Dynamic Graph-Based Software Watermarking

Christian Collberg,* Clark Thomborson,† Gregg M. Townsend‡

Technical Report TR04-08

April 28, 2004

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 this paper we describe a software watermarking technique in which a dynamic graph watermark is stored in the execution state of a program. Because of the hardness of pointer alias analysis such watermarks are difficult to attack automatically.

1 Introduction

Steganography is the art of hiding a secret message inside a *host* (or *cover*) message. The purpose is to allow two parties to communicate surreptitiously, without raising suspicion from an eavesdropper. Thus, steganography and cryptography are complementary techniques: cryptography attempts to hide the *contents* of a message while steganography attempts to hide the *existence* of a message. History provides many examples of steganographic techniques used in the military and intelligence communities. For example, secret dispatches have been sent using invisible ink, hidden tattoos, microdots, etc.

Steganography—in the form of media *watermarking* and *fingerprinting*—has also found commercial applications. In image watermarking a copyright notice that identifies the intellectual property owner is imperceptibly embedded into the host image. Web spiders have been constructed that trawl the Internet for watermarked images that have been used without permission from their copyright owner. In media fingerprinting a customer identifier (such as a credit card number) is embedded into the host. This allows the intellectual property owner to trace the original purchaser of a pirated media object.

Media watermarking algorithms have been devised for many different host types, including images, audio, video, and text [9]. Algorithms typically exploit limitations in the human perception system. Audio watermarks, for example, can be hidden in short echos which humans have difficulty detecting. Many attacks against media watermarks have also been constructed. StirMark [48], for example, is able to defeat many image watermarking schemes by applying sequences of image transforms.

*Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, `collberg@cs.arizona.edu`

†Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand. `cthombor@cs.auckland.ac.nz`

‡Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, `gmt@cs.arizona.edu`

Our interests lie in the watermarking of *software*. Conceptually, a software watermarking system consists of functions

$$\begin{aligned} embed(P, w, key) &\rightarrow P_w \\ extract(P_w, key) &\rightarrow w \\ recognize(P_w, key, w) &\rightarrow [0.0, 1.0] \\ attack(P_w) &\rightarrow P'_w \end{aligned}$$

where *embed* transforms a program P into P_w by embedding the watermark w using the secret key *key* and *extract* extracts w from P_w. In *non-blind* watermarking systems *extract* is replaced by a *recognize* function that returns the likelihood that a particular watermark occurs in P. The *attack* function attempts to destroy the watermark embedded in P_w such that $extract(P'_w, key) \not\rightarrow w$. In other words, *attack* models the capabilities of a software pirate trying to remove a watermark from a protected program prior to illegally redistributing it.

This paper makes the following contributions:

- We present the first complete implementation of a *dynamic* software watermarking algorithm (known as the Collberg-Thomborson (CT) algorithm) for Java bytecode.
- We introduce several techniques for improving the stealth of the introduced watermark code, the data rate of the embedding, and the resilience of the watermark against automatic attacks.
- We evaluate the CT algorithm with respect to stealth, data rate, and resilience to attack. This is the first software watermarking algorithm which has been evaluated to this level of detail.

The remainder of the paper is structured as follows. In Sections 1.1 through 1.5 we discuss watermarking in general, attacks on watermarking systems, the ideas behind the CT watermarking algorithm, and the design of the SandMark software protection research tool in which CT has been implemented. In Section 2 we present an informal classification of software watermarking algorithms, discuss attacks against software watermarks, and review the previous literature. In Section 3 we present the basic implementation of the CT algorithm. In Sections 4, 5, and 6 we discuss methods for improving the resilience, bit-rate, and stealth of the algorithm, respectively. In Section 7 we empirically evaluate the algorithm, in Section 8 we discuss our findings and in Section 9 we summarize.

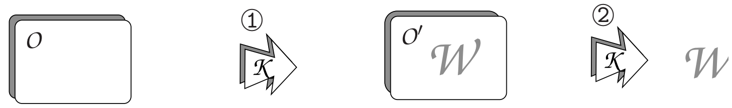
1.1 Attacks on Watermarking Systems

The strength of any steganographic system is a function of its *data rate*, *stealth*, and *resilience*. The data rate expresses the number of bits of hidden data that can be embedded within each kilobyte of 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 data rate.

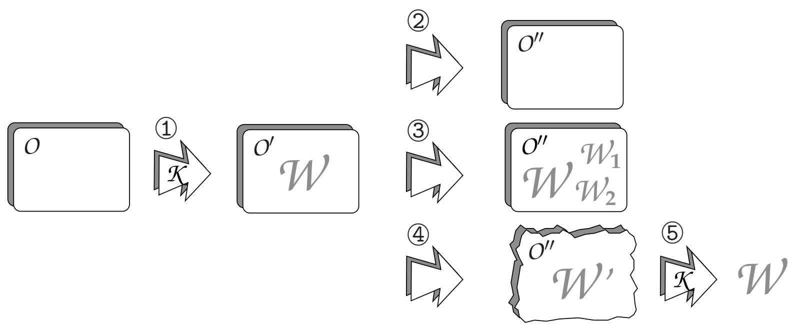
To evaluate the quality of a watermarking scheme we must also know how well it stands up to *different types* of attacks. In general, no steganographic scheme is immune to all attacks, and often several techniques have to be employed simultaneously to attain the required degree of resilience. In [9] 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 O with watermark $\mathcal{W}$ and key $\mathcal{K}$, and then sells O to Bob. Before Bob can sell 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:

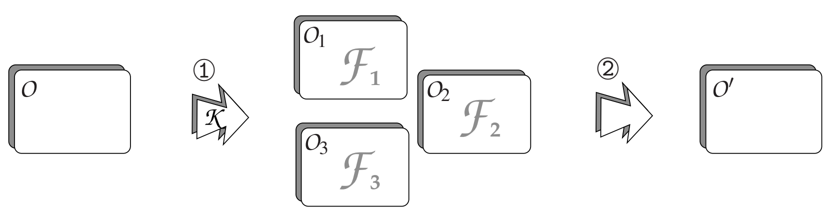
subtractive attack If Bob can detect the presence and (approximate) location of $\mathcal{W}$, he may try to *crop* it out of O. An *effective* subtractive attack is one where the cropped object has retained enough original content to still be of value to Bob.



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



(b) ② shows an effective *subtractive* attack, where Bob successfully removes $\mathcal{W}$ from 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 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 ① Alice creates several versions of 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 O_1, O_2, and O_3.

Figure 1: Attacks on watermarking systems.

distortive attack If Bob cannot locate $\mathcal{W}$ and is willing to accept some degradation in quality of 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 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. 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 [5, 47].

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 *collusive attacks*. As shown in Figure 1(c), 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 [4, 30, 28, 54, 31, 34, 38], software watermarking is an area that has received very little attention. This is unfortunate since software piracy is rampant. A sizeable fraction, estimated at 39% with a valuation of 13 billion dollars, of business application software is installed annually without a license [BSA2003,Malhotra94]

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 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 [50] and analyze. It is *easier* because Java's strong typing allows us to rely on the integrity of heap-allocated data structures.

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 [16]), *optimization*, *compression* [20], and *obfuscation* [14, 15, 13].

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?

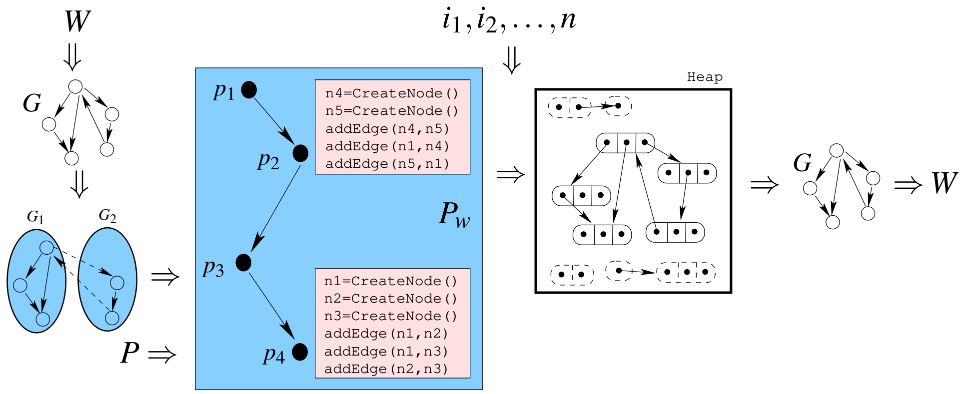


Figure 2: Overview of the CT algorithm.

1.4 The CT Algorithm

The CT software watermarking algorithm is a *dynamic* algorithm. The idea is that rather than embedding the watermark directly in the *code* of the application, code is embedded that *builds* the watermark at runtime. The algorithm assumes a secret key $\mathcal{K}$ which is necessary to extract the watermark. $\mathcal{K}$ is a sequence of inputs $I_0, I_1, \ldots$ to the application.

The embedding and extraction processes are illustrated in Figure 2. The watermark number W is embedded in the topology of a graph G; the graph is split into several components $G_1, G_2, \ldots$; each G_i is converted into Java bytecode C_i that builds it; and each C_i is embedded into the application along the execution path that is taken on the special input $I_0, I_1, \ldots$. During extraction the watermarked application is run with $I_0, I_1, \ldots$ as input, the watermark graph gets built on the heap, the graph is extracted and the watermark number is recovered.

There are several motivations for this design. First of all, because of pointer aliasing effects it is difficult to analyze code that builds graph structures [23, 52]. Thus it would be difficult for an attacker to statically analyze a watermarked program to look for code that builds a watermark graph or to destroy the graph using semantics-preserving transformations. Second, object-oriented programs typically contain a large number of the types of operations necessary to build a graph, namely object allocations and pointer assignments. Thus the code that gets inserted is likely to fit in with surrounding code. Third, since a large graph can be split into an arbitrary number of components that can be spread over the entire program, it should be possible to stealthily embed large watermarks. Fourth, and finally, since the watermark is data rather than code it should be easier to tamperproof. If the watermark graph is selected to have a particular property (such as being planar, having a certain diameter, etc.) code can be inserted to test this property. This is in contrast to static code watermarks for which tamperproofing requires the code segment of the executable to be examined. This is difficult to do stealthily.

Figure 3 shows a simple example of a what a program may look like after having been watermarked. The original program `Simple` is modified into `Simple_W` such that when run with the secret input argument `"World"` the watermark graph is built on the heap. In a typical implementation `Simple_W` and `Watermark` would be obfuscated to prevent attacks by pattern matching.

Palsberg et al. [46] present the first implementation of the CT algorithm. We compare their implementation to ours in Section 8.1.

1.5 SandMark

SandMark [11] is a tool for doing research on software protection algorithms. The goal is to provide an infrastructure for implementing and evaluating algorithms for code obfuscation, software watermarking, and tamperproofing. SandMark currently contains implementations of forty code obfuscation algorithms and fourteen watermarking algorithms. It also supports various reverse engineering tools such as a slicer, a bytecode differ [8], and a bytecode viewer. These can be used to aid *manual attacks* against software protection algorithms. A number of *software com-*

```java
public class Simple {
   static void P(String i) {
      System.out.println("Hello␣" + i);
   }
   public static void main(String args[]) {
      P(args[0]);
   }
}
```

⇓

```java
public class Simple_W {
   static void P(String i, Watermark n2) {
      if (i.equals("World")) {
         Watermark n1 = new Watermark();
         Watermark n4 = new Watermark();
         n4.edge1 = n1;
         n1.edge1 = n2;
         Watermark n3 = (n2 != null)?n2.edge1:new Watermark();
         n3.edge1 = n1;
      }
      System.out.println("Hello␣" + i);
   }
   public static void main(String args[]) {
     Watermark n3 = new Watermark();
     Watermark n2 = new Watermark();
     n2.edge1 = n3;
     n2.edge2 = n3;
     P(args[0], n2);
   }
}

class Watermark extends java.lang.Object {
  public Watermark edge1,edge2;
}
```

Figure 3: Example of a trivial class before and after being watermarked with the CT algorithm.

plexity metrics [27, 35, 40, 45, 25, 29] are provided for measuring the effect of code obfuscation and watermarking algorithms.

SandMark works on Java bytecode. A typical code obfuscation algorithm, for example, will read a Java jar-file (a collection of class files) as input and produce a modified jar-file as output.

SandMark is designed using a *plug-in* style architecture and supports a number of useful static analyses (class hierarchy, control-flow, call graph, def-use, stack-simulation, liveness, etc.) simplifying the development of new software protection algorithms. SandMark relies on BCEL [1] for bytecode editing, DynamicJava [2] for scripting, and BLOAT [44] for code optimization. SandMark is currently approximately 120,000 lines of Java code of which approximately 10,000 lines comprises the CT implementation.

The CT algorithm that will be described in this paper has been implemented within SandMark. It can be downloaded from `http://sandmark.cs.arizona.edu`. Appendix A shows a screenshot of SandMark recognizing a CT watermark.

2 Techniques for 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.

Static Data Watermarks are very common since they are easy to construct and extract. For example, several copyright notices can easily be extracted from the *Netscape 4.78* binary:

```
> strings /usr/local/bin/netscape | grep -i copyright
Copyright (C) 1998 Netscape Communications Corporation.
Copyright (c) 1996, 1997 VeriSign, Inc.
```

Media watermarks are commonly embedded in redundant bits, bits which we cannot detect due to the imperfection of our human perception. *Static 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.

Dynamic Watermarks are extracted from a program's execution state rather than from the program code or data itself. They were first introduced in [12]. The idea is that in order to extract a watermark from an application it is run with a special input sequence $I{=}I_1 \cdots I_k$ which makes it enter a state which represents the watermark. I can be thought of as a secret key known only to the owner of the software and unlikely to occur naturally.

Dynamic watermarking methods differ in which part of the program state the watermark is stored, and in the way it is extracted. When the special input sequence is entered, *Dynamic Easter Egg Watermarks* perform 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 the `Netscape 7.01` browser will make a message ⌜`And the beast shall be made legion. []`⌝ appear.

Dynamic Execution Trace Watermarks are extracted from the trace (either instructions or addresses, or both) of the program as it is being run with a particular input I. The watermark is extracted by monitoring some (possibly statistical) property of the address trace and/or the sequence of operators executed.

Dynamic Data Structure Watermarks are extracted by examining the current values held in the watermarked program's variables, after the end of the special 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.

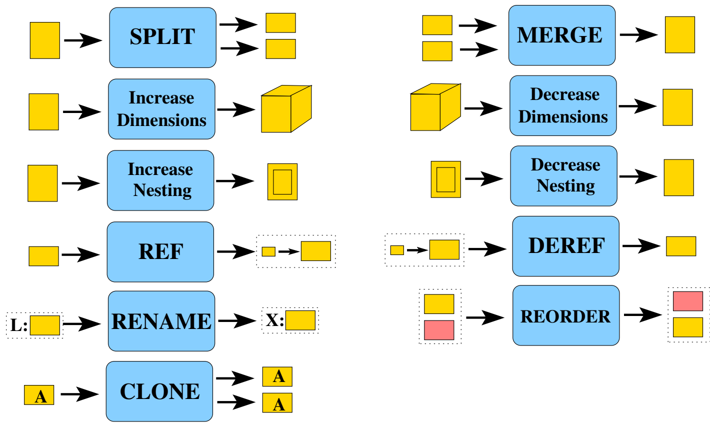
2.1 Attacks Against Software Watermarks

No known software watermarking method is completely immune to attack. In the worst case an adversary can study the input/output behavior of the watermarked application and completely rewrite it, sans the mark. We do not consider this a reasonable attack. We would, however, want the watermark to survive attacks by *translation*, *optimization*, and *obfuscation*, since tools that perform such operations are readily available [22, 21, 16, 44, 11] and can be used by even the most unsophisticated attacker.

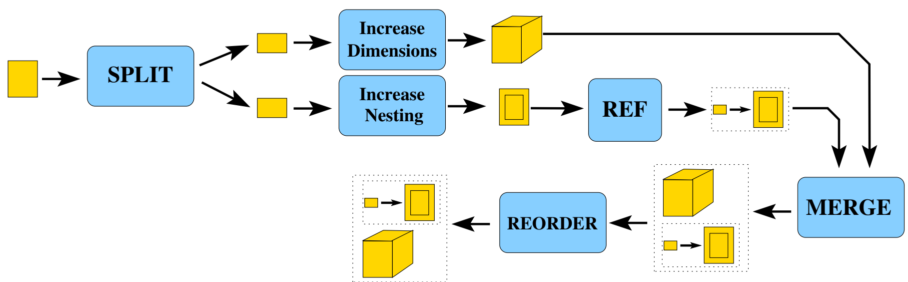
Static data watermarks are highly susceptible to distortive attacks by obfuscation. In the simplest case, an automatic obfuscator might break up all strings (and other static data) into substrings which are then scattered over the executable. This makes watermark extraction nearly impossible. A more sophisticated de-watermarking attack would convert all static data into *a program* that produces the data [14].

