

# Dynamic Graph-Based Software Fingerprinting

CHRISTIAN S. COLLBERG

University of Arizona

CLARK THOMBORSON

University of Auckland

and

GREGG M. TOWNSEND

University of Arizona

---

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

The Software Fingerprinting 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 article, we describe a software fingerprinting technique in which a dynamic graph fingerprint is stored in the execution state of a program. Because of the hardness of pointer alias analysis such fingerprints are difficult to attack automatically.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; E.1 [**Data Structures**]: *Graphs and networks*; K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection—*Proprietary rights*

General Terms: Languages; Legal Aspects, Security

Additional Key Words and Phrases: Software piracy, software protection, watermarking

---

An earlier version of this article was published as COLLBERG, C., AND THOMBORSON, C. 1999. Software watermarking: Models and dynamic embeddings. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Authors' addresses: C. S. Collberg and G. M. Townsend, Department of Computer Science, University of Arizona, Tucson, AZ 85721; email: {collberg,gmt}@cs.arizona.edu; C. Thomborson, Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand; email: cthombor@cs.auckland.ac.nz.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org). © 2007 ACM 0164-0925/2007/10-ART35 \$5.00 DOI 10.1145/1286821.1286826 <http://doi.acm.org/10.1145/1286821.1286826>

ACM Transactions on Programming Languages and Systems, Vol. 29, No. 6, Article 35, Publication date: October 2007.

**ACM Reference Format:**

Collberg, C. S., Thomborson, C., and Townsend, G. M. 2007. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.* 29, 6, Article 35 (October 2007), 67 pages. DOI = 10.1145/1286821.1286826 <http://doi.acm.org/10.1145/1286821.1286826>

---

## 1. INTRODUCTION

## 1.1 Background

*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 [Anderson and Peticolas 1998].

Steganography—in the form of media *watermarking* and *fingerprinting*—has also found commercial applications. In a typical application of image watermarking, a copyright notice identifying the intellectual property owner is imperceptibly embedded into the host image. Fingerprinting is a form of watermarking in which an individualized mark is embedded into a copy of the media. A typical fingerprint would include vendor, product, and customer identification numbers. This allows the intellectual property owner to trace the original purchaser of a pirated media object.

Our interest is the fingerprinting of *software*. Although much has been written about protection against software piracy [Albert and Morse 1982; Herzberg and Karmi 1984; Hauser 1995; Simmel and Godard 1994; Herzberg and Pinter 1987; Maude and Maude 1984; Mori and Kawahara 1990], software fingerprinting is an area that has received very little attention. This is unfortunate since software piracy is estimated to be a 15 billion dollar per year business [International Planning and Research Corporation 2003; Malhotra 1994].

Specifically, this article is concerned with a game between two adversaries, a software producer, Alice, and a software pirate, Bob. Prior to selling her program  $P$  to Bob, Alice will *embed* Bob's unique identifier  $Bob$ , yielding a new program  $P_{id=Bob}$ :

$$P \xrightarrow{Bob, key} P_{id=Bob}.$$

When Alice locates a suspected pirated copy of  $P$  she can *extract* the identity of the original purchaser from the marked copy, and use this as part of a legal argument against him:

$$P_{id=Bob} \xrightarrow{key} Bob$$

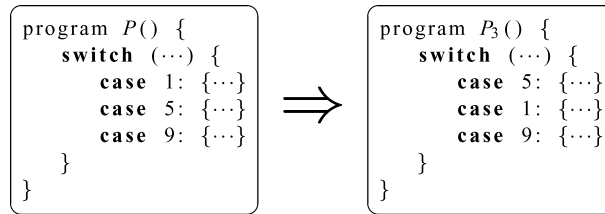
In this game, it is assumed that Bob knows the embedding and extraction algorithms used by Alice, but not the mark nor the secret *key* necessary for extraction. His goal, then, is to create a disturbed copy  $P'_{id=Bob}$  from  $P_{id=Bob}$  (without damage to its utility) to the point that it can be illicitly resold without

Alice being able to extract the mark:

$$P_{\text{id=Bob}} \longrightarrow P'_{\text{id=Bob}} \xrightarrow{\text{key}} \text{Bob}$$

Software fingerprinting differs from other anti-piracy techniques in significant ways. First of all, unlike dongles and related techniques, fingerprinting does not rely on the existence of specialized, secure, hardware. Second, fingerprinting does not *prevent* software from being pirated but rather allows tracing of perpetrators after the fact. Finally, unlike software license checking code that is embedded within the application itself (and hence provides valuable clues to the attacker about the location of the mark), a fingerprint extractor is *external* to and not delivered with the marked application.

To illustrate these points, consider the class of fingerprinting algorithms based on *reordering*. The idea is that whenever there are  $m$  language elements that can be arbitrarily reordered,  $\log_2(m!) \approx \log_2(\sqrt{2\pi m}(m/e)^m) = O(m \log m)$  bits of a fingerprint can be embedded. For example, before selling her program  $P$  below to Bob, Alice embeds his unique customer identifier, 3, resulting in the marked program  $P_3$ . The embedding is accomplished by reordering the arms of a switch-statement to yield the third permutation  $\langle 5, 1, 9 \rangle$  (in lexicographic order) of  $\langle 1, 5, 9 \rangle$ :



When Alice discovers a pirated copy of  $P_3$  she applies her fingerprint extractor in order to discover who purchased the original copy. The extractor locates the relevant switch statement and converts the permutation  $\langle 5, 9, 1 \rangle$  back into the mark 3, letting Alice know that this copy was sold to Bob. To prevent recognition, prior to resale of  $P_3$  Bob randomly permutes the arms of every switch statement in the program thereby obliterating all marks.

### 1.2 Contributions

This article makes the following contributions.

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

In the remainder of this section, we discuss software fingerprinting schemes, attacks on fingerprinting systems, and practical considerations for the use of software fingerprinting. In Section 2, we give an informal classification of software fingerprinting algorithms, discuss related work, present the ideas behind the CT fingerprinting algorithm, and show the SandMark software protection research tool in which CT has been implemented. 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, data 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.

### 1.3 Software Fingerprinting Schemes

Conceptually, a software fingerprinting system consists of functions

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

where *embed* transforms a program  $P$  into  $P_w$  by embedding the fingerprint  $w$  using a secret *key*, and where *extract* extracts  $w$  from  $P_w$ . The functionality of  $P$  and  $P_w$  must be identical: fingerprint embedding must preserve program semantics.

There are many minor variations of the definitions given above. For example, the *extract* function may return a list of fingerprints, as well as a confidence level for each fingerprint in the list. Some practical fingerprinting systems are *unkeyed*; the *key* parameter is suppressed in their embedding, extraction and recognizing functions. The *key* parameter may include a complete copy of the original (unfingerprinted) program  $P$ . This extreme case is called a *non-blind* fingerprinting system, in which the *extract* and *recognize* functions are best expressed as having another explicit parameter, this being a copy of the unfingerprinted object.

Since software fingerprinting is done in an adversarial environment, it becomes essential to properly model the goals and capabilities of the attacker. In particular, the attacker will thwart the defender's goals by discovering functions of the following form:

$$\begin{aligned} \text{detect}(P_w) &\rightarrow [0.0, 1.0] \\ \text{attack}(P_w) &\rightarrow P'_w. \end{aligned}$$

The first function models a strong (steganographic) form of *confidentiality*, whereby an adversary is unable to *detect* whether or not a possibly fingerprinted object is actually fingerprinted or not.

The second function models a form of fingerprint *availability*, whereby an adversary is unable to *attack* a suspected fingerprinted object to the point that its fingerprint becomes unextractable or unrecognisable, except by modifying the object so severely that it becomes useless.

