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

Christian S. Collberg   Clark Thomborson

February 10, 2000

University of Arizona Computer Science Technical Report
2000–03

University of Auckland Computer Science Technical Report
#170

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Department of Computer Science
University of Auckland
Auckland, New Zealand

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

Christian Collberg
Department of Computer Scince
University of Arizona
Tucson, AZ 85721
`collberg@cs.arizona.edu`

Clark Thomborson
Department of Computer Science
University of Auckland
Auckland, New Zealand
`cthombor@cs.auckland.ac.nz`

Abstract

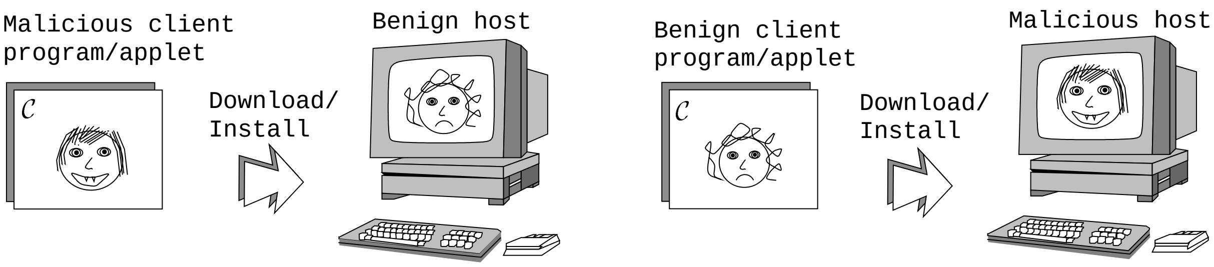
We identify three types of attack on the intellectual property contained in software, and three corresponding technical defenses. A potent defense against reverse engineering is obfuscation, a process that renders software unintelligible but still functional. A defense against software piracy is watermarking, 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 non-functional code. We briefly survey the available technology for each type of defense.

1 Background – Malicious Clients vs. Malicious Hosts

Until recently, almost all computer security research was concerned with protecting the integrity of a *benign host* from attacks by *malicious client* programs (Figure 1(a)). The Java security model, for example, 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.

Figure 1: Attacks by malicious clients and hosts.

(a) Attack by a malicious client. (b) Attack by a malicious host.



To defend itself 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 [29, 50, 51], which modifies the client code so that it is unable to write outside its designated area (the "sandbox").

Recently, researchers have begun to consider a fundamentally different kind of security threat, which we shall term a *malicious host attack* (Figure 1(b)). In a malicious host scenario, a benign client is under threat from the host on which it has been downloaded or installed. These attacks typically take 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 such scenarios.

1.1 Malicious host attacks

Software piracy, the illegal copying and resale of applications, is a 15 billion dollar per year industry [1]. 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 [19, 22, 32–35, 47, 53] 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. The remaining schemes seem to be only a minor impediment to software pirates, for whom breaking new copy protection schemes is intellectually as well as financially rewarding.

Many software developers also worry about their applications being *reverse engineered* [3,31, 44, 46, 48]. 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, programs are distributed in easily decompilable formats rather than native binary code [39, 49]. Important examples include the Java class file format and ANDF [30].

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 Figure 2:

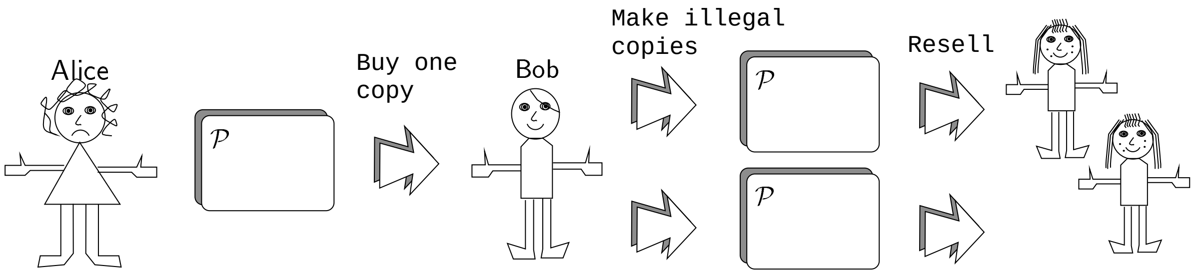
- In Figure 2(a) Bob makes copies of an application he has legally purchased from Alice, and illegally sells them to unsuspecting customers.
- In Figure 2(b) Bob decompiles and reverse engineers an application he has bought from Alice in order to reuse one of her modules in his own program.
- In Figure 2(c), finally, Bob receives a "digital container" [25, 26, 54] 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.