Many code obfuscation techniques [14, 15] will successfully thwart the extraction of code watermarks. Since software is such a fluid medium it is easy to devise transformations which will destroy just about any structure of a program. Figure 4(a) shows some of the basic code transformations on which code obfuscations can be built:

- A language construct can be *split* or two constructs can be *merged*. For example, an array or a method can be split into two halves, or two modules or two integer variables can be merged into one.



(a) Basic code obfuscation transformations.



(b) Example showing how simple obfuscating transformations can be combined.

Figure 4: Obfuscation attacks against software watermarks.

- The *dimensionality* of a construct can be increased or decreased. For example, an array can be folded or flattened.
- The *nesting level* of a construct can be increased or decreased. For example, the complexity of a method can be increased by turning a single loop into a loop nest, or a scalar variable can be boxed into a heap-allocated variable.
- A *level of indirection* can be added, for example by turning a static method into a virtual method.
- Language constructs such as methods, variables, and classes can be *renamed*.
- A compound language construct can be *reordered*. For example, two adjacent statements without data- or control-dependencies can be swapped.
- A language construct can be *cloned*. For example, a method can be duplicated, each clone can be differently obfuscated, and different calls can be made to invoke the different clones.

Most current commercial code obfuscators only perform name obfuscation. SandMark, however, supports a full-fledged obfuscation engine with transformations that affect control-flow, classes, methods, and data-structures.

While each individual obfuscation may only add a small amount of confusion, transformations can be cascaded, as shown in Figure 4(b).

Figure 5 shows the result of applying a sequence of obfuscating transformations to a simple program. The transformations were performed automatically by the SandMark tool and the resulting program was decompiled using Ahpah's `SourceAgain` decompiler. The transformations applied were:

1. *boolean splitting* (a boolean variable is split into two small integer variables);

```
public class C {
   static int gcd(int x, int y) {
      int t;
      while (true) {
         boolean b = x % y == 0;
         if (b) return y;
         t = x % y; x = y; y = t;
      }
   }
   public static void main(String[] a){
       System.out.print("Answer: ");
       System.out.println(gcd(100,10));
   }
}
```

$\Downarrow \mathcal{T}$

```
public class C {
 static Object get0(Object[] I) {
  Integer I7, I6, I4, I3; int t9, t8;
  I7=new Integer(9);
  for (;;) {
   if (((Integer)I[0]).intValue()%((Integer)I[1]).intValue()==0)
      {t9=1; t8=0;} else {t9=0; t8=0;}
   I4=new Integer(t8);
   I6=new Integer(t9);
   if ((I4.intValue()^I6.intValue())!=0)
     return new Integer(((Integer)I[1]).intValue());
   else {
     if ((((I7.intValue()+ I7.intValue()*I7.intValue())%2!=0)?0:1)!=1)
        return new Integer(0);
     I3=new Integer(((Integer)I[0]).intValue()%((Integer)I[1]).intValue());
     I[0]=new Integer(((Integer)I[1]).intValue());
     I[1]=new Integer(I3.intValue());
    }
  }
 }
 public static void main(String[] Z1) {
    System.out.print((String)Obfuscator.get0(
        new Object[] {(String)new Object[] {
"\u00AB\u00CD\u00AB\u00CD\uFF84\u2A16\u5D68\u2AA0\u388E\u91CF\u5326\u5604"
         }[0]}));
    System.out.println(((Integer)get0(new Object[]
          {(Integer)new Object[] {new Integer(100),new Integer(10)}[0],
          (Integer)new Object[] {
               new Integer(100),new Integer(10) }[1]})).intValue());
    }
}
```

Figure 5: Obfuscation example.

2. *basic block splitting* (a bogus branch protected by an *opaquely false predicate* [15] $(q+q*q) \bmod 2! = 0$ is inserted);

3. *string encoding* (the string `"Answer:"` is encoded into an unintelligible string which gets decoded at runtime);

4. *scalar promotion* (integer variables are converted to `java.lang.Integer` boxed integers);

5. *signature unification* (every method in the program is given the same `Object[]` signature, where possible);

6. *name obfuscation* (`gcd` is renamed `get0`).

It is obvious that many obfuscating transformations add an non-trivial amount of overhead. It may therefore well be the case that while a certain sequence of transformations would obliterate a particular watermark, the resulting de-watermarked program would be too slow or too large to have any value to the attacker. The goal of software watermarking research is to design marking algorithms that will be robust against semantics-preserving transformations that add an *acceptable* (to the attacker) amount of overhead. This is similar to the situation in image watermarking where obliterating a watermark by blurring the image beyond recognition is not considered a viable attack.

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 [43]. 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.

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 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 extraction routine is not shipped within an application that has been watermarked using a data structure watermark (it is linked in during watermark extraction), there is little information in the executable itself as to where the watermark may be located.

2.2 Software Watermarking Algorithms

Many simple watermarking algorithms have been based on reordering language constructs to embed the watermark. For example, by reordering the branches of an m-branch case-statement we can encode $\log_2(m!) \approx \log_2(\sqrt{2\pi m}(m/e)^m) = O(m \log m)$ watermarking bits:

```
switch (E) {                        switch (E) {
    case 1: {···}                       case 5: {···}
    case 5: {···}          ⟹           case 1: {···}
    case 9: {···}                       case 9: {···}
}                                   }
```

A similar idea is used in the first published static code watermarking algorithm, due to Davidson and Myhrvold [19]. The idea is to embed a watermark by rearranging the order in which the basic blocks of a control-flow graph (CFG) are laid out in the executable. See Figure 6(a) which shows how a watermark is encoded in the basic block sequence $\langle B_5, B_2, B_1, B_6, B_3, B_4 \rangle$. Like all other algorithms based on reordering this one is trivial to attack by randomly reordering the basic block layout of the program.

Qu and Potkonjak [51] propose to embed the watermark in the register allocation of a program. Like all software watermarking algorithms based on renaming structures of the program this algorithm is very fragile. Watermarks typically do not even survive decompiling and then recompiling the program. This algorithm also suffers from a low bit-rate [42].

Stern et al. [55] present an algorithm which uses a spread-spectrum technique to embed the watermark. The algorithm embeds the watermark by changing the frequencies of certain instruction sequences, replacing them by different but semantically equivalent sequences. A codebook gives equivalent instruction sequences that can be used to manipulate code frequencies. This algorithm is resilient to semantics-preserving transformations that only affect the high-level structure (such as the class-hierarchy and call-graph) of a program. However, it can be defeated by obfuscations that modify data-structures and data-encodings and many low-level optimizations [53]. See Figure 6(b).

Moskowitz [39] 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. See Figure 6(c).

Arboit's [6] algorithm embeds the watermark by adding special opaque predicates to the program. Extraction is done by pattern-matching on the opaque predicates, and hence attacks by pattern-matching are easily constructed.

Monden's algorithm [37, 36] adds a bogus method to the application. The bogus method is guarded by an always-false predicate, where the predicate may be generated from a programmer's assertion statement. The watermark is encoded steganographically in the opcodes and operands of the static code in the bogus method.

Pieprzyk [49] describes a digital-rights management scheme for software, with two alternative methods for watermarking. The first alternative is to embed the watermark signal in the choice of synonyms for short sequences of instructions, similarly to Stern et al. [55]. Pieprzyk notes that this method is weak against adversaries who make a "random selection of variants for each instruction."

Venkatesan's [56] algorithm constructs a CFG whose topology embeds the watermark number. This watermark CFG is attached to an existing CFG by adding bogus control-flow edges using opaque predicates. In order to be able to extract the watermark, basic blocks that belong to the watermark graph are *marked*. See Figure 6(d).

Cousot and Cousot [17] describe a watermarking process, in which the watermark is recognizable by a static analysis of program semantics in a secret model. The embedding key is a series $[n_1, n_2, ..., n_i]$ of relatively prime positive integers. The watermark c is any integer in the range $0..\Pi_i n_i - 1$. It is embedded as a series of residues $c_i = c \bmod ni$, computed in loops, where these c_i are constant in the secret semantic model used for watermark recognition. In the standard execution model, the values c_i are non-constant and may closely resemble the pseudorandom variates used in stochastic numerical computations such as Monte Carlo estimation algorithms. The watermark recognition process is essentially the static analysis done by an optimising compiler to recognize constant assignments in loops. This is a promising method in static data watermarking, however an attacker may be able to recognize the watermark data in commonly-occurring programs, because of its use of unusually complex literal constants. In a survey of 600 Java programs, we found that 92% of all literal integers are 2^k or $2^k + 1$, so the watermarking method of Cousot and Cousot would be non-stealthy in such programs. We see promise in this method, because it is not difficult to devise a method to hide arbitrarily-complex literal constants in secret, non-standard semantics for commonly-occuring code sequences in Java. The secret detection semantics for the Cousot and Cousot watermark must, however, remain simple if it is to be persuasive evidence in favour of the embedded watermark. Otherwise the attacker may construct an alternative secret semantics which fraudulently detects an arbitrary, non-existent watermark.

Many of these algorithms have been implemented within the SandMark framework.

Palsberg et al. [46] present a dynamic watermarker based on the CT algorithm. In this simplified implementation, the watermark is not dependent on a key input sequence, but is constructed unconditionally. The watermark value is represented as a planted planar cubic tree (PPCT) graph. Queries applied to the PPCT can be used as opaque predicates for obfuscation or tamperproofing. The CT algorithm was found to be practical and robust.

2.3 Tamperproofing 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 [14], various forms of loop transformations [7] and code motion are all well-known optimization techniques that will easily destroy static code watermarks.

Moskowitz [39] 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 (for example by the StirMark [48] suite of image transforms). Unfortunately, generating and executing code on the fly is unusual and unstealthy behavior for most applications. See Figure 6(c).

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

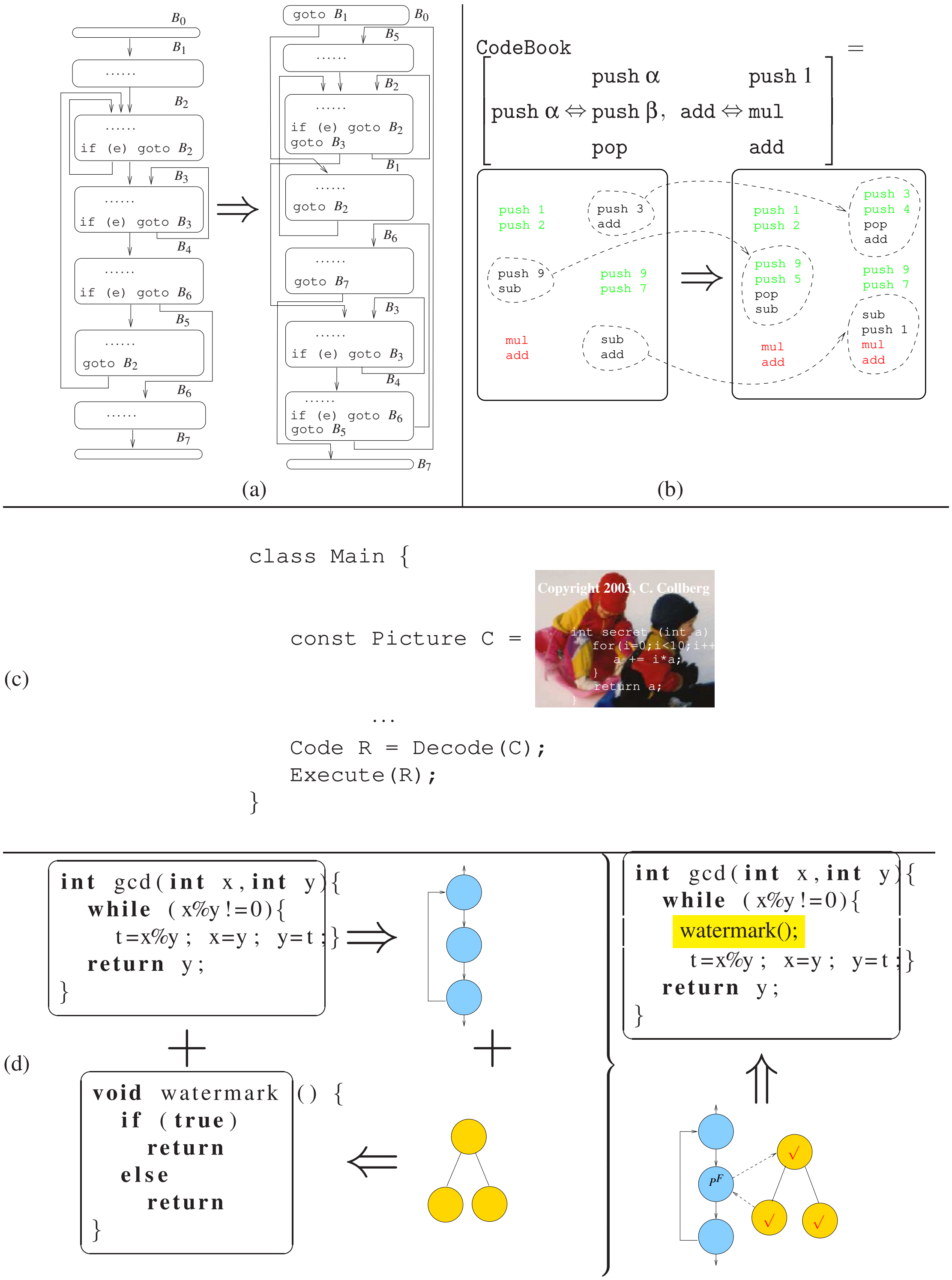


Figure 6: Watermarking algorithms. (a) Stern [55], (b) Davidson-Myhrvold [18], (c) DICE [39], (d) Venkatesan [56].

than the data segment of the executing program) and unstealthy.

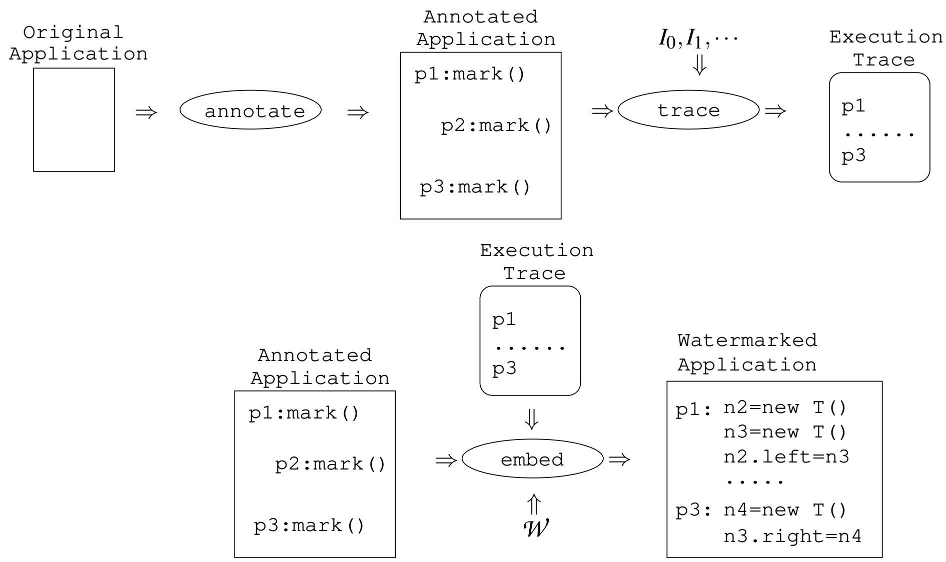


Figure 7: Overview of how the CT algorithm watermarks an application. First, the user adds *annotation points* (`mark()`-calls) to the application. These are locations where watermarking code may be inserted. Secondly, the application is run with a secret input sequence, $I_0, I_1, \ldots$ and the trace of `mark()`-calls hit during this run is recorded. Finally, code is embedded into the application (at certain `mark()`-call locations) that builds a graph G_W at runtime. The topology of $G_{\mathcal{W}}$ embeds the watermark $\mathcal{W}$.

3 The CT Algorithm — Basic Implementation

The Collberg-Thomborson watermarking algorithm (henceforth, CT) is a dynamic algorithm. The idea is that rather than embedding the watermark directly in the *code* of the application, code is embedded that *builds* the watermark at runtime. The algorithm assumes a secret key $\mathcal{K}$ which is necessary to extract the watermark. $\mathcal{K}$ is a sequence of inputs $I_0, I_1, \ldots$ to the application. As seen in Figure 2, the watermark (a graph structure) is built by the application only when the user runs it with the special input $I_0, I_1, \ldots$. Figure 3 shows a simple example of a what a program may look like after having been watermarked.

In the SandMark implementation of CT, watermark embedding and extraction runs in several steps (See Figure 7):

Annotation Before the watermark can be embedded the user must add *annotation* (or *mark*) points into the application to be watermarked. These are calls of the form

```
mark();
String S = ...;
mark(S);
long L = ...;
mark(L);
```

The `mark()` calls perform no action. They simply indicate to the watermarker locations in the code where (part of) a watermark-building code can be inserted. The argument to the `mark()` call can be any string or integer expression that (directly or indirectly) depends on user input to the application.

Tracing When the application has been annotated the user performs a *tracing* run of the program. The application is run with the chosen secret input sequence, I. During the run one or more annotation points will be hit. Some of these points will be the locations where watermark-building code will later be inserted.