We also consider collusive attacks of the forms  $\text{detect}(\overline{P_w})$  and  $\text{attack}(\overline{P_w})$  where an attacker has access to a collection  $\overline{P_w}$  of programs.

## 1.4 Evaluating Fingerprinting Systems

We define three desiderata of fingerprinting systems. A *stealthy* fingerprinting system is one that resists adversarial attempts to discover a *detect* function. A *resilient* fingerprinting system is one that resists adversarial attempts to discover an *attack* function. We cannot hope to have a precise metric for either of these aspects, in the absence of a precise (and analytically tractable) model of the adversary. Instead, we evaluate these two desiderata only in qualitative terms, except in the case of the specific and very weak adversary considered in Section 7.

Our third desideratum we call *data rate*. Data rate expresses the space efficiency of the fingerprinting system. There are two variants: *dynamic data rate* and *static data rate*. In both variants, the numerator is the number of bits in the fingerprint, and the denominator measures the overhead (in bytes) added by the fingerprint. In the static measure, we compute the increase in the compiled size of the fingerprinted program as we add fingerprinting bits. In the dynamic measure, we compute the increase in the consumption of storage space, as a function of the number of fingerprinting bits, when the fingerprinted program is run.

An ideal software fingerprinting system would have maximal data rate, stealth, and resilience. In practice, it is not possible to maximize any one of these without having some deleterious effect on one or both of the other desiderata. For example, increasing the data rate tends to decrease the stealth of a fingerprint, because the addition of more “fingerprinting code” will give more clues to an adversary’s *detect* function. Another simple trade-off is to increase resilience at the expense of a lower data rate and a lower stealth, by using several different methods to embed the same fingerprint value repeatedly in the same program. The resilience of this scheme is raised, because an adversary must remove or distort all these fingerprints. However, the data rate is certainly lower. The stealth is probably lower, since the adversary will know the program is fingerprinted if he is able to detect any one of the fingerprints.

## 1.5 Attacks on Fingerprinting Systems

The attacker’s *detect* function is an unkeyed version of the defender’s *recognize* function. The suppression of the key parameter reflects the usual assumption in a security proof: the defender will not directly reveal the key to the adversary.

Since we assume the *recognize* function is nonsecret, an adversary can trivially define a randomized *detect* function for use in a brute-force search. This randomized function is an application of *recognize()*, where the value of the *key* and *w* parameters are chosen probabilistically according to the adversary’s prior distribution on possible key-watermark pairs. Thus, we see that the adversary’s uncertainty about the key and watermark space is an important consideration in any evaluation of stealth.

The most dangerous *detect* function, from the defender’s point of view, is perfectly accurate, with neither false-negatives nor false-positives as defined

by the following two equations:

$$\begin{aligned} & \forall P, w : \text{detect}(\text{embed}(P, w)) \geq 0.5 \\ \forall X : \text{detect}(X) < 0.5 & \Rightarrow \forall S, P, w : \text{embed}_S(P, W) \neq X. \end{aligned}$$

On information-theoretic grounds, no ideal fingerprint detector can possibly exist for any finite object  $X$ , because of the unbounded range of the variables  $S$ ,  $P$ , and  $w$  in the false-positive equation above. In practical settings, however, any competent adversary will have prior probability distributions on the range of fingerprinting systems  $S$ , programs  $P$ , and fingerprints  $w$  used by the defender. The adversary will also have a prior belief in whether or not any specific program  $X$  has been fingerprinted. Such an adversary may use the output of a nonideal detector (which must be designed and configured appropriately, in light of their prior probability distributions) to generate posterior distributions of sufficient accuracy to guide a devastating attack.

To evaluate the resilience of a fingerprinting scheme we must know how well it stands up to different types of attacks. This is, by necessity, an incomplete argument. We cannot precisely specify all possible attacks by all possible adversaries, and we cannot expect a real system to provide “perfect” security against any nontrivial attack. Bender et al. [1996] characterize the security of media-fingerprinting systems as follows: “. . . 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.”

Below, we illustrate these differing attacks in the following scenario. Alice fingerprints a host program  $P$  with fingerprint  $w$  and key  $key$ , and then sells  $P$  to Bob:

$$P_w = \text{embed}(P, w, key).$$

Before Bob can resell  $P_w$  he must ensure that the fingerprint has been rendered useless, or else Alice will be able to prove that her intellectual property rights have been violated.

**1.5.1 Additive Attacks.** Bob can augment  $P$  by inserting his own fingerprint  $w$  (or several such marks). An *effective* additive attack is one in which Bob’s bogus mark seems to be present in the modified program. Formally, Bob seeks a bogus mark  $w'$ , possibly without knowing Alice’s recognition  $key$ , for the fingerprinted program  $P_w$  with the following properties:

$$\begin{aligned} \text{recognize}(\text{attack}(P_w), key, w') & > 0.5 \\ \text{recognize}(\text{attack}(P_w), key, w) & > 0.5 \end{aligned}$$

Note that Bob’s mark  $w'$ , as well as Alice’s mark  $w$ , are both recognizable in the attacked program.

**1.5.2 Subtractive Attacks.** If Bob can detect the presence and (approximate) location of  $w$ , he may try to *crop* it out of  $P_w$ . An *effective* subtractive attack is one where the cropped program has retained enough original content to still be of value to Bob. A formal statement of the subtractive attack is that Bob seeks a semantics-preserving *attack* transformation on the fingerprinted

program such that

$$\text{recognize}(\text{attack}(P_w), \text{key}, w) < 0.5$$

**1.5.3 Distortive Attacks.** If Bob cannot locate  $w$  and is willing to accept some degradation in quality of  $P$ , he can apply distortive transformations uniformly over the program and, hence, to any fingerprint it may contain. An *effective* distortive attack is one where Alice can no longer detect the degraded fingerprint, but the degraded program still has value to Bob. A distortive attack is thus very similar to a subtractive attack, but the former is an unsound attack and the latter is a sound attack, in the terminology recently proposed by Madou et al. [2005]. An attack is sound if it is guaranteed to work in all program executions. By contrast, an unsound attack will result in a program whose behavior differs on some inputs.

Of the attacks listed above, distortive attacks seem to be the most problematic for media watermarking systems. Common image transforms such as cropping, rotation, and lossy compression will cause most image fingerprints to be unrecognizable and unextractable [Anderson and Peticolas 1998; Peticolas et al. 1998]. One advantage of software fingerprinting over media fingerprinting, as we will see later in this article, is that a software fingerprint can be embedded in ways that are difficult to distort.

Bob may also indulge in combined attacks. For example, he may construct a combined additive and subtractive attack, whereby his mark  $w'$  is the only mark that is recognizable in the attacked program.

**1.5.4 Collusive Attacks.** Since every distributed copy of Alice's program embeds a different secret message, fingerprinted programs are also vulnerable to *collusive attacks*. The idea is that an adversary might attempt to gain access to several fingerprinted copies of a program, compare them to determine the location of the fingerprints, and, as a result, be able to perform an additive, subtractive, or distortive attack.