1.2 Defenses against malicious host attacks

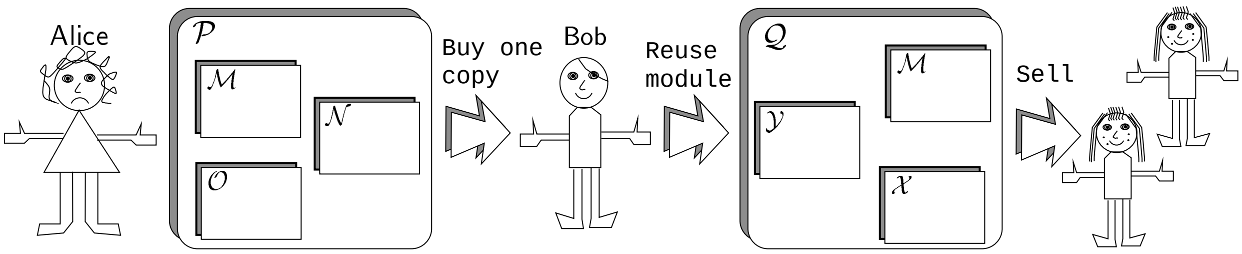
It should be noted that it is much more difficult to defend a client than it is to defend a host. To defend a host against a malicious client, all that is needed is to restrict the actions that the client is allowed to perform.

Figure 2: Attacks against software intellectual property.

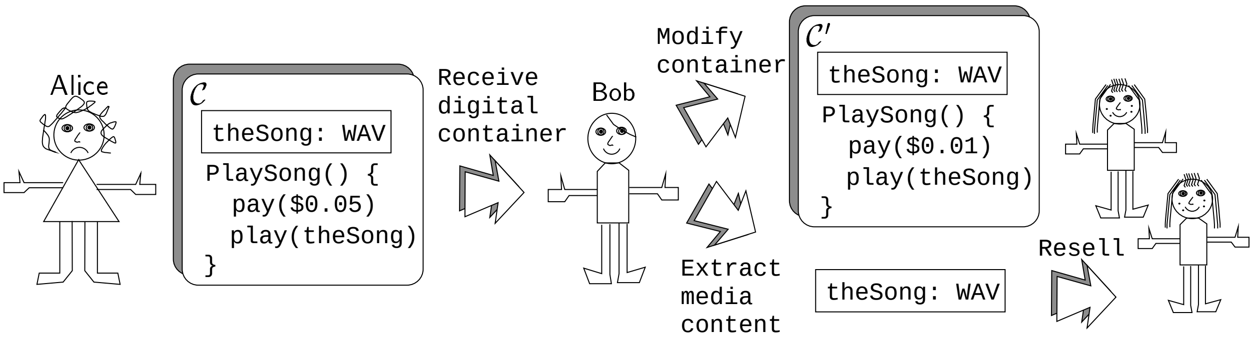
(a) Software piracy attack. Bob makes illegal copies of Alice's program $\mathcal{P}$ and resells them.



(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.