Embedding During the embedding stage the user enters a watermark, a string or an integer. A string is converted to an integer. From this number a graph is generated, such that the topology of the graph embeds the number. The graph is converted to Java bytecode that builds the graph. The relevant `mark()`-calls are replaced with this graph-building code.

Extraction During watermark extraction the application is again run with the secret input sequence as input. The same `mark()`-locations will be hit as during the tracing run. Now, however, these locations will contain code for building the watermark graph. When the last part of the input has been entered, the heap is examined for graphs that could potentially be watermark graphs. The graphs are decoded and the resulting watermark number is reported to the user.

We will next consider these tasks in detail.

3.1 Annotation

The CT watermark consists of dynamic data-structures. This means that the code inserted in the application will look like this:

```
Watermark n1 = new Watermark();
Watermark n2 = new Watermark();
n1.edge = n2;
   ...
```

Hence, we should prefer mark locations that

- allocate objects and manipulate pointers, and
- directly depend on user input.

We should avoid mark locations that

- are hot-spots, and
- are executed non-deterministically.

In other words, `mark()`-calls should be added to locations where the resulting watermark code will be fit in (is *stealthy*), will not affect performance, and will be executed consistently from run to run, depending only on user actions.

For example, the following code is undesirable since `Math.random()` may generate different values during different runs of the program:

```
if (Math.random() < 0.5) {
   ...
   mark();
}
```

Similarly, if thread scheduling, network activity, processor load, etc. can affect the order in which some locations are executed, these locations are not valid annotation points and should be avoided.

3.2 Tracing

SandMark makes heavy use of Java's `JDI` (*Java Debugging Interface*) framework. During tracing and extraction SandMark starts up the user's application as a subprocess running under debugging. This allows SandMark to set breakpoints, examine variables, and step through the application – all the operations that can be done under an interactive debugger. During tracing we are interested in obtaining a trace of the `mark()`-calls that are hit while the

```
import java.awt.event.*;
import javax.swing.*;
public class Button implements ActionListener {
    static void P(int i) {
       L0:mark(i);
    }
    static void Q(int i) {
       L1:mark(i);
       if (i < 2) P(i);
    }
    public void actionPerformed(ActionEvent e) {
       L2:mark();
       Q(3);
    }
     public static void main(String argv[]) {
        Q(1);
        Q(2);
        JFrame jw = new JFrame();
        JButton b = new JButton("w00t!");
        b.addActionListener(new Button());
        jw.getContentPane().add(b);
        jw.pack(); jw.show();
     }
}
```

#	Value	Method	Location	Thread	Stack
Ⓐ	1	Q	L_1	1	$\langle$main,Q$\rangle$
Ⓑ	1	P	L_0	1	$\langle$main,Q,P$\rangle$
Ⓒ	2	Q	L_1	1	$\langle$main,Q$\rangle$
Ⓓ	$\emptyset$	aP	L_2	2	$\langle$aP$\rangle$
Ⓔ	3	Q	L_1	2	$\langle$aP,Q$\rangle$

Figure 8: An example Java program annotated for tracing and the generated tracepoints. The method `actionPerformed` is abbreviated `aP`. The corresponding trace forest is shown in Figure 18.

---

user enters their secret input. We also want to know the argument to the `mark()`-call and the stack trace at the point of the call. During extraction we use `JDI` to examine the objects on the heap to look for watermark graphs.

At the end of tracing run we have gathered a list of *TracePoint*s which represent the `mark()`-calls that were hit during the trace. Each *TracePoint* contains three pieces of information:

1. the location in the bytecode where the `mark()`-call was located;
2. the value of the expression *e* that the user supplied as an argument to the `mark(e)`-call, or $\emptyset$ if a parameterless `mark()`-call was hit;
3. a list of the stack-frames active when the `mark()`-call was hit.

Figure 8 shows an example application and the corresponding list of trace points.

Not all locations generated during the tracing run can be used to build the watermark graph and some may have to be removed. An annotation point $\langle value, location\rangle$ is valid if

1. there is exactly one trace point at *location*, or

2. there are multiple trace points at *location*, but they all have unique *value*s.

For example, consider the following `mark()`-points:

$$\begin{array}{c}\langle \emptyset, L_0\rangle \\ \langle 1, L_1\rangle \\ \langle 1, L_1\rangle \\ \langle 10, L_2\rangle \\ \langle 11, L_2\rangle \\ \langle 12, L_2\rangle\end{array}$$

$\langle \emptyset, L_0\rangle$ is valid, because it is the only `mark()`-point at location L_0. $\langle 1, L_1\rangle$ is not valid because there are two identical annotation values at this location. If we were to insert watermark-building code at this location we would not be able to tell the difference between the first and the second time we arrive. $\langle 10, L_2\rangle, \langle 11, L_2\rangle, \langle 12, L_2\rangle$ are valid because the *value*s are unique. If there is one unique value at a location, this `mark()`-call is said to be `LOCATION`-based, otherwise it is `VALUE`-based.

3.3 Embedding

Once the application has been traced we can start embedding the watermark. The input to this phase is tracing information, a watermark $\mathcal{W}$ to be embedded, and a jar-file containing the classfiles in which to embed the mark. The embedding is divided into five phases:

1. Generate a graph G whose topology embeds $\mathcal{W}$.
2. From G generate an *intermediate code* C that builds this graph.
3. Translate the intermediate code C into a Java method M that, when executed, will build G.
4. Finally, based on the tracing information, replace one of the `mark()`-calls with a call to the M-method. The remaining `mark()`-calls are removed.

The result is a new jar-file which when executed with the special input sequence will execute the method M, and, consequently, build the watermark graph G on the heap.

In Section 5 we will extend the embedding method such that the watermark graph is split into several pieces and inserted at various `mark()`-locations.

3.4 Constructing the Graph

An ideal class of watermark graphs should

1. have a root node from which all other nodes are reachable to prevent pieces of the graph from being garbage collected,
2. have a high bit-rate so that a large watermark will result in a small graph,
3. have low out-degree to resemble common data-structures such as lists and trees,
4. have some error-correcting properties such that minor changes to the graphs by an attacker, or minor failures during extraction, will not prevent the graph from being extracted, and
5. have some internal structure that makes the graph easy to tamper-proof.

We do not expect to find a single class of graphs which simultaneously optimizes all these criteria. Instead, we are developing a library of algorithms for building watermark graphs with different sets of properties. Depending on a user's particular requirements (high bitrate, high resilience to attack, high stealth, etc.) this will allow an appropriate graph (or combination of graphs) to be found. Currently, SandMark contains an implementation of four of the five classes of watermark graphs illustrated in Figure 9.

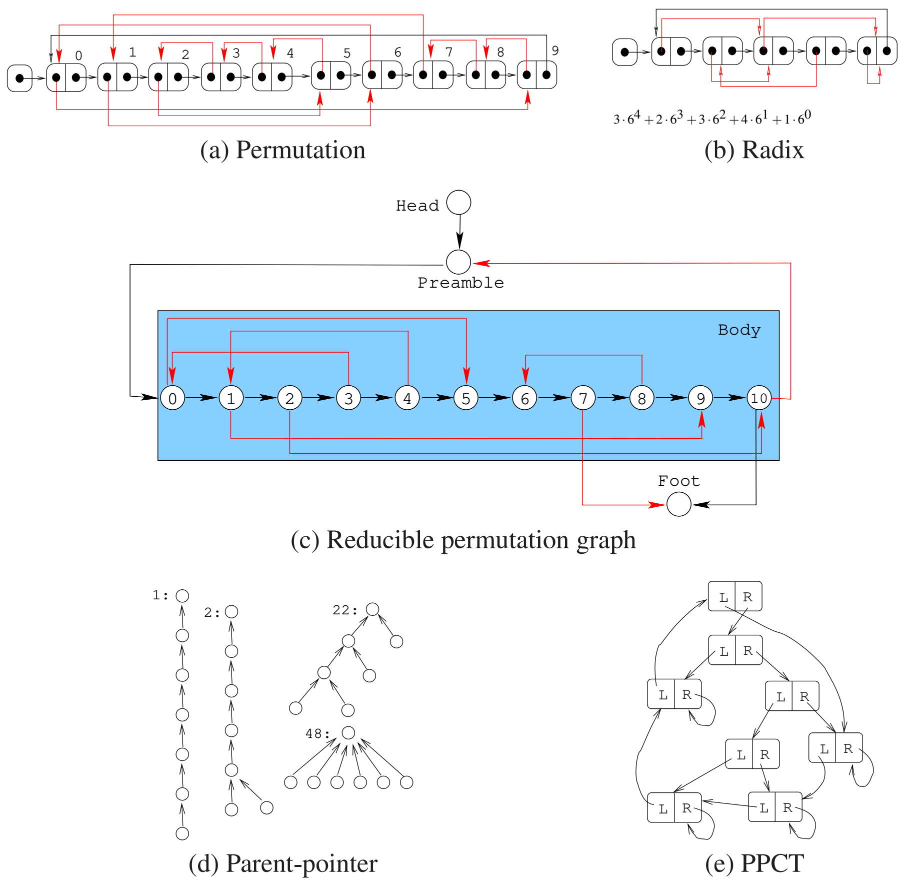


Figure 9: Graph encodings. This figure is taken from [10].

3.4.1 Permutation Encoding

A watermark integer $\mathcal{W}$, in the range $[0..n-1]$, may be represented by a permutation of the numbers $\langle 0,\ldots,n-1\rangle$. We use a natural mapping of permutations onto integers, such that (for example) the watermark

$$1024$$

is represented by the permutation

$$\pi = \langle 9,6,5,2,3,4,0,1,7,8\rangle$$

We use a singly-linked, circular list data structure to represent a permutation. We call this structure a Permutation Graph. See Figure 9(a). Each element i of the list has two pointers. Its data pointer refers to the element $\pi(i)$ to which i is mapped by the permutation π. It also has a list pointer referring to element $(i+1) \bmod n$.

The dynamic bitrate of a Permutation Graph is the number $(\lg n!)$ of watermark bits represented by an n-element list, divided by the number of bytes $(an+b)$ required to represent this list in computer memory. To embed a non-negative watermark integer w, we use $n = \min\{k : k! > w\}$ list elements. Using the first term of Stirling's approximation, we have $n = m/\lg m + O(m/\lg\lg m)$, where $m = \lceil \lg(w+1)\rceil$ is the number of bits in w. We conclude that the dynamic bitrate $r(m) = (\lg m)/a + O(\lg m/\lg\lg m)$ is a slowly increasing function of m.

The coefficient a in our dynamic bitrate expression is a small integer whose exact value will depend on implementation details. At minimum, a must be 16 bytes on a 32-bit computer architecture, because each list element contains two 4-byte pointers. If list elements are dynamically allocated, then the dynamic storage allocator will require at least one additional pointer field for each allocated object, and a would be "rounded up" to 16 bytes in such implementations. The list-overhead coefficient b is another small integer whose value is irrelevant in the asymptotic limit, even though it will be important when m is small.

The static bitrate is the number of watermark bits, divided by the number of bytes of code required to build the watermark. In a straightforward implementation, there would be an overhead of b' code bytes to create the list header, plus a' code bytes per list element. Thus the static bitrate will have the same asymptotic form (with different constants) as the dynamic bitrate. This is true of all our watermark-encoding methods, so we will not treat dynamic bitrates again in this section. However in Section 7 we analyse our experimental data to estimate the constants governing static and dynamic bitrates for each of our encoding methods.

Permutation Graph watermarks have a modest resilience to attacks on its pointer fields. Any change to one of its list pointers will disrupt its circular-list property. Any change to one of its data pointers will disrupt its permutation property. We summarise these observations by saying that Permutation Graphs are single-error detecting.

3.4.2 Radix Encoding

Figure 9(b) illustrates a Radix Graph in a circular linked list of length n. The data pointer field encodes a base-n 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. Note that this is the same data structure as a Permutation Graph, however Radix Graph watermarks have higher bitrate and less error-detection capacity, because their data pointers are less constrained.

A Radix Graph of length n can represent any integer in the range $0\ldots(n+1)^n-1$. The list requires $an+b$ words, so its dynamic bit-rate, as a function of n, is $(n-1)\lg(n+1)/(an+b) \approx (\lg n)/a$. For $n=511$, if $a=b=8$ bytes, we can hide $m=8\times 511=2040$ bits in 2048 bytes of storage. This is a dynamic bitrate of almost exactly one watermark bit per byte of data structure. Smaller dynamic bitrates would be observed if a were larger than 8 bytes, or if n were smaller than 511 (due to the increasing importance of the overhead term b and the convexity of the $\lg n$ term). Somewhat larger bitrates would be observed for larger n.

The constants in the static data rate is more difficult to estimate, since these will depend heavily on the encoding. As an example, we will consider Java bytecode. Allocating a node and initializing two pointer fields may be accomplished with a 24-byte sequence of bytecodes. To hide a 2040-bit watermark, we must build a 255-element list. This requires $24\times 255=6120$ bytes of straight-line bytecode, plus a few dozens of additional bytes to construct the list header, for a static bit-rate of 0.33 hidden bits per codebyte.

3.4.3 Parent-Pointer Trees

All our watermark embedding methods can be described as enumerations of graphs [26]. 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 have effective algorithms to

1. given n, generate the w:th graph in the enumeration, and
2. given G, extract its index w 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 classes of graphs, in addition to Permutation Graphs and Radix 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 [32, Section 2.3.4.4]. Figure 9(d) illustrates these trees.

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

The number a_n of parent-pointer trees with n nodes is asymptotically (for $c\approx 0.44$ and $1/\alpha\approx 2.956$) $a_n = c(1/\alpha)^{n-1}/n^{3/2}+O((1/\alpha)^n/n^{5/2})$. Thus we can encode an arbitrary 1024-bit integer w in a graphic watermark with

$1024/\lg 2.956 \approx 655$ data pointers. This might require as little as 2620 bytes on a 32-bit architecture, if these data pointers were added to objects that were allocated by the original unwatermarked program. (These data pointers would, however, slow the de-allocation processes because it would prevent any dead objects from being de-allocated until all its children objects in the watermark tree were also dead.) The dynamic bit-rate of a Parent Pointer Tree watermark is thus expected to be $1024/2620 \approx 0.4$ hidden bits per data byte. This is about half the bitrate of Radix Graphs. It is approximately the same bitrate as Permutation Graphs, but not all single errors in data pointers can be detected in a Parent-Pointer Tree. Because Parent-Pointer Trees are inferior to Permutation Graphs in bitrate, and are no better in resiliency, Parent Pointer Trees have not been a high priority for implementation in SandMark.

In Section 4 we discuss the bitrates and error-correcting properties of the more complex graphs illustrated in Figures 9(e) and 9(c).

3.5 Generating Intermediate Code

We could, of course, generate Java code directly from the graph components. However, it turns out to be advantageous to insert one intermediate step. From the watermark graph we generate a list of *intermediate code instructions*, much in the same way a compiler might generate an intermediate representation of a program, in anticipation of code generation and optimization. In a compiler the intermediate code separates the front-end from the back-end, improving retargetability, and also provides a target-independent representation for optimizing transformations. Similarly, our intermediate representation provides

1. retargetability, to allow future generation of code for other languages; and
2. transformability, i.e. the ability to optimize or otherwise transform the intermediate code prior to generating Java code.

In fact, in our implementation we start by generating straight-forward intermediate code and then run several transformations over the code to optimize it.

Given the graph in Figure 10(a) we would generate the intermediate code in Figure 10(b). Nodes are named n_1, n_2, etc. The n=`CreateNode()` instruction creates a new node n. The `AddEdge(`$n \xrightarrow{\text{edge}} m$`)` instruction adds an edge from node n to node m. Since the graphs are multi-graphs the out-edges are named `edge1`, `edge2`, etc. The instruction `SaveNode(`n`,`L`)` is used to store the root node n in a global storage location L, such as a hash table, vector, etc. This ensures the liveness of every node so that the graph will not be reclaimed by the garbage collector. As we will in Section 6.1, we can often do away with these global pointers by passing root nodes as method arguments. This is much stealthier since most programs have few global variables but many method parameters.

To generate intermediate code from a graph G we perform a depth-first search from the root of the graph. `CreateNode()`-instructions are generated from each node, in a reverse topological order. We issue an `AddEdge(`$n \xrightarrow{\text{edge}} m$`)`-instruction as soon as the instructions m=`CreateNode()` and n=`CreateNode()` have both been generated.

The complete set of intermediate code instructions is given in Table 1. These will be discussed in more detail in conjunction with the splitting of graphs into multiple pieces.

3.6 Generating and Inserting Java Code

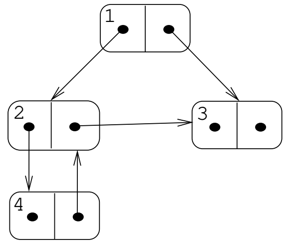
Generating Java code from the intermediate representation is relatively straight-forward. We use the `BCEL` library to generate a bytecode class `Watermark`. From the intermediate code in Figure 10(b) we would generate the Java code in Figure 10(c).

