On the Limits of Software Watermarking

Techical Report #164

Christian Collberg Clark Thomborson

Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland, New Zealand.
Phone: +64-9-373-7599
{*collberg,cthombor*}*@cs.auckland.ac.nz*

August 26, 1998

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 semantics preserving transformations such as code 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 [15] and a few recent US patents [10,20, 27,32], very little (publicly available) information seems to exist on *software watermarking* in which a copyright message is embedded into a program. This is in contrast to media watermarking which is a very active area of research [4,6,21, 29].

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.*

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. There are 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 is willing to accept some degradation in quality of $\mathcal{O}$, he can distort it so that it becomes impossible for Alice to detect the presence of $\mathcal{W}$ in $\mathcal{O}$. An *effective* distortive attack is one where the degraded watermark can no longer be detected 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.

Alice might, in some cases, be able to *tamperproof* her object against attacks from Bob. Tamper-proofing is any technique used by Alice specifically to render de-watermarking attacks ineffective. Figure 1 (a) illustrates these attacks and counter-measures.

Most media watermarking schemes seem vulnerable to attack by distortion. For example, image transforms (such as cropping and lossy compression) will distort the image enough to render many watermarks unrecoverable [4,29].

1.2 Attacks on Fingerprinting Systems

Fingerprinting is similar to watermarking, except a different watermark is embedded in every cover message. This may allow us to not only detect that theft has occurred, but also to trace the copyright violator. Fingerprinting objects make them vulnerable to *collusion attacks*. As shown in Figure 1 (b), 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 [2,17–19,25,26,33], 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 [3,23,24,34].

The choice of software watermarking technique will depend in part on the kind of object code we want to protect. 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 [31] and analyze. It is *easier* because Java's strong typing allows us to rely on the integrity of heap-allocated data structures.

In order to be able to watermark her code there are several questions Alice has to answer:

- In what kind of language structure should the watermark be embedded?
- How do we locate a suitable place within the application where we can store the watermark?
- How do we extract the watermark and prove that it is ours?
- How do we prevent Bob from removing or distorting the watermark?
- How do we prevent Bob from adding his own watermark?

The purpose of this paper is to examine these questions in detail and to survey the watermarking techniques available to Alice and the de-watermarking techniques available to Bob.

In particular, we will show that software watermarks are vulnerable to distortive attacks by *semantics preserving transformations*. Some simple watermarks that are stored in the code section of an executable can be destroyed by common optimizing transformations. *Obfuscating* transformations such as presented in [7–9], will, at some time/space penalty, effectively destroy most any kind of program structure. As a consequence, any software watermarking technique must be evaluated with respect to its resilience to attack from optimizing and obfuscating transformations.

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

2 Static Software Watermarking

Static watermarks are stored in the application executable itself. In a Unix environment this is typically within the initalized 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.

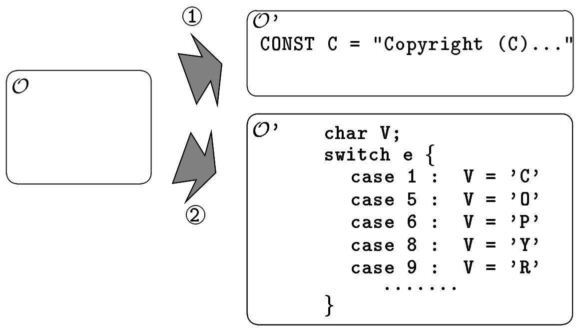

Figure 2: Static watermarks. In ① Alice embeds a watermark in the initialized data (string) section of her program. In ② the watermark is embedded in the text (code) section of the program.

