuscon, AZ 85721. E-mail: collberg@cs.arizona.edu.*
- *C. Thomborson is with the Department of Computer Science, University of Auckland, Auckland, New Zealand. E-mail: cthombor@cs.auckland.ac.nz.*

Manuscript received 1 July 2000; accepted 26 Nov. 2001.
Recommended for acceptance by M. Reiter.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112393.

A recent surge of interest in "mobile agent" systems has caused researchers to focus attention on a fundamentally different view of security [17], [88]. See Fig. 1b, illustrating a benign client code being threatened by the host on which it has been downloaded or installed. A "malicious host attack" typically takes the form of *intellectual property violations*. The client code may contain trade secrets or copyrighted material that, should the integrity of the client be violated, will incur financial losses to the owner of the client. We will next consider three malicious-host attack scenarios.

1.1 Malicious Host Attacks

Software piracy, the illegal copying and resale of applications, is a 12 billion dollar per year industry [67]. Piracy is therefore a major concern for anyone who sells software. In the early days of the personal computer revolution, software developers experimented vigorously with various forms of technical protection [34], [37], [56], [57], [58], [59], [79], [96] against illegal copying. Some early copy protection schemes have been abandoned since they were highly annoying to honest users who could not even make backup copies of legally purchased software, or who lost the hardware "dongle" required to activate it. A number of dongle manufacturers are still in business; one is frank enough to state "... the software interrogating the dongle [may be] the weakest part of the system... Any dongle manufacturer who claims that their system is unbeatable is lying" [83].

Software piracy is likely to continue so long as it continues to be easy, delivers immediate tangible or intangible rewards to the pirate, and is socially acceptable [51]. Our goal in this paper is to make piracy more difficult. We note that software piracy is socially acceptable in settings that encourage a belief in insiders' entitlement [72], price discrimination [33], "cooperation is more important than copyright" [81], or traditional Confucian ethics [16]—but, also see [85].

Many software developers also worry about their applications being *reverse engineered* [4], [55], [76], [78], [87]. Several court cases have been tried in which a valuable piece of code was extracted from an application and incorporated into a competitor's code. Such threats have recently become more of a concern since, more and more,

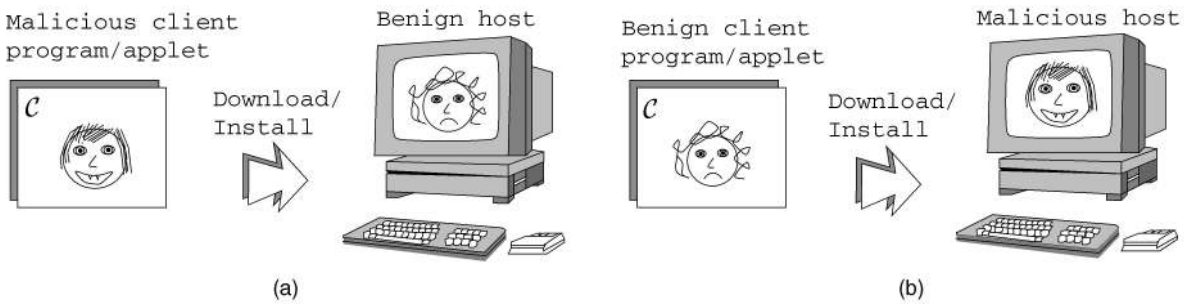

Fig. 1. Attacks by malicious clients and hosts. (a) Attack by a malicious client. (b) Attack by a malicious host.

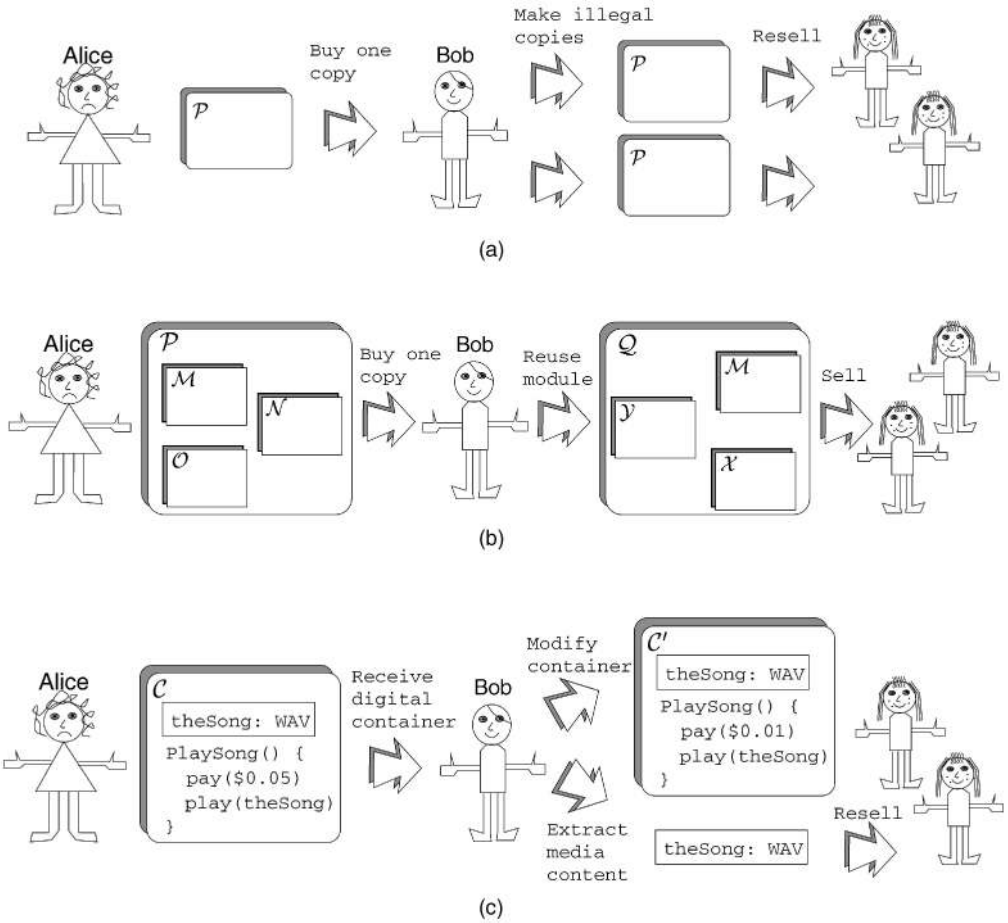

Fig. 2. Attacks against software intellectual property. (a) Software piracy attacks. Bob makes illegal copies of Alice's program $\mathcal{P}$ and resells them. (b) Malicious reverse engineering attack. Bob extracts a module $\mathcal{M}$ from Alice's program $\mathcal{P}$ and reuses it in his own application $\mathcal{Q}$. (c) Tampering attack. Bob either extracts the media content from the digital container $\mathcal{C}$ or modifies $\mathcal{C}$ so that he has to pay less for playing the media.

programs are distributed in easily decompilable formats rather than native binary code [68], [89]. Important examples include the Java class file format and ANDF [54].

A related threat is *software tampering*. Many mobile agents and e-commerce application programs must, by their very nature, contain encryption keys or other secret information. Pirates who are able to extract, modify, or otherwise tamper with this information can incur significant financial losses to the intellectual property owner.

These three types of attack (*software piracy*, *malicious reverse engineering*, and *tampering*) are illustrated in Fig. 2:

- In Fig. 2a, Bob makes copies of an application he has legally purchased from Alice and illegally sells them to unsuspecting customers.
- In Fig. 2b, Bob decompiles and reverse engineers an application he has bought from Alice in order to reuse one of her modules in his own program.
- In Fig. 2c, finally, Bob receives a "digital container" [27], [43], [44], [98] (also known as *Cryptolope* and *DigiBox*) from Alice, consisting of some digital media content as well as code that transfers a certain amount of electronic money to Alice's account whenever the media is played. Bob can attempt to tamper with the digital container either to modify the amount that he has to pay or to extract the media content itself. In the latter case, Bob can continue to enjoy the content for free or even resell it to a third party.