The chosen `mark()`-location is replaced by a call to `Watermark.Create()`. A final obfuscation pass over the watermarked application will inline the call, rename the watermark class, etc., to prevent attacks by pattern-matching.

To insert the call to `Watermark.Create()` two cases must be considered, depending on whether the `mark()`-call is `LOCATION`-based or `VALUE`-based. A `LOCATION`-based `mark()`-call is simply replaced by a call

```
Watermark.Create();
```

A `VALUE`-based `mark(expr)`-call is replaced by the call



(a)

create()
n_4=CreateNode()
n_2=CreateNode()
AddEdge($n_2 \xrightarrow{\text{edge1}} n_4$)
AddEdge($n_4 \xrightarrow{\text{edge2}} n_2$)
n_3=CreateNode()
AddEdge($n_2 \xrightarrow{\text{edge2}} n_3$)
n_1=CreateNode()
AddEdge($n_1 \xrightarrow{\text{edge1}} n_2$)
AddEdge($n_1 \xrightarrow{\text{edge2}} n_3$)
SaveNode(n_1, Array)

(b)

```
public class Watermark {
    public Watermark edge1, edge2;
    public static Watermark[] array;

    public static void Create() {
        Watermark n4=new Watermark();
        Watermark n2=new Watermark();
        n2.edge1=n4;
        n4.edge2=n2;
        Watermark n3=new Watermark();
        n2.edge2=n3;
        Watermark n1=new Watermark();
        n1.edge1=n2;
        n1.edge2=n3;
        array[1]=n1;
    }
}
```

(c)

Figure 10: (a) shows a watermark graph, (b) the corresponding intermediate code, and (c) the resulting Java code.

```
if (expr==value)
    Watermark.Create();
```

Code is also inserted to create the hashtables, arrays, vectors, etc. that are used to store watermark graph root nodes.

3.7 Extraction

To extract the watermark the watermarked application is run as a subprocess under debugging, again using Java's JDI debugging framework. The user enters their secret input sequence $I_0, I_i, \ldots$ exactly as they did during the tracing phase. This causes the method `Watermark.Create()` to be executed and the watermark graph to be constructed on the heap. When the last input has been entered it is the extractor's task to locate the graph on the heap, decode it, and present the watermark value to the user.

There may, of course, be an enormous number of objects on the heap and it would be impossible to examine them all. To cut down the search space we rely on the observation that the root node of the watermark graph will be one of the very last objects to be added to the heap. Hence, a good strategy is to examine the heap objects in reverse allocation order. Unfortunately, `JDI` does not yet provide support for examining the heap in this way.

An elegant and efficient approach would be to modify the constructor for Java's root class `java.lang.Object` to include a counter:

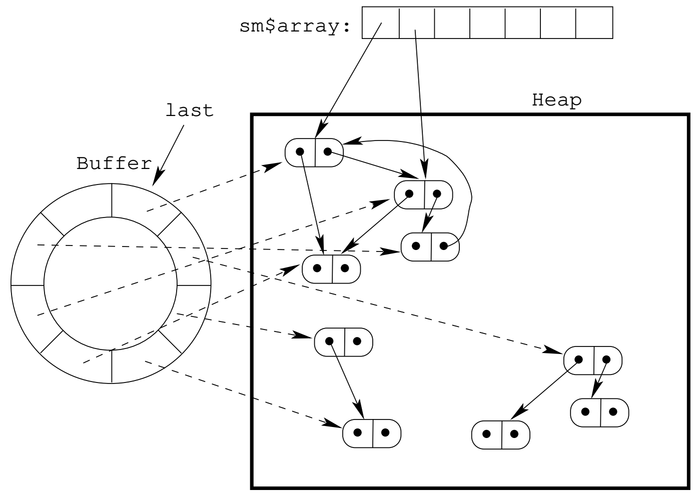


Figure 11: A view of memory during extraction. A circular linked buffer holds the last allocated objects. The extractor examines the objects in reverse allocation order and extracts the subgraph reachable from each object. This is decoded into the watermark.

```
package java.lang;
public class Object {
    public static long objCount = 0;
    public long allocTime;
    public Object() {
       allocTime = objCount++;
    }
}
```

Since every constructor must call `java.lang.Object.<init>` (the class constructor) this means that we have assigned an allocation order to the objects on the heap at the cost of only an extra add and assign per allocation.

We've shied away from this approach, however, since it would require modifying the Java runtime library. Also, it is conceivable that some Java compilers may optimize away calls to `java.lang.Object.<init>` under the assumption that this constructor does nothing.

Instead, we rely on a more heavyweight but portable solution. Using `JDI` we add a breakpoint to every constructor in the program. Whenever an allocation occurs we add a pointer to the new object to a circular linked buffer. This way, we always have the last 1000 (say) allocated objects available. This is illustrated in Figure 11. The downside is a fairly substantial slowdown due to the overhead incurred by handling the breakpoints.

To extract the watermark graph we consider each object on the circular buffer in reverse allocation order, extracting the reachable subgraph. Since every watermark graph has a root from which every node is reachable we are guaranteed to eventually find the graph. From the graph the watermark number is extracted, by the algorithm in Figure 12.

4 Improving Resilience

To properly design and evaluate a watermarking algorithm it is essential to precisely define an attack model. This has been a failing of most previous work on software watermarking. In this paper we will assume that the watermarked program is too large for manual inspection by the adversary. In other words, it is infeasible for the adversary to read the (decompiled) source to locate and destroy the watermark code. Rather, we want to protect the watermarked program against *class attacks* — the construction of automated methods of destroying the watermark. For example,

```
for each graph codec C do
   for each node n on the buffer, in reverse allocation order do
      G = heap graph with root n, edges [e_1,e_2,\ldots,e_k];
      for each pair (e_i,e_j) of edges in [e_1,e_2,...,e_k] do
         G' = subgraph of G with edges labeled (e_i,e_j);
         if watermark decoding w=C(G') succeeds then
            yield w
```

Figure 12: Watermark extraction algorithm. We assume that all graph codecs generate graphs with outdegree two, which is the case in the SandMark system. Since an adversary might add extra outgoing edges to the graph nodes we try all possible sugraphs of outdegree two. k is the outdegree of G. In cases where the user knows which graph codec was used during embedding, the outermost loop is removed.

---

if a particular code transformation can be shown to destroy a watermark then it is an easy task for an adversary to construct a program that will destroy *every* watermark in *every* program.

One nice consequence of our approach is that many translating, optimizing, and obfuscating transformations 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 would like to be resilient against. To confuse the extractor an adversary can

1. add extra pointers to the nodes of linked structures (Figure 13(a)) to make it hard for the extractor to identify the real graph edges within many extra bogus pointer fields;
2. rename and reorder instance variables (Figure 13(b));
3. add levels of indirection, for example by splitting nodes into several linked parts (Figure 13(c));
4. add extra bogus nodes pointing into our graph, preventing us from finding the root.

Figure 14 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. For example, splitting a node costs 12 bytes per allocated node (one 4-byte pointer cell plus approximately 8 bytes of overhead for current Java implementations). 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.

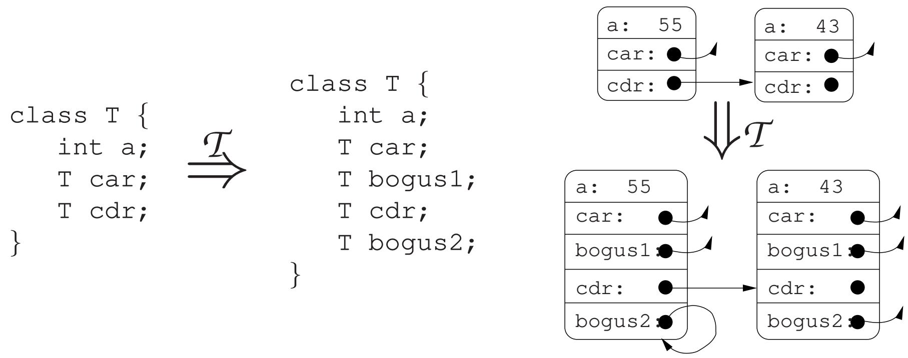
4.1 De-watermarking by Field Reordering

A simple way to protect against field reordering is to consider every order of the fields during watermark extraction. This could, however, lead to an increased false positive rate. A better approach is to choose an unlabeled class of graphs for which the order of outgoing edges do not affect the watermark value. Reducible Permutation Graphs and Planted Plane Cubic Trees have this property. Unfortunately, as we will see these graphs have a lower bit-rate than Permutation Graphs and Radix Graphs.

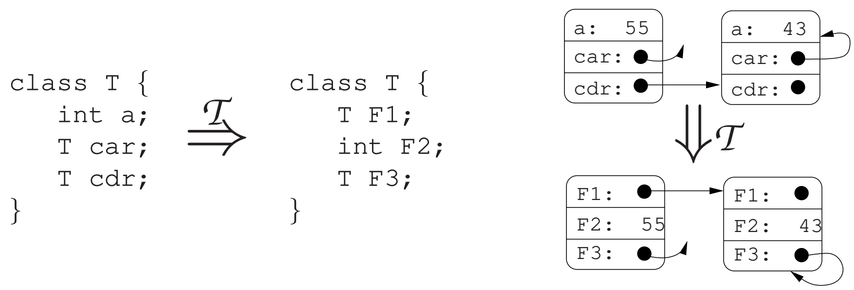
4.1.1 Reducible Permutation Graph

Figure 9(c) shows a *reducible permutation graph* (RPG) [10]. An RPG is a reducible flow graph with a Hamiltonian path consisting of four pieces :

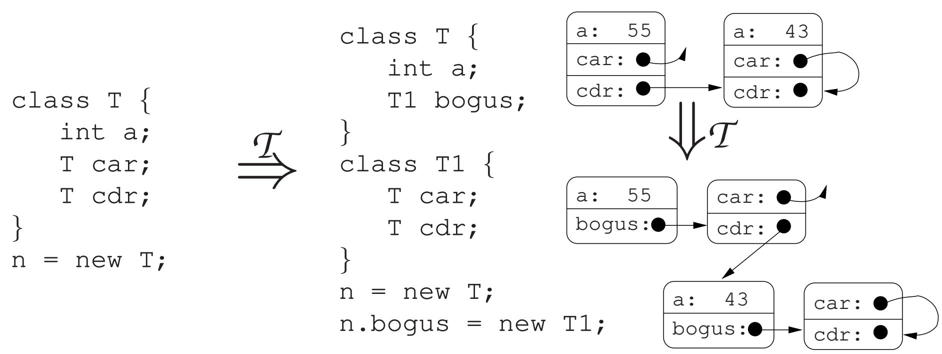
A header node The root node of the graph having out-degree one from which every other node in the graph is reachable. Every control-flow graph has such a node.



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



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



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

Figure 13: Obfuscation attacks against graphic watermarks.

The preamble The rest of the graph is only reachable from the header node through all preamble nodes. Thus, any node from the body can have an edge to any node in the preamble, and the graph is still reducible.

The body Edges among the body, from the body to the preamble, and from the body to the footer node encode a permutation that is its own inverse.

A footer node A node with out-degree zero that is reachable from every other node of the graph. Every control-flow graph has such a node, representing the method exit.

Every node of an RPG has one outgoing "list edge" and one outgoing "permutation edge". Thus each node can be represented by a data structure with two pointers per element.

There is a one-to-one correspondence between self-inverting permutations and isomorphism classes of RPGs. We have developed a low-degree polynomial-time algorithm for encoding any integer *w* as the RPG corresponding to the

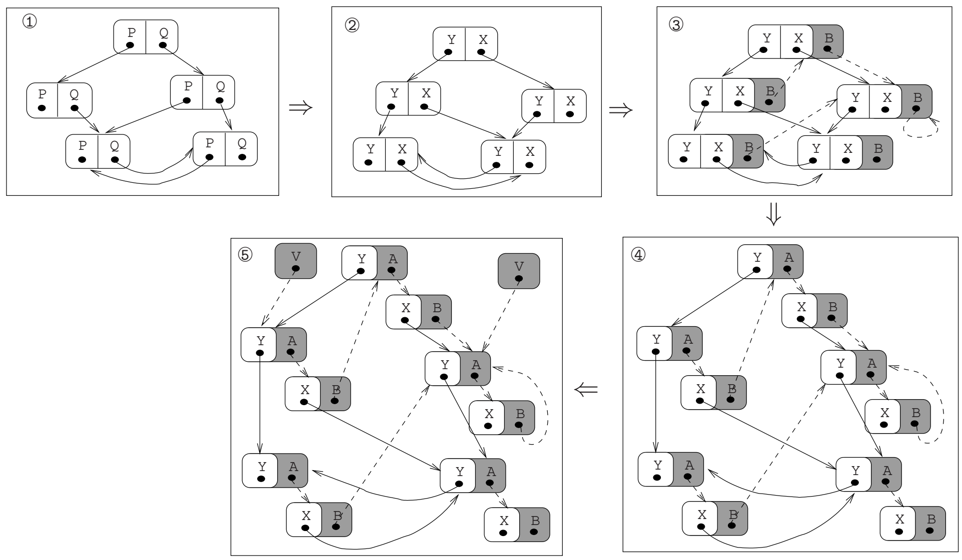


Figure 14: Example obfuscation attack against the watermark graph in ①. The adversary renames and reorders node pointer fields (②), adds a bogus pointer field `B` (③), and splits nodes by adding a bogus pointer field `A` (④). Finally, in ⑤ bogus pointers into the graph obscure the root node.

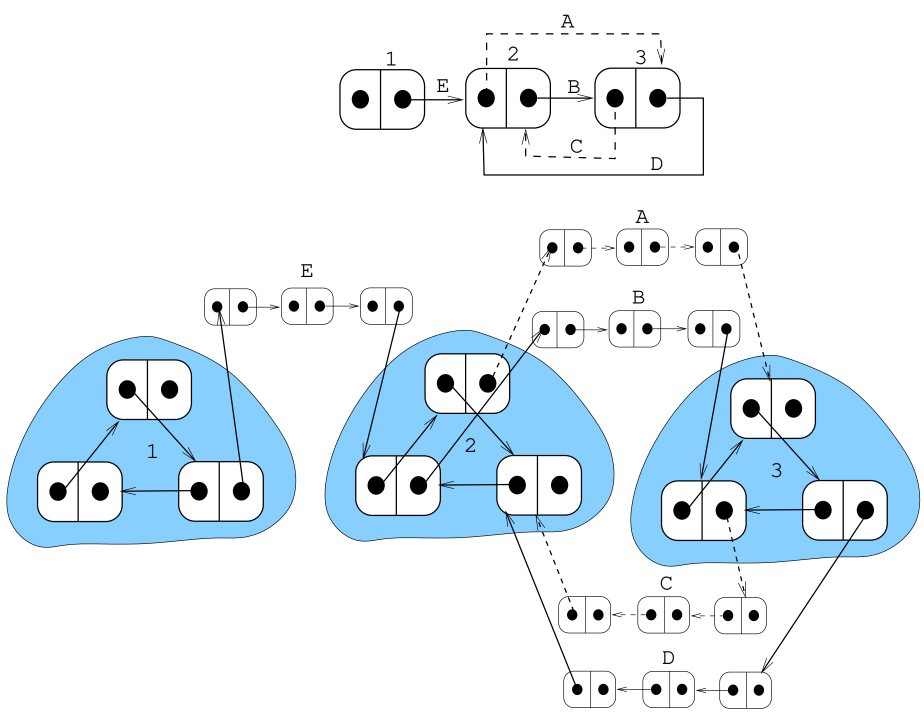


Figure 15: The top-most graph shows a radix encoding of the value 3. The bottom-most graph shows how this encoding is made more resilient against node-split attacks by turning each node into a 3-cycle and each edge into a path of length three. For identification purposes nodes have been labeled 1–3 and edges A-E in this figure. Edges on the spine are shown solid and edges encoding a radix digit are shown dashed.

wth self-inverting permutation in this correspondence. We also have an efficient algorithm for decoding w from an RPG.

An RPG encoding a permutation on n elements has a bitrate of approximately $\frac{3}{8}\lg n - 0.62$ bits per node, plus or minus approximately $\frac{1}{8}\lg n$ bits per node depending on the size of the preamble required for the RPG corresponding to the watermark w. This is, very roughly, half the bitrate of a Permutation Graph.

Each node in an RPG has exactly one incoming "list edge" and exactly one incoming "permutation edge". Thus any single change to an edge pointer will cause some node to have an indegree less than two, and the erroneous edge can be easily identified and corrected. Thus RPGs can correct single errors.

The preamble of an RPG was designed to make its Hamiltonian unique, which gives RPGs a strong error-correction ability against adversaries who reordering the pointer fields in node elements. We call such a reordering of pointer fields an "edge-flip" because it flips the coloring of a node's outgoing edges.

Permutation Graphs do not have the RPG's capacity to correct an arbitrary number of edge-flip errors. Indeed, the watermark in a Permutation Graph may not survive as few as two (carefully or luckily) chosen edge-flips, for a graph g' obtained by two edge-flips from a Permutation Graph $g(w)$ may be a legal Permutation Graph encoding a watermark $w' \neq w$.