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
```

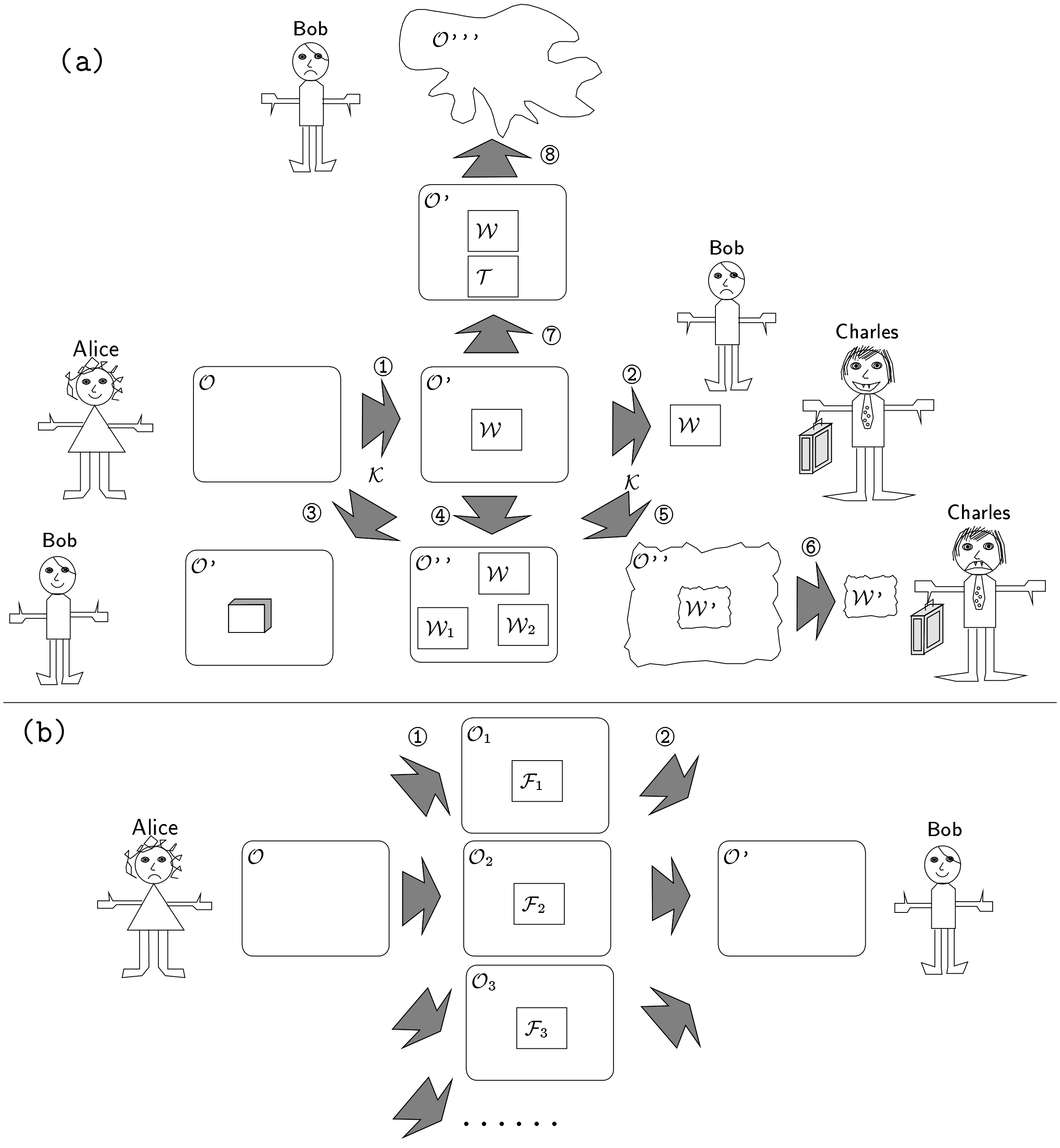

Figure 1: (a) shows attacks on watermarks and counter-measures against such attacks. At ① Alice adds a watermark $\mathcal{W}$ using key $\mathcal{K}$ to her object $\mathcal{O}$ to make $\mathcal{O}$'. At ② Bob steals a copy of $\mathcal{O}$' and Charles extracts the watermark using the key $\mathcal{K}$ to show that $\mathcal{O}$' is owned by Alice. ③ shows an effective *subtractive* attack, where Bob successfully removes $\mathcal{W}$ from $\mathcal{O}$. ④ 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. At ⑤ 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 ⑥ Charles attempts to extract the watermark from the distorted object, and either fails completely or gets a distorted watermark. At ⑦ Alice adds tamperproofing $\mathcal{T}$ to $\mathcal{O}$. ⑧ 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. (b) shows collusive attacks on fingerprints. At ① Alice creates several versions of her object $\mathcal{O}$, each with a different fingerprint (serial-number) $\mathcal{F}$. At ② Bob steals three copies of $\mathcal{O}$ and by comparing them is able to extract the original object, minus the fingerprint.

Moskowitz [27] 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 (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], as shown in Figure 3.

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 $S_1; S_2$, they can be flipped in either order. A watermarking bit could then be encoded in whether $S_1; S_2$ are in lexicographic order or not (Figure 4 ①).

There are many variations of this technique. When litigating against software pirates who had copied their PC-AT ROM, IBM [12] 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 (Figure 4 ②).

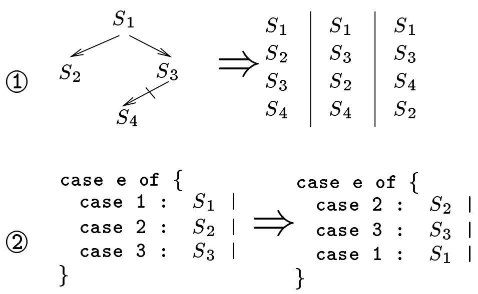

Figure 4: In ① four statements are reordered subject to data- and control-dependency constraints. In ② we show one possible reordering of three switch-statement cases.

Davidson [10] describes a similar code watermark 5 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 [10] 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, it is easy to destroy the apparent flow-of-control of a routine by inserting bogus predicated branches which break up basic blocks (see Figure 6).

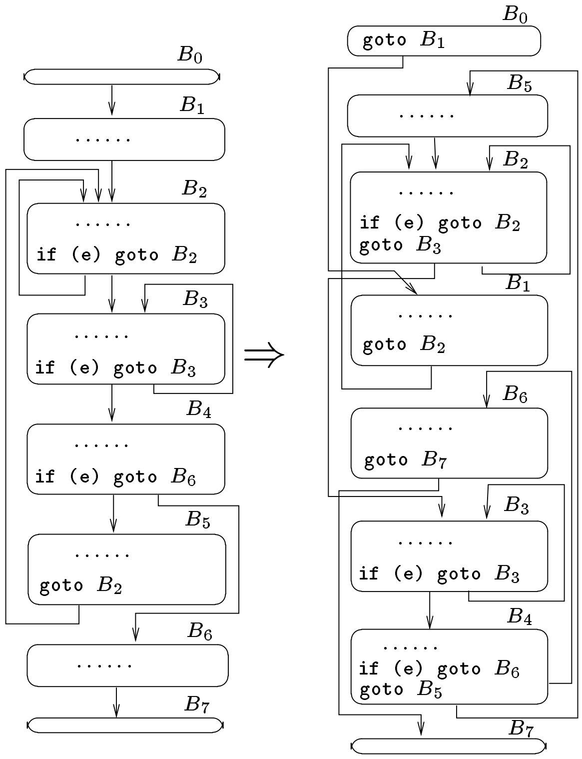

Figure 5: Encoding a signature into the control flow graph of a procedure [10]. Many simple optimizing transformations (inlining, outlining, etc.) will destroy the signature.

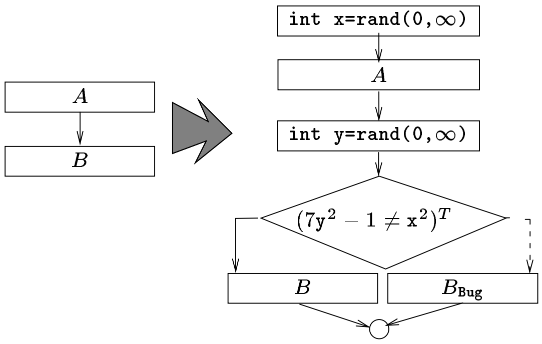

Figure 6: Splitting a basic block consisting of two statements A and B by inserting bogus predicates. In the example an opaque predicate $(y^2 - 1 \neq x^2)^T$ is inserted. This particular predicate is always true although this is difficult to work out statically.

2.3 Tamperproofing Static Watermarks

Our experience with obfuscation tells us that all static structures of a program can be successfully scrambled by obfus-