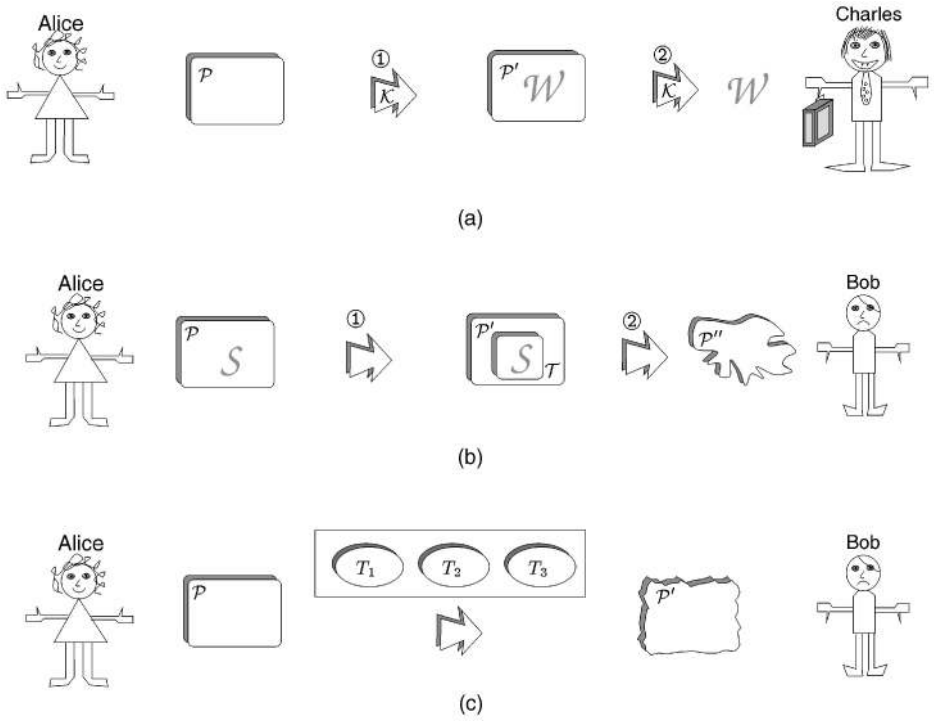

Fig. 3. Defenses against malicious host attacks. (a) Software watermarking. Alice watermarks her program using a secret key $\mathcal{K}$. Charles extracts the watermark using the same key. (b) Tamper-proofing. Alice protects a secret $\mathcal{S}$ by adding tamper-proofing code $\mathcal{T}$ that makes the program fail if $\mathcal{S}$ has been tampered with. (c) Obfuscation. Alice transforms her program into an equivalent one (using obfuscation transformations $T_1 \cdots T_3$) to prevent Bob from reverse engineering it.

1.2 Defenses against Malicious Host Attacks

It should be noted that it is much more difficult to defend a client than it is to defend a host. To defend a host against a malicious client, all that is needed is to restrict the actions that the client is allowed to perform.

Unfortunately, no such defense is available to protect a client against a host attack. Once the client code resides on the host machine, the host can make use of *any* conceivable technique to extract sensitive data from the client, or to otherwise violate its integrity. The only limiting factors are the computational resources the host can expend on analyzing the client code.

While it is generally believed that complete protection of client code is an unattainable goal [7], recent results (by ourselves and others) have shown that some degree of protection *can* be achieved. Recently, *software watermarking* [22], [28], [35], [60], *tamper-proofing* [5], [6], [38], [77], and *obfuscation* [20], [23], [24], [25], [38], [49], [65], [92] have emerged as feasible technical means for the intellectual property protection of software. (Other promising techniques, such as traitor tracing [14], secret sharing [8], reference states [39], and secure evaluation [2], [77] are still in the hands of theorists.) Obfuscation attempts to transform a program into an equivalent one that is harder to reverse engineer. Tamper-proofing causes a program to malfunction when it detects that it has been modified. Software watermarking embeds a *copyright notice* in the software code to allow the owners of the software to assert their intellectual property rights. *Software fingerprinting* is a similar technique that embeds a unique *customer identification number* into each distributed copy of an application in order to facilitate the tracking and prosecution of copyright violators.

These three types of defenses (*software watermarking*, *obfuscation*, and *tamper-proofing*) are illustrated in Fig. 3:

- In Fig. 3a, Alice watermarks her program $\mathcal{P}$. At 1, the watermark $\mathcal{W}$ is incorporated into the original program, using a secret key $\mathcal{K}$. At 2, Bob steals a copy of $\mathcal{P}'$ and Charles extracts its watermark using $\mathcal{K}$ to show that $\mathcal{P}'$ is owned by Alice.
- In Fig. 3b, Alice attempts to protect a secret $\mathcal{S}$ stored in her program by adding special tamper-proofing code. This code is able to detect if Bob has tampered with $\mathcal{S}$ and, if that is the case, the code will make the program fail.
- In Fig. 3c, Alice protects her program from reverse engineering by obfuscating it. The obfuscating transformations make the program harder for Bob to understand, while maintaining its semantics.

It should be noted that there is a vast body of practical research and development on *hardware assisted* software protection. The available art for hardware-assisted security is based on memory devices such as floppy disks [31], CD-ROMs [97], modern removable storage media (hard disk, recordable DVD, secure digital memory card), and video playback units [1]: secure processors [50], smart cards [10], and hardware dongles [15], [58], [83].[1] Many of these schemes are trade secrets, although some are the subject of patent disclosures. They have received little attention in the academic literature. Some hardware-based schemes—in

1. A dongle is a device sold with a software application to protect against piracy. The device attaches to a computer's I/O port and is queried by the application at regular intervals. If the dongle does not respond, the application will refuse to run.

particular, dongles—have had some commercial success, especially for high-end, low quantity software. In general, however, they have a poor reputation among users who find them cumbersome and intrusive. We will not consider hardware-based software protection methods further in this paper.

We note, in passing, that software piracy, reverse engineering, and tampering can be discussed from the traditional "benign-host worldview" of computer security, in which every computer system is (or should be) designed to provide security. In this worldview, a software pirate is defined as anyone who subverts system security for the purpose of stealing intellectual property in software on that system. However, this worldview begs the question of "who defines the security objectives for a system?," a question which is especially vexing for any system with multiple designers and administrators such as the world-wide web. Accordingly, in this paper, we avoid the vexed question by dismissing the traditional underlying assumption of a "benign host." Instead, we assume that any host may be malicious, as we analyze computer security in this paper from the viewpoint of the owner of the client software. We leave it to others to commence the study of computer security from the diverse viewpoints of a representative range of *users* of an open system. We expect that such user-centric security analysis will be a challenging and important field of study, if only because some users will surely disagree with some of the security design objectives of any (possibly-compromised) software client running on any possibly-hostile host.

2 Obfuscation

Security through obscurity has long been viewed with disdain in the security and cryptography communities. There are, however, situations where higher levels of protection than that achievable through obscurity at the present time does not seem achievable. For example, the media player in Fig. 2c accepts digital containers which hold, among other things, the media itself (encrypted), (partial) cryptographic keys, and a set of *business rules* describing if the consumer may play the media, resell it, store it, etc. The media player itself also contains partial cryptographic keys as well as algorithms for decoding, decrypting, and playing the media. To prevent illegal use of the contents, these algorithms and keys must be protected from an adversary. If the media player is implemented in software, this means preventing an attacker from reverse engineering the security sensitive part of the code. As far as we know, there do not exist any techniques for preventing attacks by reverse engineering stronger than what is afforded by obscuring the purpose of the code.

In [25] and [24], we explore new approaches to *code obfuscation*, based on the following statement of the code obfuscation problem.

Given a set of obfuscating transformations $\mathcal{T} = \{\mathcal{T}_1, \cdots, \mathcal{T}_n\}$ and a program $\mathcal{P}$ consisting of source code objects (classes, methods, statements, etc.) $\{\mathcal{S}_1, \cdots, \mathcal{S}_k\}$, find a new program $P' = \{\cdots, \mathcal{S}'_j = \mathcal{T}_i(\mathcal{S}_j), \cdots\}$ such that:

- P' has the same observable behavior as P, i.e., the transformations are *semantics-preserving*.
- The *obscurity* of P' maximized, i.e., understanding and reverse engineering P' will be strictly more time-consuming than understanding and reverse engineering P.
- The *resilience* of each transformation $\mathcal{T}_i(\mathcal{S}_j)$ is maximized, i.e., it will either be difficult to construct an automatic tool to undo the transformations or executing such a tool will be extremely time-consuming.
- The *stealth* of each transformation $\mathcal{T}_i(\mathcal{S}_j)$ is maximized, i.e., the statistical properties of $\mathcal{S}'_j$ are similar to those of $\mathcal{S}_j$.
- The *cost* (the execution time/space penalty incurred by the transformations) of P' is minimized.

Code obfuscation is very similar to *code optimization*, except that, with obfuscation, we are maximizing obscurity while minimizing execution time; whereas, with optimization, we are just minimizing execution time.