**1.5.5 Protocol Attacks.** In cases where the defender Alice lacks complete security over the choice of recognition (or extraction) apparatus and keys, Bob may indulge in a *protocol attack*. There are several variations [Craver et al. 1998]. Bob may produce a bogus key  $k'$  that purports to prove the existence of his mark  $w'$  in the unchanged program  $P_w$ , using Alice's recognition function:

$$\text{recognize}(P_w, k', w') > 0.5$$

Alternatively, Bob may produce a bogus recognition function that purports to prove the existence of his mark  $w'$  in the unchanged program  $P_w$ :

$$\text{recognize}'(P_w, \text{key}, w') > 0.5.$$

Bob may subvert several elements of the protocol simultaneously:

$$\text{extract}'(P_w, k') \rightarrow w'.$$

Protocol attacks may be combined with an additive, subtractive or distortive attack on the program itself. For example, Bob might develop an additive attack

that works in combination with a protocol attack to confuse a courtroom test of ownership in a case where Alice is unable to require the use of her extraction function and her key:

$$\text{extract}'(\text{attack}(P_w), k') \rightarrow w'.$$

Protocol attacks on the recognition or extraction function are outside the scope of the remainder of this article, because the best defense is to use a well-defined protocol that is secured against adversarial change. For example, when a recognizer or extractor is subjected to a judicial test, we would expect the court to call an expert witness. Bob should be required to give this witness a functional version of his purported embedder, recognizer, and extractor; he should explain their design and operating principles to the satisfaction of this witness; and he should explain how and why he chose his key and fingerprint. This requirement to justify a protocol design would preclude many protocol attacks. For example it would expose any attempt by Bob to make evidentiary claims on the basis of a constant-valued *extract'* function that reports his fingerprint  $w'$  is present in an arbitrary software program.

**1.5.6 Tamperproofing.** Alice might, in some applications, be able to *tamperproof* her program against attacks from Bob. An program is effectively tamperproofed if the adversary is unable to modify the fingerprinted program without fatal damage to its use-value. More formally, we would say a program  $P_w$  has been tamperproofed if Alice has transformed it into  $P'_w$  in such a way that the set of semantics-preserving transformations available to Bob for  $P'_w$  is smaller than the set available to Bob for transforming the untamperproofed  $P_w$ . In Section 2, we will see that some types of software fingerprinting techniques are more amenable to tamperproofing than others.

## 1.6 Security Properties

When the field of software fingerprinting is fully mature, we would have mathematically-sound proofs, similar to those currently expected of cryptographic systems, for important security properties of fingerprinting systems. As with cryptographic proofs, fingerprinting proofs will be based on assumptions about the adversary's capabilities. In particular, a secure system generally includes one or more "black boxes" for storing and using secret keys. The security of such systems rests critically on an assumption that the adversary is unable to observe or control operations "inside" a black box.

However, no mathematically sound security proofs for software fingerprinting systems are yet in existence, essentially for the same reason that so-called "white-box cryptography" is currently infeasible. We do not yet know how to construct a truly secure black box entirely out of software. Constructing such boxes will require significant advances in the theory and practice of program transformation and semantic analysis, by automated and semi-automated methods, in an adversarial model.

Anderson and Peticolas [1998] described a fundamental problem of software fingerprinting in the following way. "[A]n active warden can completely block the stego channel. For example, if (a) his model of the communication at least



as good as the prisoners’ [and] (b) the covertext information separates cleanly from the covert information, then [the warden] can replace the latter with noise. This is the case of a software pirate who has a better code mangler than the software author.”

We cannot hope to prove a software fingerprinting system is secure against Anderson’s active-warden attack, except by introducing limits to the warden’s analytic powers. However, after we introduce limits, we can argue security. For example, if the warden is not a human but is, instead, a computer program designed based on any known technique in program transformation and semantic analysis, then this article’s dynamic software fingerprinting system can secure complex programs against the active-warden attack. See Section 4 of this article, where we argue that the second step of Anderson’s attack is infeasible. Below, we sketch the main lines of this argument.

Anderson’s active-warden attack can be viewed as an attempt to rewrite the fingerprinted program in a “clean room”, with reference only to its semantic description. However, suitably precise and complete semantic descriptions of large software objects are generally unavailable to attackers, because even the designers of such systems do not often (if ever) have such descriptions. A fundamental difficulty is the nonexistence of general-purpose automated tools for extracting a complete semantic description from a compiled program. Lacking a complete semantic description, an automated active warden will be unable to build a clean-room version of the fingerprinted program, from which all possibly fingerprinted data structures have been deleted.

### 1.7 Practical Considerations

In addition to the data-rate and security issues described in the previous section, we must consider some other issues in any practical fingerprinting scheme. We tentatively classify these issues under two headings, and indicate some sample questions under each heading.

*Cover Program.* Will the program be distributed in a typed architecture-neutral virtual machine code, or in an untyped native binary code? Does the cover program execute mostly floating-point operations, or does it execute operations that are more typical of a user-interface program?

*Logistics.* How do we generate and distribute keys to our fingerprint detectors? How do we generate and distribute fingerprinted programs, and how do we handle debugging issues arising in these?

This article will sidestep such complications, by making simplifying assumptions which will allow us to argue the feasibility of secure dynamic fingerprinting for a class of typical Java programs that can be found by a web-search. However, we warn readers that the “devil is in the details” for fingerprinting system design, as with any other real-world design problem. For example, an adversary with sufficient knowledge of the covertext class of programs can use this knowledge to mount an effective attack against the fingerprinting system. Such attacks may be as simple as an exhaustive search for an unfingerprinted program with the same functionality (over a finite but sufficiently large testset

of possible inputs) as a fingerprinted program. Thus, the generic security analyses of this article do *not*, and cannot, supplant the need for a specific security analysis of any application of a fingerprinting system designed along the lines suggested in this article.

In this article, we will assume that Alice’s object  $\mathcal{O}$  is a program distributed to Bob as a collection of Java class files. As we shall see, fingerprinting Java class files is at the same time easier and harder than fingerprinting stripped native object code. It is *harder* because class files are simple for an adversary to decompile [Proebsting and Watterson 1997] and analyze. It is *easier* because Java’s strong typing allows us to rely on the integrity of heap-allocated data structures.

Although we have defined a wide range of possible attacks in this section, in our subsequent security analyses we assume a threat-model consisting of the distortive attacks that an adversary might construct from widely-known *semantics-preserving* code transformations. This assumption is similar in spirit to the use of the StirMark [Petitcolas 2004] benchmark to evaluate the resilience of an image fingerprint. Our threat-model transformations consist of the most common *translations* (compilation, decompilation, and binary translation Compaq [2004]), *optimizations*, *compressions* [Debray et al. 2000], and *obfuscations* [Collberg et al. 1998a,b, 1997].

Under these assumptions, we will examine various software fingerprinting techniques and attempt to answer the following questions:

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

## 2. TECHNIQUES FOR SOFTWARE FINGERPRINTING

*Static Fingerprints* 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 Fingerprints* 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 fingerprints are commonly embedded in redundant bits, bits which we cannot detect due to the imperfection of our human perception. *Static Code Fingerprints* 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 fingerprinting bit could then be encoded in whether  $S_1$  and  $S_2$  are in lexicographic order or not.

*Dynamic Fingerprints* are extracted from a program's execution state rather than from the program code or data itself. They were first introduced in Collberg and Thomborson [1999]. The idea is that in order to extract a fingerprint from an application it is run with a special input sequence  $\mathcal{I} = \mathcal{I}_1 \cdot \dots \cdot \mathcal{I}_k$ , which makes it enter a state which represents the fingerprint.  $\mathcal{I}$  can be thought of as a secret key known only to the owner of the software and unlikely to occur naturally.

Dynamic fingerprinting methods differ in which part of the program state the fingerprint is stored, and in the way it is extracted. When the special input sequence is entered, *Dynamic Easter Egg Fingerprints* perform some action that is immediately perceptible by the user, making fingerprint 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 Fingerprints* are extracted from the trace (either instructions or addresses, or both) of the program as it is being run with a particular input  $\mathcal{I}$ . The fingerprint is extracted by monitoring some (possibly statistical) property of the address trace and/or the sequence of operators executed.