```
main() {
   String S1,S2,S3,S4;
   S1 = "AAA";
   S2 = "BAAA";
   S3 = "CCB";
   S4 = "CCB";
}
```

$\stackrel{\mathcal{T}}{\Rightarrow}$

```
static String G (int n) {
   int i=0;
   int k;
   char[] S = new char[20];
   while (true) {
      L1:  if (n==1) {S[i++]='A'; k=0; goto L6};
      L2:  if (n==2) {S[i++]='B'; k=-2; goto L6};
      L3:  if (n==3) {S[i++]='C'; goto L9};
      L4:  if (n==4) {S[i++]='X'; goto L9};
      L5:  if (n==5) {S[i++]='C'; goto L11};
           if (n>12) goto L1;
      L6:  if (k++<=2) {S[i++]='A'; goto L6}
           else goto L8;
      L8:  return String.valueOf(S);
      L9:  S[i++]='C'; goto L10;
      L10: S[i++]='B'; goto L8;
      L11: S[i++]='C'; goto L12;
      L12: goto L10;
   }
}
```

$\Downarrow\mathcal{T}$

```
main() {
   String S1,S2,S3,S4,S5;
   S1 = G(1);
   S2 = G(2);
   S3 = G(3);
   S4 = G(5);
}
```

Figure 3: De-watermarking static data watermarks. To obfuscate the static strings "AAA", "BAAAA", and "CCB" we construct a function G which produces the values G(1)="AAA", G(2)="BAAAA", G(3)=G(5)="CCB", and G(4)="XCB".

cating 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.

A further complication is that it is very difficult to tamperproof 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. Hence, 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 7): **Data Structure Watermark**, **Execution Trace Watermark**, and **Easter Egg Watermark**. While Easter Egg watermarks are very popular [28], there seems to be no published accounts of data structure or execution trace watermarks.

3.1 Easter Egg Watermark

Figure 7 ① 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 [28].

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 7 ② 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,

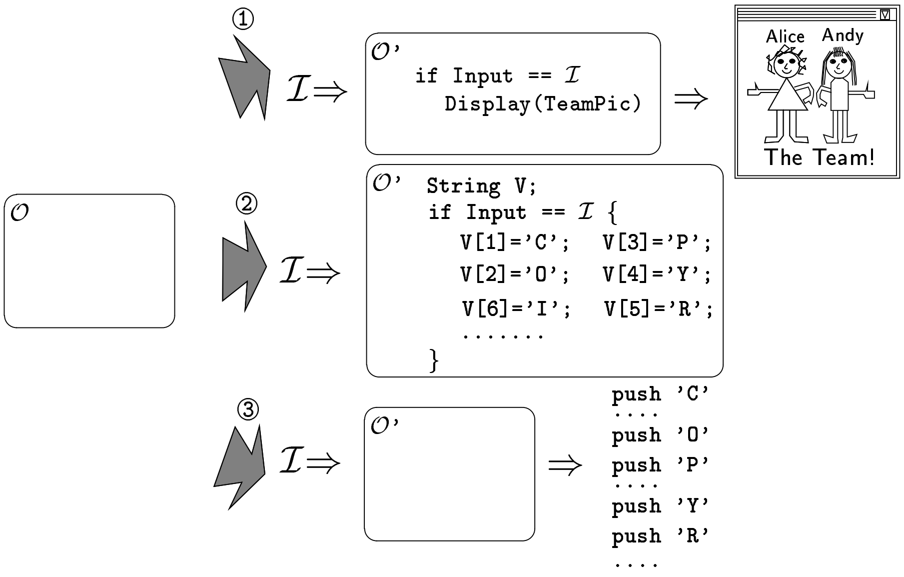

Figure 7: In ① the watermark is embedded in the unexpected behavior (an "Easter Egg") of the program when it is run with input $\mathcal{I}$. In ② the watermark gets embedded in a global variable V when the program is run with input $\mathcal{I}$. In ③ the watermark is embedded in the execution trace when the program is run with input $\mathcal{I}$.

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 wich will effectively destroy the dynamic state and make watermark recognition impossible. For example, in [8] we show how one variable can be split into several variables (Figure 8) and how several variables can be merged into one (Figure 9). 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 7 ③ 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.

Many of the same transformations that can be used to obfuscate code will also effectively obfuscate an instruction trace. Figure 10 shows another, more potent, transformation. The idea 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. The execution trace of the new virtual machine running the obfuscated program will be completely different from that of the original program. In most cases this will not be a practical attack because of the extra overhead of interpretation.

4 A Formal Model of Watermarking

In the next section we will build construct new techniques which are resilient to a variety of 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 after having processed input I. Let $|S(P, I)|$ be the size of this state, in accessible words. □

A program transformation is *semantics preserving* if it preserves input-output behavior. It is *state preserving* if internal state is 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:

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

Similarly, $\mathbb{T}_{\text{stat}} \subset \mathbb{T}$ is the set of *state preserving* transformations:

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

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

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