RPGs are thus preferable to Permutation Graphs, in situations where resilience to attacks is more important than bitrate.

4.1.2 Planted Plane Cubic Trees

Figure 9(e) shows an example of the class G_n of *planted plane cubic trees* (PPCT) on n leaf nodes $v_1, v_2, \ldots, v_n$. PPCT graphs are enumerated in [24]. Such trees have $n-1$ internal nodes and one root node v_0, so there are $2n$ nodes in each $w \in G_p$. We would represent w by using $2n$ objects, where each object holds two pointers l and r. this data structure requires $4n$ 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, leaf node indices are discoverable in linear time by following an $(n+1)$-cycle on l-links: $l(v_i) = v_{(i+1) \bmod (n+1)}$.

PPCTs have a dynamic bitrate of approximately 1 bit per node. Note that this is asymptotically much worse than the bitrate of our other encoding methods, for it lacks the logarithmic term. The nodes in a PPCT are the same size as the other encoding methods in SandMark, because they hold two user-addressible pointer fields. If each node takes 16 bytes, then the bit-rate of a PPCT is approximately 1 bit per 16 bytes.

PPCTs have the same edge-flip correction property as RPGs, by the following argument. There is only one $(n+1)$-cycle in any *PPCT* on n leaf nodes, and this cycle is the outercycle linking all of its leaves with its root. This outercycle can be discovered quite efficiently: in linear time with a depth-first search. The colors (L,R) on the edges in the interior of the tree can be assigned unambiguously, also in linear time, by maintaining planarity at successively higher levels in the tree. Thus the correct colors (L,R) can always be recovered, even if these are modified by an adversary.

PPCTs have a local consistency property that could be checked, reasonably stealthily and efficiently, by tamper-detection code. The idea is to test the PPCT's planarity locally, for any 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. Any disruption of the PPCT's planarity (as may occur when an adversary modifies the PPCT in an attempt to obliterate its watermark) may be detected by some subsequent planarity test. When such a planarity violation is discovered, this could trigger a 'logic bug' or an outright crash in the watermarked program.

4.2 De-Watermarking by Node Splitting

Node-Splitting is an effective attack against our watermark graphs but one which will have serious detrimental effect on the performance of the de-watermarked program. Even so, Figure 15 shows another representation which, at the expense of a lower data rate, will increase a graph watermark's resilience to node-splitting attacks. The idea is to turn every node into a 3-cycle and every edge into a path of length 3 during embedding. We call such graphs *cycled graphs*. During watermark extraction the cycles and paths are contracted back to the original graph. Any node or an edge split by an adversary will be ignored by this process.

In our SandMark implementation any of our graph encodings can be turned into a cycled graph. The algorithm for node and edge contractions is shown in Figure 16.

```
while there exist cycles without supernodes
    replace the smallest cycle not containing a supernode
    with a new supernode;

P := empty set of edges;
for each supernode s do
    for each outgoing edge (s,t0) do
       for each supernode c do
          if ∃ path ⟨t0,t1,...,tn,c⟩ such that t1,...,tn are not supernodes then
            add (s,c) to P

remove all non−supernodes from the graph;
for each edge (s,c) ∈ P do
    add an edge (s,c) to the graph;
```

Figure 16: Algorithm to contract edges and cycles for cycled graphs.

4.3 De-Watermarking by Bogus Field Addition

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, including the addition of bogus fields in the graph nodes. 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();
```

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

Reflection can also be used to protect against reordering and renaming attacks The idea is to 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;
    }
```

Because reflection is unusual in most programs these techniques are of limited usefulness. For this reason, they have not yet been implemented in the current version of SandMark.

INSTRUCTION	DESCRIPTION
`AddEdge(`$n \xrightarrow{\texttt{edge}} m$`)`	Add an edge from node n to node m. Since the graphs are multi-graphs the out-edges are named.
n `= CreateNode()`	Create node n.
`CreateStorage(`S`)`	Create the global storage structure S.
`FollowLink(`$n \xrightarrow{\texttt{edge}} m$`)`	Return m by following the edge `edge` from n.
`LoadNode(`n, L`)`	Load node n from global storage location L.
`SaveNode(`n, L`)`	Save node n in global storage location L.

Table 1: Intermediate code instructions.

4.4 Future work: Manipulating Graphs

One of our main motivations for using dynamically built graph structures to represent the watermark is that the code that builds the watermark will be difficult to analyze. The reason is the inherent hardness of alias analysis. However, current alias analysis algorithms do very well with non-circular data structures (such as linear lists and trees) and purely *constructive* code. That is, an algorithm may be successful in analyzing code that builds a linear list, but may fail if that list is taken apart and reassembled [23].

We could increase our resilience to pattern-matching attacks by exploiting these weaknesses in current alias analysis algorithms. For example, along the special execution path we can not only merge graph components, but also split components into sub-parts, which are later reassembled.

5 Increasing Watermark Size

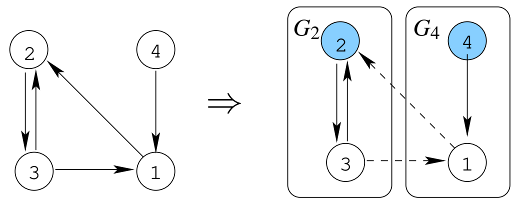
While the operations by which the watermark graph is built (`new`s and pointer assignments) are stealthy in themselves, a large number of such operations concentrated to one place in the code would arouse suspicion. For large watermark values we therefore split the graph G into several components $G_0, G_1, \ldots$ whose code is spread along the special execution path. There are several issues to consider:

1. The subgraphs should be of roughly equal size.[1]
2. The splitting of G should be done in such a way that each subgraph has a root, a special node from which all other nodes in the graph can be reached. This allows us to store only pointers to root nodes to prevent the garbage collector from collecting the subgraphs.
3. We should attempt to split G in such a way that the number of edges between subgraphs is minimized. The reason for this restriction is that the more edges there are between subgraphs, the more Java code we will have to generate in order to connect the subgraphs into the complete graph G.

We use an algorithm [33] by Kundu and Misra to partition the watermark graphs. Given the graph on the left in Figure 17(a) it would produce the two graph components G_2 and G_4 on the right. Figure 17(b) shows the generated intermediate code. The main intermediate code instructions are given in Table 1. As before, `SaveNode` instructions are used to store the root node of each graph component in a global structure such as a hash table, vector, etc. `LoadNode` instructions are used to reload the nodes. We do this to protect against garbage collection, but also so that the graph components can be connected. Note how root node n_2 from graph G_2 has been loaded from global storage in order to connect n_1 to n_2. Note also how the `FollowLink` instruction is used to traverse G_2 from the root node n_2 to get to node n_3, which can then be connected to n_1 in G_4.

The intermediate code for a subgraph G_i contains instructions connecting G_i to all the previous subgraphs $G_0, \ldots, G_{i-1}$. If there is an inter-subgraph edge $m \xrightarrow{\texttt{edge}} n$ from G_k to G_i (i.e. m is a node in G_k and n is in G_i) we generate

[1] It is actually not completely clear that this is a reasonable requirement. For stealth reasons it might be better if the components are of random size. The current implementation, however, splits in equal-size pieces.



(a)

```
create(G2)
    n3=CreateNode()
    n2=CreateNode()
    SaveNode(n2, Array)
    AddEdge(n2 --edge1--> n3)
    AddEdge(n2 --edge2--> n3)

create(G4)
    n1=CreateNode()
    n4=CreateNode()
    SaveNode(n4, Hash)

    AddEdge(n4 --edge1--> n1)
    n2=LoadNode(Array)
    AddEdge(n1 --edge1--> n2)
    n3=FollowLink(n2 --edge1--> n3)

    AddEdge(n3 --edge1--> n1)
```

(b)

```java
public class Watermark {
  public Watermark edge1, edge2;
  public static Hashtable hash;
  public static Watermark[] array;

  public static void Create_G2(){
     Watermark n3=new Watermark();
     Watermark n2=new Watermark()
     Watermark.array[1]=n2;
     n2.edge1=n3;
     n2.edge2=n3;
  }

  public static void Create_G4(){
     Watermark n1=new Watermark();
     Watermark n4=new Watermark();
     Watermark.hash.put(
        new Integer(4), n4);
     n4.edge1=n1;
     Watermark n2=array[1];
     n1.edge1=n2;
     Watermark n3=(n2!=null)?
        n2.edge1:new Watermark();
     n3.edge1=n1;
  }
}
```

(c)

Figure 17: (a) shows a watermark graph that has been split into two components. Root nodes have been shaded and inter-component edges have been dashed. The graphs are identified by their root nodes. (b) shows the intermediate code for each component and (c) the corresponding Java code.

---

1. one or more `FollowLink()`-instructions to reach node m by traversing G_k starting at its root node, and
2. a final `AddEdge()` instruction to link m to n.

This is done by finding the shortest path from k (the root of subgraph G_k) to m and issuing a `FollowLink()`-instruction for each edge on the path.

5.1 Generating Java Code

Table 2 lists possible translations of the intermediate instructions and Figure 17(c) shows the Java class generated from the intermediate code in Figure 17(b). Method `Create_G2` builds subgraph G_2 and `Create_G4` subgraph G_4. Note,

<table>
<tr><th>INSTRUCTION</th><th>JAVA</th></tr>
<tr><td><code>AddEdge</code>($n \xrightarrow{\texttt{edge}} m$)</td><td>Generate <code>n.edge = m</code></td></tr>
<tr><td><code>CreateNode</code>(n)</td><td>Generate <code>Watermark n = new Watermark()</code></td></tr>
<tr><td><code>CreateStorage</code>(G, S)</td><td>Generate one of
1. <code>static java.util.Hashtable hash = new java.util.Hashtable();</code>
2. <code>static Watermark arr = new Watermark[</code>m<code>];</code>
3. <code>static java.util.Vector vec = new java.util.Vector(</code>m<code>); vec.setSize(</code>m<code>);</code>
4. <code>static Watermark n1,n2,...;</code>
where m is the number of nodes in the graph and <code>n1,n2,...</code> are the root nodes of the subgraphs.</td></tr>
<tr><td>FollowLink($n \xrightarrow{\texttt{edge}} m$)</td><td>Generate
<code>Watermark m = n.edge</code>
or if <code>n</code> can be <code>null</code> at this point, generate one of
1. <code>Watermark m = (n!=null)?n.edge:new Watermark();</code>
2. <code>try {</code>
<code>   Watermark m = n.edge;</code>
<code>   // Any code referencing m</code>
<code>} catch (Exception e){};</code>
3. <code>if (n != null) {</code>
<code>   Watermark m=n.edge;</code>
<code>   // Any code referencing m</code>
<code>}</code></td></tr>
<tr><td><code>LoadNode</code>(n, S)</td><td>Generate one of
1. <code>Watermark</code> n <code>= (Watermark) Watermark.hash.get(new java.lang.Integer(</code>k<code>));</code>
2. <code>Watermark</code> n <code>= Watermark.arr[</code>$k-1$<code>];</code>
3. <code>Watermark</code> n <code>= (Watermark) Watermark.vec.get(</code>$k-1$<code>);</code>
4. <code>Watermark</code> n <code>= Watermark.n</code>k
depending on how n is stored. k is n's node number.</td></tr>
<tr><td><code>SaveNode</code>(n, L)</td><td>Generate one of
1. <code>hash.put(new java.lang.Integer(</code>k<code>),</code> n<code>);</code>
2. <code>Watermark.arr[</code>$k-1$<code>] =</code> n;
3. <code>Watermark.vec.set(</code>$k-1$<code>,</code> n<code>);</code>
4. <code>Watermark.n</code>k <code>=</code> n
depending on how n is stored. k is n's node number.</td></tr>
</table>

Table 2: Translation from intermediate code instructions to Java.

in particular, the statements which link nodes n_3 and n_1. To get access to G_2's node n_3 we follow the edge from G_2's root node n_2 to n_3. This will work provided G_2 has been created at this point. However, if we are not doing a extraction run (i.e. the input sequence in *not* exactly $I_0, I_1, \ldots$) then G_2 may not have been created, and n_2 may be `null`. Table 2 shows several ways to protect against `null`-pointer exceptions. It is useful to have a whole library of such protection mechanisms to prevent attacks by pattern matching.

5.2 Future work: Splitting the Watermark Number

There is an alternative approach to increasing the size of the watermark that can be inserted in a given program. Rather than splitting the graph, we split the watermark number n into several (smaller) numbers $n_0, n_1, \ldots, n_k$ using the Chinese Remaindering theorem [41]. The numbers are encoded into graphs $G_0, G_1, \ldots, G_k$ which are embedded along the special execution path. The problem is that during extraction all the graphs need to be found, which precludes us from examining only the last few allocated objects as was done in Section 3.7. Instead, we may want to produce k traces using k special input sequences, and embed each subgraph along one of the resulting special execution paths. This approach, however, will be more onerous for the user, both during embedding and extraction.

6 Improving Stealth

While our main goal is to protect the watermark from automatic means of destruction, we would also like to protect it against manual attack whenever possible. If an attacker can construct an automatic analysis tool that will pinpoint the location of the watermark with some degree of accuracy (for example, with 90% assurance, the watermark is located within these 10% of the code) it may become feasible for the attacker to manually search the decompiled program for the watermark code. In this section we will consider three techniques for improving the stealth of our graph watermarks.

6.1 Avoiding Global Variables

We have so far assumed that the roots of subgraphs are stored in static fields. In Figure 17(b), for example, the roots of subgraph are stored in global variables `array` and `hash`. This is likely to be un-stealthy since programs written in a modern object-oriented style typically contain only a few globals. Instead, we would like to pass roots in the formal parameters of methods. This means we are going to have to find paths through the call-graphs from one `mark()`-call to the next.

To facilitate finding the right method calls along the special execution path to modify, we use the information collected during tracing to build a precise call-graph. In the general case, this graph is a forest of directed acyclic graphs. The root of each DAG represents either the `main` method (which was invoked by the user), or a method that was invoked by the Java runtime system in response to an asynchronous event such as the user interacting with the graphical user interface. There are four kinds of call-graph nodes: ENTER and EXIT nodes represent the entry into and return from a method, and CALL and RETURN represent the invocation of a method. It should be noted that, in contrast to the conservative call-graphs built by static analysis tools, our graphs are exact.

Figure 18 shows the trace forest generated from the trace points in Figure 8. Solid lines represent the path actually taken during tracing. Dashed lines represent paths along which information can be passed as method arguments.