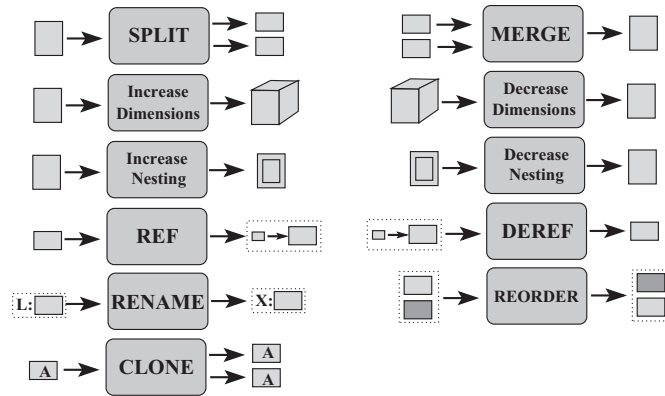
*Dynamic Data Structure Fingerprints* are extracted by examining the current values held in the fingerprinted program's variables, after the end of the special input sequence has been reached. This can be done using either a dedicated fingerprint extraction routine which is linked in with the executing program, or by running the program under a debugger.

## 2.1 Attacks Against Software Fingerprints

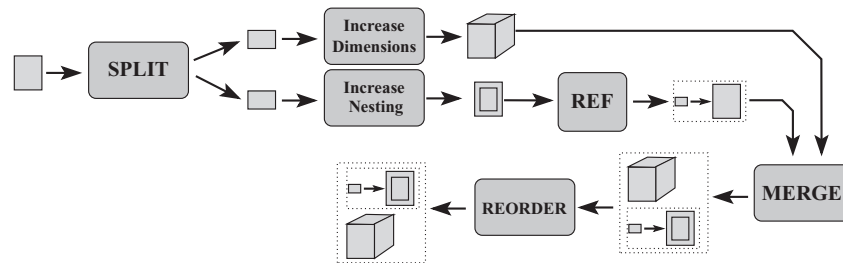
No known software fingerprinting method is completely immune to attack. In the worst case an adversary can study the input/output behavior of the fingerprinted application and completely rewrite it, sans the mark. We do not consider this a reasonable attack. We would, however, want the fingerprint to survive attacks by *translation*, *optimization*, and *obfuscation*, since tools that perform such operations are readily available [Debray et al. 2001a,b; Compag 2004; Nystrom 2004; Collberg et al. 2003b] and can be used by even the most unsophisticated attacker.

Static data fingerprints 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 that are then scattered over the executable. This makes fingerprint extraction nearly impossible. A more sophisticated de-fingerprinting attack would convert all static data into a *program* that produces the data [Collberg et al. 1998a].

Many code obfuscation techniques [Collberg et al. 1998a,b] will successfully thwart the extraction of code fingerprints. Since software is such a fluid medium it is easy to devise transformations which will destroy just about any structure of a program. Figure 1(a) shows some of the basic code transformations on which code obfuscations can be built:



(a) Basic code obfuscation transformations.



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

Fig. 1. Obfuscation attacks against software fingerprints.

- 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.
- 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 1(b).

Figure 2 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 Ahpah [2005] decompiler. The transformations applied were:

- (1) *Boolean splitting* (a Boolean variable is split into two small integer variables);
- (2) *basic block splitting* (a bogus branch protected by an *opaquely false predicate* [Collberg et al. 1998b]  $((q + q * q) \bmod 2) \neq 0$  is inserted);
- (3) *string encoding* (the string "Answer:" is encoded into an unintelligible string that 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 a nontrivial amount of overhead. It may therefore well be the case that while a certain sequence of transformations would obliterate a particular fingerprint, the resulting defingerprinted program would be too slow or too large to have any value to the attacker. The goal of software fingerprinting research is to design marking algorithms that will be robust against semantics-preserving transformations that add an *acceptable* amount of overhead, that is, without slowing the program down (or increasing its size) to the point that it is unsaleable by the pirate attacker. A similar situation arises in robust image fingerprinting, where an attacker would be uninterested in obliterating a fingerprint by blurring the image to the point that it is apparently damaged. The borderline of acceptability is a heuristic, not an exact calculation, and will depend on the performance sensitivity of the application. In lightweight applications, a 100x slowdown might be acceptable. However as little as a 2% slowdown might be completely unacceptable if the application is running very near some hard realtime limit. As a rule of thumb, we would expect an attacked application to be unacceptable if it becomes "heavy" (consuming more than 50% of some resource such as CPU cycles) for its target computing platform, in the cases where the unattacked application was "light" (consuming less than 25% of that resource). If the unattacked application consumes more than 25% of some critical resource, then its attacker probably won't accept a doubling of its consumption. As noted in Section 3.2, an acceptability criterion is also necessary in an implementation of a fingerprint embedder, which must not introduce a slowdown that is unacceptable to the defender of the fingerprint.

Classic *Easter egg fingerprints* are revealed to the end-user if a special input sequence is entered. The main problem with this class of fingerprints is that they seem to be easy to locate. There are even web-site repositories of such fingerprints [Nagy-Farkas 2004]. Unless the effects of the Easter egg are really

```

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));
    }
}

```



```

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());
}
}

```

Fig. 2. Obfuscation example.

subtle (in which case, it will be hard to argue that they indeed constitute a fingerprint and are not the consequence of bugs or random programmer choices), it is often immediately clear when a fingerprint has been found. Once the right input sequence has been discovered, standard debugging techniques will allow us to trace the location of the fingerprint in the executable and then remove or disable it completely.

*Data structure fingerprints* have some nice properties. In particular, since no output is ever produced, it is not immediately evident to an adversary when the special input sequence  $\mathcal{I}$  has been entered. This is in contrast to Easter egg fingerprints, 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 fingerprinted using a data structure fingerprint (it is linked in during fingerprint extraction), there is little information in the executable itself as to where the fingerprint may be located.

## 2.2 Software Fingerprinting Algorithms

Many simple fingerprinting algorithms have been based on reordering language constructs to embed the fingerprint. The first such published static code fingerprinting algorithm is due to Davidson and Myhrvold [1996b]. The idea is to embed a fingerprint by rearranging the order in which the basic blocks of a control-flow graph (CFG) are laid out in the executable. See Figure 5(a), which shows how a fingerprint 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 [1998] propose to embed the fingerprint in the register allocation of a program. Like all software fingerprinting algorithms based on renaming structures of the program this algorithm is very fragile; if Alice can rename a program structure to embed the mark, then Bob can rename it to remove the mark. Fingerprints typically do not even survive decompiling and then recompiling the program. This algorithm also suffers from a low bit-rate [Myles and Collberg 2003].

Stern et al. [1999] present an algorithm that uses a spread-spectrum technique to embed the fingerprint. The algorithm embeds the fingerprint 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 [Sahoo and Collberg 2004]. See Figure 5(b).

Moskowitz and Cooperman [1996] describe a data fingerprinting method in which the fingerprint is embedded in an image (or other digital media such as audio or video) using one of the many media fingerprinting algorithms. This image is then stored in the static data section of the program. See Figure 5(c).

Arboit’s algorithm [Arboit 2002] embeds the fingerprint 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 [Monden et al. 1998; 2000] 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 fingerprint is encoded steganographically in the opcodes and operands of the static code in the bogus method.

Pieprzyk [1999] describes a digital-rights management scheme for software, with two alternative methods for fingerprinting. The first alternative is to embed the fingerprint signal in the choice of synonyms for short sequences of instructions, similarly to Stern et al. [1999]. Pieprzyk notes that this method is weak against adversaries who make a “random selection of variants for each instruction.”

Venkatesan’s algorithm [Venkatesan et al. 2001] constructs a CFG whose topology embeds the fingerprint number. This fingerprint CFG is attached to an existing CFG by adding bogus control-flow edges using opaque predicates. In order to be able to extract the fingerprint, basic blocks that belong to the fingerprint graph are *marked*. See Figure 5(d).

