Software Watermarking: Models and Dynamic Embeddings

Christian Collberg* Clark Thomborson

Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland, New Zealand.
{*collberg, cthombor*}*@cs.auckland.ac.nz*

Abstract

Watermarking embeds a secret message into a cover message. In media watermarking the secret is usually a copyright notice and the cover a digital image. Watermarking an object discourages intellectual property theft, or when such theft has occurred, allows us to prove ownership.

The Software Watermarking problem can be described as follows. Embed a structure W into a program P such that: W can be reliably located and extracted from P even after P has been subjected to code transformations such as translation, optimization and obfuscation; W is stealthy; W has a high data rate; embedding W into P does not adversely affect the performance of P; and W has a mathematical property that allows us to argue that its presence in P is the result of deliberate actions.

In the first part of the paper we construct an informal taxonomy of software watermarking techniques. In the second part we formalize these results. Finally, we propose a new software watermarking technique in which a dynamic graphic watermark is stored in the execution state of a program.

1 Introduction

Apart from Grover [16] and a few recent US patents [10,21, 28,33], very little (publicly available) information seems to exist on *software watermarking* in which a copyright notice or customer identification number is embedded into a program. This is in contrast to media watermarking which is a very active area of research [4,6,22,30].

In the present paper we will try to bring together what little information does exist in the form of a taxonomy of software watermarking techniques, provide a formalization of software watermarking, and present new results on *dynamic data structure watermarking.*

*Author's present address: Department of Computer Science, University of Arizona, Tucson, AZ 85721. email: collberg@cs.arizona.edu

1.1 Attacks on Watermarking Systems

The strength of any steganographic system is a function of its *data rate*, *stealth*, and *resilience*. The data rate expresses the quantity of hidden data that can be embedded within the cover message, the stealth expresses how imperceptible the embedded data is to an observer, and the resilience expresses the hidden message's degree of immunity to attack by an adversary. All steganographic systems exhibit a trade-off between these three metrics in that a high data rate implies low stealth and resilience. For example, the resilience of a watermark can easily be increased by exploiting redundancy (i.e. including it several times in the host message) but this will result in a reduction in bandwidth.

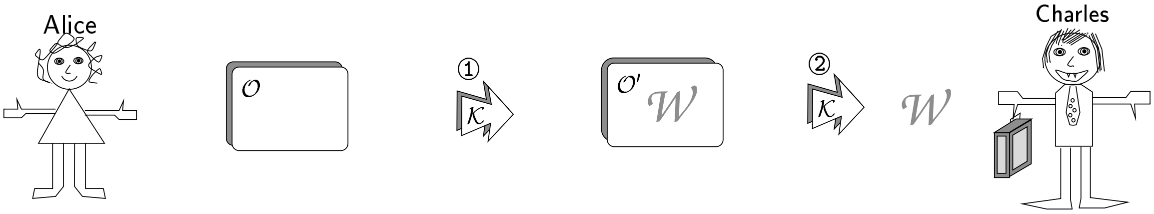
To evaluate the quality of a watermarking scheme we must also know how well it stands up to *different types* of attacks. In general, no steganographic scheme is immune to all attacks, and often several techniques have to be employed simultaneously to attain the required degree of resilience. In [6] Bender writes about media watermarking: "[] all of the proposed methods have limitations. The goal of achieving protection of large amounts of embedded data against intentional attempts at removal may be unobtainable."

To illustrate these concepts we will assume the following scenario. Alice watermarks a host object $\mathcal{O}$ with watermark $\mathcal{W}$ and key $\mathcal{K}$, and then sells $\mathcal{O}$ to Bob. Before Bob can sell $\mathcal{O}$ on to Douglas he must ensure that the watermark has been rendered useless, or else Alice will be able to prove that her intellectual property rights have been violated. Figure 1 shows the three principal kinds of attacks Bob can launch against the watermark:

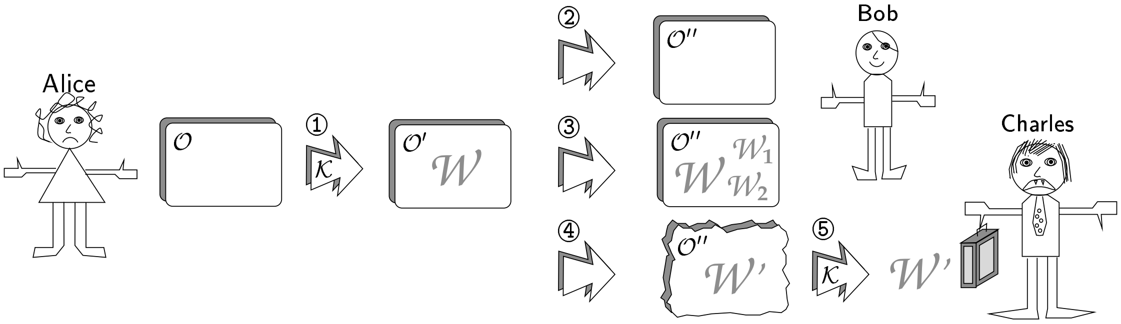
subtractive attack If Bob can detect the presence and (approximate) location of $\mathcal{W}$, he may try to *crop* it out of $\mathcal{O}$. An *effective* subtractive attack is one where the cropped object has retained enough original content to still be of value to Bob.

distortive attack If Bob cannot locate $\mathcal{W}$ and is willing to accept some degradation in quality of $\mathcal{O}$, he can apply distortive transformations uniformly over the object and, hence, to any watermark it may contain. An *effective* distortive attack is one where Alice can no longer detect the degraded watermark, but the degraded object still has value to Bob.

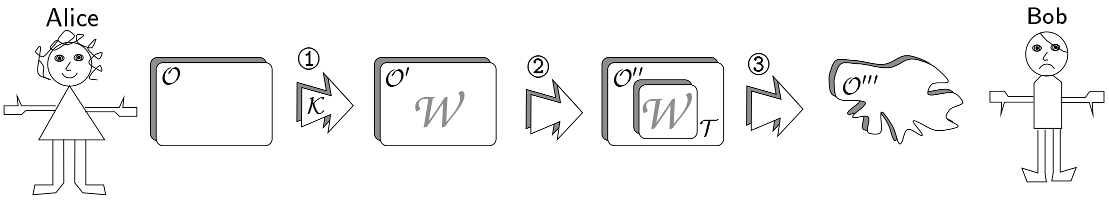
additive attack Finally, Bob can augment $\mathcal{O}$ by inserting his own watermark $\mathcal{W}'$ (or several such marks). An *effective* additive attack is one in which Bob's mark completely overrides Alice's original mark so that it can no longer be extracted, or where it is impossible to detect that Alice's mark temporally precedes Bob's.



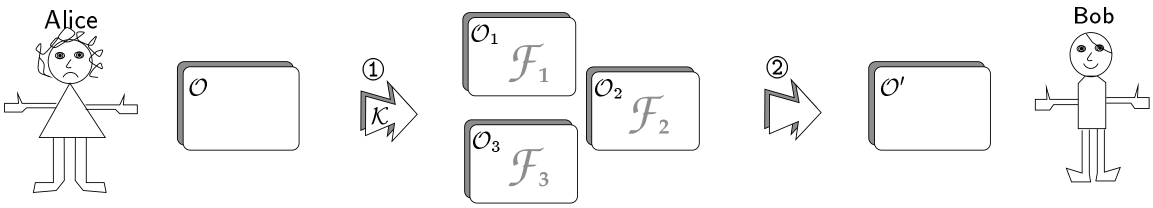
(a) At ① Alice creates a watermarked object $\mathcal{O}'$, by adding watermark $\mathcal{W}$ using key $\mathcal{K}$ to her original object $\mathcal{O}$. At ② Bob steals a copy of $\mathcal{O}'$ and Charles extracts its watermark using $\mathcal{K}$ to show that $\mathcal{O}'$ is owned by Alice.



(b) ② shows an effective *subtractive* attack, where Bob successfully removes $\mathcal{W}$ from $\mathcal{O}'$. ③ shows an effective *additive* attack, where Bob adds new watermarks $\mathcal{W}_1$ and $\mathcal{W}_2$ to make it hard for Charles to prove that $\mathcal{W}$ is Alice's original watermark. ④ shows an effective *distortive* attack, where Bob transforms $\mathcal{O}'$ (and $\mathcal{W}$) to make it difficult for Charles to detect or extract $\mathcal{W}$. At ⑤ Charles attempts to extract $\mathcal{W}$ from the distorted object, and either fails completely or gets a distorted watermark.