We first need to identify locations in the program where the structures that store subgraph nodes can be created. We call these *Storage Structures*. For example, if we want to store subgraph roots in a vector, we need to insert the code

```
static java.util.Vector vec = new java.util.Vector(m);
vec.setSize(m);
```

at a location where `vec` can be passed on to all the locations in the program where it is needed. More precisely, the code can be placed at any call-forest node that dominates all the `mark()`-nodes being used. This allows us to pass the storage structures in a method parameter from the point of creation to all the chosen `mark()`-calls. In a language that (unlike Java bytecode) supports pass-by-reference parameters a simpler strategy could be used: the storage structure could simply be created at the same point as the first created graph component. Pass-by-reference parameters can,

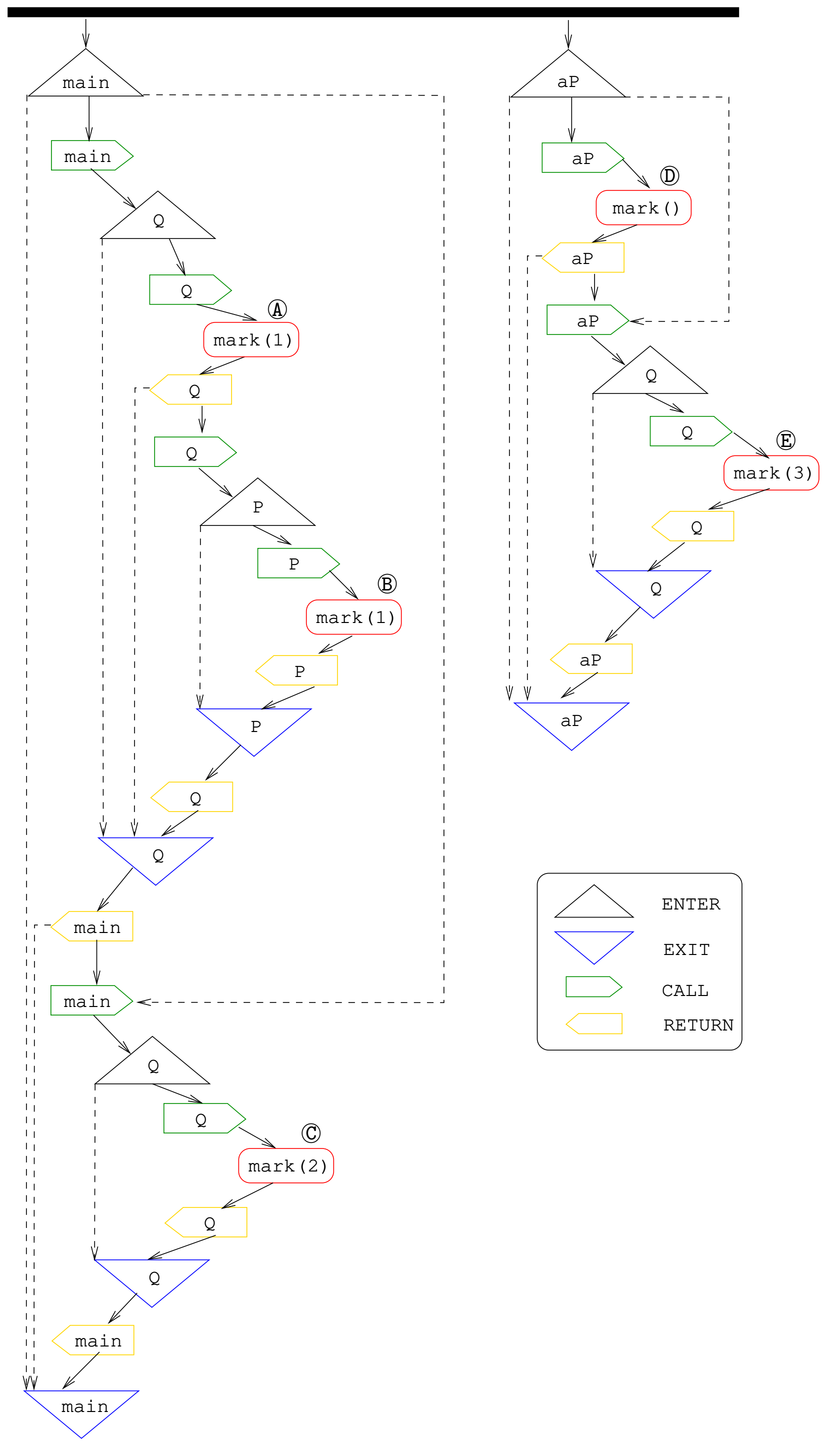


Figure 18: The trace forest generated from the trace points in Figure 8.

bytecode	frequency
`aload*`	29%
`putfield`	10%
`astore*`	8%
`new`	7%
`invokespecial`	7%
`dup`	7%
`getstatic`	6%
`invokevirtual`	6%
`iconst*,ldc`	5%
`ifnull`	3%
`pop`	2%
`return`	2%
`getfield`	2%
`checkcast`	1%
`goto`	1%

Table 3: The distribution of Java bytecode instructions in generated graph-building code.

of course, be simulated in Java bytecode by adding an extra level of indirection, but we believe this would be a less stealthy solution.

The next question that arises is which of the available `mark()`-calls should be selected for graph construction. To allow us to select the most stealthy locations we add weights to the nodes and edges in the call-forest. The watermarking code that the CT algorithm inserts into a program has approximately the distribution shown in Table 3. The weight of a method that contains a `mark()`-call is calculated to be proportional to how similar its code is to this distribution.

We also have to take into account to which methods it would be *allowable* to add an extra storage structure argument, and for which methods it would be *stealthy* to do so. It is not legal to change the signature of a method which — directly or indirectly — overrides a method in the Java standard library. For example, we cannot change the signature of `actionPerformed` in Figure 8. We must also be careful not to "hide" a method by changing a method signature such that spurious overloading is introduced into the program. By building the complete inheritance tree of the program we can compute which are legal signature changes.

The weight of an edge from a CALL-node to an ENTER-node is computed based on the stealthiness of adding a storage structure argument to the called method. This depends on factors such as the number of arguments the method has and the types of these arguments. In general, adding yet another argument to a method that already has several should be good. Also, since we will be adding a pointer argument (a reference to an array, vector, hash table, etc.) it is probably stealthy if the method already has one or more such parameters.

We then heuristically select the `mark()`-nodes based on the node weight, the path weight, and the distance between `mark()`-nodes. Since we are passing storage structures as formals we need to identify all the methods that appear in the path between the point of creation and the selected `mark()`-node. The signature of these methods are modified accordingly, and all calls to these methods are modified to pass the storage structures as actual arguments in addition to their existing arguments. Calls to these methods which are *not* on the path between `mark()`-calls are also modified by adding dummy actuals.

6.2 Protecting Against Collusive Attacks

All the calls to the watermark creation methods are inlined. Additionally, the complete watermarked program can be obfuscated using any sequence of SandMark's obfuscators, complicating attacks by pattern-matching. One important advantage of dynamic graph watermarking is that typical obfuscating transformations (reorder statements, split/merge methods, split/merge classes, etc.) will have no effect on the inserted watermark code. This is in contrast to most other software watermarking methods where obfuscation after watermark embedding will destroy the mark.

SandMark also contains several *obfuscation executives*, loops that automatically select sequences of obfuscating transformations in a way that attempts to maximize the amount of confusion and minimize the amount of computational overhead introduced. The sequences of transformations are generated based on user input (which parts of the program are security and performance critical), profiling data, measurements of the level of obfuscation that each transformation incurs (using software complexity metrics), and a seed that initializes a random number generator. This allows us to obfuscate each differently fingerprinted version of a program using a different sequence of obfuscating transformations. As a result, collusive attacks become more difficult.

6.3 Protecting Against Pattern Matching

Figure 3 shows how a special class `Watermark` has been generated which is used to create nodes in the watermark graph. Obviously, this is not very stealthy. Instead, we search the application to be watermarked for a class that closely resembles the `Watermark` class. Ideally, the class should already have one or more fields of the appropriate reference type.If there are several candidate classes we break ties based on which class has more static instantiations. If there is no class with enough pointer fields, extra fields are added to the most appropriate class.

6.4 Future work: Avoiding Weak Cuts

An important part of Venkatesan's watermarking algorithm [56] is to bind the watermark code (in their case, a control-flow graph) tightly to the application. The reasoning is that segments of weakly connected code are unusual in real programs and are easy to find using existing graph algorithms. To prevent this attack Venkatesan et al. connect the watermark code to the application by adding a number of bogus control-flow edges realized by opaque predicates.

Our current implementation of the CT algorithm does not try to connect the graph building code to the application. If it is indeed the case (as Venkatesan et al. conjecture) that weakly connected code is unusual, this would leave us open to attacks that attempt to locate weak cuts in the control-flow graphs.

While it is certainly possible to use Venkatesan's technique to remedy this problem, there are far easier and stealthier methods. Since the structure of the watermark graph is known at each point in the program we can use it as a source of opaque values. For instance, a literal integer 3 in the application can be replaced by computation that uses the watermark graph as input to compute the value 3, perhaps as a function of the path length from the root to a leaf.

7 Evaluation

There is no widely accepted method for measuring the strength of a software watermarking algorithm. As a result, most previous publications in this area contain little or no theoretical or empirical evaluation. While measuring the data rate of an embedding is relatively straight-forward, measuring stealth and resilience is much harder, since this requires a *model* of how an adversary might measure and transform the watermarked program. Future work in this area will have to develop and validate such models. Stealth, in particular, is an elusive quantity, as any model needs to be validated using human subjects.

This section contains embryonic evaluation techniques for stealth, data rate, and resilience. While these techniques have yet to be validated, we believe they are a vast improvement over previous attempts, and form a solid basis for future study.

7.1 Stealth

Many different definitions of stealth are possible. In this paper we will define two possible measures. We say that an algorithm exhibits a high degree of *steganographic stealth* if, given access to a watermarking algorithm A and a watermarked program P_w, an adversary cannot determine if P_w has been watermarked with A or not. We say that an algorithm exhibits a high degree of *local stealth* if, given access to a watermarking algorithm A and a program P_w known to be watermarked with A, an adversary cannot determine the location of w within P_w.

In practice, a particular algorithm will produce stealthy marks for some host programs, and unstealthy ones for others. For example, a watermarking algorithm which encodes the watermark in the number of `xor` instructions in the program is likely to be unstealthy for most host programs (since `xor` instructions are unusual in common code), but stealthy for programs that contain cryptographic and graphic primitives. Our measure of stealth will therefor be based on a universe U of real programs: an algorithm is deemed to have a high degree of stealth if it is stealthy for a large fraction of the programs in U.

Stealth is also closely related to data rate. For example, using algorithm A, a stealthy embedding of a 4-bit watermark in program P might be possible, while a 5-bit embedding might not.

Definition 7.1.1 (local stealth) *Let A be a watermarking algorithm, b the length (in bits) of a watermark, k a secret key, U a universe of benchmark programs, and P a program in U. Let* $\mathrm{LS}_{A,b}(P)$ *be 0 if an adversary can locate a b-bit watermark w embedded in P using A and k, and 1 otherwise. We define the* local stealth *of A with respect to U and b as*

$$\text{local\_stealth}_{U,b}(A) = \frac{\sum_{P\in U} \mathrm{LS}_{A,b}(P)}{|U|} \quad (1)$$

Definition 7.1.2 (steganographic stealth) *Let A be a watermarking algorithm, b the length (in bits) of a watermark, k a secret key, U a universe of benchmark programs, and P a program in U. Let* $\mathrm{SS}_{A,b}(P)$ *be 0 if an adversary can determine that* P_b *(P watermarked with a b-bit watermark using algorithm A and key k) has been watermarked with A, and 1 otherwise. We define the* steganographic stealth *of A with respect to U and b as*

$$\text{stego\_stealth}_{U,b}(A) = \frac{\sum_{P\in U} \mathrm{SS}_{A,b}(P)}{|U|} \quad (2)$$

Depending on the exact definitions of LS and SS we obtain different models of the capability of an adversary. We can, for example, imagine a worst-case scenario where an adversary decompiles, reads, and learns to understand an entire program in order to locate redundant watermark code. Or, we can imagine a scenario where an adversary constructs a class attack which computes static statistics of a program from which a heuristic determines if the program contains a watermark or not.

7.1.1 Experimental Setup

We collected a universe of 622 Java jar-files from the Internet. The programs range in size from 6 to 40858 methods. We conjecture that since these programs come from a variety of sources, were written by a large number of programmers, are both *applets* and applications, and are of a wide range of sizes, they, in fact, form a reasonable random sampling of real Java programs.

For each jar-file a set of *instruction windows* was computed by sweeping a peephole of size 1-4 instructions over the instruction stream. The frequency of each window (the number of times it occurred in each jar-file) was recorded.

Prior to computing the windows, similar instructions were put into equivalence classes. This prevents anomalies resulting from different applications being compiled with different compilers, using different code-generation strategies. For example, to push the value 2, one compiler might generate `iconst_2`, while another might generate `iconst 2`. Therefore, all `iconst` instructions were put in the `ICONST` class, all `iload` instructions in the `ILOAD` class, all integer arithmetic instructions in the `IARITH` class, all branch instructions in the `IF` class, etc.

We generated watermark classes (as in Figure 3) for uncycled Radix Graph watermarks of size 4, 16, 32, and 64 bits. The graphs were split into three components. Instructions were put in equivalence classes, and instruction window frequencies were computed for the generated classes, just as for the downloaded applications. Figure 19 shows that instruction patterns involving `aload` and `astore` instructions are very common in the graph building code.

To simulate the embedding of a graph watermark in an application, the window frequencies of the watermark class were added to the frequencies of each application. It is important to note that this simulation is only approximate. First of all, in an actual embedding some windows in the application would be broken up by the inserted watermarking code. Also, before and/or after an actual embedding we would typically apply code obfuscation to increase stealth. However, since our implementation is not completely automatic (it requires the user to annotate the code and then run it with a secret input sequence to produce a trace), it is impractical to produce a more accurate set of windows.

7.1.2 Steganographic Stealth

For the purpose of this paper we only computed steganographic stealth. It will be the subject of future research to compare all known software watermarking algorithms (many of them implemented within SandMark) with respect to different measures of stealth.

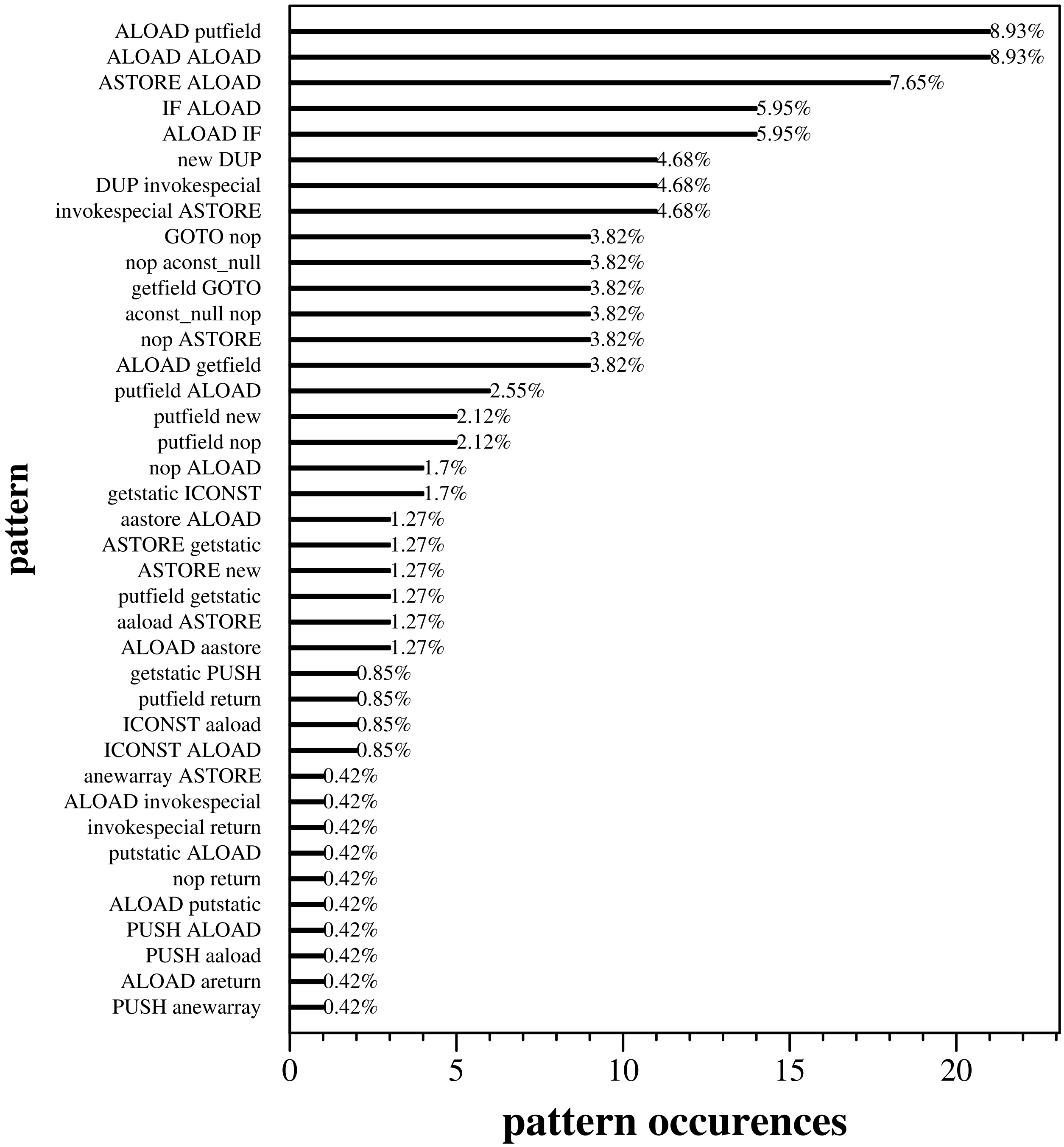


Figure 19: Instruction window frequencies generated for a 32-bit CT watermark. The window size is two instructions.