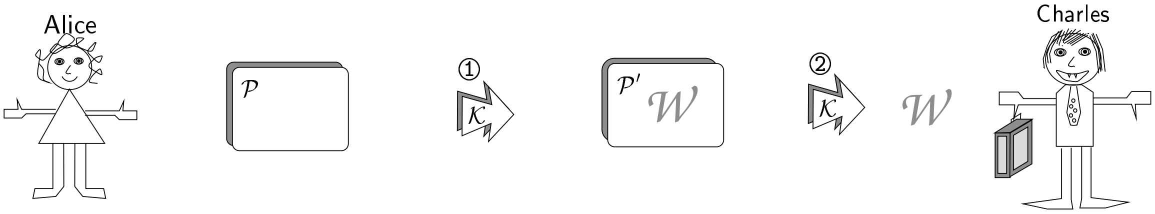


Unfortunately, no such defense is available to protect a client against a host attack. Once the client code resides on the host machine, the host can make use of *any* conceivable technique to extract sensitive data from the client, or to otherwise violate its integrity. The only limiting factors are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

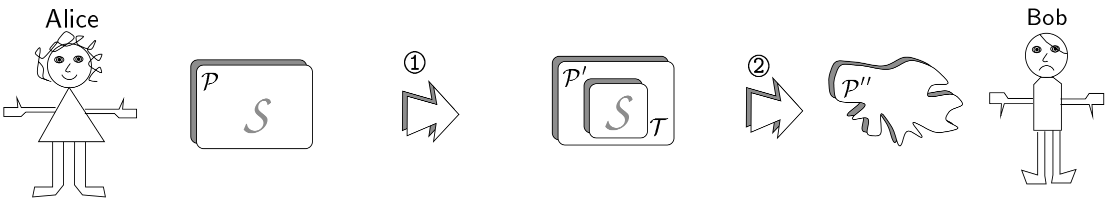
While it is generally believed that complete protection of client code is an unattainable goal, recent results (by ourselves and others) have shown that some degree of protection *can* be achieved. Recently, *software watermarking* [12, 16, 21, 36], *tamper-proofing* [4, 5, 23, 45], and *obfuscation* [10, 13–15] have emerged as alternatives to other forms of intellectual property protection of software. 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

Figure 3: Defenses against malicious host attacks.

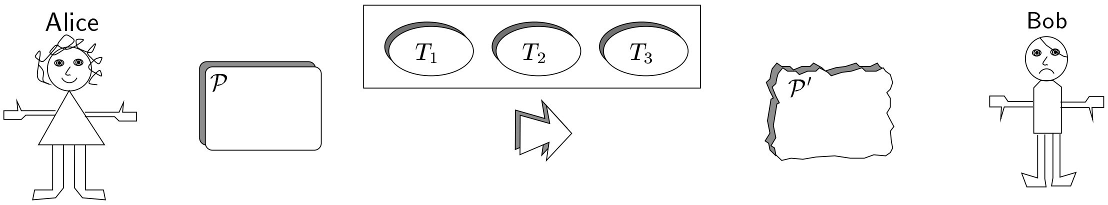
(a) Software watermarking. Alice watermarks her program using a secret key $\mathcal{K}$. Charles extracts the watermark using the same key.



(b) Tamperproofing. 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 obfuscating transformations $T_1 \cdots T_3$) to prevent Bob from reverse engineering it.



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 Figure 3:

- In Figure 3(a) 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 Figure 3(b), 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 Figure 3(c), 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.

2 Obfuscation

Security through obscurity has long been viewed with disdain in the security and cryptography communities. As we saw in Section 1.1, however, there are applications where higher levels of protection than that achievable through obscurity is simply not possible.

In [15] and [14] 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 $P' = \{\cdots, S'_j = \mathcal{T}_i(\mathcal{S}_j), \cdots\}$ 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.