An upper bound on the time needed by an adversary to reverse engineer a program P is the time needed to do a *black-box* study of P (i.e., study the input-output relations of P) plus the time needed to encode the discovered relations in a new program. Most malicious reverse engineers, however, will also do a *white-box* study of P, i.e. they will decompile and examine the code itself. The ultimate goal of code obfuscation is to construct P' (an obfuscated version of P) for which a white-box study will yield no useful information. In other words, we would like the actual time needed to reverse engineer P' to approach the upper bound needed to reverse engineer P.

2.1 Lexical Transformations

The advent of Java, whose strongly typed bytecode and architecture-neutral class files make programs easy to decompile, has left programmers scurrying for ways to protect their intellectual property. On our website [21], we list a number of "Java obfuscation tools," most of which modify only the lexical structure of the program. Typically, they do nothing more than scramble identifiers. Such lexical transforms will surely be annoying to a reverse engineer and, therefore, will prevent some thievery of intellectual property in software. However, any determined reverse engineer will be able to read past the scrambling of identifiers in order to discover what the code is really doing.

2.2 Control Transformations

In [25], we introduced several control-altering transformations. These control transformations rely on the existence of *opaque predicates*. A predicate P is opaque if its outcome is known at obfuscation time, but is difficult for the deobfuscator to deduce. We write P^F $\left(P^T\right)$ if P always evaluates to `False` (`True`) and $P^?$ if P may sometimes evaluate to `True` and sometimes to `False`.

Given such opaque predicates, it is possible to construct obfuscating transformations that break up the flow-of-control of a procedure. In Fig. 4a, we split up the block $A; B$ by inserting an opaquely true predicate P^T which makes it appear as if B is only executed sometimes. In Fig. 4b, B is split into two *different* obfuscated versions B and B'. The opaque predicate $P^?$ selects either of them at runtime. In Fig. 4c, finally, P^T always selects B over B_{Bug}, a buggy version of B.

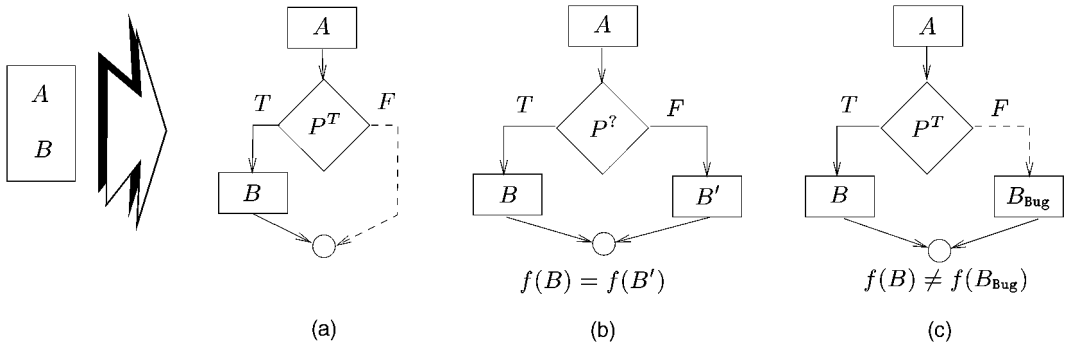

Fig. 4. Control transformation by opaque predicate insertion.

There are many control transformations similar to those in Fig. 4, some of which are discussed in [25]. The resilience of these transformations is directly related to the resilience of the opaque predicates on which they rely. It is therefore essential that we are able to manufacture strong opaque predicates.

Equally important is the *cost* and *stealth* of opaque predicates. An introduced predicate that differs wildly from what is in the original program will be unacceptable since it will be easy for a reverse engineer to detect. Similarly, a predicate is unacceptable if it introduces excessive computational overhead.

Since we expect most deobfuscators to employ various static analysis techniques, it seems natural to base the construction of opaque predicates on problems which these techniques cannot handle well. In particular, precise static analysis of pointer-based structures and parallel regions is known to be intractable [29], [41], [70]. In [25], we discuss two general methods for generating resilient and cheap opaque predicates that are based on the intractability of these static analysis problems.

Fig. 5 shows a simple example of how strong opaque predicates can be constructed based on the intractability of alias analysis. The basic idea is to extend the program to be obfuscated with code that builds a set of complex dynamic structures. A number of global pointers reference nodes within these structures. The introduced code will occasionally update the structures (modifying pointers, adding nodes, splitting and merging structures, etc.), but will maintain certain invariants, such as "pointers p and q will never refer to the same heap location," or "there may be a path from pointer p to pointer q," etc. These invariants are then used to manufacture opaque predicates as needed.

For example, in Fig. 5a, 5b, and 5c, we can ask the opaque query `if` $(\texttt{f} == \texttt{g})^?$ `then` $\cdots$ since the two pointers `f` and `g` move around in the same structure and could possibly alias each other. Then, after the one component in Fig. 5c is split into two components in Fig. 5d, we can ask the query `if` $(\texttt{f} == \texttt{g})^F$ `then` $\cdots$ since `f` and `g` now move around in different structures. Finally, in Fig. 5f, the two components have been merged and we can again ask the query `if` $(\texttt{f} == \texttt{g})^?$ `then` $\cdots$.

It should be noted that, although theoretical studies have shown alias analysis to be hard, it is to the best of our knowledge unknown whether a *random instance of aliasing* is hard. However, many years of experience from optimizing compilers have shown us that it is very difficult to construct fast and precise alias analyzers. We also have a good understanding where such analyzers fail: They typically have problems with deeply nested recursive structures and with destructive operations (such as `Split`, `Merge`, and `Delete` above). We therefore conjecture that carefully constructed instances of aliasing will be a good source of opaque predicates. It is an open problem whether alias analysis will eventually be shown to be *average case complete* [94].

2.3 Data Transformations

In [24], we present several transformations that obfuscate data structures. As an example, consider the *Variable Splitting* transformation in Fig. 6. In this example, a Boolean variable V is split into two integer variables p and q using the new representation shown in Fig. 6a. Given this new representation, we create new implementations for the built-in Boolean operations. Only the implementation of `&` is shown in Fig. 6b.

In Fig. 6c, we show the result of splitting three Boolean variables `A`, `B`, and `C` into short variables `a1` and `a2`, `b1`, and `b2`, and `c1` and `c2`, respectively. An interesting aspect of our chosen representation is that there are several possible ways to compute the same Boolean expression. Statements (2′) and (3′), for example, look different, although they both assign `False` to a variable. Similarly, while statements (4′) and (5′) are completely different, they both compute $A\&B$.

2.4 Other Defenses against Reverse Engineering

In a general sense, we can consider an "obfuscation" to be anything that slows down or dissuades a reverse engineer. We list two such methods below and we describe two others near the end of Section 3.

Antidisassembly. We may write the machine code in an executable distribution in such a way that a disassembler is unlikely to be able to print out an accurate rendition of the assembly code. Cohen describes several such techniques, applicable to any machine codes with variable-length byte-addressible instructions, such as the Intel x86 series. For example, "... we can include jump instructions whose last byte (usually the address jumped to) corresponds to a desired operation code and places the code jumped to appropriately so that the jump works and the proper return address is in the middle of the previously executed instruction. In this way, we reuse the last bytes of the jump location as operation codes on the next pass through the code" [20].

Antidebugging. We may disable or confuse a debugger, for example, by writing code that actively uses all available