(c) At ② Alice adds tamperproofing $\mathcal{T}$ to her watermark. ③ shows an ineffective *subtractive* attack, where Bob tries to remove $\mathcal{W}$ from $\mathcal{O}''$, but, due to the tamper-proofing, $\mathcal{O}'''$ is rendered useless.



(d) At ① Alice creates several versions of $\mathcal{O}$, each with a different fingerprint (serial-number) $\mathcal{F}$. ② shows a *collusive* attack, where Bob is able to remove the fingerprint by comparing $\mathcal{O}_1$, $\mathcal{O}_2$, and $\mathcal{O}_3$.

Figure 1: Attacks on watermarks and counter-measures against such attacks.

Alice might, in some cases, be able to *tamperproof* her object against attacks from Bob. Tamperproofing is any technique used by Alice specifically to render de-watermarking attacks ineffective.

Most media watermarking schemes seem vulnerable to attack by distortion. For example, image transforms (such as cropping and lossy compression) will distort an image enough to render many watermarks unrecoverable [4,30].

1.2 Attacks on Fingerprinting Systems

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 would include vendor, product, and customer identification numbers.

Fingerprinting objects makes them vulnerable to *collusion attacks*. As shown in Figure 1(d), an adversary might attempt to gain access to several fingerprinted copies of an object, compare them to determine the location of the fingerprints, and, as a result, be able to reconstruct the original object.

1.3 Software Watermarking

Our interest is the watermarking and fingerprinting of *software*. Although much has been written about protection against software piracy [2,18–20,26,27,34], software watermarking is an area that has received very little attention. This is unfortunate since software piracy is estimated to be a 15 billion dollar per year business [3,24,25,35].

There are three main issues at stake when designing a software watermarking technique:

required data rate How large is the watermark or fingerprint compared to the size of the program?

form of cover program Will the program be distributed in a typed architecture-neutral virtual machine code or an untyped native binary code?

expected threat-model What kinds of de-watermarking attacks can we expect from Bob?

There are also logistic issues that need to be addressed. For example, how do we generate and distribute a large number of uniquely fingerprinted programs, and how do we handle bug-reports for these? This paper will ignore such complications.

In this paper we will assume that Alice's object $\mathcal{O}$ is an application distributed to Bob as a collection of Java class files. As we shall see, watermarking Java class files is at the same time easier and harder than watermarking stripped native object code. It is *harder* because class files are simple for an adversary to decompile [32] and analyze. It is *easier* because Java's strong typing allows us to rely on the integrity of heap-allocated data structures.

We will furthermore assume that a watermark or fingerprint can be encoded in no more than a thousand bits. Much smaller fingerprints will be sufficient in many cases. A 64-bit fingerprint, for example, would allow us to encode a 32-bit customer number, and 16 bits each of vendor and product identification.

Finally, 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 [12]), *optimization*, and *obfuscation* [7–9].

Based on these assumptions, we will examine various software watermarking techniques and attempt to answer the following questions:

- In what kind of language structure should the watermark be embedded?
- How do we extract the watermark and prove that it is ours?
- How do we prevent Bob from distorting the watermark?
- How does the watermark affect the performance of the program?

The remainder of the paper is structured as follows. In Section 2 we discuss static watermarking, in which marks are stored directly into the data or code sections of a native executable or class file. In Section 3 we turn to dynamic watermarking, in which marks are stored in the run-time structures of a program. In Section 4 we construct a formal model of software watermarking. In Section 5 we present a new dynamic watermarking technique that encodes watermarks in dynamic linked data structures. We show that this method, when properly tamperproofed, is resilient against many types of distortive de-watermarking attacks. In Section 6 we discuss the implications of our results and in Section 7 we summarize.

2 Static Software Watermarking

Static watermarks are stored in the application executable itself. In a Unix environment this is typically within the initialized data section (where static strings are stored), the text section (executable code), or the symbol section (debugging information) of the executable. In the case of Java, information could be hidden in any of the many sections of the class file format: constant pool table, method table, line number table, etc.

In our software watermark taxonomy we will distinguish between two basic types of static watermarks (see Figure 2): **code watermarks** which are stored in the section of the executable that contains instructions, and **data watermarks** which are stored in any other section, including headers, string sections, debugging information sections, etc.

2.1 Static Data Watermarks

Data watermarks (Figure 2 ①) are very common since they are easy to construct and recognize. For example, the JPEG group's copyright notice can be easily extracted from the *Netscape* binary:

```
> strings /usr/local/bin/netscape | \
     grep -i copyright
Copyright (C) 1995, Thomas G. Lane
```

Moskowitz [28] describes a data watermarking method in which the watermark is embedded in an image (or other digital media such as audio or video) using one of the many media watermarking algorithms. This image is then stored in the static data section of the program.

Unfortunately, static data watermarks are highly susceptible to distortive attacks by obfuscation. In the simplest case, an automatic obfuscator might break up all strings

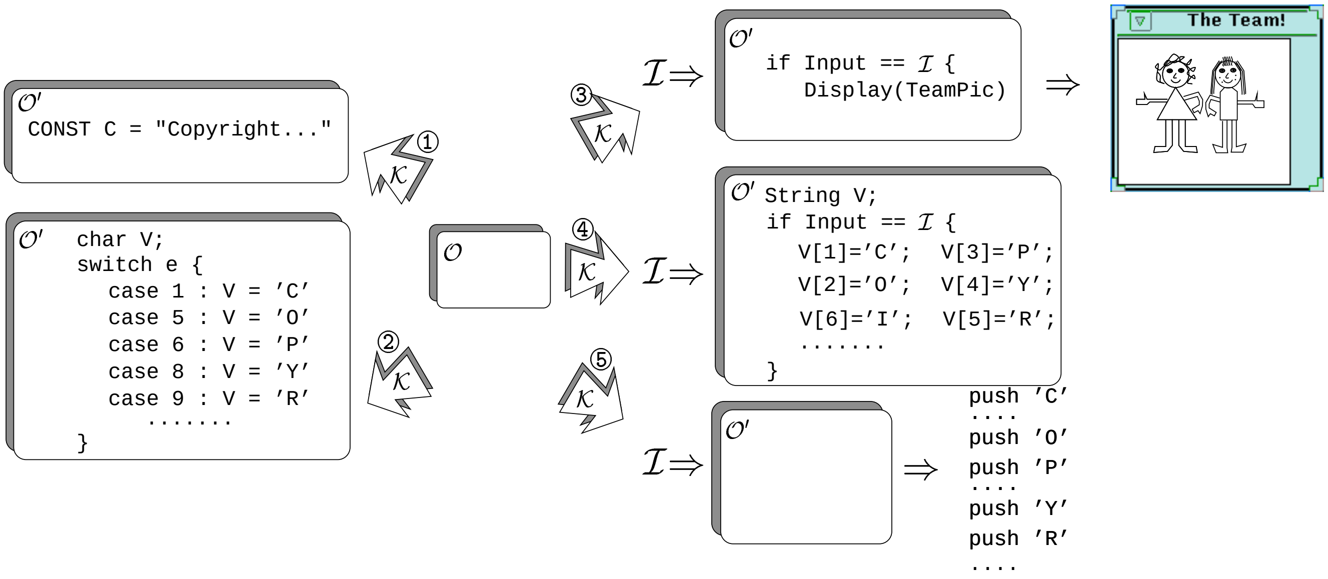


Figure 2: Static (① and ②) and dynamic (③, ④ and ⑤) watermarks. In ① Alice embeds a watermark in the initialized data (string) section of her program. In ② the watermark is embedded in the text (code) section of the program. In ③ the watermark is embedded in the unexpected behavior (an "Easter Egg") of the program when it is run with input $\mathcal{I}$. In ④ the watermark gets embedded in a global variable `V` when the program is run with input $\mathcal{I}$. In ⑤ the watermark is embedded in the execution trace when the program is run with input $\mathcal{I}$.

(and other static data) into substrings which are then scattered over the executable. This makes watermark recognition nearly impossible.

An even more sophisticated de-watermarking attack is to convert all static data into *a program* that produces the data [8].

2.2 Code Watermarks

Media watermarks are commonly embedded in redundant bits, bits which we cannot detect due to the imperfection of our human perception. Code watermarks can be constructed in a similar way, since object code also contains redundant information. For example, if there are no data or control dependencies between two adjacent statements S_1 and S_2, they can be flipped in either order. A watermarking bit could then be encoded in whether S_1 and S_2 are in lexicographic order or not.