2.1 Lexical Transformations

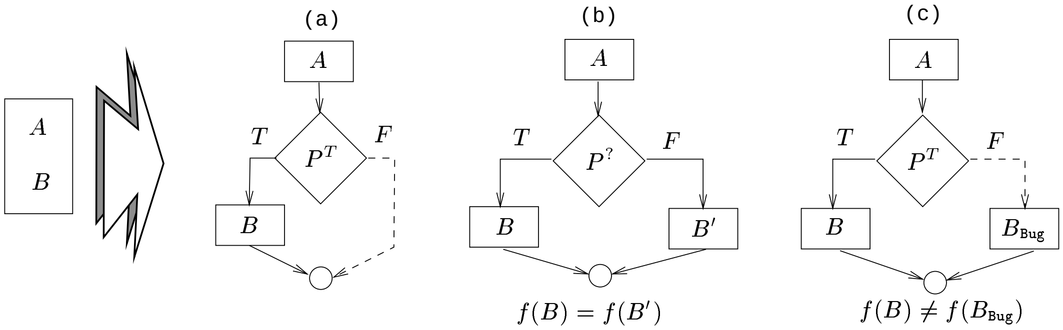
The advent of Java, whose strongly typed bytecode and architecture-neutral class files make programs easy to decompile, has left programmers scurrying for ways to protect their intellectual property. On our website [11], we list a number of "Java obfuscation tools," most of which modify only the lexical structure of the program. Typically, they do nothing more than to scramble identifiers. Such lexical transforms will surely be annoying to a reverse engineer, and therefore will prevent some theivery 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 [15] 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 (P^T) 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 Figure 4(a) we split up the block $\ulcorner A; B \urcorner$ by inserting an opaquely true predicate P^T which makes it appear as if B is only executed sometimes. In Figure 4(b), B is split into two *different* obfuscated versions B and B'. The opaque predicate $P^?$ selects either of them at runtime. In Figure 4(c), finally, P^T always selects B over $B_{\texttt{Bug}}$, a buggy version of B.

Figure 4: Control transformation by opaque predicate insertion.



There are many control transformations similar to those in Figure 4, some of which are discussed in [15]. 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. In [15] we discuss two general methods for generating resilient and cheap opaque predicates that are based on the intractability of these static analysis problems.

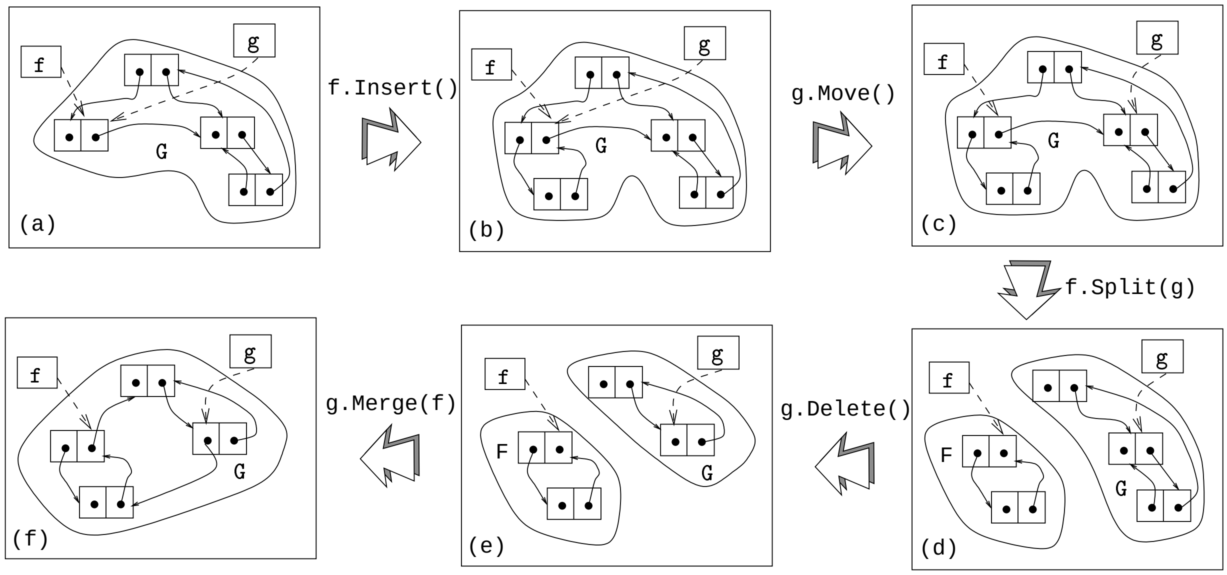
Figure 5 shows a simple example of how strong opaque predicates can be constructed based on the intractability of alias analysis. The basic idea is to extend the program to be obfuscated with code that builds a set of complex dynamic structures. A number of global pointers reference nodes within these 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 Figure 5(a) through (c) we can ask the opaque query $\ulcorner$`if (f==g)`$^{?}$ `then` $\cdots\urcorner$, 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 (c) is split into two components in (d), we can ask the query $\ulcorner$`if (f==g)`^{F} `then` $\cdots\urcorner$, since `f` and `g` now move around in different structures. Finally, in Figure 5(f), the two components have been merged, and we can again ask the query $\ulcorner$`if (f==g)`$^{?}$ `then` $\cdots\urcorner$. Current static alias analysis algorithms typically fail for `Split`, `Merge`, `Delete` and other kinds of destructive update operations.