Cousot and Cousot [2004] describe a fingerprinting process, in which the fingerprint is recognizable by a static analysis of program semantics in a secret model. The embedding key is a series  $[n_1, n_2, \dots, n_i]$  of relatively prime positive integers. The fingerprint  $c$  is any integer in the range  $0.. \prod_i n_i - 1$ . It is embedded as a series of residues  $c_i = c \bmod n_i$ , computed in loops, where these  $c_i$  are constant in the secret semantic model used for fingerprint 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 fingerprint recognition process is essentially the static analysis done by an optimizing compiler to recognize constant assignments in loops. This is a promising method in static data fingerprinting, however an attacker may be able to recognize the fingerprint 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 fingerprinting method of Cousot and Cousot would be nonstealthy 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, nonstandard semantics for commonly occurring code sequences in Java. The secret detection semantics for the Cousot and Cousot fingerprint must, however, remain simple if it is to be persuasive evidence in favor of the embedded fingerprint. Otherwise, the attacker may construct an alternative secret semantics that fraudulently detects an arbitrary, nonexistent fingerprint.

### 2.3 The CT Algorithm

The CT software fingerprinting algorithm is the primary focus of this article. It is a *dynamic* algorithm, that is, rather than embedding the fingerprint directly



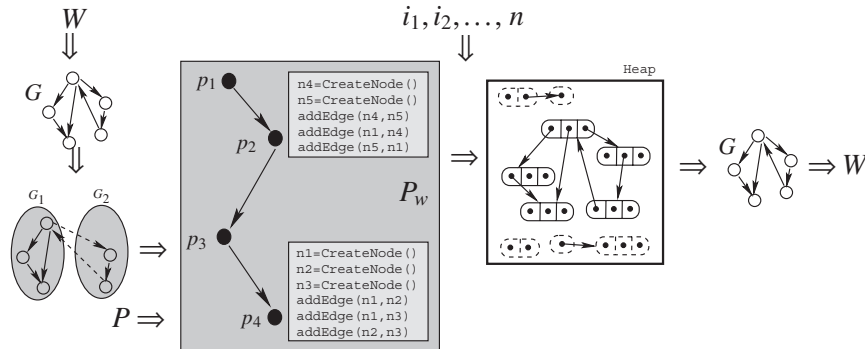


Fig. 3. Overview of the CT algorithm.

in the *code* of the application, code is embedded that *builds* the fingerprint at runtime. The algorithm requires its user to select a secret key that is sufficient to extract the fingerprint. The key is a sequence of legal inputs  $I_0, I_1, \dots$ , to the application.

The embedding and extraction processes are illustrated in Figure 3. The fingerprint number  $W$  is embedded in the topology of a graph  $G$ ; the graph is split into several components  $G_1, G_2, \dots$ ; 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  $I_0, I_1, \dots$ . During extraction the fingerprinted application is run with  $I_0, I_1, \dots$  as input, the fingerprint graph gets built on the heap, the graph is extracted and the fingerprint 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 [Ghiya and Hendren 1996; Ramalingam 1994]. Thus, it would be difficult for an attacker to statically analyze a fingerprinted program to look for code that builds a fingerprint 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 (which would be easy to spot were it embedded in only one place) 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 fingerprints. Fourth, and finally, since the fingerprint is data rather than code it should be easier to tamperproof. If the fingerprint 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 fingerprints for which tamperproofing requires the code segment of the executable to be examined. This is difficult to do stealthily. In Java, for example, it is not possible to write code that examines the code segment of the running program, in order to determine if a fingerprint code has been tampered with.

```

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

```



```

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;
}

```

Fig. 4. Example of a trivial class before and after being fingerprinted with the CT algorithm.

Figure 4 shows a simple example of what a program may look like after having been fingerprinted. The original program `Simple` is modified into `Simple_W` such that when run with the secret input argument "World" the fingerprint 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. [2000] present a dynamic fingerprinter based on the CT algorithm. In this simplified implementation, the fingerprint is not dependent on a key input sequence, but is constructed unconditionally. The fingerprint value is a data structure representing a graph. Pointers into a decoy structure of similar type to the fingerprint graph were used in opaque predicates for tamperproofing. The idea is that Bob cannot simply remove all "fingerprint-like" graph structures from this tamperproofed program, because this will invalidate some opaque predicates on the decoy structure, and these predicates are required for program correctness. The CT algorithm was found to be practical

and robust, in Palsberg’s experimentation. We compare their implementation to ours in Section 8.2.

## 2.4 SandMark

SandMark [Collberg et al. 2003b] 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 fingerprinting, and tamperproofing. SandMark currently contains implementations of forty code obfuscation algorithms and fourteen fingerprinting algorithms. It also supports various reverse engineering tools such as a slicer, a bytecode differ [Baker and Manber 1998], and a bytecode viewer. These can be used to aid *manual attacks* against software protection algorithms. A number of *software complexity metrics* [Harrison and Magel 1981; McCabe 1976; Munson and Kohshgoftaar 1993; Oviedo 1980; Halstead 1977; Henry and Kafura 1981] are provided for measuring the effect of code obfuscation and fingerprinting 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.

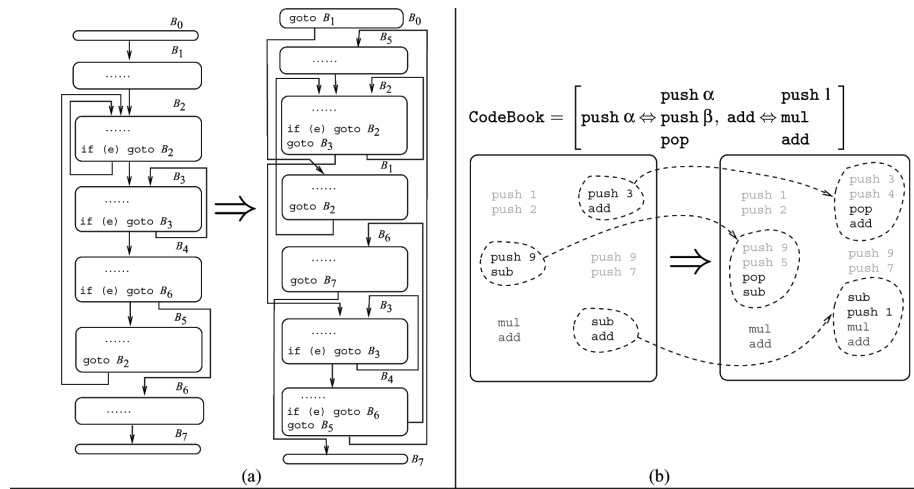
SandMark is designed using a *plug-in* style architecture and supports a number of useful static analyses (class hierarchy, control-flow, call graph, def-use, stack-simulation, liveness, etc.) simplifying the development of new software protection algorithms. SandMark relies on BCEL [2004] for bytecode editing, DynamicJava [Dyn 2004] for scripting, and BLOAT [Nystrom 2004] for code optimization. SandMark is currently approximately 120,000 lines of Java code of which approximately 10,000 lines comprises the CT implementation. The source code can be downloaded from `sandmark.cs.arizona.edu`.

## 2.5 Tamperproofing Static Fingerprints


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 [Collberg et al. 1998a], various forms of loop transformations [Bacon et al. 1994] and code motion are all well-known optimization techniques that will easily destroy static code fingerprints. Thus, static code fingerprints must be tamperproofed to be secure against adversaries who modify the fingerprinted code.