```
(1) bool A,B,C;
(2) B = False;
(3) C = False;
(4) C = A & B;
(5) C = A & B;
(6) if (A) ...;
(7) if (B) ...;
```

$\xRightarrow{\mathcal{T}}$

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

Figure 8: Variable splitting example. We show one possible choice of representation for split boolean variables. The table indicates that boolean variable V has been split into two short integer variables p and q. If $p = q = 0$ or $p = q = 1$ then V is `False`, otherwise, V is `True`. Given this new representation, we devise substitutions for the built-in boolean operations. In the example, we provide a run-time lookup table for each operator. Given two boolean variables $V_1 = [p, q]$ and $V_2 = [r, s]$, $\ulcorner V_1 \& V_2 \urcorner$ is computed as $\ulcorner \mathtt{AND}[2p + q, 2r + s] \urcorner$.

$$
\begin{aligned}
\mathtt{Z}(\mathtt{X}+r,\mathtt{Y}) &= 2^{32}\cdot\mathtt{Y}+(r+\mathtt{X}) &&= \mathtt{Z}(\mathtt{X},\mathtt{Y})+r\\
\mathtt{Z}(\mathtt{X},\mathtt{Y}+r) &= 2^{32}\cdot(\mathtt{Y}+r)+\mathtt{X} &&= \mathtt{Z}(\mathtt{X},\mathtt{Y})+r\cdot 2^{32}\\
\mathtt{Z}(\mathtt{X}\cdot r,\mathtt{Y}) &= 2^{32}\cdot\mathtt{Y}+\mathtt{X}\cdot r &&= \mathtt{Z}(\mathtt{X},\mathtt{Y})+(r-1)\cdot\mathtt{X}\\
\mathtt{Z}(\mathtt{X},\mathtt{Y}\cdot r) &= 2^{32}\cdot\mathtt{Y}\cdot r+\mathtt{X} &&= \mathtt{Z}(\mathtt{X},\mathtt{Y})+(r-1)\cdot 2^{32}\cdot\mathtt{Y}
\end{aligned}
$$

```
(1) int X=45;
    int Y=95;
(2) X += 5;
(3) Y += 11;
(4) X *= c;
(5) Y *= d;
```

$\xRightarrow{\mathcal{T}}$

```
(1') long Z=167759086119551045;

(2') Z += 5;
(3') Z += 47244640256;
(4') Z += (c-1)*(Z & 4294967295);
(5') Z += (d-1)*(Z & 18446744069414584320);
```

Figure 9: Merging two 32-bit variables `X` and `Y` into one 64-bit variable `Z`. `Y` occupies the top 32 bits of `Z`, `X` the bottom 32 bits. If the actual range of either `X` or `Y` can be deduced from the program, less intuitive merges could be used. First we give rules for addition and multiplication with `X` and `Y`, then show some simple examples.