2.3 Data Transformations

In [14] we present several transformations that obfuscate data structures. As an example, consider the *Variable Splitting* transformation in Figure 6. In this example a boolean variable V is split into two integer variables p and q, using the new representation shown in Figure 6(a).

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



Given this new representation, we create new implementations for the built-in boolean operations. Only the implementation of `&` is shown in Figure 6(b).

In Figure 6(c) 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`⌝.

3 Watermarking

Watermarking embeds a secret message into a cover message. In *media watermarking* [2,6,27,38] 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* [12,16,20,21,24,28,36,43], 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 de-watermarking attacks);
- W is large (the embedding has a high *data rate*);

Figure 6: A data transformation that splits boolean variables.

(a)

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

(b)

	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

(c)

```
(1) bool A,B,C;          (1') short a1,a2,b1,b2,c1,c2;
(2) B = False;           (2') b1=0; b2=0;
(3) C = False;     T     (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) ···;
```

- 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.

3.1 Threat-model

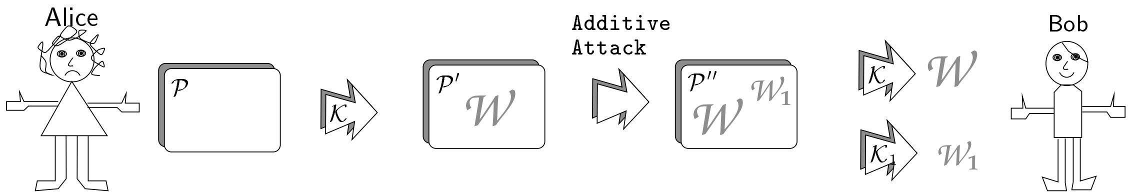
To evaluate the resilience of a watermarking technique (how well the mark will resist intentional attempts at removal), we must first define our *threat-model.* In other words, what constitutes a reasonable level of attack, and what specific techniques is an attacker likely to employ? It is generally 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. Figure 7 illustrates the kinds of de-watermarking attacks available to Bob:

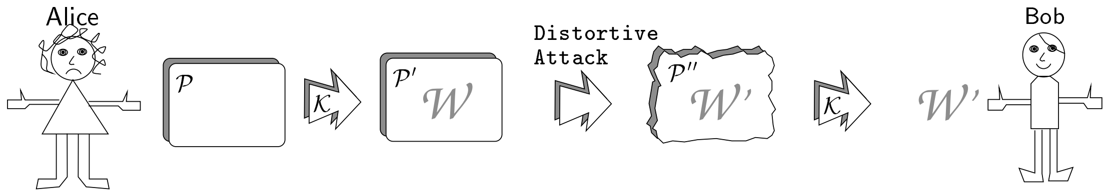
- In Figure 7(a) Bob launches an *additive attack* by adding his own watermark $\mathcal{W}_1$ to Alice's watermarked program $\mathcal{P}'$. This is an effective attack if it is impossible to detect that Alice's mark temporally precedes Bob's.
- In Figure 7(b) Bob launches a *distortive attack* on Alice's watermarked program $\mathcal{P}'$. A distortive attack applies a sequence of *semantics-preserving transformations* uniformly over

Figure 7: Attacks on watermarks and fingerprints.