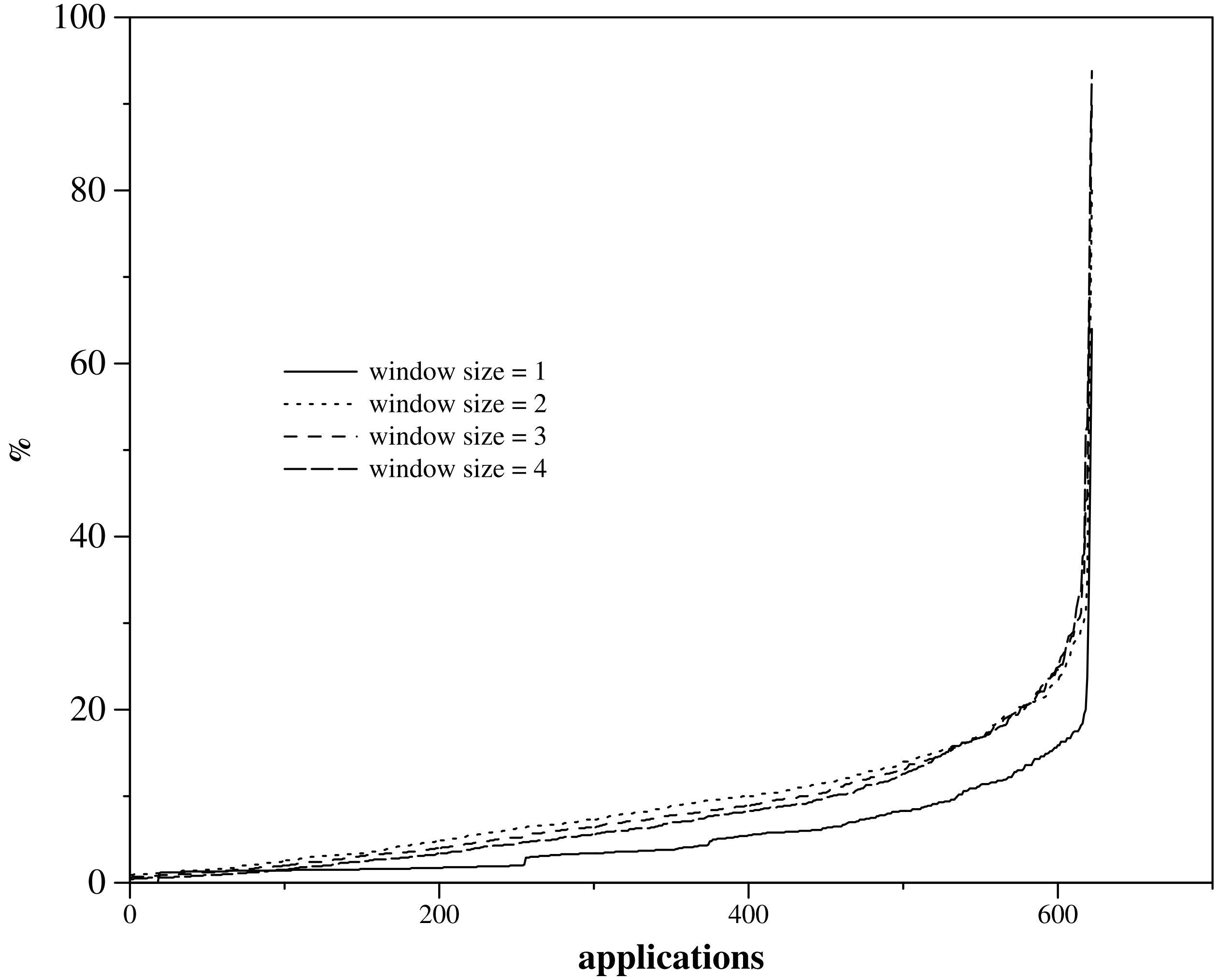


Figure 20: Steganographic stealth evaluation. For each of the 622 applications in our benchmark universe, the graph shows the fraction of instruction windows which occur in the embedded 32-bit CT watermark code, but which do not occur in the application itself.

We define

$$\mathrm{SS}_{A,b}(P) \quad = \quad \begin{cases} 1, & \text{if } \frac{|\text{window types that occur in a } b\text{-bit watermark but not in } P|}{|\text{window types that occur in } P_b|} < \delta \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Intuitively, $\mathrm{SS}_{A,b}(P) = 1$ if both P and P_b contain identical sets of window types, and 0 if none of the windows that occur in CT code occur in P.

Figure 20 shows that if we set $\delta = 0.1$, it would be stealthy to embed a 32-bit watermark in 395 out of the 622 applications in our benchmark universe.

7.2 Data Rate

Figure 21 shows the runtime size of the different graph structures as a function of the size of the watermark.

We used linear regressions to estimate the parameters a, b of the best-fit equation $S(m) = am + b$ for each of our encoding methods. Here m is the number of bits in the watermark integer w, that is, $m = \lceil \lg(w+1) \rceil$ when w is

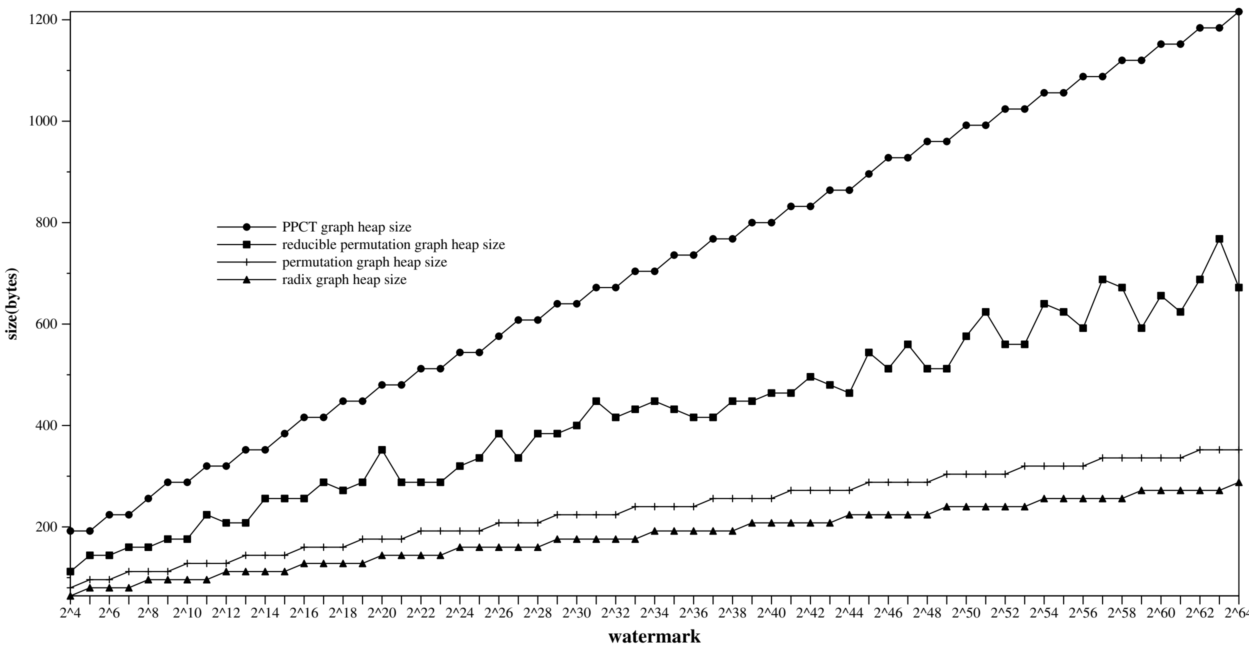


Figure 21: Sizes of the graph as a function of the size of the watermark.

any non-negative integer. As noted below, all our estimated parameters are in good agreement with the theoretical expectations for dynamic bitrate we developed in earlier sections.

As expected, the PPCTs have the lowest bitrate. Our regression line is $S_{ppct}(m) = 17m + 127$. We had expected a bitrate of 16 bytes per bit, not the 17 bytes per bit of this equation, however this expectation was based on an asymptotic approximation that ignored low-order terms. The R^2 metric for this regression is 0.999, indicating that all our observations are very accurately predicted by this equation, and implying that the low-order terms in the bitrate expression for PPCTs are almost negligible.

Our theoretical analysis predicted that the bitrates of our other three methods would be slowly increasing functions of the number of bits m in the watermark. This was reflected in our experimental data, however the nonlinearity is so slight over the range of our experimentation that it is almost imperceptible in our plots.

Our regression line for the Reducible Permutation Graph is $S_{RPG}(m) = 9.3m + 104$, showing that RPGs have roughly twice the bitrate of PPCTs. We see a modest amount ($R^2 = .970$) of unexplained variance. A tiny amount of this unexplained variance is the expected nonlinearity, but the majority is clearly visible as "noise" in Figure 21. We believe this apparent noise is due to the dependence of the size of the preamble of an RPG, on the value of the watermark w being encoded as an RPG: two watermarks w with the same number of bits can have differing sizes of RPGs. Our asymptotic analysis indicated that the RPG size would vary by +/- 33% for any given number of bits m, however this was an asymptotic upper bound and our experiments indicate that the variability is perhaps half of this for m in the range of our measurements.

Our regression lines for Permutation Graphs and Radix Graphs are similar, but with Radix Graphs having slightly higher bitrate, in accordance with our theory. We find $S_{PG}(m) = 4.4m + 84$ and $S_R(m) = 3.4m + 67$, with $R^2 = .991$ in both cases. The unexplained variance in this regression is almost entirely attributable to the slight nonlinearity that was predicted by our theory.

We conclude that PGs and RGs have somewhat more than twice the bitrate of RPGs, and that RPGs have about twice the bitrate of PPCTs. None of our methods required more than 1.3 KB of dynamically allocated storage to embed a 64-bit watermark.

Figure 22 shows the size of the generated bytecode for each of our encoding methods. We fit regression lines to this dataset to estimate the static bitrates of our watermarking methods. As expected, the static bitrates are in proportion to the dynamic bitrates: 21 codebytes per watermark bit for PPCTs, 12 for RPGs, 6.3 for PGs, and 4.9 for Radix Graphs. Each regression line had a modest additive "overhead" term of between 77 and 150 codebytes.

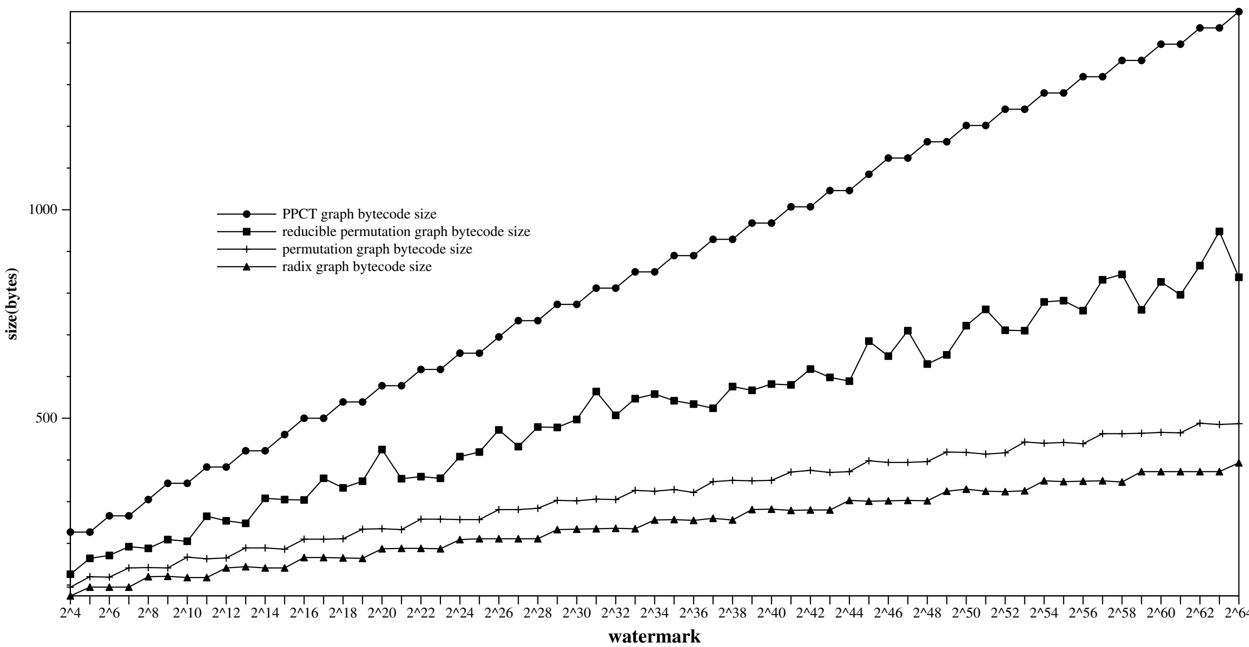


Figure 22: Sizes of the graph building bytecode as a function of the size of the watermark.

The amount of bytecode necessary to build a graph of a particular type of n nodes and m edges can differ due to the structure of the graph. In particular, the size can depend on the number of edges between low-numbered nodes. This is due to the fact that there are two kinds of JVM store instructions: a one-byte instruction is sufficient to store into low-numbered local variables, but a two-byte instruction is necessary to store into high-numbered locals. This explains the non-monotonic nature of the radix graph curve in Figure 22.

Figure 23 shows the size of the generated bytecode as a function of k, the number of components into which the graph is split. Our statistical analysis, with a generalized linear model, revealed that the average increase in bytecode size was $22(k-1)\%$. Thus, for example, the 4.9 codebytes per watermark bit of the Radix Graph method becomes approximately 6 codebytes per watermark bit if the Radix Graph is built up in two components (which are subsequently merged) rather than in a single component.

The observed linear dependence on k is easily explained: the more components the graph is split into, the more extra code needs to be generated to merge the components together.

Watermarking with many small components is probably more stealthy than using a single component. However if k is very large, then the total amount of watermarking code may become large enough that an attacker may be able to recognize some frequently-repeated patterns.

7.3 Resilience

SandMark currently contains approximately forty code obfuscators. They perform a wide variety of transformations on code and data, such as merging methods, splitting classes, splitting arrays, changing the signature of methods, turning scalars into objects, etc. None of these transformations prevent extraction of a watermark inserted by the CT algorithm, except for the NODESPLITTER transformation, described in Section 4.2. Using cycled rather than plain graphs counters this attack. However, as shown in Figure 24, this resilience comes at a significant cost: the dynamic data rate is reduced by a factor of nine, and the static data rate is reduced by a factor of seven.

Figure 25 shows the overhead of applying multiple node-splittings to the SpecJVM benchmark suite. These results show that on many applications an attacker can easily apply one or two node-splits without having to worry about performance overhead. The variance is high, however. Applying two node-splits makes `_228_jack` a 16% slower and `_227_mtrt` 286% slower. Therefore, for less performance critical applications it may in many cases be worthwhile to use cycled graphs, if the lower data rate can be tolerated.

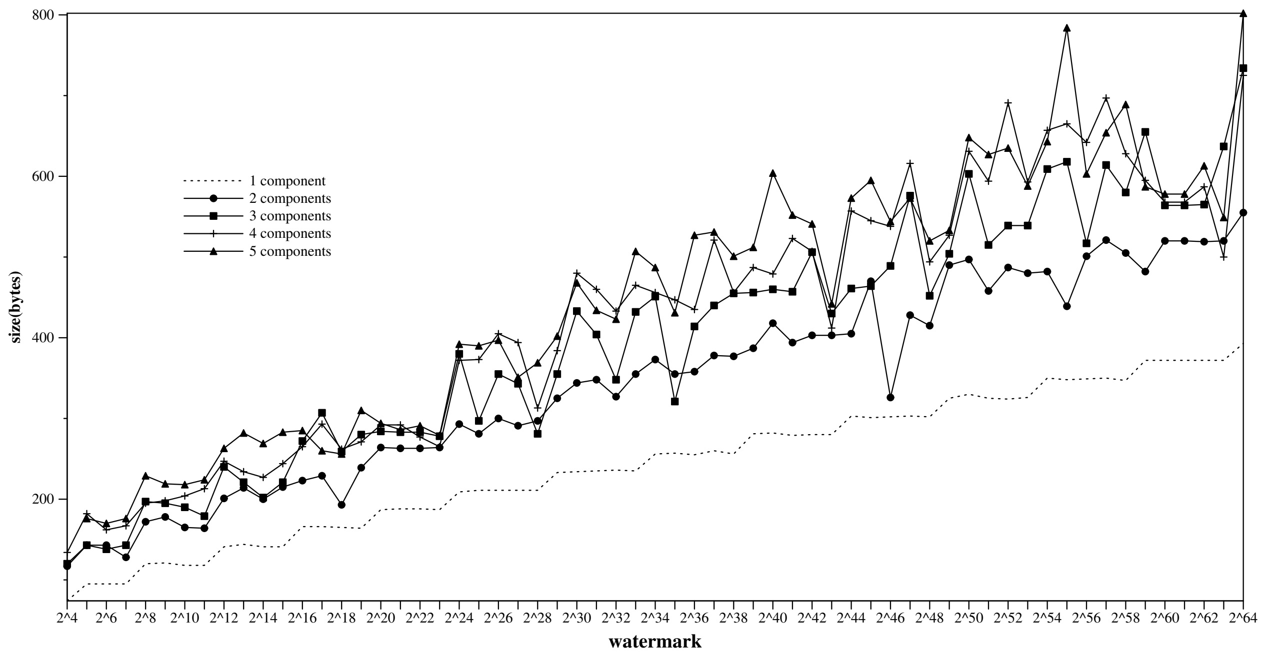


Figure 23: Size of the bytecode for building a Radix Graph, as a function of the number of components into which the graph is split.

The BLOAT [44] bytecode optimizer is included in the SandMark system. It also has no effect on the success of watermark extraction.

8 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 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.