```
int Sum(int A[]) {
  int i, sum=0;
  int n=A.length;
  for (i=0;i<n;i++)
    sum += A[i];
  return sum;
}
```

$\xRightarrow{\mathcal{T}}$

```
int Sum(int A[]) {
  int sum=0, i=0, pc=0;
  int s[]=new int[5], sp=-1;
  loop:  while (true)
    switch("fcgabced".charAt(pc)) {
      case 'a':  sum += s[sp--]; pc++; break;
      case 'b':  i++; pc++; break;
      case 'c':  s[++sp] = i; pc++; break;
      case 'd':  if (s[sp--] > s[sp--]) pc -= 6;
                 else break loop; break;
      case 'e':  s[++sp] = A.length; pc++; break;
      case 'f':  pc += 5; break;
      case 'g':  s[sp] = A[s[sp]]; pc++; break;
    }
  return sum;
}
```

Figure 10: The Java method `Sum` on the left is obfuscated by translating it into the bytecode `"fcgabced"`. This code is then executed by a stack-based interpreter specialized to handle this particular virtual machine code. This technique is similar to Proebsting's superoperators [30].

□

In [29] Peticolas writes: "the problem [with watermarking] is not so much inserting the marks as recognizing them afterwards". Hence, watermark recognition is defined with respect to the set of transformations under which the watermark can be extracted:

DEFINITION 4 (WATERMARK RECOGNIZER) $\mathcal{R}_T(P_w, S(P_w, I))$ is a *recognizer* of $w \in \mathbb{W}$ in $P_w \in \mathbb{P}$ with input I wrt a set of transformations $T \subset \mathbb{T}$, if,

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

□

This notation allows us to define several useful recognizers:

- $\mathcal{R}_\emptyset(P_w, S(P_w, I))$ is the *trivial* recognizer that cannot extract w if *any* transformations have been performed on P_w.
- $\mathcal{R}_T(P_w, \emptyset)$ 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.
- $\mathcal{R}_{\mathbb{T}_{\text{sem}}}(P_w, S(P_w, I))$ is a *strong* recognizer that is resilient to any semantics preserving transformation.

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 5 (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. □

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 6 (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.

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

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

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

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

□

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. We should therefore concentrate on constructing watermarks that are likely to survive under a variety of threat models, rather than setting our sights on methods that are completely resistant to all kinds of attack. 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 and subtractive 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 11 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 the P_w, the recognizer extracts n from P_w, and factors n. A similar *static* watermarking scheme base on public-key cryptography has been proposed by Samson [32]. Obviously, $p(w)$ can be based on other hard graph problems, such as the lattice problems described in [1,13].

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 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 issue 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 12 illustrates a Radix-6 encoding. The structure of the graph is a circular linked list with an extra pointer field which encodes a base-k digit. 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 bit-rate is $\log_2(m +$

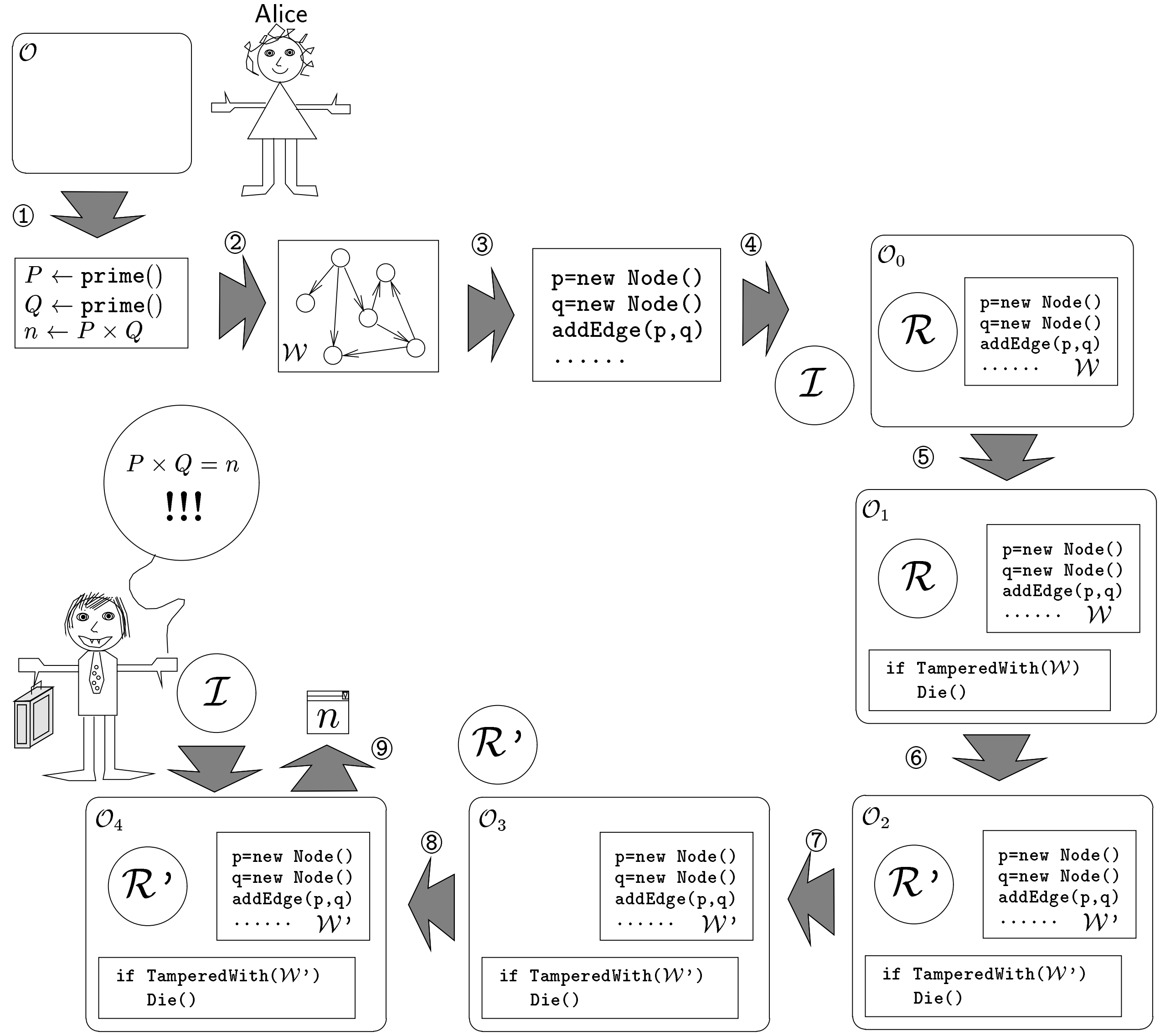