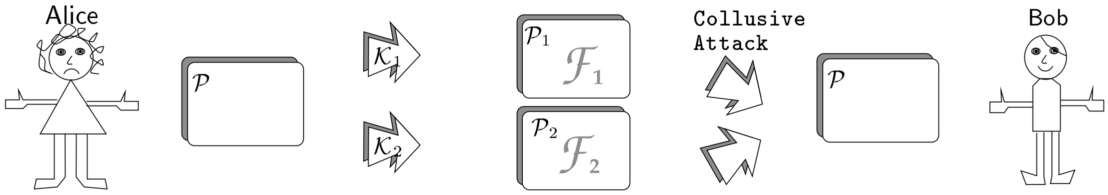
(a) An effective *additive* attack.



(b) An effective *distortive* attack.



(c) An effective *collusive* attack.



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 Figure 7(c) 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 [17]), *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 [12].

Moskowitz [36] and Davidson [16] are two techniques representative of typical static watermarks. Moskowitz describes 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 [16] describes a static code watermark in which a fingerprint is encoded in the basic block sequence of a program's control flow graphs.

Unfortunately, all static watermarks are susceptible to simple distortive de-watermarking 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 [37].

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

3.4 Dynamic Graph Watermarking

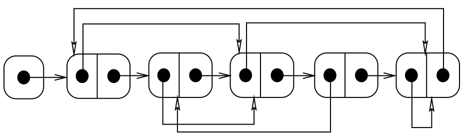
In [12] we describe a new Data Structure watermarking technique called *Dynamic Graph Watermarking*. The central idea is to embed a watermark in the *topology* of a dynamically built graph structure. Code that builds this graph is then inserted into the program to be watermarked. Because of pointer aliasing effects, the graph-building code will be hard to analyze and detect, and it can be shown that it will be impervious to most de-watermarking 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. Figure 8(a) shows a Radix-k embedding in a circular linked list, and Figure 8(b) shows how we can embed n by selecting the n:th graph

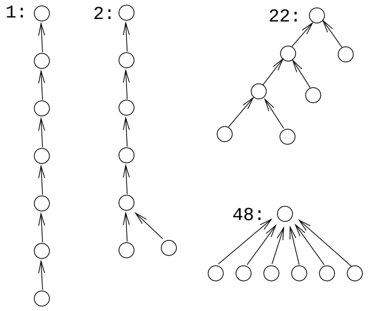
Figure 8: Graphic embeddings of watermarks.

(a) Radix-6 encoding. The right pointer field holds the `next` field, the left pointer encodes a base-k digit.



$61 \times 73 = 3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0$

(b) Enumeration encoding. These are the 1st, 2nd, 22nd, and 48th trees in an enumeration of the oriented trees with seven vertices.



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 [12] 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 ⌜`if (tampered_with()) 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 [40] 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* [7–9, 18, 41, 42, 52] and has been touted as an alternative to program verification and testing.

3. We can generate the executable on the fly, in the hope that even minor changes to the generating program will produce code that cannot be executed.

Aucsmith [4, 5] was the first to suggest 3. Intel's implementation (for their *content protection architecture* [25]) 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 pseudo-random values generated from a key. After a segment has been executed it is re-encrypted so that only one segment is ever in plaintext. 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.

Aucsmith's technique is not well suited to type-safe distribution formats such as Java bytecode. While generating, loading, and jumping to a block of Java bytecode on the fly is possible, it cannot be done stealthily, since it will always involve a call to a class loader from the standard Java library.

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 [12], in the context of tamper-proofing software watermarks.

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 or trade secrecy laws. However legal defenses are not always feasible or economical. 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 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 unauthorised 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 anyone who is willing to undertake a search through the "cracks" newsgroups on the internet.

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 our 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 unauthorised 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 tamperproofing 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 which extent they can be improved.

References

[1] Business Software Alliance. The cost of software piracy: BSA's global enforcement policy. `http://www.rad.net.id/bsa/piracy/globalfact.html`, 1996.

[2] Ross J. Anderson and Fabien A.P. Peticolas. On the limits of steganography. *IEEE J-SAC*, 16(4), May 1998.