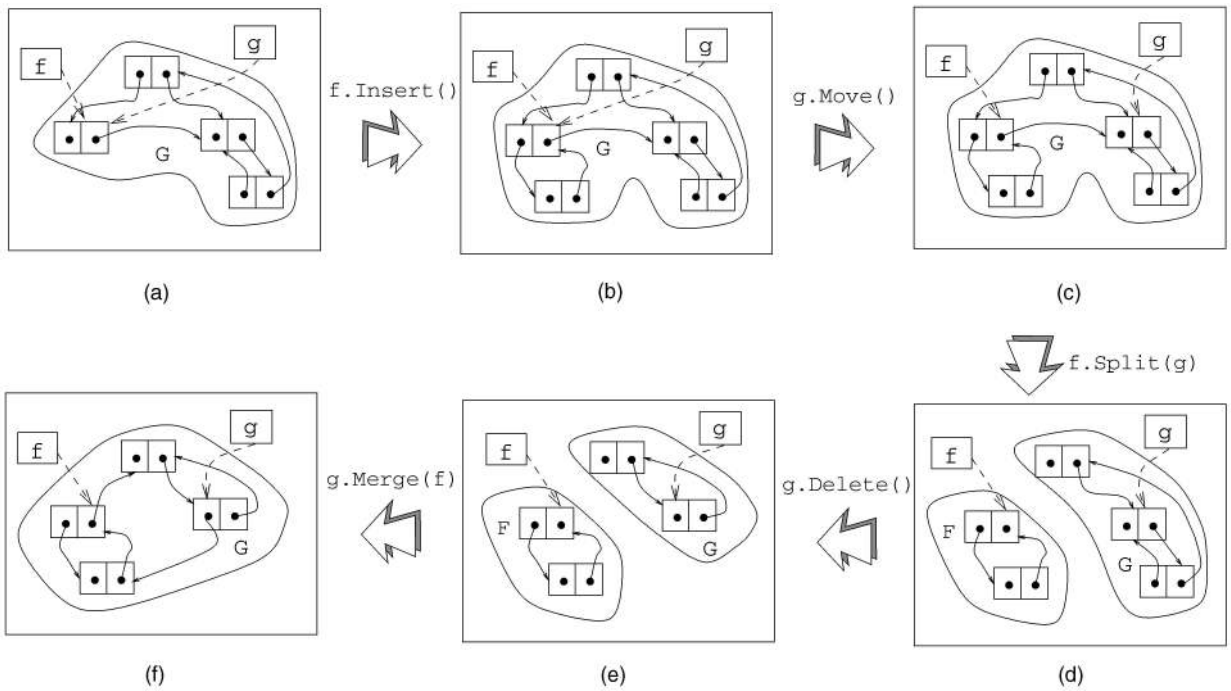

Fig. 5. Strong opaque predicates based on the intractability of alias analysis.

$g(V)$ p	q	$f(p,q)$ V	$2p+q$
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

(a)

AND[A,B]		A 0	1	2	3
	0	3	0	0	0
B	1	3	1	2	3
	2	0	2	1	3
	3	3	0	0	3

(b)

```
(1) bool A,B,C;          (1') short a1,a2,b1,b2,c1,c2;
(2) B = False;           (2') b1=0; b2=0;
(3) C = False;     ⇒     (3') c1=1; c2=1;
(4) C = A & B;           (4') x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(5) C = A & B;           (5') c1=(a1 ^ a2) & (b1 ^ b2); c2=0;
(6) if (A) ···;          (6') x=2*a1+a2; if ((x==1) || (x==2)) ···;
(7) if (B) ···;          (7') if (b1 ^ b2) ···;
```

(c)

Fig. 6. A data transformation that splits Boolean variables.

interrupts (including the breakpoint interrupt) [84]. If we know what debugger the reverse-engineer is likely to be using, then our code may be able to "attack" the debugger by writing into its address area [20]. We can write time-sensitive code, for example, involving races between processes on two CPUs that will terminate its operation whenever a debugger is probing a process intensively—because this will slow down its operation [84].

3 Watermarking

Watermarking embeds a secret message into a cover message. In *media watermarking* [3], [9], [45], [66], the secret is usually a copyright notice and the cover a digital image or an audio or video production. Watermarking an object discourages intellectual property theft or, when such theft has occurred, allows us to prove ownership.

Fingerprinting is similar to watermarking, except a different secret message is embedded in every distributed cover message. This may allow us not only to detect when theft has occurred, but also to trace the copyright violator. A typical fingerprint includes a vendor, product, and customer identification numbers.

Our interest is in the watermarking and fingerprinting of *software* [22], [28], [35], [40], [46], [60], [69], [75], a problem that has received much less attention than media watermarking. We can describe the software watermarking problem as follows:

Embed a structure W (the watermark) into a program P such that:

- W can be reliably located and extracted from P (the embedding is *resilient* to dewatermarking attacks).
- W is large (the embedding has a high *data rate*).
- Embedding W into P does not adversely affect the performance of P (the embedding is *cheap*).
- Embedding W into P does not change any statistical properties of P (the embedding is *stealthy*).
- W has a mathematical property that allows us to argue that its presence in P is the result of deliberate actions.

Any software watermarking technique will exhibit a trade-off between resilience, data rate, cost, and stealth. For example, the resilience of a watermark can easily be increased by exploiting redundancy (i.e., including the mark several times in the cover program), but this will result in a reduction in bandwidth. In many situations, a high data rate may be unnecessary. For example, a 32-bit integer could provide 10 bits of vendor and product information as well as a 22-bit customer identification number. In other situations, however, we might want the watermark to contain some internal structure that will allow us to detect tampering (parity or error-correcting bits may be used for this purpose) or which allows us to argue that the watermark has some unique property that could not have occurred by chance. For example, the watermark could be the product of two large primes which only the owner of the watermarked program knows how to factor.

It should be noted that there are two possible interpretations of stealth, *static stealth* and *dynamic stealth*. A watermark is statically stealthy if a static analysis reveals no statistical differences between the original and the watermarked program. Similarly, the watermark is dynamically stealthy if a execution trace of the program (say, an address trace or an instruction trace) reveals no differences. An attacker may, of course, analyze a program statically or dynamically or both in order to discover the location of a watermark. In this paper, we (like many others, for example, Wang et al. [93]) will mostly consider attacks by static analysis.

Goldreich and Ostrovsky [32], on the other hand, provide a theoretical treatment of *oblivious machines*, computational devices that leak no runtime information to an attacker about the internal execution of a program. In particular, when running an oblivious program twice on different inputs (that have the same running time), the program will access exactly the same sequence of memory locations.

Similarly, Aucsmith [5] presents a practical obfuscation method designed not to leak information about execution paths and data accesses. The idea is to break up the program in segments which are continuously relocated in memory. Every time a particular piece of code is executed, it will be moved to a different segment of memory, therefore making an address trace of no value to an adversary. We will discuss this technique further in Section 4.

3.1 Threat-Model

To evaluate the resilience of a watermarking technique (how well the mark will resist intentional attempts at removal), we must first define our *threat-model*. In other words, what constitutes a reasonable level of attack and what specific techniques is an attacker likely to employ? It is generally accepted that no software protection scheme will withstand a determined *manual attack*, where the software is inspected by a human reverse engineer for an extensive period of time. Of more interest are *automated* or *class* attacks where an automated watermark removal tool that is effective against a whole class of watermarks is constructed.

Assume the following scenario: Alice watermarks a program $\mathcal{P}$ with watermark $\mathcal{W}$ and key $\mathcal{K}$ and then sells $\mathcal{P}$ to Bob. Before Bob can sell $\mathcal{P}$ on to Douglas, he must ensure that the watermark has been rendered useless or else Alice will be able to prove that her program has been stolen. Fig. 7 illustrates the kinds of dewatermarking attacks available to Bob:

- In Fig. 7a, Bob launches an *additive attack* by adding his own watermark $\mathcal{W}_1$ to Alice's watermarked program $\mathcal{P}'$. This is an effective attack if it is impossible to detect that Alice's mark temporally precedes Bob's.
- In Fig. 7b, Bob launches a *distortive attack* on Alice's watermarked program $\mathcal{P}'$. A distortive attack applies a sequence of *semantics-preserving transformations* uniformly over the entire program, in the hope that
  a. the distorted watermark $\mathcal{W}'$ can no longer be recognized and
  b. the distorted program $\mathcal{P}''$ does not become so degraded (i.e., slow or large) that it no longer has any value to Bob.
- In Fig. 7c, Bob buys several copies of Alice's program $\mathcal{P}$, each with a different fingerprint (serial number) $\mathcal{F}$. By comparing the different copies of the program, Bob is able to locate the fingerprints and can then easily remove them.

We will assume a threat-model consisting primarily of distortive attacks, in the form of various types of semantics-preserving code transformations. Ideally, we would like our watermarks to survive *translation* (such as compilation, decompilation, and binary translation [26]), *optimization*, and *obfuscation*.

3.2 Static Watermarking Techniques

Software watermarks come in two flavors, *static* and *dynamic*. Static watermarks are stored in the application executable itself; whereas, dynamic watermarks are constructed at runtime and stored in the dynamic state of the program. While static watermarks have been around for a long time, dynamic marks were only introduced recently in [22].

Moskowitz and Cooperman [60] and Davidson and Myhrvold [28] are two techniques representative of typical static watermarks. Moskowitz and Cooperman describe a static data watermarking method in which the watermark is embedded in an image using one of the many media watermarking algorithms. This image is then stored in the static data section of the program. Davidson and Myhrvold [28] describe a static code watermark in which a fingerprint is encoded in the basic block sequence of a program's control flow graphs.