Moskowitz and Cooperman [1996] describe how to tamperproof their static-data software fingerprinting method, which embeds the fingerprint in an image included in a static data area of the application. The basic idea is to embed a fingerprint that evaluates to an “essential” piece of code. See Figure 5(c). This code is occasionally extracted and executed, making the program fail if an adversary tampers with the image to destroy or modify its fingerprint. The adversary might make use of the StirMark Petitcolas [2004] suite, which contains a number of image transforms selected for their fingerprint-destroying properties.

The Moskowitz tamperproofing process must emit sequences of instructions that decode, then execute, a fingerprint extracted from a static string. These



```

class Main {
    ...
    const Picture C = 
    ...
    Code R = Decode(C);
    Execute(R);
}
    
```

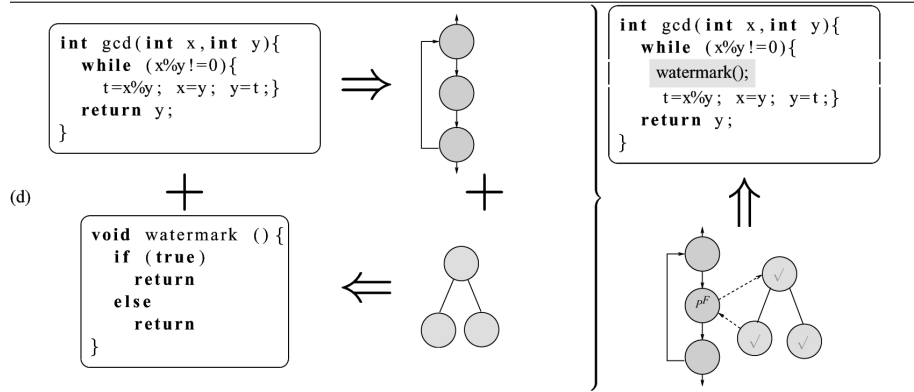


Fig. 5. Fingerprinting algorithms. (a) [Stern et al. 1999], (b) [Davidson and Myhrvold 1996a], (c) DICE [Moskowitz and Cooperman 1996], (d) [Venkatesan et al. 2001].

instructions must be cloaked or disguised somehow, otherwise a *detect* function could work by simple pattern-matching.

We conclude that Moskowitz’s tamperproofing method would not be effective in increasing either stealth or resilience against a highly skilled adversary, unless it is coupled with a strong obfuscation method to prevent a pattern-matching *detect* followed by an automated *attack* transformation in which the fingerprinted static string is replaced by its decoded version. Another method

for tamperproofing involves introspection: a code might “watch itself” or its data structures for signs of adversarial change [Horne et al. 2001; Chang and Atallah 2001].

Software obfuscation is a potent method for tamperproofing software fingerprints. As our focus in this article is on software fingerprinting, we limit our discussion of obfuscation to those aspects that bear directly on software fingerprinting. For example, as mentioned in our literature survey above, Arboit’s software fingerprint [Arboit 2002] is extracted by a pattern-match on its opaque predicates. If these opaque predicates were obfuscated, they would not be recognizable by the extractor. So this form of software fingerprinting can be attacked, but not defended, by software obfuscation. This illustrates an intrinsic problem with static fingerprints: stronger obfuscation methods will generally assist the adversary more than the defender. When Bob obfuscates code containing a static fingerprint, he can make it difficult, or even impossible, for Alice to recognize or extract her fingerprint. However, if Alice instead uses a dynamic software fingerprint, then she can obfuscate her fingerprinted code to defend against Bob’s code analysis. Strong obfuscation will make it more difficult for Bob to analyze the code well enough to design a distortive attack. We will return to this topic in Section 4, where we argue that a well-designed dynamic data fingerprint can be recognized or extracted even after the fingerprint has been attacked by obfuscations that affect the data structures of the fingerprint program.

### 3. THE CT ALGORITHM—BASIC IMPLEMENTATION

In this section, we discuss our implementation of the CT algorithm described in the previous section. The embedder must introduce code that *builds* the fingerprint at runtime. The algorithm assumes a secret key  $\mathcal{K}$  which is used to extract the fingerprint, where  $\mathcal{K}$  is a sequence of inputs  $I_0, I_1, \dots$  to the application. See Figure 3, where the fingerprint (a graph structure) is built by the fingerprinted application  $P_w$  when its execution takes path  $p_1, p_2, p_3, p_4$  as a result of special input  $I_0, I_1, \dots$ . Figure 4 shows a simple example of what a program may look like after having been fingerprinted.

In the SandMark implementation of CT, fingerprint embedding and extraction runs in several steps (see Figure 6):

*Annotation.* Before the fingerprint can be embedded the user must add *annotation* (or *mark*) points into the application to be fingerprinted. 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 fingerprinter locations in the code where (part of) a fingerprint-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

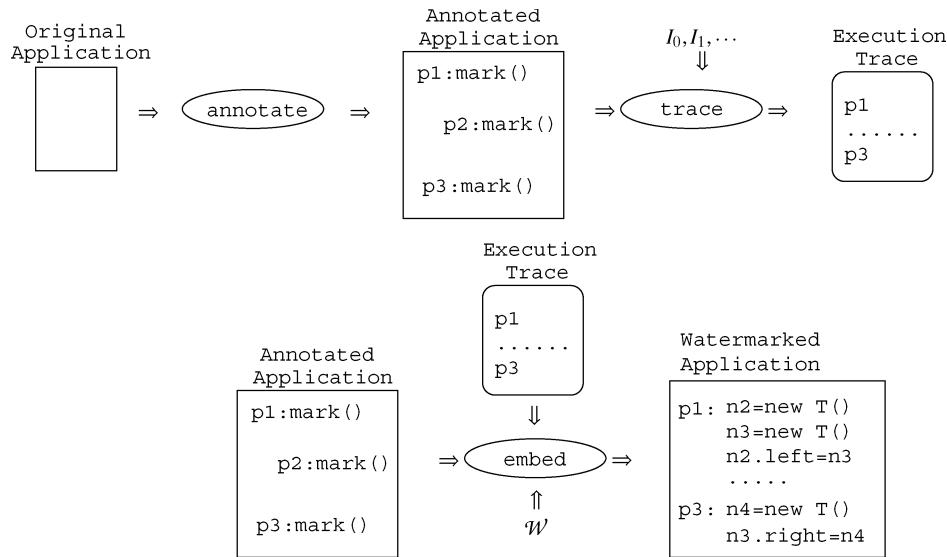


Fig. 6. Overview of how the CT algorithm fingerprints an application. First, the user adds *annotation points* (`mark()`-calls) to the application. These are locations where fingerprinting code may be inserted. Secondly, the application is run with a secret input sequence,  $I_0, I_1, \dots$  and the 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  $G_{\mathcal{W}}$  at runtime. The topology of  $G_{\mathcal{W}}$  embeds the fingerprint  $\mathcal{W}$ .

the application. These arguments help distinguish paths that were taken through the program for the special input sequence (that should trigger the building of the fingerprint graph), and those paths that were the result of normal (non-fingerprint recognition) execution.

**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,  $\mathcal{I}$ . During the run one or more annotation points will be hit. Some of these points will be the locations where fingerprint-building code will later be inserted.

**Embedding.** During the embedding stage the user enters a fingerprint, 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 fingerprint 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 fingerprint graph. When the last part of the input has been entered, the heap is examined for graphs that could potentially be fingerprint graphs. The graphs are decoded and the resulting fingerprint number is reported to the user.

We will next consider these tasks in detail.

### 3.1 Annotation

The CT fingerprint 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 nondeterministically.

In other words, `mark()`-calls should be added to locations where the resulting fingerprint 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 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 fingerprint graphs.