There are many variations of this technique. When litigating against software pirates who had copied their PC-AT ROM, IBM [13] argued that the order in which registers were pushed and popped constituted a signature of their software. Similarly, by reordering the branches of an m-branch case-statement we can encode $\log_2(m!) \approx \log_2(\sqrt{2\pi m}(m/e)^m) = \mathcal{O}(m \log m)$ watermarking bits.

Davidson [10] describes a similar code watermark in which a software serial number is encoded in the basic block sequence of a program's control flow graphs.

Many code watermarks are susceptible to simple distortive de-watermarking attacks. For example, Davidson's method is easily destroyed by many locality-improving optimizations, such as described in Davidson [11]. This method also provides no protection against additive attacks; if we reorganize the basic block structure to encode our own watermark it is clear the original watermark can no longer be retrieved.

Many code obfuscation techniques [8,9] will also successfully thwart the recognition of code watermarks. For example, Davidson's method relies on one being able to reliably recognize the individual basic blocks of a control flow graph. But, it is easy to break up a basic block by inserting an opaquely true predicated branch:

$$
\begin{array}{lcl}
\texttt{void P() \{} & & \texttt{void P() \{} \\
\quad S_1\texttt{;} & & \quad S_1\texttt{;} \\
\quad S_2\texttt{;} & \overset{\mathcal{T}}{\Longrightarrow} & \quad \texttt{if } (P^T)\ S_2\texttt{;} \\
\quad S_3\texttt{;} & & \quad S_3\texttt{;} \\
\quad \ldots & & \quad \ldots \\
\texttt{\}} & & \texttt{\}}
\end{array}
$$

The construction of strong opaque predicates is discussed in [9].

2.3 Tamperproofing Static Watermarks

Our experience with obfuscation tells us that all static structures of a program can be successfully scrambled by obfuscating transformations. And, in cases where obfuscation is deemed too expensive, inlining and outlining [8], various forms of loop transformations [5] and code motion are all well-known optimization techniques that will easily destroy static code watermarks.

Moskowitz [28] describes how their software watermarking method (which embeds the watermark within an image included with the application) can be tamperproofed. The idea is to also embed an "essential" piece of code within the image. This code is occasionally extracted and executed, making the program fail if the image (and hence the watermark) has been tampered with. Unfortunately, generating and executing code on the fly is unusual and unstealthy behavior for most applications.

A further complication is the difficulty of tamperproofing code watermarks against these types of semantics-preserving transformations. This is particularly true in Java, since, for security reasons, Java programs are not able to inspect their own code. In other words, in Java we cannot write ⌜`if (instruction #99 != "add") die()`⌝. Even in languages like C where this is possible, such code would be highly unusual (since it examines the code rather than the data segment of the executing program) and unstealthy.

As a result, in spite of their simplicity and popularity, we believe static watermarks to be inherently flawed.

3 Dynamic Software Watermarking

As we have seen, static watermarks suffer from being easily attacked by semantics-preserving transformations. We therefore now turn to *dynamic* watermarks which have received even less attention than static ones. Dynamic watermarks are stored in a program's execution state, rather than in the program code itself. As we shall see, this makes (some of) them easier to tamperproof against obfuscating transformations.

There are three kinds of dynamic watermarks. In each case, the application $\mathcal{O}$ is run with a predetermined input sequence $\mathcal{I}=\mathcal{I}_1\cdots\mathcal{I}_k$ which makes the application enter a state which represents the watermark. The methods differ in which part of the program state the watermark is stored, and in the way it is extracted.

In our taxonomy we will distinguish between three dynamic watermarking techniques (see Figure 2): **Easter Egg Watermarks**, **Data Structure Watermarks**, and **Execution Trace Watermarks**. While Easter Egg watermarks are very popular [29], there seem to be no published accounts of data structure or execution trace watermarks.

3.1 Easter Egg Watermark

Figure 2 ③ shows a watermark encoded in an *Easter Egg*, a piece of code that gets activated for a highly unusual input to the application. The defining characteristic of an Easter Egg watermark is that it performs some action that is immediately perceptible by the user, making watermark extraction trivial. Typically, the code will display a copyright message or an unexpected image on the screen. For example, entering the URL ⌜`about:mozilla`⌝ in `Netscape 4.0` will make a fire-breathing creature appear [29].

The main problem with Easter Egg watermarks is that they seem to be easy to locate. There are even several website repositories of such watermarks. Unless the effects of the Easter Egg are really subtle (in which case it will be hard to argue that they indeed constitute a watermark and are not the consequence of bugs or random programmer choices), it is often immediately clear when a watermark has been found. Once the right input sequence has been discovered, standard debugging techniques will allow us to trace the location of the watermark in the executable and then remove or disable it completely.

3.2 Dynamic Data Structure Watermark

Figure 2 ④ shows a watermark being embedded within the state (global, heap, and stack data, etc.) of a program $\mathcal{O}$ as it is being run with a particular input $\mathcal{I}$. The watermark is extracted by examining the current values held in $\mathcal{O}$'s variables, after the end of the input sequence has been reached. This can be done using either a dedicated watermark extraction routine which is linked in with the executing program, or by running the program under a debugger.

Data structure watermarks have some nice properties. In particular, since no output is ever produced it is not immediately evident to an adversary when the special input sequence $\mathcal{I}$ has been entered. This is in contrast to Easter Egg watermarks, where, at least in theory, it would be possible to generate input sequences at random and wait for some "unexpected" output to be produced. Furthermore, since the recognition routine is not shipped within the application (it is linked in during watermark extraction), there is little information in the executable itself as to where the watermark may be located.

Unfortunately, data structure watermarks are also susceptible to attacks by obfuscation. Several obfuscating transformations have been devised which will effectively destroy the dynamic state and make watermark recognition impossible. In [8] we show how one variable can be split into several variables. This transformation requires us to provide functions f and f^{-1} for converting between the original and the split data representations, and functions $g^{\oplus}$ that implement any built-in operator $\oplus$ on the new split representation:

$$\begin{array}{l} \texttt{main() \{} \\ \quad \texttt{T x,y,z;} \\ \quad \texttt{x} \leftarrow e\texttt{;} \\ \quad \cdots \leftarrow \texttt{x;} \\ \quad \texttt{x} \leftarrow \texttt{y} \oplus \texttt{z;} \\ \texttt{\}} \end{array} \quad \stackrel{\mathcal{T}}{\Longrightarrow} \quad \begin{array}{l} \texttt{main() \{} \\ \quad \texttt{T1 x1,x2,y1,y2,z1,z2;} \\ \quad \texttt{x1,x2} \leftarrow f_{k\in\{1,2\}}(e)\texttt{;} \\ \quad \cdots \leftarrow f^{-1}(\texttt{x1,x2})\texttt{;} \\ \quad \texttt{x1,x2} \leftarrow g^{\oplus}(\texttt{y1,y2,z1,z2})\texttt{;} \\ \texttt{\}} \end{array}$$

For example, to split a boolean variable `x` into two short variables $\langle$`x1,x2`$\rangle$ we let `T=bool`, `T1=short`, `True` = $\langle 0,1\rangle$ or $\langle 1,0\rangle$, and `False` = $\langle 0,0\rangle$ or $\langle 1,1\rangle$. We furthermore have to provide new implementations of any built-in operators:

$$\begin{array}{l} \langle\texttt{short,short}\rangle\ f_1\texttt{(bool x)=\{return x?}\langle 0,1\rangle\texttt{:}\langle 0,0\rangle\texttt{\}} \\ \langle\texttt{short,short}\rangle\ f_2\texttt{(bool x)=\{return x?}\langle 1,0\rangle\texttt{:}\langle 1,1\rangle\texttt{\}} \\ \texttt{bool } f^{-1}\texttt{(short x1,x2)=\{return x1\^{}x2==1\}} \\ \langle\texttt{short,short}\rangle\ g^{\&\&}\texttt{(short x1,x2,y1,y2)=\{} \\ \quad \texttt{return } \langle\texttt{(x1\^{}x2)\&(y1\^{}y2)),0}\rangle\texttt{\}.} \end{array}$$