Venkatesan et. al. [86] present what appears to be the strongest known static software watermarking technique. The idea is to treat the source program as a control flow

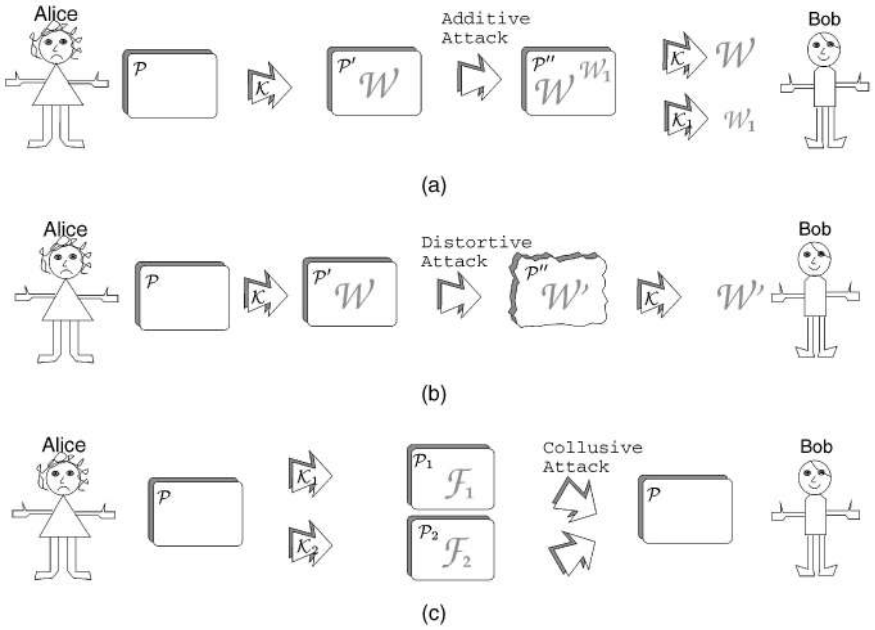

Fig. 7. Attacks on watermarks and fingerprints. (a) An effective *additive* attack. (b) An effective *distortive* attack. (c) An effective *collusive* attack.

graph G of basic blocks, to which a watermark graph W is added forming a new control flow graph H. See Fig. 8b which shows how G and W are merged by adding code to the watermarked program that introduces new control flow edges between the two graphs. These edges (dashed in the figure) can be realized, for example, by using opaque predicates.

To detect the watermark of Venkatesan et. al., the extractor needs to

a. reconstruct the control flow graph of the watermarked program,
b. identify which of the nodes of the control flow graph belong to the watermark graph (or, at least identify most of these nodes), and
c. reconstruct the watermark graph itself.

To identify the watermark nodes, the authors suggest "stor[ing] one or more bits at a node that flags when a node is in W by using some padded data...." This is a serious weakness of the algorithm. If the method by which watermark nodes are marked is publically known (and we would normally expect this to be the case), then destroying the mark is trivial: Simply scramble the mark bits in each watermark node. Even if the exact marking method is unknown, an adversary can apply a variety of local code optimization techniques (such as peephole optimization, register reallocation, and instruction scheduling), that will completely restructure every basic block of the program. This will make watermark recognition virtually impossible.

Thus, unfortunately, all currently known static watermarks are susceptible to simple distortive dewatermarking attacks. For example, any code motion optimization technique will destroy Davidson's method. Code obfuscation techniques that radically change the control flow or reorganize data will also successfully thwart the recognition of static watermarks.

3.3 Dynamic Watermarking Techniques

There are three kinds of dynamic watermarks. In each case, the mark is recognized by running the watermarked program with a predetermined input sequence $\mathcal{I} = \mathcal{I}_1 \cdots \mathcal{I}_k$. This highly unusual input makes the application enter a state which represents the watermark.

There are three dynamic watermarking techniques:

Easter Egg Watermarks. The defining characteristic of an Easter Egg watermark is that, when the special input sequence is entered, it performs some action that is immediately perceptible by the user. Typically, a copyright message or an unexpected image is displayed. For example, entering the URL `about : mozilla` in `Netscape 4.0` will make a fire-breathing creature appear. The main problem with Easter Egg watermarks is that they seem to be easy to locate. There are even several Web site repositories of such watermarks, e.g., [62].

Execution Trace Watermarks. Unlike Easter Egg watermarks, Execution Trace watermarks produce no special output. Instead, the watermark is embedded within the trace (either instructions or addresses, or both) of the program as it is being run with the special input $\mathcal{I}$. The watermark is extracted by monitoring some (possibly statistical) property of the address trace and/or the sequence of operators executed. Unfortunately, many simple optimizing and obfuscating transformations will obliterate Execution Trace watermarks.

Data Structure Watermarks. Like Execution Trace watermarks, Data Structure watermarks do not generate any output. Rather, the watermark becomes embedded within the state (global, heap, and stack data, etc.) of the program as it is being run with the special input $\mathcal{I}$. The watermark is extracted by examining the current values held in the program's variables after the end of the input sequence has been reached. Unfortunately, many data structure watermarks are also susceptible to attacks by obfuscation. Several obfuscating transformations have

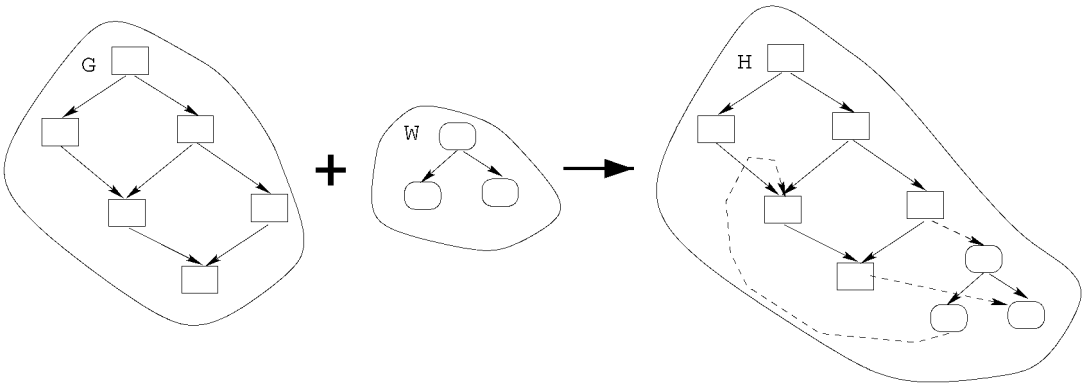

Fig. 8. Static watermarking algorithm of Venkatesan et al. [86].

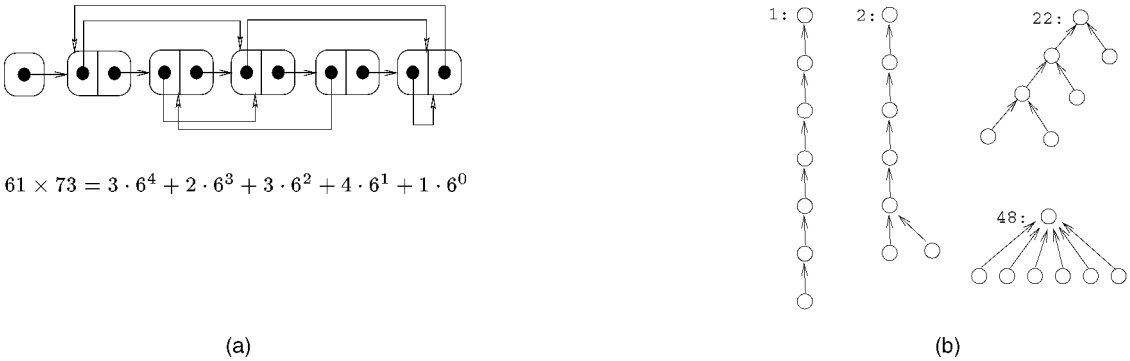

Fig. 9. Graphic embeddings of watermarks. (a) Radix-6 encoding. The right pointer field holds the `next` field, the left pointer encodes a base-*k* digit. (b) Enumeration encoding. These are the first, second, 22nd, and 48th trees in an enumeration of the oriented trees with seven vertices.

been devised which will effectively destroy the dynamic state (while maintaining semantic equivalence) and make watermark recognition impossible.

3.4 Dynamic Graph Watermarking

In [22], we describe a new Data Structure watermarking technique called *Dynamic Graph Watermarking*. The central idea is to embed a watermark in the *topology* of a dynamically built graph structure. Code that builds this graph is then inserted into the program to be watermarked. Because of pointer aliasing effects, the graph-building code will be hard to analyze and detect and it can be shown that it will be impervious to most dewatermarking attacks by code optimization and code obfuscation.