At the end of tracing run, we have gathered a list of *TracePoints* that 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;

```

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	$\langle \text{main}, Q \rangle$
Ⓑ	1	P	$L_0$	1	$\langle \text{main}, Q, P \rangle$
Ⓒ	2	Q	$L_1$	1	$\langle \text{main}, Q \rangle$
Ⓓ	$\emptyset$	aP	$L_2$	2	$\langle \text{aP} \rangle$
Ⓔ	3	Q	$L_1$	2	$\langle \text{aP}, Q \rangle$

Fig. 7. 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 17.

- (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 7 shows an example application and the corresponding list of trace points.

Not all annotation marks encountered during a tracing run can be used to build the fingerprint graph. There are various reasons for unacceptability, which we itemize below. An annotation point  $\langle \text{value}, \text{location} \rangle$  is unique if

- (1) there is exactly one trace point at  $\text{location}$ , or
- (2) there are multiple trace points at  $\text{location}$ , but they all have unique  $\text{values}$ .

An annotation point is acceptable only if it is unique; otherwise code embedded at that location would generate the fingerprint data structure many times, at an unnecessarily high cost in runtime and heap space. For example, consider



the following `mark()`-points:

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

$\langle 0, L_0 \rangle$  is unique, because it is the only `mark()`-point at location  $L_0$ . The marks  $\langle 10, L_2 \rangle$ ,  $\langle 11, L_2 \rangle$ ,  $\langle 12, L_2 \rangle$  are unique because the mark values (10, 11, 12) are unique.  $\langle 1, L_1 \rangle$  is not unique because there are two identical annotation values at this location. If we were to insert fingerprint-building code at this location we would need to introduce a state variable to distinguish between its first and second invocation. This introduces a stealthiness vulnerability, because an attacker may be able to distinguish the code manipulating these state variables from the code in the original, unwatermarked program. If the recognition is fairly accurate, the attacker will be able to construct a `detect()` function. If the recognition is extremely accurate, then the attacker will be able to build an effective `attack()` function that would distort or remove the watermark without affecting program correctness.

If there is just one value observed at a marked location, this `mark()`-call is said to be LOCATION-based; otherwise, it is VALUE-based. At a LOCATION-based `mark()`, the watermark-building code is unconditionally executed. By contrast, the watermark-building code at a VALUE-based `mark()` must be guarded by a predicate that tests the value. We consider a VALUE-based `mark()` to be marginally acceptable. Its guard predicate is not as unstealthy as the state-variable testing predicate that would be used at a nonunique `mark()` location, because the variable in a VALUE-based `mark` was present in the unwatermarked program, and the value being tested has been observed (in our trace) to occur in the original program. By contrast, the predicate guarding a nonunique `mark()` must be constructed out of whole cloth.

In addition to uniqueness, acceptable marks have three other properties. Acceptable marks must be reproducible, that is, they must recur reliably when the program is run again on the same input. Acceptable marks must be efficient, that is, the marked locations must not occur too many times on any input in the program author's test suite. Finally, acceptable marks should be specific, that is, the marked locations should not occur on any (or at least not on very many) inputs other than the special key input. If there are no acceptable marks, then the tracing step must be repeated with a new set of tracepoints and/or a new secret input.

With the exception of uniqueness, all our acceptability tests are heuristic. The threshold of acceptability, the total number of tracing runs, and the choice of inputs in the test suite, are all decisions that can be made appropriately in the context of a particular application for software fingerprinting. We discuss these heuristics very briefly below.

The threshold for acceptable specificity must be set to zero for any program which takes no input, for it can have no input-specific marks, the threshold for

specificity must be set to zero in such a case. A program that takes multiple inputs can have a highly specific fingerprint, however, for we can select a sequence of several marks that has been observed (in our traces) to occur reliably only for a single input sequence. As we increase the specificity of a fingerprint, we increase its stealth against some adversarial attacks, for example by someone who would be able to operate a fingerprint extractor on the fingerprinted program if they knew (or guessed) an input that would cause the fingerprint to be built.

The threshold for acceptable efficiency will depend on the performance constraints of the application. If an unfingerprinted application is already very close to the limits of acceptable performance, then any amount of fingerprinting overhead may be unacceptable. However, we believe that performance-critical applications are unlikely to be written in Java. Our experiments indicate that a static data rate of five codebytes per fingerprint bit is achievable. See Figure 21. If all of this fingerprint-building code is positioned at location-based marks, then it is only executed once. We conclude that the efficiency of our fingerprint, when placed at location-based marks, will be sufficient for all but the most performance-sensitive applications. However, when fingerprint code is placed at value-based marks, there will be additional runtime overheads due to the conditional branching that avoids building the fingerprint more than once. Thus marks are not acceptable in inner loops in performance-sensitive code.

The threshold for reproducibility will depend on one's tolerance for false-negative outputs from the fingerprint recognizer. Generally, we have not found reproducibility to be a problem in the codes we have fingerprinted and inspected, because these do not have race conditions or other real-time behavior that would make traces unreproducible.

### 3.3 Embedding

Once acceptable mark points have been selected for our application, we can start embedding the fingerprint. The input to this phase is a list of acceptable marks, a fingerprint  $\mathcal{W}$  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, replace one of the acceptable `mark()`-calls with a call to the  $M$ -method. The remaining `mark()`-calls are ignored.

The result is a new jar-file that, when executed with the special input sequence, will execute the method  $M$ . Consequently, it will build the fingerprint graph  $G$  on the heap.

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

### 3.4 Constructing the Graph

An ideal class of fingerprint 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 data rate so that a large fingerprint 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,
- (5) have some internal structure that makes the graph easy to tamper-proof.
- (6) have computationally feasible “unranking” and “ranking” functions Myrvold and Ruskey [2001] based on some enumeration Harary and Palmer [1973] of all graphs in the class, so that a fingerprinting integer can be converted to the corresponding graph during embedding or recognition, and so that a fingerprinting graph can be converted to the corresponding integer during an extraction, and
- (7) have some computationally feasible algorithm for graph isomorphism method, for use during recognition.

We do not expect to find a single class of graphs that simultaneously optimizes all these criteria. Instead, we are developing a library of algorithms for building fingerprint graphs with different sets of properties. Depending on a user’s particular requirements (high data rate, 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 fingerprint graphs illustrated in Figure 8.