In a similar manner, several variables can be merged into one. We have to provide functions f_k and f_k^{-1} to insert and extract the original variable k from the merged data representation, and functions $g_k^{\oplus}$ that implement any built-in operator $\oplus$ on variable k in the new representation:

$$\begin{array}{l} \texttt{main() \{} \\ \quad \texttt{T y,z;} \\ \quad \texttt{x} \leftarrow e\texttt{;} \\ \quad \texttt{y} \leftarrow e\texttt{;} \\ \quad \cdots \leftarrow \texttt{x;} \\ \quad \texttt{x} \leftarrow \texttt{x} \oplus c\texttt{;} \\ \texttt{\}} \end{array} \quad \stackrel{\mathcal{T}}{\Longrightarrow} \quad \begin{array}{l} \texttt{main() \{} \\ \quad \texttt{T1 z;} \\ \quad \texttt{z} \leftarrow f_{\texttt{x}}(\texttt{z},e)\texttt{;} \\ \quad \texttt{z} \leftarrow f_{\texttt{y}}(\texttt{z},e)\texttt{;} \\ \quad \cdots \leftarrow f_{\texttt{x}}^{-1}(\texttt{z})\texttt{;} \\ \quad \texttt{z} \leftarrow g_{\texttt{x}}^{\oplus}(\texttt{z,c})\texttt{;} \\ \texttt{\}} \end{array}$$

For example, to merge two 32-bit `x` and `y` integers into a 64-bit integer `z` we let `T=unsigned int32` and `T1=unsigned int64`, and provide new implementations of the built-in operators on the merged type:

$$\begin{array}{l} \texttt{T1 } f_{\texttt{x}}\texttt{(T1 z, T x)=\{return z\&FFFFFFFF00000000}_{16}\texttt{|x\}} \\ \texttt{T1 } f_{\texttt{y}}\texttt{(T1 z, T y)=\{return y<<32 | z\&FFFFFFFF}_{16}\texttt{\}} \\ \texttt{T } f_{\texttt{x}}^{-1}\texttt{(T1 z)=\{return z\&FFFFFFFF}_{16}\texttt{\}} \\ \texttt{T } f_{\texttt{y}}^{-1}\texttt{(T1 z)=\{return z>>32\}} \\ \texttt{T1 } g_{\texttt{x}}^{*}\texttt{(T1 z, T } c\texttt{)=\{return } f_{\texttt{x}}(z,c*f_{\texttt{x}}^{-1}(\texttt{z}))\texttt{\}} \end{array}$$

Other transformations will merge or split arrays, modify the inheritance hierarchy of an object oriented program, etc.

3.3 Dynamic Execution Trace Watermark

In Figure 2 ⑤ a watermark is embedded within the trace (either instructions or addresses, or both) of the program as it is being run with a particular input $\mathcal{I}$. The watermark is extracted by monitoring some (possibly statistical) property of the address trace and/or the sequence of operators executed.

The obfuscating transformations that we have already presented, as well as many optimizing and translating transformations, will effectively obliterate any structure embedded in an instruction trace. An even more potent, and more expensive, transformation is to convert a section of code (Java bytecode in our case) into a *different* virtual machine code. The new code is then executed by a virtual machine interpreter (included with the obfuscated application) which is specialized to handle this particular virtual machine code:[1]

```
                      void P() {
                        Stack S; sp ← 0; pc ← 0;
void P() {              C ← ⟨op3, op5, op1, ···⟩;
  S1;                   for(;;)
  S2;        T            switch C[pc++] of {
             ⇒              op1   :  T_op1 ; break;
  ⋮                         op2   :  T_op2 ; break;
  Sn;                               ···
}                         }
                      }
```

While the behavior of the the new virtual machine running the obfuscated program will be the same as the original program, i.e.

$$S_1; S_2; \cdots \equiv T_{\texttt{C[0]}}; T_{\texttt{C[1]}}; T_{\texttt{C[2]}}; \cdots,$$

the execution trace will be completely different. In most cases this will not be a practical attack because of the extra overhead of interpretation.

4 A Formal Model of Software Watermarking

In the next section we will be constructing new techniques which are resilient to a variety of semantics-preserving de-watermarking attacks. Before we do so we will formalize our notion of a watermark and what it means to *recognize* a watermark in a program.

In order to be able to legally argue ownership of a watermarked program, we must be able to show that our recognition of the watermark is not a chance occurrence:

Definition 1 (Software Watermark) Let $\mathbb{W}$ be a set of mathematical structures, and p a predicate such that $\forall w \in \mathbb{W} : p(w)$. We choose p and $\mathbb{W}$ such that the probability of $p(x)$ for a random $x \notin \mathbb{W}$ is small. □

As we have seen, watermarks can be embedded both in the program text and in the state of the program as it is run with a particular set of inputs. Furthermore, *attacks* can be launched both on the program text and the state.

Definition 2 (Programs) Let $\mathbb{P}$ be the set of programs. P_w is an embedding of a watermark $w \in \mathbb{W}$ into $P \in \mathbb{P}$.

Let $\texttt{dom}(P)$ be the set of input sequences accepted by P. Let $\texttt{out}(P, I)$ be the output of P on input I.

Let $S(P, I)$ be the internal state of program P (drawn from a set of states $\mathbb{S}$) after having processed input I. Let $|S(P, I)|$ be the size of this state, in accessible words. □

[1] This technique is similar to Proebsting's superoperators [31].

4.1 Watermark Recognition

The resilience of a watermark w in a program P_w will be defined in terms of the de-watermarking attacks that can be launched against P_w. Attacks are *program transformations* that can be *semantics-preserving* (if they preserve input-output behavior), *state-preserving* (if internal state is preserved), or *cropping* (if input-output behavior is not preserved).

Definition 3 (Program Transformations) Let $\mathbb{T}$ be the set of transformations from programs to programs.

$\mathbb{T}_{\text{sem}} \subset \mathbb{T}$ is the set of *semantics-preserving* transformations, $\mathbb{T}_{\text{stat}} \subset \mathbb{T}$ is the set of *state-preserving* transformations, and $\mathbb{T}_{\text{crop}} \subset \mathbb{T}$ is the set of transformations which do not preserve semantics:

$$\begin{aligned}
\mathbb{T}_{\text{sem}} = \{t : \mathbb{T} \quad | \quad & P \in \mathbb{P}, I \in \texttt{dom}(P), \\
& \texttt{dom}(P) = \texttt{dom}(t(P)), \\
& \texttt{out}(P, I) = \texttt{out}(t(P), I)\}. \\
\mathbb{T}_{\text{stat}} = \{t : \mathbb{T} \quad | \quad & P \in \mathbb{P}, I \in \texttt{dom}(P), \\
& S(P, I) = S(t(P), I)\}. \\
\mathbb{T}_{\text{crop}} = \{t : \mathbb{T} \quad | \quad & \exists P \in \mathbb{P}, \exists I \in \texttt{dom}(P), \\
& (I \notin \texttt{dom}(t(P))\vee \\
& \texttt{out}(P, I) \neq \texttt{out}(t(P), I))\}.
\end{aligned}$$

□

State-preservation implies semantics-preservation but many transformations (such as code optimizing transformations) will preserve semantics but not state, i.e. $\mathbb{T}_{\text{stat}} \subset \mathbb{T}_{\text{sem}}$.

In [30] Peticolas writes: "the problem [with watermarking] is not so much inserting the marks as recognizing them afterwards". Hence, the strength of a watermark is defined with respect to the set of transformations under which the watermark can be recognized:

Definition 4 (Watermark Recognition) A watermark $w \in \mathbb{W}$ in a program $P_w \in \mathbb{P}$ is recognizable wrt a set of transformations $T \subset \mathbb{T}$ if there exists a recognizer

$$\mathcal{R}_T : (\mathbb{P} \times \mathbb{S}) \to \mathbb{W}$$

and an input I such that

$$\forall t \in T : p(\mathcal{R}_T(t(P_w), S(t(P_w), I))) = p(w).$$

□

This definition allows us to distinguish several useful subclasses of recognizers:

- $\mathcal{R}_\emptyset(P_w, S(P_w, I))$ is a *trivial* recognizer that is not guaranteed to recognize w if *any* transformations have been performed on P_w.
- $\mathcal{R}_{\mathbb{T}_{\text{sem}}}(P_w, S(P_w, I))$ is a *strong* recognizer that is resilient to any semantics-preserving transformation.
- $\mathcal{R}_{\mathbb{T}}(P_w, S(P_w, I))$ is an *ideal* recognizer that is resilient to any transformation.
- $\mathcal{R}_T(P_w, \emptyset)$ is a *static* recognizer that can only examine the text of P_w, not its execution state.
- $\mathcal{R}_T(\emptyset, S(P_w, I))$ is a *pure dynamic* recognizer that can only examine the execution state of P_w, not its text.