Figure 11: At ① Alice selects two large primes P and Q, and computes their product n. At ② she embeds n in the topology of a graph. This graph is her watermark $\mathcal{W}$. At ③ $\mathcal{W}$ is converted to a program which builds the graph. At ④ the program is embedded into the original program $\mathcal{O}$, such that when $\mathcal{O}_0$ is run with $\mathcal{I}$ as input, $\mathcal{W}$ is built. Also, a recognizer program $\mathcal{R}$ is constructed, which is able to identify $\mathcal{W}$ on the heap, and extract n from it. At ⑤ tamperproofing is added, to prevent an adversary to transform the graph to such an extent that $\mathcal{R}$ cannot identify it. At ⑥ the application (including the watermark, tamperproofing code, and recognizer) is obfuscated to prevent a attacks by pattern-matching. At ⑦ the recognizer is removed from the application. $\mathcal{O}_3$ is the version of Alice's program that is distributed. At ⑧ Charles links in the recognizer program $\mathcal{R}$ with $\mathcal{O}_3$. At ⑨ the application is run with $\mathcal{I}$ as input, and the recognizer $\mathcal{R}$ produces n. Since Charles is the only one who can factor n, he can prove the legal origin of Alice's program.

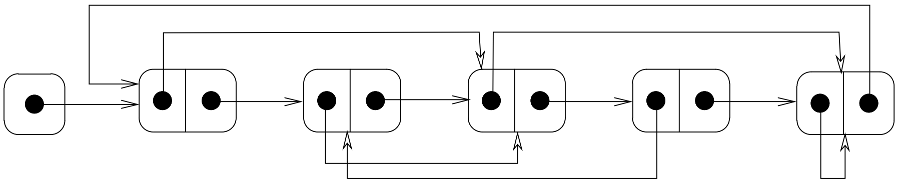

Figure 12: Radix-k encoding of a number into a graph structure. The structure is essentially a linked list where the rightmost pointer of each node is the `next` field. The leftmost pointer encodes a digit in the length of the path from the node back to itself. A `null` pointer represents 0, a self-pointer represents 1, a pointer to the next node represents 2, etc. This allows us to encode a value $61 * 73 = 4453_{10}$ as the base-6 value 32341_6.

$1)^m/(2m+1) \approx (\log_2 m)/2$. For $m = 255$ we can hide $255 \cdot 8 = 2040$ bits in 511 words of storage, or 4 hidden bits per word.

5.2.2 Enumeration Encoding

Our second embedding method uses results from graph enumeration [16]. The idea is to let the watermark number n be represented by the *index* of the watermark graph G in some convenient enumeration. The requires us to be able to (a) given n, generate the n:th graph in the enumeration, and (b) 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 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 [22, 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 bit-rate of $1024/1.56 \approx 1.56$ hidden bits per word.

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 13 for an example.

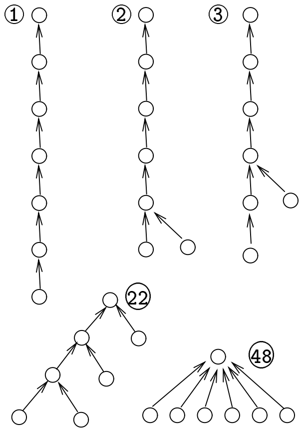

Figure 13: Some of the trees in the proposed enumeration of the oriented trees with seven vertices.

5.3 Recognizing the Watermark

At step ④ of Figure 11 we select the length k of the input sequence $\mathcal{I}$ and separate G into k components, $\mathcal{W}_1 \cdots \mathcal{W}_k$. The code to build these components is now 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 (see Figure 14 (a)).

It might seem that in order to identify G we whould need to examine all reachable heap objects, which, of course, would be intractable. In fact, Figure 14 (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 $\mathcal{W}_k$, 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 types of obfuscating transformations discussed in Section 2 and 3 will have no effect on the dynamic 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. This will make it hard for the recognizer to identify the real graph within a lot of extra bogus pointer fields.

(a)

if (input = $\mathcal{I}_1$) $\mathcal{W}_1 = \cdots$;
if (input = $\mathcal{I}_2$) $\mathcal{W}_2 = \cdots$;
if (input = $\mathcal{I}_3$) $\mathcal{W}_3 = \mathcal{W}_2 \oplus \mathcal{W}_3$;
......
if (input = $\mathcal{I}_k$) $\mathcal{W} = \mathcal{W}_1 \oplus \mathcal{W}_3 \oplus \cdots$;

(b)

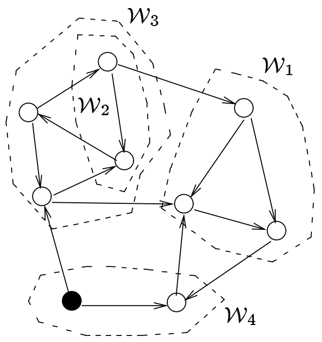

Figure 14: Code to build the watermark graph (a) and the graph and its components (b). The root node is black.

2. rename and reorder the fields in the node, again making it hard to recognize the real watermark.
3. add levels of indirection, for example by splitting nodes into several linked parts.
4. add extra bogus nodes pointing into our graph, preventing us from finding the root.

These transformations are illustrated in Figure 15.

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 13, for example, is resilient to renaming and reordering attacks since each node only has one pointer. Figure 16 shows another representation which is resilient to node-splitting attacks.

5.5.1 Tamperproofing by Reflection

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

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

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