[3] Atari games corp. and Tengen, inc. v. Nintendo of America inc. and Nintendo co., ltd., September 1992.

[4] David Aucsmith. Tamper resistant software: An implementation. In Ross J. Anderson, editor, *Information Hiding, First International Workshop*, pages 317–333, Cambridge, U.K., May 1996. Springer-Verlag. Lecture Notes in Computer Science, Vol. 1174.

[5] David Aucsmith and Gary Graunke. Tamper resistant methods and apparatus. US patent 5,892,899, 1999. Assignee: Intel Corporation.

[6] W. Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *IBM Systems Journal*, 35(3&4):313–336, 1996.

[7] Manuel Blum. Program checking. In Somenath Biswas and Kesav V. Nori, editors, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 560 of *LNCS*, pages 1–9, Berlin, Germany, December 1991. Springer.

[8] Manuel Blum. Program result checking: A new approach to making programs more reliable. In Svante Carlsson Andrzej Lingas, Rolf G. Karlsson, editor, *Automata, Languages and Programming, 20th International Colloquium*, volume 700 of *Lecture Notes in Computer Science*, pages 1–14, Lund, Sweden, 5–9 July 1993. Springer-Verlag.

[9] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995.

[10] Frederick B. Cohen. Operating system protection through program evolution. `http://all.net/books/IP/evolve.html`, 1992.

[11] Christian Collberg. The obfuscation and software watermarking home page. `http://www.cs.arizona.edu/~collberg/Research/Obfuscation/index.html`, 1999.

[12] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, San Antonio, TX, January 1999. `http://www.cs.auckland.ac.nz/~collberg/Research/ Publications/CollbergThomborson99a/index.html`.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergTh%omborsonLow97a`.

[14] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages, ICCL'98*, Chicago, IL, May 1998. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergTh%omborsonLow98b/`.

[15] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1998. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergTh%omborsonLow98a/`.

[16] Robert L. Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, September 1996. Assignee: Microsoft Corporation.

[17] Compaq Digital. Freeport express. `http://www.digital.com/amt/freeport/index.html`.

[18] Funda Ergün, Sampath Kannan, S. Ravi Kumar, Ronitt Rubinfeld, and Mahesh Vishwanathan. Spot-checkers. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 259–268, New York, May 23–26 1998. ACM Press.

[19] James R. Gosler. Software protection: Myth or reality? In *CRYPTO'85 — Advances in Cryptology*, pages 140–157, August 1985.

[20] G.Qu and M.Potkonjak. Analysis of watermarking techniques for graph coloring problem. In *IEEE/ACM International Conference on Computer Aided Design*, pages 190–193, November 1998. `http://www.cs.ucla.edu/~gangqu/publication/gc.ps.gz`.

[21] Derrick Grover. Program identification. In *The protection of computer software: its technology and applications*, The British Computer Society monographs in informatics. Cambridge University Press, 2nd edition, 1992. ISBN 0-521-42462-3.

[22] Amir Herzberg and Shlomit S. Pinter. Public protection of software. *ACM Transactions on Computer Systems*, 5(4):371–393, November 1987.

[23] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, pages 92–113. Springer-Verlag, 1998. Lecture Notes in Computer Science, Vol. 1419.

[24] Keith Holmes. Computer software protection. US Patent 5,287,407, February 1994. Assignee: International Business Machines.

[25] Intel. Software integrity system. `http://developer.intel.com/software/security`.

[26] InterTrust. Digital rights management. `http://www.intertrust.com/de/index.html`.

[27] Neil F. Johnson and Sushil Jajodia. Computing practices: Exploring steganography: Seeing the unseen. *Computer*, 31(2):26–34, February 1998. `http://www.isse.gmu.edu/~njohnson/pub/r2026.pdf`.

[28] 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. In *35th ACM/IEEE DAC Design Automation Conference (DAC-98)*, pages 776–781, June 1999. `http://www.cs.ucla.edu/~gangqu/ipp/c79.ps.gz`.