4.2 Watermark Resilience

We are particularly interested in evaluating the strength of a watermark written dynamically into the state of a program. Such watermarks may be attacked by adversaries who write other information into the state. If the watermark can only be obliterated by adversaries that increase the size of the program's dynamic state by a factor r, then we say the watermark is r-space resilient:

Definition 5 (Watermark r-Space Resilience) A watermark $w \in \mathbb{W}$ in a program program P_w is r-space resilient wrt a set of transformations $T \subset \mathbb{T}$ if there exists a recognizer $\mathcal{R}_T$ and an input I such that

$$\forall t \in T : (p(\mathcal{R}_T(t(P_w), S(t(P_w), I))) \neq p(w)) \Rightarrow$$
$$\frac{|S(t(P_w), I)|}{|S(P_w, I)|} \geq r$$

□

Note that any 1-space resilient watermark has a strong recognizer. However, if $r > 1$, our parameter r is a measure of the weakness of the watermarking system.

Some watermarks that are written in the program text or static data are susceptible to attacks that increase the static size of the code:

Definition 6 (Watermark r-Size Resilience) A watermark $w \in \mathbb{W}$ in a program program P_w is r-size resilient wrt a set of transformations $T \subset \mathbb{T}$ if there exists a recognizer $\mathcal{R}_T$ and an input I such that

$$\forall t \in T : (p(\mathcal{R}_T(t(P_w), S(t(P_w), I))) \neq p(w)) \Rightarrow$$
$$\frac{|t(P_w)|}{|P_w|} \geq r$$

□

Finally, many attacks on a watermarked program P_w will increase its runtime; however if the runtime is increased too much (for at least one input in the domain of P) then the attack is not particularly worrisome:

Definition 7 (Watermark r-Runtime Resilience) A watermark $w \in \mathbb{W}$ in a program P_w is r-runtime resilient wrt a set of transformations $T \subset \mathbb{T}$ if there exists a recognizer R_T and an input I such that

$$\forall t \in T : (p(\mathcal{R}_T(t(P_w), S(P_w, I))) \neq p(w)) \Rightarrow$$
$$\exists i \in \mathtt{dom}(P) \frac{\mathrm{Time}(t(P_w), i)}{\mathrm{Time}(P_w, i)} \geq r$$

□

4.3 Watermark Stealth

Certain types of watermarks are vulnerable to attack by statistical analysis. If the static or dynamic instruction mix of P_w is radically different from what one would expect from a program of P_w's type, we may suspect that the watermark might be hidden in the more frequently occurring instructions.

Definition 8 (Watermark Stealth) A watermark w is *statically stealthy* for program P wrt statistical measure M, if $M(P) - M(P_w)$ is insignificant.

Similarly, a watermark w is *dynamically stealthy* if $M(S(P, I)) - M(S(P_w, I))$ is insignificant. □

4.4 Watermark Data Rate

It is essential that the watermark encodes as much information as possible, while at the same time not increasing the size of the program text or the working set size of the executing program.

Definition 9 (Watermark Coding Efficiency) $H(w) = \log_2 |\mathbb{W}|$ is the *entropy* of w, in bits, when w is drawn with uniform probability from $\mathbb{W}$.

Let $|P|$, $P \in \mathbb{P}$ be the size (in words) of P as expressed in some encoding.

Let $|S(P)| = \max_{I \in \mathtt{dom}(P)} |S(P, I)|$ be the least upper bound on the size of P's internal state.

An embedding of P_w of w in P has a *high static data rate* if

$$\frac{H(w)}{\max(1, |P_w| - |P|)} \geq 1.$$

An embedding P_w of w in P has a *high dynamic data rate* if

$$\frac{H(w)}{\max(1, |S(P_w)| - |S(P)|)} \geq 1.$$

□

Note that data rate is measured in "hidden bits" per "extra" word added in the watermarking process.

5 Dynamic Graph Watermarking

As we have seen from the previous discussion, all software watermarking techniques (with the exception of Easter Egg watermarks) are susceptible to distortive attacks by semantics-preserving transformations. This is similar to the situation in media watermarking.

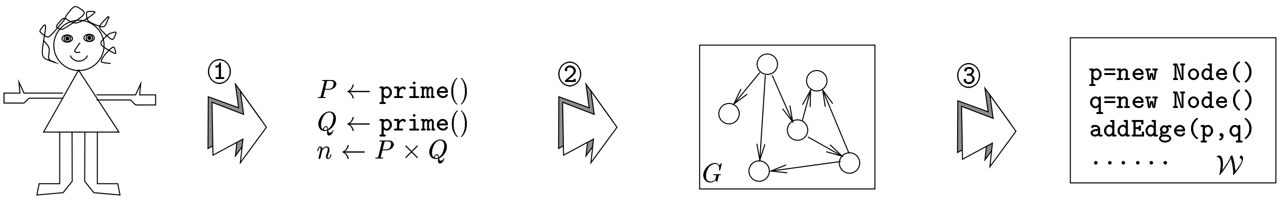
In this section we will discuss, in detail, new techniques for embedding software watermarks in dynamic data structures. It is our belief that these techniques are the most promising for withstanding distortive de-watermarking attacks. In particular, we will see that it is possible to exactly describe the types of attacks that are possible against this method, and devise counter-measures that will protect against reasonable levels of attack.

5.1 Overview

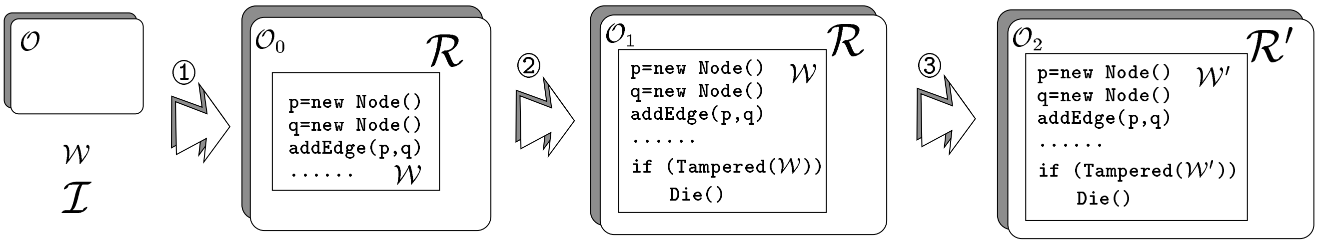
The central idea of *Dynamic Graph Watermarking* is to embed a watermark in the *topology* of a dynamically built graph structure. Because of pointer aliasing effects, code which manipulates dynamic graph structures is hard to analyze. As a result, semantics-preserving transformations that make fundamental changes to a graph will be hard to construct. Moreover, it is easier to tamperproof such structures than tamperproofing code or scalar data.

Figure 3 illustrates our technique. The signature property $p(w)$ we propose to embed in a graph-watermark w is that the topology of the graph represents the product n of two large primes P and Q. To prove the legal origin of P_w, the recognizer extracts n from P_w, and factors n. A similar signature property based on public-key cryptography has been proposed by Samson [33] for a *static* watermarking scheme. Obviously, $p(w)$ can be based on other hard graph problems, such as the lattice problems described in [1,14].

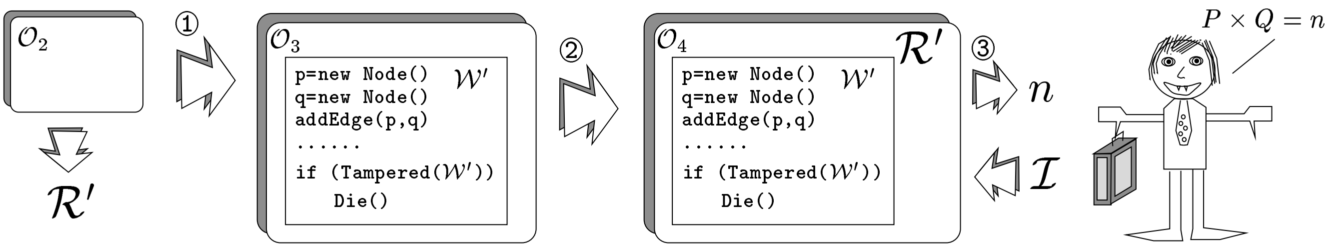
As always, the main problem of watermarking is recognizing and extracting the mark. To extract w from P_w our recognizer $\mathcal{R}_T(\emptyset, S(P_w, I))$ will primarily examine the runtime *object heap* as the program is being run with the watermark key input sequence I. When the end of this sequence



(a) At ①, Alice selects n, the product of two large primes P and Q. At ②, n is embedded in the topology of a graph G. At ③, a program $\mathcal{W}$ which builds G is constructed.