```
Field[] F = C.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 = C.class.getFields();
int n=0;
for(int i=0; i<F.length; i++)
 if (F[i].getType().isAssignableFrom(C.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 Non-Semantics-Preserving 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 he must apply obfuscation 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 inserting extra nodes or edges. To thwart these sort of attacks, P_w should occasionally check the integrity of G.

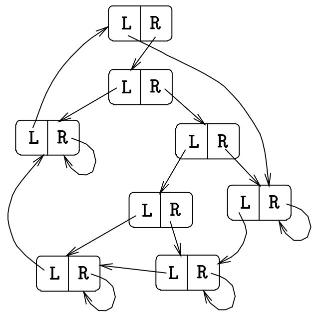

Figure 17: A planted plane cubic tree on $2m = 8$ nodes.

For example, consider the class G_m of *planted plane cubic trees* (See Figure 17) on m leaf nodes $v_1, v_2, ...v_m$, as enumerated in [14]. 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 its right subtree is l-linked to the right-most child of its left subtree.

5.6 Discussion

Obfuscating linked structures has some very serious consequences to the memory requirement of an adversary's de-watermarked program. For example, splitting a node costs one pointer cell plus the usual object overhead (2-3 words in Java). Furthermore, since we can assume that an adversary does not know in which dynamic structure our watermark is hidden, he is going to have to obfuscate every dynamic memory allocation in the *entire* program in order to be certain the watermark has been obliterated. This could easily double the program's memory requirement.