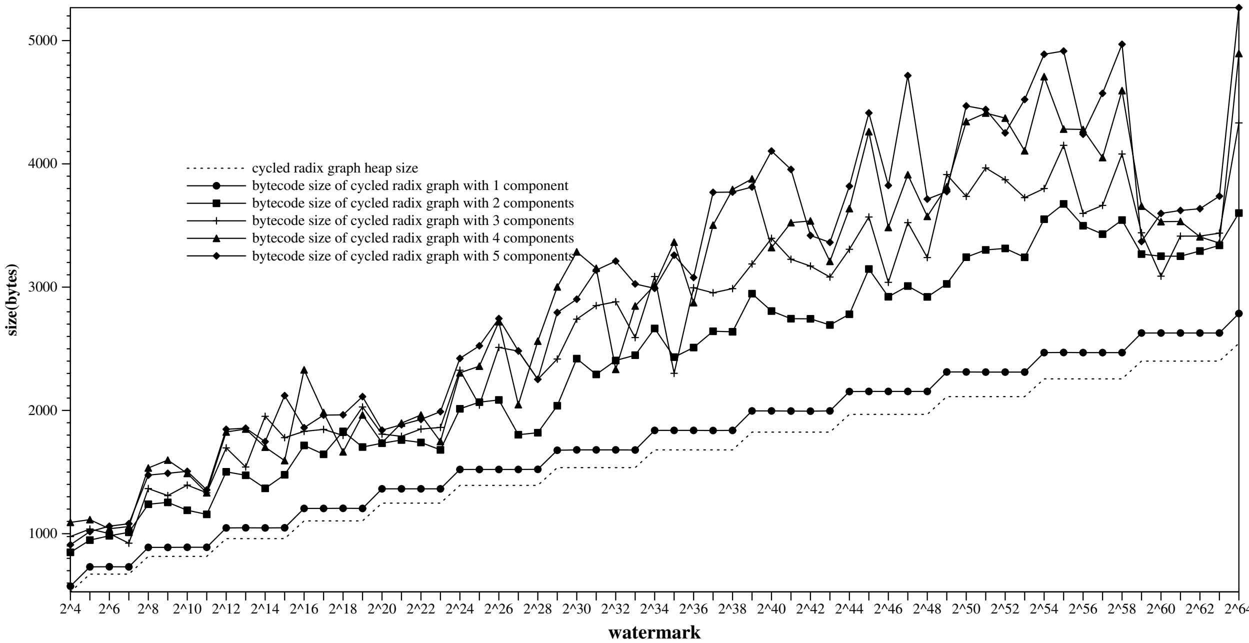


Figure 24: Bytecode size, and runtime heap size, for cycled radix graphs.

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 [19] 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.

8.1 SandMark and JavaWiz

The first implementation based on the CT algorithm was JavaWiz [3]. This section contrasts the SandMark implementation with that of JavaWiz.

The JavaWiz implementation is unkeyed. In the SandMark implementation, a particular input sequence serves as a key for embedding and is required for watermark extraction. User annotation of the source program is required to make the watermark input-dependent.

Both implementations embed an arbitrary bit string as the watermark, treating it as a single large integer. JavaWiz requires the input of an integer; SandMark accepts either an integer or an arbitrary text string.

JavaWiz encodes the watermark in a Planted Plane Cubic Tree (PPCT). SandMark offers a choice of four encodings, including PPCT. SandMark also offers an option to use a "cycled graph" encoding (of any of its representations)

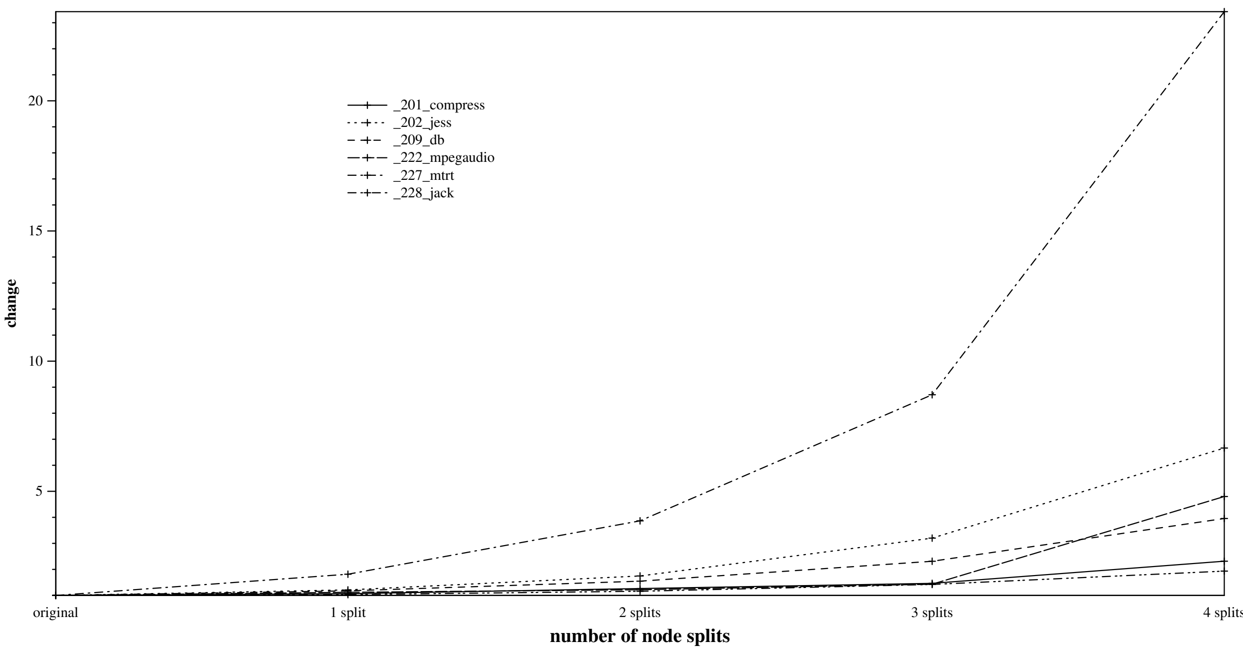


Figure 25: Overhead of applying sequence of node-splittings to the SpecJVM benchmark suite. The `javac` benchmark was excluded since it currently fails on the NODESPLITTER obfuscation.

---

as a defense against node-splitting attacks.

SandMark generates several code fragments which are inserted at user-specified locations. Code generation requires some care to allow the segments to be executed out of order. JavaWiz constructs the graph in a single sequence.

JavaWiz uses a user-specified class for graph nodes. SandMark attempts to deduce one automatically.

Both implementations require access to portions of the source code of the target application: SandMark for annotating, and JavaWiz for modification.

Both implementations rely on features of the Sun reference implementation of the Java language. SandMark uses JDI tracing facilities as part of both the embedding and recovery stages. JavaWiz uses the heap dump facility to recover a constructed watermark.

Both implementations generate relatively straightforward code under the assumption that it will be subsequently obfuscated.

9 Summary

Software watermarking embeds an identifying value in a program. The ideal watermark is easily extractable with a key but difficult to detect otherwise. It imposes little program overhead and is robust against a wide variety of program transformations.

Most software watermarks are static: They are applied to, and detected in, an executable binary file. In this paper we have described the CT algorithm, where a dynamic watermark is encoded by a graph that is built during program execution. Graph construction is driven by a specific input sequence that serves as the key. A dynamic watermark is much harder to detect than a static watermark and is also much more robust against transformations such as obfuscation and optimization.

The CT watermarking scheme has been implemented in the SandMark package, a large collection of tools for modifying Java programs. The SandMark implementation provides several options for configuring the watermark, including a choice of graph encodings. User annotation of the target program specifies the dependence on program input and the locations for code insertion. Incorporation of watermarking as part of SandMark allows obfuscation after modification.

We have explored tradeoffs among graph encodings and watermark sizes and evaluated their effects on code size and memory requirements. We have measured execution-time overhead and found a real but not impractical increase. We have presented a model for measuring the stealth of a software watermark and found that the code introduced by the CT algorithm is stealthy for a large fraction of real Java programs.

We tested the CT watermark's resilience against SandMarkś suite of obfuscators, and found it invulnerable to all but node splitting. This somewhat costly obfuscation is in turn overcome by use of a more redundant graph encoding.

The SandMark package in which CT has been implemented can be downloaded from `sandmark.cs.arizona.edu`.

Acknowledgments: We thank Edward Carter, Andrew Huntwork, Kamlesh Kantilal, and Jasvir Nagra for implementation assistance.

References

[1] BCEL, February 2004. `http://jakarta.apache.org/bcel`.

[2] DynamicJava, February 2004. `http://koala.ilog.fr/djava`.

[3] JavaWiz, February 2004. `http:www.cs.purdue.edu/homes/madi/wm/`.

[4] D.J. Albert and S.P. Morse. Combating software piracy by encryption and key management. *IEEE Computer*, 17(4):68–73, April 1982.

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

[6] Geneviève Arboit. A method for watermarking Java programs via opaque predicates (extended abstract). In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002. `http://citeseer.nj.nec.com/arboit02method.html`.

[7] 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.

[8] Brenda S. Baker and Udi Manber. Deducing similarites in java sources from bytecodes. In *In Proc. of Usenix Annual Technical Conf.*, 1998.

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

[10] Christian Collberg, Edward Carter, Stephen Kobourov, and Clark Thomborson. Error-correcting graphs. In *Workshop on Graphs in Computer Science (WG'2003)*, June 2003.

[11] Christian Collberg, Ginger Myles, and Andrew Huntwork. SANDMARK — A tool for software protection research. *IEEE Magazine of Security and Privacy*, 1(?), August 2003.

[12] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, San Antonio, TX, January 1999.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997. `http://citeseer.nj.nec.com/collberg97taxonomy.html`.

[14] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages, ICCL'98*, Chicago, IL, May 1998.

[15] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1998.

[16] Compaq. FreePort Express. `hhttp://www.support.compaq.com/amt/tools/migrate-cover.html`, February 2004.

[17] Patrick Cousot and Radhia Cousot. An abstract interpretation-based framework for software watermarking. In *ACM Principles of Programming Languages*, 2004.

[18] R.L. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, Assignee: Microsoft Corporation, 1996. `http://www.delphion.com/details?pn=US05559884__`.

[19] 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.

[20] Saumya Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[21] Saumya Debray, Robert Muth, Scott Watterson, and Koen De Bosschere. ALTO: A link-time optimizer for the Compaq Alpha. *Software — Practice and Experience*, 31:67–101, January 2001.

[22] Saumya Debray, Benjamin Schwarz, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*, September 2001.

[23] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL'96*, pages 1–15, St. Petersburg Beach, Florida, 21–24 January 1996.

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

[25] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

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

[27] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Notices*, 16(3):63–74, 1981.

[28] 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.

[29] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

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

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

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

[33] Sukhamay Kundu and Jaydev Misra. A linear tree partitioning algorithm. *SIAM Journal of Computing*, 6(1):151–154, March 1997.

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

[35] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[36] A. Monden, H. Iida, K. Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking Java programs. In *24th Computer Software and Applications Conference*, 2000.

[37] Akito Monden, Hajimu Iida, Ken ichi Matsumoto, Koji Torii, and Yuuji Ichisugi. Watermarking method for computer programs. In *Proceedings of the 1998 Symposium on Cyptography and Information Security (SCIS'98 - 9.2A)*, January 1998. "In Japanese".

[38] Ryoichi Mori and Masaji Kawahara. Superdistribution: the concept and the architecture. *The Transactions of the IEICE*, 73(7), July 1990. `http://www.virtualschool.edu/mon/ElectronicProperty/MoriSuperdist.html`.

[39] 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.

[40] John C. Munson and Taghi M. Kohshgoftaar. Measurement of data structure complexity. *Journal of Systems Software*, 20:217–225, 1993.

[41] Hirofumi Muratani. A collusion-secure fingerprinting code reduced by chinese remaindering and its random-error resilience. In *Information Hiding: 4th International Workshop, IHW 2001*, pages 303–315, Pittsburgh, PA, April 2001.

[42] Ginger Myles and Christian Collberg. Software watermarking through register allocation: Implementation, analysis, and attacks. In *International Conference on Information Security and Cryptology*, 2003.

[43] David Nagy-Farkas. The Easter egg archive. `http://www.eeggs.com`, February 2004.

[44] Nathaniel Nystrom. BLOAT – the Bytecode-Level Optimizer and Analysis Tool. `http://www.cs.purdue.edu/s3/projects/bloat`, February 2004.

[45] E. I. Oviedo. Control flow, data flow, and program complexity. In *Proceedings of IEEE COMPSAC*, pages 146–152, November 1980.

[46] Jens Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of ACSAC'00, 16th Annual Computer Security Applications Conference*, pages 308–316, 2000. `http://citeseer.nj.nec.com/323325.html`.

[47] 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.

[48] Fabien A. P. Petitcolas. Stirmark 3.1. `http://www.cl.cam.ac.uk/~fapp2/watermarking/stirmark`, February 2004.

[49] Josef Pieprzyk. Fingerprints for copyright software protection. In *Proceedings of the Second International Workshop on Information Security, ISW'99*, pages 178–. Springer, 1999. LNCS 1729.

[50] 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.

[51] Gang Qu and Miodrag Potkonjak. Analysis of watermarking techniques for graph coloring problem. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 190–193. ACM Press, 1998.

[52] G. Ramalingam. The undecidability of aliasing. *ACM TOPLAS*, 16(5):1467–1471, September 1994.

[53] Tapas Sahoo and Christian Collberg. Software watermarking in the frequency domain: Implementation, analysis, and attacks. Technical Report TR04-07, Department of Computer Science, University of Arizona, March 2004.

[54] 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.

[55] Julien P. Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999.

[56] Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, Pittsburgh, PA, April 2001.

A An Example

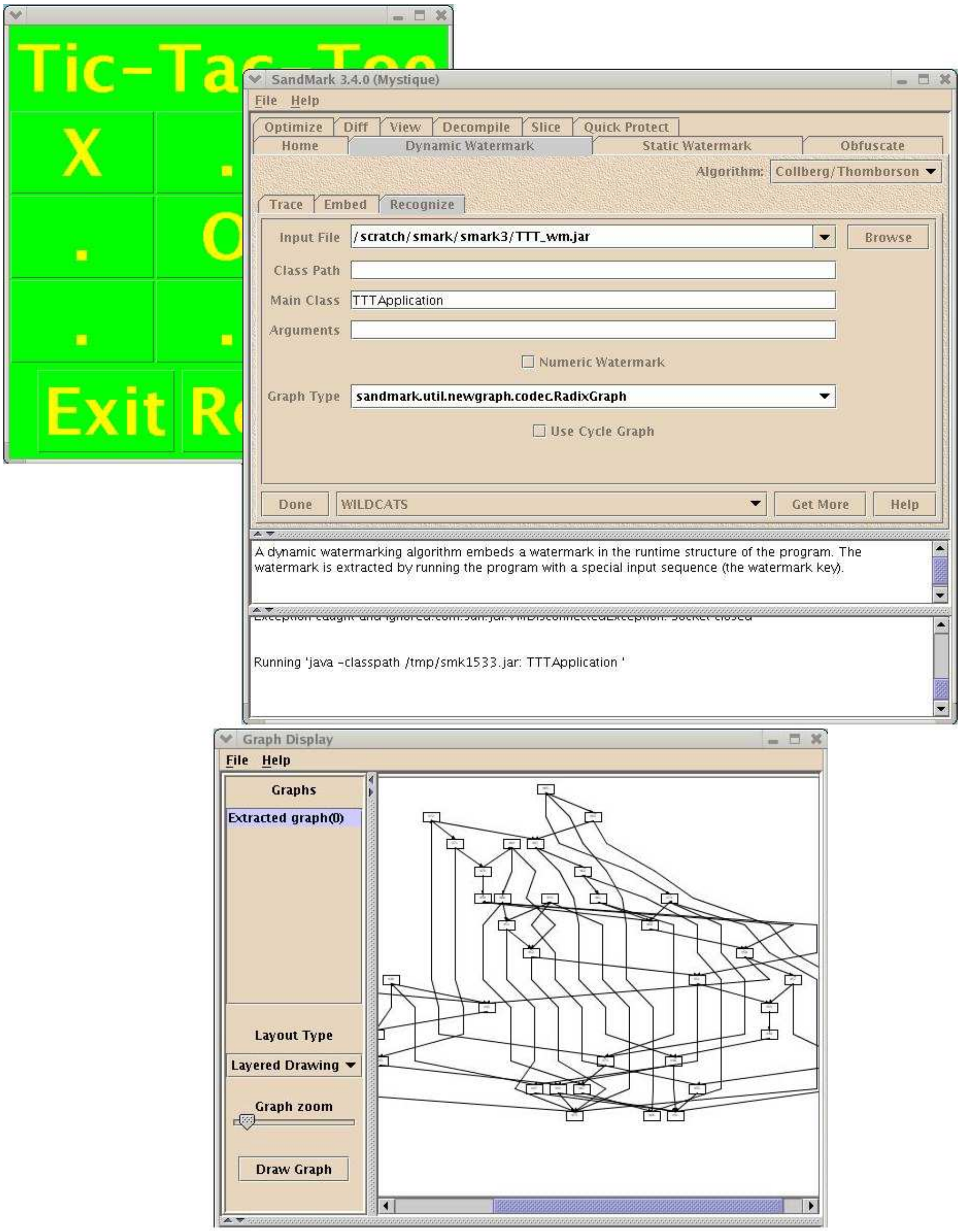


Figure 26: Screenshot of SandMark recognizing the watermark `"WILDCATS"` in a tic-tac-toe application. The user has entered the secret input sequence (clicking `X-O-X` along the diagonal) The bottom pane shows the recognized graph.