(b) At ①, $\mathcal{W}$ is embedded into the original program $\mathcal{O}$, such that when $\mathcal{O}_0$ is run with $\mathcal{I}$ as input, G is built. Also, a recognizer $\mathcal{R}$ which can identify G on the heap, is constructed. At ②, tamperproofing is added, to prevent de-watermarking attacks. At ③, the application (including the watermark, tamperproofing code, and recognizer) is obfuscated to prevent attacks by pattern-matching.



(c) At ①, the recognizer is removed from the application to make $\mathcal{O}_3$, the version of $\mathcal{O}$ that Alice sells. At ②, Charles links in $\mathcal{R}$ with $\mathcal{O}_3$. At ③, $\mathcal{O}_4$ is run with $\mathcal{I}$ as input, and the recognizer $\mathcal{R}$ produces n. Charles proves ownership of $\mathcal{O}_3$ by factoring n.

Figure 3: Overview of proposed dynamic graph watermarking scheme.

is reached we know that one of the (possibly many) linked object structures on the heap will represent w. The main difficulty will be to recognize our graph out of the many other structures on the heap. In the next few sections we will discuss this and other issues in more detail.

5.2 Embedding the Watermark

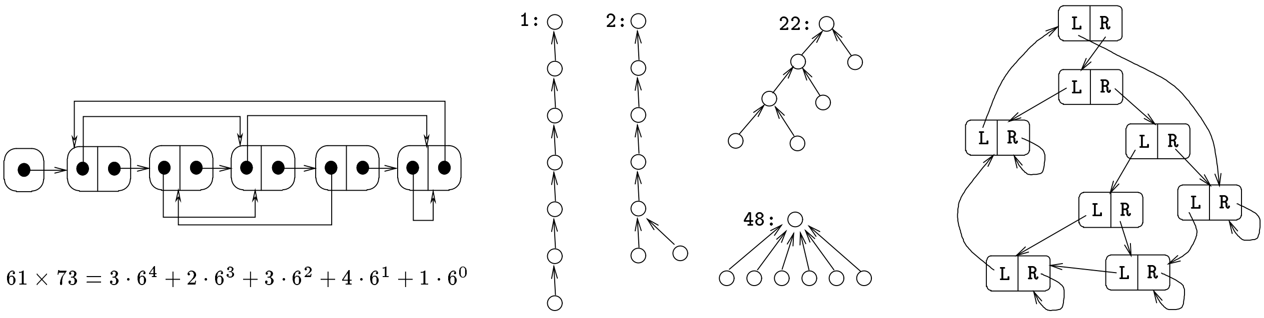
In this section we show two ways of embedding a number n in the topology of a graph G. There are obviously many ways of doing this, and, in fact, a watermarking tool should have a library of many such techniques to choose from to prevent attacks by pattern-matching.

5.2.1 Radix-k Encoding

Figure 4(a) illustrates a Radix-k encoding in a circular linked list. An extra pointer field encodes a base-k digit in the length of the path from the node back to itself. A null-pointer encodes a 0, a self-pointer a 1, a pointer to the next node encodes a 2, etc.

A list of length m can encode any integer in the range $0 \ldots (m+1)^m - 1$. The list requires $2m+1$ extra words, if we assume no overhead heap cells. The dynamic bit-rate is $\log_2(m+1)^m/(2m+1) \approx (\log_2 m)/2$. For $m = 255$ we can hide $255 \times 8 = 2040$ bits in 511 words of storage, or 4 hidden bits per word.

The static data rate is harder to determine, since this will depend heavily on the encoding. As an example, we will consider Java bytecode. Allocating a node and initializing two pointer fields requires a 24-byte bytecode sequence. To hide a 2040-bit watermark we build a 255-element list which requires $24 \times 255 = 6120$ bytes of straight-line bytecode, for a static bit-rate of 0.33 hidden bits per byte.



(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 1st, 2nd, 22nd, and 48th trees in an enumeration of the oriented trees with seven vertices.

(c) Enumeration encoding in a planted plane cubic tree on $2m = 8$ nodes.

Figure 4: Graphic embeddings of watermarks.

5.2.2 Enumeration Encoding

Our second embedding method uses results from graph enumeration [17]. The idea is to let the watermark number n be represented by the *index* of the watermark graph G in some convenient enumeration. This requires us to be able to

1. given n, generate the n:th graph in the enumeration, and

2. given G, extract its index n in the enumeration.

Both operations must be efficient, since we expect n to be large. This rules out many classes of graphs due to the intractability of sub-graph isomorphism.

Several restricted classes of graphs allow efficient enumeration and indexing. For example, we can let G be an oriented "parent-pointer" tree, in which case it is enumerable by the techniques described in Knuth [23, Section 2.3.4.4].

The number a_m of oriented trees with m nodes is asymptotically $a_m = c(1/\alpha)^{n-1}/n^{3/2} + \mathcal{O}((1/\alpha)^n/n^{5/2})$ for $c \approx 0.44$ and $1/\alpha \approx 2.956$. Thus we can encode an arbitrary 1024-bit integer n in a graphic watermark with $1024/\log_2 2.956 \approx 655$ extra words. This is a dynamic bitrate of $1024/1.56 \approx 1.56$ hidden bits per word.[2]

We construct an index n for any enumerable graph in the usual way, that is, by ordering the operations in the enumeration. For example, we might index the m-node trees in "largest subtree first" order, in which case the path of length $m-1$ would be assigned index 1. Indices 2 through a_{m-1} would be assigned to the other trees in which there is a single subtree connected to the root node. Indices $a_{m-1}+1$ through $a_{m-1}+a_{m-2}$ would be assigned to the trees with exactly two subtrees connected to the root node, such that one of the subtrees has exactly $m-2$ nodes. The next $a_{m-3}a_2 = a_{m-3}$ indices would be assigned to trees with exactly two subtrees connected to the root node, such that one of the subtrees has exactly $m-3$ nodes. See Figure 4(b) for an example.

[2] This assumes that the graph is stored on an untraced heap. In languages with only traced heaps extra pointers will be necessary to avoid leaves being collected.

5.3 Recognizing the Watermark

In Figure 3(b) we select the length k of the input sequence $\mathcal{I}$ and separate G into k components, $G_1 \cdots G_k$. The code to build these components, $\mathcal{W}_1 \cdots \mathcal{W}_k$, is then inserted into the application, such that when the end of the input sequence $\mathcal{I}{=}\mathcal{I}_1 \cdots \mathcal{I}_k$ is reached, all graph components have been built and assembled into the complete watermark (Figure 5(a)).

It might seem that in order to identify G we would need to examine all reachable heap objects, which, of course, would be intractable. In fact, Figure 5(b) shows that we can do better than that. If we assume that G has a distinguished node (this is the case of the embeddings in the previous section), and this *root* node is part of G_k, we only have to examine the nodes built during the processing of $\mathcal{I}_k$.

5.4 Attacks Against the Watermark

One nice consequence of our approach is that the translating, optimizing, and obfuscating transformations discussed in Sections 2 and 3 will have no effect on the heap-allocated structures that are being built. There are, however, other techniques which can obfuscate dynamic data, particularly for languages with typed object code, like Java. There are four types of obfuscating transformations that we will need to tamperproof against. An adversary can

1. add extra pointers to the nodes of linked structures (Figure 6(a)). This will make it hard for the recognizer to identify the real graph edges within a lot of extra bogus pointer fields.

2. rename and reorder the fields in the node, again making it hard to recognize the real watermark (Figure 6(b)).

3. add levels of indirection, for example by splitting nodes into several linked parts (Figure 6(c)).

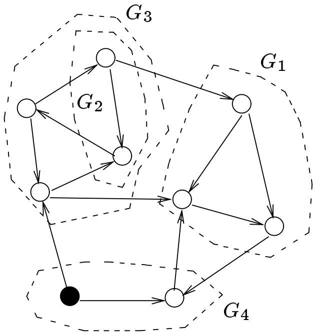
4. add extra bogus nodes pointing into our graph, preventing us from finding the root.