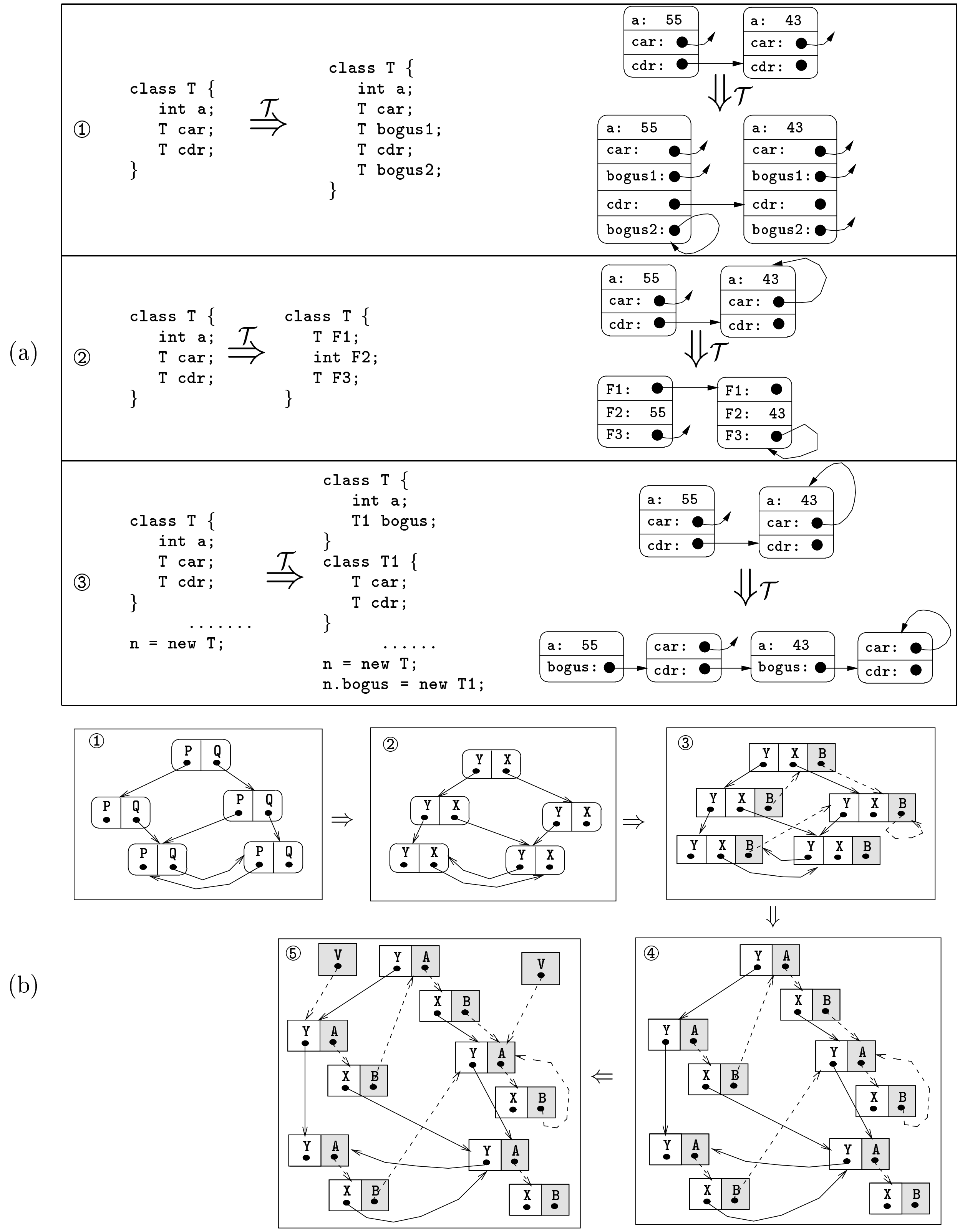

Figure 15: (a) shows various types of obfuscation attacks against linked structures. In ① we add bogus pointer fields to all nodes of type T. In ② we rename and reorder fields. In ③ we add a level of indirection by splitting all nodes in two. In (b) we give an example. ① shows our original watermark graph. In ② an adversary has renamed and reordered node pointer fields. In ③ each node has received a bogus pointer field B and bogus edges have been added. In ④ each node has been split in two by adding a bogus pointer field A. Finally, in ⑤ bogus nodes have been allocated which point into the graph, obscuring which node is the root.

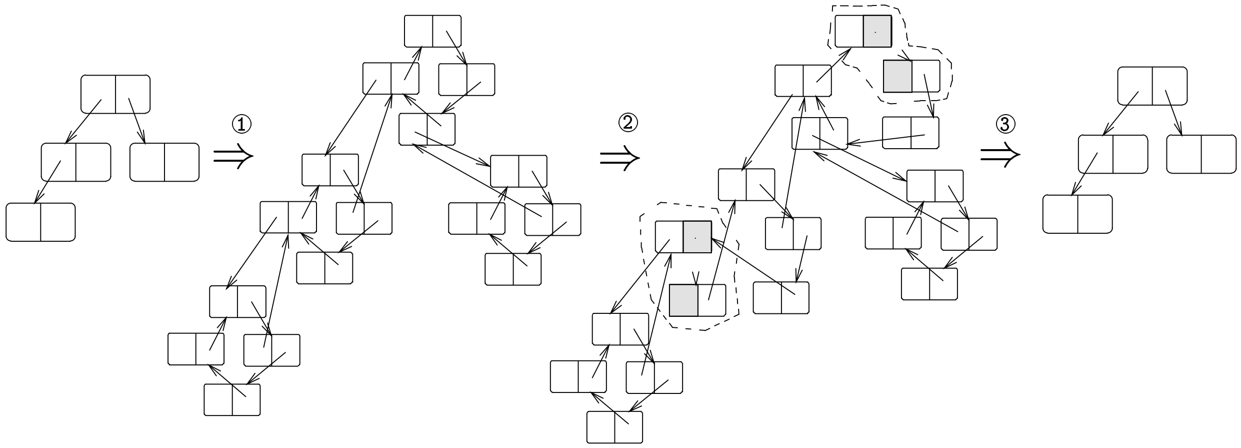

Figure 16: Tamperproofing against node-splitting. At ① we expand each node of our original watermark tree into a 4-cycle. At ② an adversary splits two nodes. The structure of the graph ensures that these nodes will still fall on a cycle. At ③ the recognizer shrinks the biconnected components of the underlying (undirected) graph. The result is a graph isomorphic to our original watermark.

6 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 semantics-preserving 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) all describe trivial schemes in which copyright notices are embedded in the object code of a program. None of these methods are resilient to even the simplest program transformations.

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 practical software watermarking techniques in which marks are embedded within the topology of dynamic heap data structures.

Acknowledgment: We would like to thank P. Gibbons, S. Cheng, and the members of STAR Lab 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/CollbergTh%omborsonLow97a`.

[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/CollbergTh%omborsonLow98b/`.

[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/CollbergTh%omborsonLow97c/`.

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

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

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

[15] Derrick Grover. *The Protection of Computer Software – Its Technology and Applications*, chapter 6, pages 122–154. The British Computer Society Monographs in Informatics. Cambridge University Press, second edition, 1992. ISBN 0 521 42462 3.

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

[17] 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.

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

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

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

[21] 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`.

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

[23] 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.

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

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

[26] 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`.

[27] 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.

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

[29] 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.

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

[31] 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.

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

[33] 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.

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