The watermarking algorithm runs in three steps:

1. Select a number n with a unique signature property. For example, let $n = p \times q$, where p and q are prime.
2. Embed n in the topology of a graph G. Fig. 9a shows a Radix-*k* embedding in a circular linked list and Fig. 9b shows how we can embed n by selecting the nth graph in a particular enumeration of a particular class of graphs. Many other such embeddings are possible.
3. Construct a program W which builds G. Embed W in the program to be watermarked such that, when the program is run with a particular input sequence $\mathcal{I}$, G is built.

To recognize the mark, the watermarked program is run with $\mathcal{I}$ as input, G is extracted from the heap, n is extracted from G, and n is factored. We refer to [22] for a more detailed exposition.

4 Tamper-Proofing

There are many situations where we would like to stop anyone from executing our program if it has been altered in any way. For example, a program P should not be allowed to run if 1) P is watermarked and the code that builds the mark has been altered, 2) a virus has been attached to P, or 3) P is an e-commerce application and the security-sensitive part of its code has been modified. To prevent such *tampering attacks* we can add *tamper-proofing code* to our program. This code should

a. **detect** if the program has been altered and
b. cause the program to **fail** when tampering is evident.

Ideally, detection and failure should be widely dispersed in time and space to confuse a potential attacker. Simple-minded tamper-proofing code like

$$\texttt{if } (\texttt{tampered\_with}()) \texttt{ i} = 1/0$$

is unacceptable, for example, because it is easily defeated by locating the point of failure and then reversing the test of the detection code.

There are three principal ways to detect tampering:

1. We can examine the executable program itself to see if it is identical to the original one. To speed up the test, a message-digest algorithm such as MD5 [71] can be used.
2. We can examine the validity of intermediate results produced by the program. This technique is known as *program* (or *result*) *checking* [12], [13], [30], [73], [74], [95] and has been touted as an alternative to program verification and testing.

3. We can encrypt the executable, thereby preventing anyone from modifying it successfully unless they are able to decrypt it. The decryption routines must be protected from reverse-engineering by hardware means, by obfuscation, or both.

In our literature search, we find only a few publications that describe tamper-proofing methods. One commercial website describes a tamper-proofing system that uses continual self-examination and encryption (methods 1 and 3 above), along with its own keyboard handler for password input [63]. Torrubia and Mora describe an encryption method in which the protected code is arranged in nested loops and the loop-entry code decrypts the body of the loop [84]. In both cases, antidebugging measures are taken to make it difficult for a reverse-engineer to examine the code.

Aucsmith [5] and Aucsmith and Graunke's encryption method [6] breaks up a binary program into individually encrypted segments. The tamper-proofed program is executed by decrypting and jumping to segments based in part on a sequence of pseudorandom values generated from a key. After a segment has been executed, it is reencrypted so that only one segment is ever in plain text. The process is constructed so that any state the program is in is a function of all previous states. Thus, should even one bit of the protected program be tampered with, the program is virtually guaranteed to eventually fail and the point of failure may occur millions of instructions away from the point of detection.

Tamper-proofing of type-safe distribution formats such as Java bytecode is more difficult than tamper-proofing assembly code. For example, Aucsmith's technique requires a decryption and jump, which is possible in Java bytecode; however, it cannot be done stealthily since it will always involve a call to a class loader from the standard Java library. One hacker claims to have quickly defeated several of the early protection schemes for Java bytecodes generally by disassembling to discover the authentication code and then by applying patches either to jump around it or to ignore its result [47].

Tamper-proofing by program checking is more likely to work well in Java since it does not require us to examine classfiles directly. Some such detection techniques were discussed in [22], in the context of tamper-proofing software watermarks.

4.1 Tamper-Proofing Viruses

It is interesting to compare the work done on software protection with the on-going struggle between virus writers and virus detector writers. A computer virus [11], [18], [19], [80] is a piece of code that has the ability to reproduce by attaching itself to other programs, disks, data files, etc. Most viruses are *malicious*, performing destructive operations on infected systems, although *good* viruses have also been discussed.

Virus writers employ many obfuscation-like techniques to protect a virus from detection and tamper-proofing-like techniques to protect it from being easily removed from the infected host program. So-called *armored viruses* add extra code to a virus to make it difficult to analyze. *Polymorphic* (or *self-mutating*) viruses generate new versions of themselves as part of the infection process. This is, in many ways, similar to Aucsmith's technique, although viruses tend to mutate only on infection while Aucsmith mutates the code continuously.

5 Discussion

We have identified three types of attacks by malicious hosts on the intellectual property contained in software. Any of these attacks may be dissuaded by legal means if the software is protected by patent, copyright, contract, or trade secrecy laws. However, it is generally difficult to discover that an attack on intellectual property in software has occurred. After an attack is discovered, it may be expensive or even impossible to obtain a remedy in courtroom proceedings. For example, a charge of copyright infringement may fail if reverse engineering is accepted by the court as a defense. For these reasons, we believe that technical defenses (known in legal circles as "self-help") will continue to be important for any software developer who is concerned about malicious hosts.

The most common attack on intellectual property in software is software piracy. This typically takes the form of unauthorized copying. Nowadays, most licensed software has a weak form of technical protection against illegal copying, typically a password activation scheme. Such schemes can generally be circumvented easily by an expert hacker [47], or indeed by anyone who is willing to undertake a search for "warez" sites and "cracks" newsgroups on the Internet [72].

Software watermarking provides an alternate form of protection against piracy. To the extent that a watermark is stealthy, a software pirate will unwittingly copy the watermark along with the software being stolen. To the extent that a watermarks is resilient, it will survive a pirate's attempts at removal. The watermark must also be detectable by the original developer of the software. In this paper, we have argued that dynamic watermarking techniques are more stealthy and more resilient than the existing alternative technology of static watermarks.

A second form of attack on intellectual property in software is reverse engineering. A malicious reverse engineer seeks to understand a software product well enough to use its secret methodology without negotiating for a license. Reverse engineers can be discouraged slightly by lexical transformations on the software, such as the scrambling or "stripping" of variable names. In this paper, we have described many other, more powerful obfuscations that obscure the control and data structures of the software.

We identify tampering as a third form of attack on intellectual property in software. Sometimes tampering will occur in conjunction with the other forms of attack. For example, a reverse engineer may tamper with code in order to extract the modules of interest, or in order to "see how it works." Also, a software pirate may tamper with code in an attempt to remove its watermark. However, tampering may occur independently of the other attacks, for example, if someone wishes to corrupt an e-commerce application so that it provides unauthorized discounts or free services. In all cases, an appropriate technical self-help is to render the code tamper-proof. If a tamper-proof code is modified in any way, it will no longer be functional. In this paper, we have described several previously published methods for tamper-proofing code.

All of the methods described in this paper provide at least a modicum of protection for software against attacks by malicious hosts. Future research will show exactly which attacks these methods are vulnerable to and to what extent they can be improved. Particularly interesting are recent theoretical results [7], [36], [61] which point the way toward a deeper understanding of the limits of software protection.

ACKNOWLEDGMENTS

The authors are grateful for the extensive and insightful comments of two anonymous referees. This material is based upon work supported by the US National Science Foundation under Grant No. 0073483.

REFERENCES

[1] 4C Entity, "Content Protection System Architecture,"revision 0.81, available http://www.4centity.com/data/tech/cpsa/cpsa081.pdf, Aug. 2001.

[2] M. Abadi and J. Feigenbaum, "Secure Circuit Evaluation: A Protocol Based on Hiding Information from an Oracle," *J. Cryptology,* vol. 2, no. 1, pp. 1–12, 1990.

[3] R.J. Anderson and F.A.P. Peticolas, "On the Limits of Steganography," *IEEE J. Selected Areas Comm.,* vol. 16, no. 4, May 1998.

[4] Atari Games Corp. and Tengen, Inc. v. Nintendo of America Inc. and Nintendo Co., Ltd., United States Court of Appeals for the Federal Circuit, Sept. 1992.

[5] D. Aucsmith, "Tamper Resistant Software: An Implementation," *Information Hiding, First Int'l Workshop,* R.J. Anderson, ed., pp. 317–333, May 1996.

[6] D. Aucsmith and G. Graunke, "Tamper Resistant Methods and Apparatus," US patent 5,892,899, Assignee: Intel Corporation, 1999.

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)possibility of Obfuscating Programs (Extended Abstract)," *Advances in Cryptology—CRYPTO 2001,* J. Kilian, ed., 2001.

[8] A. Beimel, M. Burmester, Y. Desmedt, and E. Kushilevitz, "Computing Functions of a Shared Secret," *SIAM J. Discrete Math.,* vol. 13, no. 3, pp. 324–345, 2000.