[29] Steven Lucco, Robert Wahbe, and Oliver Sharp. Omniware: A universal substrate for web programming. In *WWW4*, 1995.

[30] Stavros Macrakis. Protecting source code with ANDF. `ftp://riftp.osf.org/pub/andf/andf_coll_papers/ProtectingSourceCode.ps`, January 1993.

[31] Apple's QuickTime lawsuit. `http://www.macworld.com/pages/june.95/News.848.html` and `may.95/News.705.html`, May–June 1995.

[32] Y. Malhotra. Controlling copyright infringements of intellectual property: the case of computer software. *J. Syst. Manage. (USA)*, 45(6):32–35, June 1994. part 1, part 2: No 7, Jul. pp. 12–17.

[33] J. Martin. Pursuing pirates (unauthorized software copying). *Datamation*, 35(15):41–42, August 1989.

[34] Tim Maude and Derwent Maude. Hardware protection against software piracy. *Communications of the ACM*, 27(9):950–959, September 1984.

[35] Ryoichi Mori and Masaji Kawahara. Superdistribution: the concept and the architecture. Technical Report 7, Inst. of Inf. Sci. & Electron (Japan), Tsukuba Univ., Japan, July 1990. `http://www.site.gmu.edu/~bcox/ElectronicFrontier/MoriSuperdist.html`.

[36] Scott A. Moskowitz and Marc Cooperman. Method for stega-cipher protection of computer code. US Patent 5,745,569, January 1996. Assignee: The Dice Company.

[37] David Nagy-Farkas. The easter egg archive. `http://www.eeggs.com/lr.html`, 1998.

[38] Fabien A.P. Peticolas, Ross J. Anderson, and Markus G. Kuhn. Attacks on copyright marking systems. In *Second Workshop on Information Hiding*, Portland, Oregon, April 1998.

[39] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java (Does bytecode reveal source?). In *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, June 1997.

[40] Ronald Rivest. The md5 message-digest algorithm. `http://www.ietf.org/rfc/rfc1321.txt`, 1992. The Internet Engineering Task Force RFC 1321.

[41] Ronitt Rubinfeld. Batch checking with applications to linear functions. *INFORMATION PROCESSING LETTERS*, 42(2):77–80, May 1992.

[42] Ronitt Rubinfeld. Designing checkers for programs that run in parallel. *ALGORITHMICA*, 15(4):287–301, April 1996.

[43] Peter R. Samson. Apparatus and method for serializing and validating copies of computer software. US Patent 5,287,408, February 1994. Assignee: Autodesk, Inc.

[44] Pamela Samuelson. Reverse-engineering someone else's software: Is it legal? *IEEE Software*, pages 90–96, January 1990.

[45] T. Sander and Chr. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, 1998. Springer-Verlag, Lecture Notes in Computer Science 1419.

[46] Sega enterprises ltd. v. Accolade, inc., July 1992.

[47] Sergiu S. Simmel and Ivan Godard. Metering and Licensing of Resources - Kala's General Purpose Approach. In *Technological Strategies for Protecting Intellectual Property in the Networked Multimedia Environment*, The Journal of the Interactive Multimedia Association Intellectual Property Project, Coalition for Networked Information, pages 81–110, MIT, Program on Digital Open High-Resolution Systems, January 1994. Interactive Multimedia Association, John F. Kennedy School of Government.

[48] Vermont Microsystems inc. v. AutoDesk inc., January 1996.

[49] Hans Peter Van Vliet. Mocha — The Java decompiler. `http://web.inter.nl.net/users/H.P.van.Vliet/mocha.html`, January 1996.

[50] Robert Wahbe and Steven Lucco. Methods for safe and efficient implementation of virtual machines. US Patent 5,761,477, 1999. Assignee: Microsoft Corporation.

[51] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *SOSP'93*, pages 203–216, 1993.

[52] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[53] S. P. Weisband and Seymour E. Goodman. International software piracy. *Computer*, 92(11):87–90, November 1992.

[54] Xerox. ContentGuard. `http://www.contentguard.com/productmenu.htm`.