Figure 6(d) illustrates a combination of such attacks.

With the exception of renaming and reordering, these attacks can have some very serious consequences for the memory requirement of an adversary's de-watermarked program.

if (input = $\mathcal{I}_1$) $G_1 = \cdots$;
if (input = $\mathcal{I}_2$) $G_2 = \cdots$;
if (input = $\mathcal{I}_3$) $G_3 = G_2 \oplus G_3$;
......
if (input = $\mathcal{I}_k$) $G = G_1 \oplus G_3 \oplus \cdots$;

(a) Code to build the watermark graph. Each input in the key sequence builds one graph component. After $\mathcal{I}_k$ has been processed, the entire graph has been built.



(b) The complete watermark graph and its components. The root node (colored black) is built during processing of the last key input, $\mathcal{I}_k$.

Figure 5: Building and recognizing the watermark graph.

For example, splitting a node costs one pointer cell plus the usual object overhead (2-3 words in Java). Furthermore, since we are assuming that an adversary will not know in which dynamic structure our watermark is hidden, he is going to have to apply the transformations uniformly over the *entire* program in order to be certain the watermark has been obliterated. In other words, programs with high allocation rate are likely to be resilient to these types of attacks, since the de-watermarked program will have a much higher memory requirement than the original one.

5.5 Tamperproofing the Watermark

A variety of techniques can be used to protect the watermark graph against attack. The most attractive methods are those where the structure of the graph itself renders certain types of attacks ineffective. The parent-pointer representation of Figure 4(b), for example, is resilient to renaming and reordering attacks since each node only has one pointer. Figure 7 shows another representation which, at the expense of a lower data rate, will increase a graphic watermark's resilience to node-splitting attacks.

5.5.1 Tamperproofing by Reflection

The *reflection* capabilities of Java (and other languages like Modula-3 and Icon) give us a simple way of tamperproofing a graph watermark against many types of attack. Assume that we have a graph node `Node`:

```
class Node {public int a; public Node car, cdr;}
```

Then the Java reflection class lets us check the integrity of this type at runtime:

```
Field[] F = Node.class.getFields();
if (F.length != 3) die();
if (F[1].getType() != Node.class) die();
```

To prevent reordering and renaming attacks we can access watermark pointers through reflection. For example, rather than ⌜`O.car=V`⌝, we let `car` be represented by the first relevant pointer in the node `O`:

```
Field[] F = Node.class.getFields();
int n=0;
for(int i=0; i<F.length; i++)
 if (F[i].getType().isAssignableFrom(Node.class))
  { F[i].set(O, V); break; }
```

Obviously, this type of code is unstealthy in a program that does not otherwise use reflection.

5.5.2 Cropping Attacks

So far, we have assumed that all attacks preserve the semantics of P_w. This is reasonable, since if the adversary has no knowledge of the location of w his only hope is to apply semantics-preserving transformations uniformly over all of P_w. If, however, the adversary can locate the code that builds the watermarking graph G, he can easily destroy it by adding extra nodes or edges. To thwart this kind of attack, P_w should occasionally check the integrity of G.

For example, consider the class G_m of *planted plane cubic trees* on m leaf nodes $v_1, v_2, \ldots, v_m$, as enumerated in [15] and illustrated in Figure 4(c). Such trees have $m-1$ internal nodes and one root node v_0, so there are $2m$ nodes in each $w \in G_p$. We would represent w by using $2m$ objects, where each object holds two pointers l and r; this data structure requires $4m$ words. A leaf node v_i is recognizable by its self-loop $r(v_i) = v_i$. The root node v_0 can be found from any leaf node by following l-links. Furthermore, the leaf node indices are discoverable by following an m-cycle on l-links: $l(v_i) = v_{(i+1) \bmod m}$. This watermark has a bit-rate of $(\log_2 |\{w : w \in G_m\}|)/4m \approx (2m - 1.5\log_2 m)/4m \approx 0.5$. The planarity restriction may be tested for each internal node x by confirming that the left-most child of x's right subtree is l-linked to the right-most child of its left subtree.

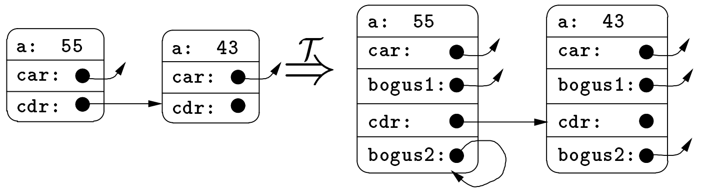
6 Discussion

Because software watermarking is a new field, many fundamental issues have yet to be resolved. From a practical point of view, the most important question is what constitutes a

```
class T {
   int a;
   T car;
   T cdr;
}
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
class T {
   int a;
   T car;
   T bogus1;
   T cdr;
   T bogus2;
}
```

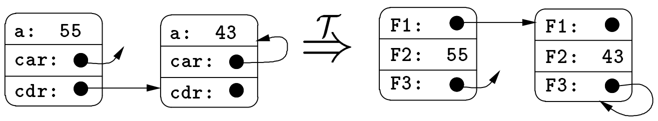


(a) Add bogus pointer fields to all nodes of type `T`.

```
class T {
   int a;
   T car;
   T cdr;
}
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
class T {
   T F1;
   int F2;
   T F3;
}
```

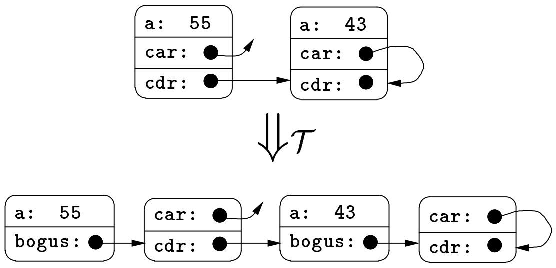


(b) Rename and reorder fields in all nodes of type `T`.

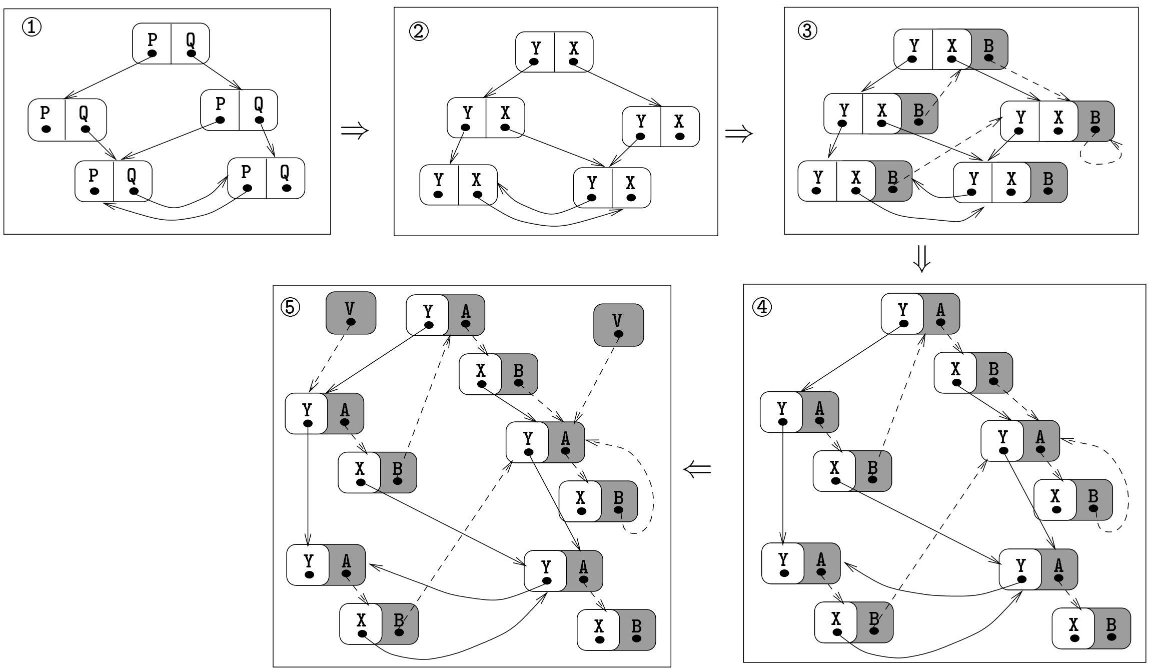
```
class T {
   int a;
   T car;
   T cdr;
}
n = new T;
```

$\stackrel{\mathcal{T}}{\Longrightarrow}$

```
class T {
   int a;
   T1 bogus;
}
class T1 {
   T car;
   T cdr;
}
n = new T;
n.bogus = new T1;
```



(c) Add a level of indirection by splitting all nodes of type `T` in two.