[9] W. Bender, D. Gruhl, N. Morimoto, and A. Lu, "Techniques for Data Hiding," *IBM Systems J.,* vol. 35, nos. 3 & 4, pp. 313–336, 1996.

[10] P. Bieber, J. Cazin, P. Girard, J.L. Lanet, V. Wiels, and G. Zanon, "Checking Secure Interactions of Smart Card Applets," *Proc. Sixth European Symp. Research in Computer Security (ESORICS),* 2000.

[11] M. Bishop, "An Overview of Computer Viruses in a Research Environment," technical report, Dept. of Math. and Computer Science, Dartmouth College, 1992.

[12] M. Blum, "Program Result Checking: A New Approach to Making Programs More Reliable," *Proc. 20th Int'l Colloquium Automata, Languages, and Programming,* S. Carlsson, A. Lingas, and R.G. Karlsson, eds., pp. 1–14, July 1993.

[13] M. Blum and S. Kannan, "Designing Programs that Check Their Work," *J. ACM,* vol. 42, no. 1, pp. 269–291, Jan. 1995.

[14] D Boneh and M.K. Franklin, "An Efficient Public Key Traitor Tracing Scheme," *Advances in Cryptology—Crypto '99,* pp. 338–353, 1999.

[15] T.A. Budd, "Protecting and Managing Electronic Content with a Digital Battery," *Computer,* vol. 34, no. 8, pp. 2–8, Aug. 2001.

[16] L.K. Chen, "Computer Software Protection against Piracy in Taiwan," *J. Asian Law,* vol. 8, no. 1, 1994, http://www.columbia.edu/cu/asiaweb/v8n1chen. htm.

[17] D.M. Chess, "Security Issues in Mobile Code Systems," *Mobile Agents and Security,* pp. 1–8, 1998.

[18] F. Cohen, "Computer Viruses—Theory and Experiments," *IFIP-TC11, Computers and Security,* pp. 22–35, 1987.

[19] F. Cohen, "Current Trends in Computer Viruses," *Proc. Int'l Symp. Information Security,* 1991.

[20] F.B. Cohen, *Operating System Protection through Program Evolution.* http://all.net/books/IP/evolve.html, 1992.

[21] C. Collberg, "The Obfuscation and Software Watermarking Home Page," http://www.cs.arizona.edu/collberg/Research/Obfuscation/index.html, 1999.

[22] C. Collberg and C. Thomborson, "Software Watermarking: Models and Dynamic Embeddings," *Principles of Programming Languages (POPL '99),* Jan. 1999, http://www.cs.auckland.ac.nz/collberg/Research/nz/~collberg/Research/Publications/CollbergThomborson99a/index.html.

[23] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Technical Report 148, Dept. of Computer Science, Univ. of Auckland, July 1997, http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a.

[24] C. Collberg, C. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures," *Proc. IEEE Int'l Conf. Computer Languages (ICCL '98),* May 1998, http://www.cs.auckland.ac.nz/collberg/Research/Publications/CollbergThomborsonLow98b/.

[25] C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," *Proc. Symp. Principles of Programming Languages (POPL '98),* Jan. 1998. http://www.cs.auckland.ac.nz/collberg/Research/Publications/CollbergThomborsonLow98a/.

[26] Compaq, "FreePort Express," http://www.support.compaq.com/amt/freeport/.

[27] Convera, "Software Integrity System,"http://convera.com/.

[28] R.L. Davidson and N. Myhrvold, "Method and System for Generating and Auditing a Signature for a Computer Program," US Patent 5,559,884, Assignee: Microsoft Corporation, Sept. 1996.

[29] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond *k*-Limiting," *Proc. SIGPLAN Conf. Programming Language Design and Implementation (PLDI '94),* pp. 230–241, June 1994.

[30] F. Ergün, S. Kannan, S.R. Kumar, R. Rubinfeld, and M. Vishwanathan, "Spot-Checkers," *Proc. 30th Ann. ACM Symp. Theory of Computing (STOC '98),* pp. 259–268, May 1998.

[31] Ernie , "Disk Copy Protection," Mar. 1997, Usenet:comp. misc, http://groups.google.com/groups?selm=33256CC1.7EE040mitre.org.

[32] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM,* vol. 43, no. 3, pp. 431–473, 1996.

[33] R.D. Gopal and G.L. Sanders, "Global Software Piracy: You Can't Get Blood Out of Turnip," *Comm. ACM,* vol. 43, no. 9, pp. 83–89, Sept. 2000.

[34] J.R. Gosler, "Software Protection: Myth or Reality? *CRYPTO'85—Advances in Cryptology,* pp. 140–157, Aug. 1985.

[35] D. Grover, "Program Identification," *The Protection of Computer Software: Its Technology and Applications,* The British Computer Soc. Monographs in Informatics, Cambridge Univ. Press, second ed., 1992.

[36] S. Hada, "Zero-Knowledge and Code Obfuscation," *AsiaCrypt 2000,* pp. 443–457, 2000, http://link.springer.de/link/service/series/0558/papers/1976/19760443.pdf.

[37] A. Herzberg and S.S. Pinter, "Public Protection of Software," *ACM Trans. Computer Systems,* vol. 5, no. 4, pp. 371–393, Nov. 1987.

[38] F. Hohl, "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts," *Mobile Agents and Security,* pp. 92–113, vol. 1419, Lecture Notes in Computer Science, Springer-Verlag, 1998.

[39] F. Hohl, "A Framework to Protect Mobile Agents by Using Reference States," *Proc. 20th Int'l Conf. Distributed Computing Systems,* pp. 410–417, 2000.

[40] K. Holmes, "Computer Software Protection," US Patent 5,287,407, Assignee: International Business Machines, Feb. 1994.

[41] S. Horwitz, "Precise Flow-Insensitive May-Alias Analysis is NP-Hard," *TOPLAS,* vol. 19, no. 1, pp. 1–6, Jan. 1997.

[42] J.D. Howard, "An Analysis of Security Incidents on the Internet, 1989-1995," PhD thesis, Dept. of Eng. and Public Policy, Carnegie-Mellon Univ., Apr. 1997.

[43] IBM, "Cryptolopes,"http://www.ibm.com/software/security/cryptolope/.

[44] InterTrust, "Digital Rights Management," http://www.intertrust.com/de/index.html.

[45] N.F. Johnson and S. Jajodia, "Computing Practices: Exploring Steganography: Seeing the Unseen," *Computer,* vol. 31, no. 2, pp. 26–34, Feb. 1998, http://www.isse.gmu.edu/njohnson/pub/r2026.pdf.

[46] A.B. Kahng, J. Lach, W.H. Mangione-Smith, S. Mantik, I.L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Watermarking Techniques for Intellectual Property Protection," *Proc. 35th ACM/IEEE DAC Design Automation Conf. (DAC '98),* pp. 776–781, June 1999, http://www.cs.ucla.edu/gangqu/ipp/c79.ps.gz.

[47] M.D. LaDue, "The Maginot License: Failed Approaches to Licensing Java Software over the Internet," http://www.geocities.com/securejavaapplets/maginot.html, Copyright 1997.

[48] C.E. Landwehr, A.R. Bull, J.P. McDermott, and W.S. Choi, "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys,* vol. 26, no. 3, pp. 211–254, Sept. 1994.

[49] D. Libes, *Obfuscated C and Other Mysteries.* Wiley, 1993.

[50] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *Architectural Support for Programming Languages and Operating Systems,* pp. 168–177, Nov. 2000.

[51] M. Limayem, M. Khalifa, and W.W. Chin, "Factors Motivating Software Piracy: A Longitudinal Study," *Proc. 20th Int'l Conf. Information Systems,* pp. 124–131, 1999.
[52] U. Lindqvist and E. Jonsson, "How to Systematically Classify Computer Security Intrusions," *Proc. 1997 IEEE Symp. Security and Privacy,* pp. 154–163, 1997, http://www.ce.chalmers.se/staff/ulfl/pubs/sp97.
[53] S. Lucco, R. Wahbe, and O. Sharp, "Omniware: A Universal Substrate for Web Programming," *Proc. WWW4,* 1995.
[54] S. Macrakis, "Protecting Source Code with ANDF," ftp://riftp.osf.org/pub/andf/andf_coll_papers/ProtectingSourceCode.ps, Jan. 1993.
[55] Apple's QuickTime lawsuit, http://www.macworld.com/pages/june.95/News.848.html and may.95/News.705.html, May–June 1995.
[56] Y. Malhotra, "Controlling Copyright Infringements of Intellectual Property: The Case of Computer Software," *J. Systems Management,* part 1, vol. 45, no. 6, pp. 32–35, June 1994, part 2: no. 7, pp. 12–17, July 1994.
[57] J. Martin, "Pursuing Pirates (Unauthorized Software Copying)," *Datamation,* vol. 35, no. 15, pp. 41–42, Aug. 1989.
[58] T. Maude and D. Maude, "Hardware Protection against Software Piracy," *Comm. ACM,* vol. 27, no. 9, pp. 950–959, Sept. 1984.
[59] R. Mori and M. Kawahara, "Superdistribution: The Concept and the Architecture," Technical Report 7, Inst. of Information Sci. & Electron, Tsukuba Univ., Japan, July 1990, http://www.site.gmu.edu/bcox/ElectronicFrontier/MoriSuperdist.html.
[60] S.A. Moskowitz and M. Cooperman, "Method for Stega-Cipher Protection of Computer Code," US Patent 5,745,569, Assignee: The Dice Company, Jan. 1996.
[61] D. Naccache, A. Shamir, and J.P. Stern, "How to Copyright a Function?" *Public Key Encryption '99,* Hideki Imai, ed., Lecture Notes in Computer Science, Springer-Verlag, 1999.
[62] D. Nagy-Farkas, "The Easter Egg Archive," http://www.eeggs.com/lr.html, 1998.
[63] NetSafe, "EXE Guardian™,"http://members.ozemail.com.au/netsafe/guardian_detailed_information.html, Copyright 1996.
[64] P.G. Neumann, *Computer-Related Risks.* ACM Press, 1995.
[65] L.C. Noll, S. Cooper, P. Seebach, and L.A. Broukhis, "The International Obfuscated C Code Contest," http://www.ioccc.org/index.html, 2000.
[66] F.A.P. Peticolas, R.J. Anderson, and M.G. Kuhn, "Attacks on Copyright Marking Systems," *Proc. Second Workshop Information Hiding,* Apr. 1998.
[67] International Planning and Research Corporation, "Sixth Annual BSA Global Software Piracy Study," http://www.bsa.org/resources/2001-05-21.55.pdf, 2001.
[68] T.A. Proebsting and S.A. Watterson, "Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)," *Proc. Third USENIX Conf. Object-Oriented Technologies and Systems (COOTS),* June 1997.
[69] G. Qu and M. Potkonjak, "Analysis of Watermarking Techniques for Graph Coloring Problem," *Proc. IEEE/ACM Int'l Conf. Computer Aided Design,* pp. 190–193, Nov. 1998, http://www.cs.ucla.edu/gangqu/publication/gc. ps.gz.
[70] G. Ramalingam, "The Undecidability of Aliasing," *ACM Trans. Programming Languages and Systems,* vol. 16, no. 5, pp. 1467–1471, Sept. 1994.
[71] R. Rivest, "The MD5 Message-Digest Algorithm," The Internet Eng. Task Force RFC 1321, http://www.ietf.org/rfc/rfc1321.txt, 1992.
[72] H. Rosner, "Steal this Software," *The.Standard.com,* June 2000. http://www.thestandard.com/article/article_print/1,1153,16039,00.html.
[73] R. Rubinfeld, "Batch Checking with Applications to Linear Functions," *Information Processing Letters,* vol. 42, no. 2, pp. 77–80, May 1992.
[74] R. Rubinfeld, "Designing Checkers for Programs that Run in Parallel," *Algorithmica,* vol. 15, no. 4, pp. 287–301, Apr. 1996.
[75] P.R. Samson, "Apparatus and Method for Serializing and Validating Copies of Computer Software," US Patent 5,287,408, Assignee: Autodesk, Inc., Feb. 1994.
[76] P. Samuelson, "Reverse-Engineering Someone Else's Software: Is It Legal?" *IEEE Software,* pp. 90–96, Jan. 1990.
[77] T. Sander and C.F. Tschudin, "Protecting Mobile Agents against Malicious Hosts," *Mobile Agents and Security,* 1998.
[78] Sega Enterprises Ltd. v. Accolade, Inc., United States Court of Appeals for the Ninth Circuit, July 1992.
[79] S.S. Simmel and I. Godard, "Metering and Licensing of Resources—Kala's General Purpose Approach," *Technological Strategies for Protecting Intellectual Property in the Networked Multimedia Environment,* pp. 81–110, Jan. 1994.
[80] E.H. Spafford, "Computer Viruses as Artificial Life," *Artificial Life,* vol. 1, no. 3, pp. 249–265, 1994.
[81] R. Stallman, "Why Software Should not Have Owners," http://www.gnu.org/philosophy/why-free.html, 1994.
[82] M. Swanson and B. Guttman, "Generally Accepted Principles and Practices for Securing Information Technology Systems," technical report, Nat'l Inst. Standards and Technology, Dept. of Commerce, US Government, Sept. 1996, http://www. auerbach-publications. com/white-papers/nist-security-guidelines.pdf.
[83] Locksmith Tools, "Advanced Detailed Description of Dinkey Dongle for Software Protection, Software Security, and License Management," Copyright 1988-1999, http://www.locksmithshop.com/lsdddetail.htm.
[84] A. Torrubia and F.J. Mora, "Information Security in Multiprocessor Systems Based on the x86 Architecture," *Computers and Security,* vol. 19, no. 6, pp. 559–563, Oct. 2000.
[85] R.E. Vaughn, "Defining Terms in the Intellectual Property Protection Debate: Are the North and South Arguing Past Each Other When We Say 'Property'? A Lockean, Confucian, and Islamic Comparison," *ILSA J. Comparative and Int'l Law,* vol. 2, no. 2, p. 308, Winter 1996, http://www.nsulaw.nova.edu/student/organizations/ILSAJournal/2-2/Vaughan202-2.htm.
[86] R. Venkatesan, V. Vazirani, and S. Sinha, "A Graph Theoretic Approach to Software Watermarking," *Proc. Fourth Int'l Information Hiding Workshop,* Apr. 2001.
[87] Vermont Microsystems Inc. v. AutoDesk Inc., United States Court of Appeals for the Second Circuit, Jan. 1996.
[88] G. Vigna, "Introduction," *Mobile Agents and Security,* pp. xi–xii, 1998.
[89] H.P. Van Vliet, "Mocha—The Java Decompiler," http://web.inter.nl.net/users/H.P.van.Vliet/mocha.html, Jan. 1996.
[90] R. Wahbe and S. Lucco, "Methods for Safe and Efficient Implementation of Virtual Machines," US Patent 5,761,477, Assignee: Microsoft Corporation, 1999.
[91] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation," *Proc. Symp. System Principles (SOSP '93),* pp. 203–216, 1993.
[92] C. Wang, "A Security Architecture for Survivability Mechanisms," PhD thesis, Univ. of Virginia, School of Eng. and Applied Science, Oct. 2000, www.cs.virginia.edu/survive/pub/wangthesis.pdf.
[93] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software Tamper Resistance: Obstructing Static Analysis of Programs," Technical Report CS-2000-12, Univ. of Virginia, Dec. 2000.
[94] J. Wang, "Average-Case Complexity Forum," http://www.uncg.edu/mat/acc-forum, 1999.
[95] H. Wasserman and M. Blum, "Software Reliability via Run-Time Result-Checking," *J. ACM,* vol. 44, no. 6, pp. 826–849, Nov. 1997.
[96] S.P. Weisband and S.E. Goodman, "Int'l Software Piracy," *Computer,* vol. 92, no. 11, pp. 87–90, Nov. 1992.
[97] CD Media World, "CD Protections," http://www.cdmediaworld.com/hardware/cdrom/cd_protections.shtml, Copyright 1998–2001.
[98] Xerox, "ContentGuard," http://www.contentguard.com.

Christian S. Collberg received the BSc and PhD degrees from Lund University, Sweden. He was previously on the faculty at the University of Auckland, New Zealand. Currently, he is a faculty member at the Unversity of Arizona, Tuscon. His primary research interests are compiler and programming language design, software security, and domain-specific web search engines. He is a member of the IEEE Computer Society.

Clark Thomborson (SM'96, M'86) received the PhD degree in computer science in 1980, under his birth name Clark Thompson, from Carnegie-Mellon University, Pennsylvania. He emigrated to New Zealand in 1996, to take up his current position as professor of Computer Science at the University of Auckland, New Zealand. He has published more than 70 refereed papers on topics in software security, computer systems performance analysis, VLSI algorithms, data compression, and connection networks. He is a senior member of the IEEE and a member of the IEEE Computer Society.