(d) Example obfuscation attack against the watermark graph in ①. The adversary renames and reorders node pointer fields (②), adds a bogus pointer field `B` (③), and splits nodes by adding a bogus pointer field `A` (④). Finally, in ⑤ bogus pointers into the graph obscure the root node.

Figure 6: Obfuscation attacks against graphic watermarks.

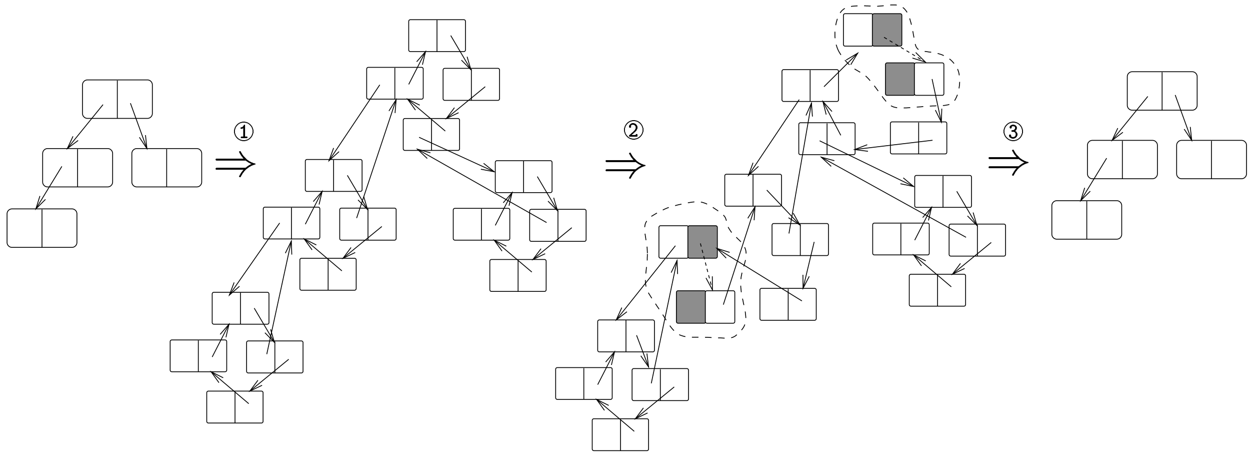


Figure 7: Tamperproofing against node-splitting. At ① we expand each node of our original watermark tree into a 4-cycle. At ② an adversary splits two nodes. The structure of the graph ensures that these nodes will fall on a cycle. At ③ the recognizer shrinks the biconnected components of the underlying (undirected) graph. The result is a graph isomorphic to our original watermark.

reasonable threat-model. In this paper we have identified several types of threats:

1. Distortive attacks by semantics-preserving transformations such as translation, optimization, and obfuscation.

2. Statistical attacks which attempt to locate a watermark by identifying anomalies in the distribution of instructions or computations.

3. Collusive attacks which attempt to locate a fingerprint by comparing several differently fingerprinted copies of a program.

4. Cropping attacks which remove a located watermark or extract an individual module from a watermarked application.

5. Additive attacks which insert new bogus watermarks into an already watermarked program.

None of the methods we have presented are immune to all types of attacks. Easter Egg watermarks and dynamic graph watermarks are highly resilient against distortive attacks, but, by their very nature, they watermark *complete applications*, not individual modules. Hence, cropping a particularly valuable module from an application for illegal reuse is likely to be a successful attack against these methods.

Static watermarks, on the other hand, are easily duplicated many times in an application and can thus be made to protect individual modules or even parts of modules. Unfortunately, static watermarks are highly susceptible to distortive attacks.

Whether a statistical attack is successful or not will depend on the nature of the watermark, and the nature of the application. Dynamic graph watermarks are stealthy in typical object-oriented programs which tend to create large and complex heap structures. They would be very unstealthy, and hence susceptible to statistical attacks, in programs that are primarily numerical in nature. Davidson's [10] method (in which a serial number is encoded in the order of basic blocks) is also prone to statistical attacks since the resulting control flow graphs tend to appear convoluted and sub-optimal.

It is interesting to note that the problems we face in software watermarking are often quite different from those that arise in watermarking media. The reason is the fluidity of software, which allows us to make quite sweeping changes to the text of a program without changing its behavior. For example, it is quite difficult to protect against a collusive attack on an image fingerprint, since, by their very nature, all fingerprinted copies must appear identical. Software watermarks do not face this problem. We can easily protect against collusive attacks by applying a different set of obfuscating transformations to each distributed copy of an application. Thus, comparing several fingerprinted copies of the same application is unlikely to reveal the location of the fingerprint, since the text of each distributed copy will appear completely different.

For similar reasons, distortive attacks are a *less* serious threat to media watermarks than to software watermarks. A distortive attack on a media object is restricted to making imperceptible changes, whereas an obfuscation attack on a program is only restricted to preserving its semantics.

We are aware of no media or software watermarking technique that is immune to additive attacks.

7 Conclusion

Software watermarking is the process of embedding a large number into a program such that: (a) the number can be reliably retrieved after the program has been subjected to program transformations, (b) the embedding is imperceptible to an adversary, and (c) the embedding does not degrade the performance of the program.

This is a challenging problem that, to the best of our knowledge, has not previously been addressed in the academic literature. The few published accounts of which we are aware (mostly software patents) describe schemes in which watermarks or fingerprints are embedded in the object code of a program. These static techniques are susceptible to attacks such as translation, optimization, or obfuscation.

In this paper we have constructed a taxonomy of software watermarking techniques based on how marks are embedded, retrieved, and attacked. We have furthermore provided a formalization of software watermarking that we believe will form the basis for further research in the field. The most interesting result, however, is a new family of software watermarking techniques in which marks are embedded within the topology of dynamic heap data structures.

Acknowledgment: We'd like to thank P. Gibbons, S. Cheng, members of STAR Lab, S. Moskowitz, M. Cooperman, and the anonymous referees for valuable input.

References

[1] Miklos Ajtai. Generating hard instances of lattice problems. In *Proceedings of The Twenty-Eighth Annual ACM Symposium On The Theory Of Computing (STOC '96)*, pages 99–108, New York, USA, May 1996. ACM Press.

[2] D.J. Albert and S.P. Morse. Combating software piracy by encryption and key management. *IEEE Computer*, April 1982.

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

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

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994. `http://www.acm.org/pubs/toc/Abstracts/0360-0300/197406.html`.

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

[7] 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/CollbergThomborsonLow97a`.

[8] 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/CollbergThomborsonLow98b/`.

[9] 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/CollbergThomborsonLow98a/`.

[10] 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.

[11] Robert L. Davidson, Nathan Myhrvold, Keith Randel Vogel, Gideon Andreas Yuval, Richard Shupak, and Norman Eugene Apperson. Method and system for improving the locality of memory references during execution of a computer program. US Patent 5,664,191, September 1997. Assignee: Microsoft Corporation.

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

[13] Council for IBM Corporation. Software birthmarks. Talk to BCS Technology of Software Protection Special Interest Group. Reported in [4], 1985.

[14] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction systems. In *Electronic Colloquium on Computational Complexity, technical reports*, 1996.

[15] I. P. Goulden and D. M. Jackson. *Combinatorial Enumeration.* Wiley, New York, 1983.

[16] 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.

[17] Frank Harary and E. Palmer. *Graphical Enumeration.* Academic Press, New York, 1973.

[18] Ralf C. Hauser. Using the Internet to decrease Software Piracy - on Anonymous Receipts, Anonymous ID Cards, and Anonymous Vouchers. In *INET'95 The 5th Annual Conference of the Internet Society The Internet: Towards Global Information Infrastructure*, volume 1, pages 199–204, Honolulu, Hawaii, USA, June 1995.

[19] A. Herzberg and G. Karmi. On software protection. In *4th Jerusalem Conference on Information Technology*, Jerusalem, Israel, April 1984.

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

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

[22] 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`.

[23] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, USA, third edition, 1997.

[24] 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.

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

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

[27] 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`.

[28] 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.

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

[30] 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.

[31] Todd Proebsting. Optimizing ANSI C with superoperators. In *POPL'96*. ACM Press, January 1996.

[32] 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.

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

[34] 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.

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