**3.4.1 Permutation Encoding.** A fingerprint integer  $\mathcal{W}$ , in the range  $[0..n - 1]$ , may be represented by a permutation of the numbers  $\langle 0, \dots, n - 1 \rangle$ . Our embedder can use any convenient mapping of permutations onto integers. For example, Myrvold’s [Myrvold and Ruskey 2001] `unrank1` function would encode the fingerprint 180398 as the permutation  $\pi = \langle 9, 6, 5, 2, 3, 4, 0, 1, 7, 8 \rangle$ . The same unranking function must be used during a fingerprint recognition, and the corresponding ranking function, `rank1` in this case, must be used during a fingerprint extraction.

We use a singly-linked, circular list data structure to represent a permutation. We call this structure a Permutation Graph. See Figure 8(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  $\pi$ . It also has a list pointer referring to element  $(i + 1) \bmod n$ .

During the recognition phase there is a need to distinguish the list pointers from the data pointers. Our circular structure allows us to do so by looking for the longest simple cycle in the graph.

The dynamic data rate of a Permutation Graph is the number  $(\lg n!)$  of fingerprint bits represented by an  $n$ -element list, divided by the number of bytes

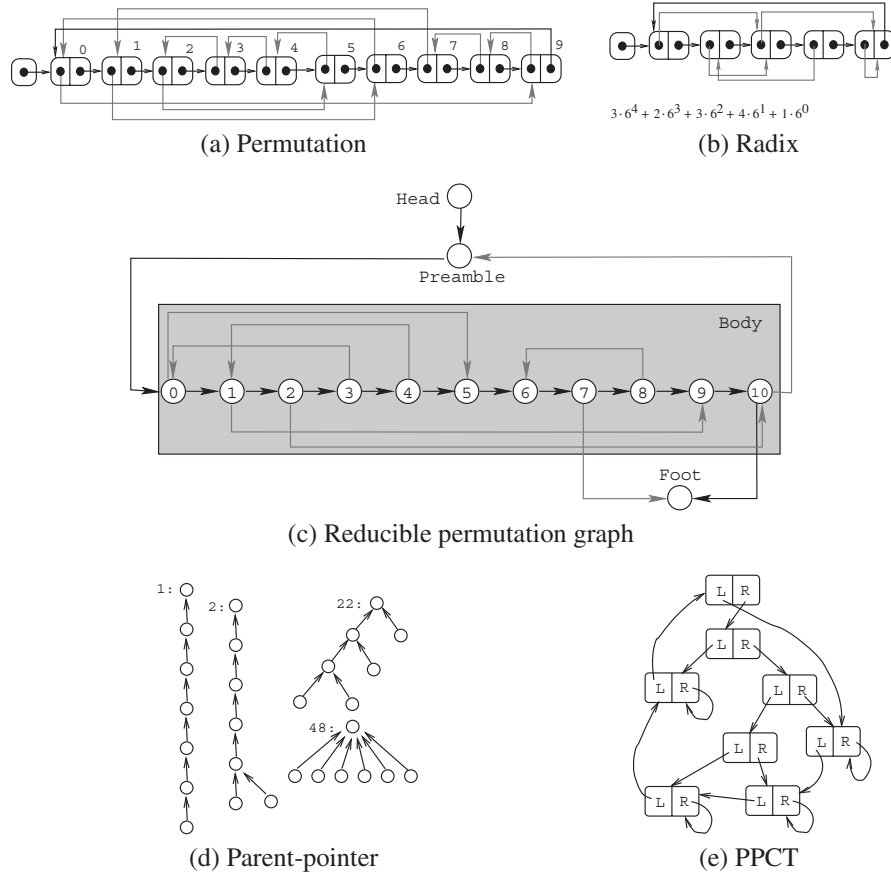


Fig. 8. Graph encodings. This figure is taken from Collberg et al. [2003a].

$(an + b)$  required to represent this list in computer memory. To embed a non-negative fingerprint 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  $w$ , so the dynamic data rate  $r(m) = (\lg m)/a + O(\lg m/\lg \lg m)$  is a slowly increasing function of  $m$ .

We can estimate the dynamic data rate by estimating the values of  $a$  and  $b$ . Both are small integers whose exact value will depend on implementation details. On a computer with 32-bit addresses, we expect  $a = 16$  bytes, because dynamic storage allocators generally use frame sizes that are a power of two, and each list cell has two pointers plus some overhead for the storage allocator. The list-overhead coefficient  $b$  is also expected to be small, perhaps 16 bytes. The exact value of  $b$  is irrelevant when computing data rates for large fingerprints, and it is not very important even when  $m$  is small. On the basis of these estimates, we conclude that Permutation Graphs have a dynamic data rate of  $\lg(n!)/(an + b) = 32.5/224 = 0.15$  hidden bits/overhead byte, when fingerprints of length  $m = 32.5$  bits are embedded in Permutation Graphs with

$n = 13$  list elements. This data rate rises slowly, in proportion to  $\lg m$ , for larger fingerprints. In Section 7, we confirm our estimates for  $a$  and  $b$  by analyzing experimental measurements on our implementation.

The static data rate is the number of fingerprint bits, divided by the number of bytes of code required to build the fingerprint. 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 data rate will have the same asymptotic form (with different constants) as the dynamic data rate.

Permutation Graph fingerprints 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 summarize these observations by saying that Permutation Graphs are single-error detecting.

**3.4.2 Radix Encoding.** Figure 8(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 fingerprints have higher data rate 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  $0 \cdots (n + 1)^n - 1$ . The list requires  $an + b$  words, so its dynamic data rate, as a function of  $n$ , is  $n \lg(n + 1)/(an + b) \approx (\lg n)/a$ . We expect the values of  $a$  and  $b$  to be similar for Radix Graphs and Permutation Graphs, and this is confirmed by our experimental data of Section 7.

If  $a = b = 16$  bytes, a Radix Graph of length  $n = 9$  will hide  $m = n \lg(n + 1) = 29.9$  bits of information in  $an + b = 160$  bytes of information, for a dynamic data rate of 0.19 bits/byte. As with Permutation Graphs, the dynamic data rate grows as  $\lg m$ . We note that Radix Graphs are more efficient than Permutation Graphs, because the former is the less restrictive organization, and therefore carries more information for any fixed number of list elements: every Permutation Graph is a Radix Graph, but not all Radix Graphs are Permutation Graphs.

We expect the static data rate of a Radix Graph, in any implementation, to be slightly better than the static data rate of a Permutation Graph in a similar implementation, because similar code will build both structures but the latter organization holds less hidden information per list element.

**3.4.3 Parent-Pointer Trees.** Many other classes of graphs, in addition to Permutation Graphs and Radix Graphs, allow efficient fingerprinting schemes. In Section 4, we discuss the data rates and error-correcting properties of the more complex graphs illustrated in Figures 8(e) and 8(c).

In this subsection, we consider the case where the fingerprinting graph  $G$  is an oriented “parent-pointer” tree, enumerable by the techniques described in Knuth [1997, Section 2.3.4.4]. See Figure 8(d) for an illustration of these trees.

We construct the rank or index  $w$  of a parent-pointer tree in the usual way, that is, by ordering the operations in an enumeration. When we enumerate the

$n$ -node parent-pointer trees in “largest subtree first” order, then the unbranching tree (a path of length  $m - 1$ ) is assigned index 1. Indices 2 through  $a_{n-1}$  are assigned to the other trees in which there is no branching at the root, that is, when there is a single subtree of size  $n - 1$  connected to the root node. Indices  $a_{n-1} + 1$  through  $a_{n-1} + a_{n-2}$  are 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 are 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  $1/\alpha \approx 2.956$ )  $a_n = c(1/\alpha)^{n-1}/n^{3/2} + \mathcal{O}((1/\alpha)^n/n^{5/2})$ . Thus we can encode an arbitrary 1024-bit integer  $w$  in a graphic fingerprint 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 unfingerprinted program, rather than being stored in separately allocated objects. We note that these fingerprinting data pointers might significantly slow de-allocations, and thereby hugely decrease the dynamic data rate, if they prevented some large, otherwise-dead, objects from being de-allocated until all its children objects in the fingerprint tree were also dead.

The dynamic data rate of a Parent Pointer Tree fingerprint might thus be as high as  $\lg(2.956)/4 \approx 0.4$  hidden bits per overhead byte, for large fingerprints such as the  $m = 1024$  case considered immediately above. The dynamic data rate would be much lower than this if the Parent Pointer Tree structure is separately allocated, rather than being carried as extra pointers in an already-existing data structure. The dynamic data rate should not change appreciably with  $m$ , although for small  $m$  it will be smaller than 0.4, because our asymptotic approximations have ignored some low-order terms that are important when  $m$  is small. Even so, Parent Pointer Trees may be preferable, from the point of view of dynamic data rate, to Radix Graphs and Permutation Graphs, for small and moderate  $m$ . This data-rate advantage will disappear for larger  $m$ . According to our estimates from the previous subsection, Radix Graphs of length  $n = 128$  are large enough to be more efficient than Parent Pointer Trees, for a Radix Graph of this size can hold  $m = 897$  bits of hidden information, at a dynamic data rate of 0.43.

Parent Pointer Trees have not yet been implemented in SandMark, so we have not yet conducted an experimental validation of the asymptotic analysis and model developed above.

### 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 fingerprint 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

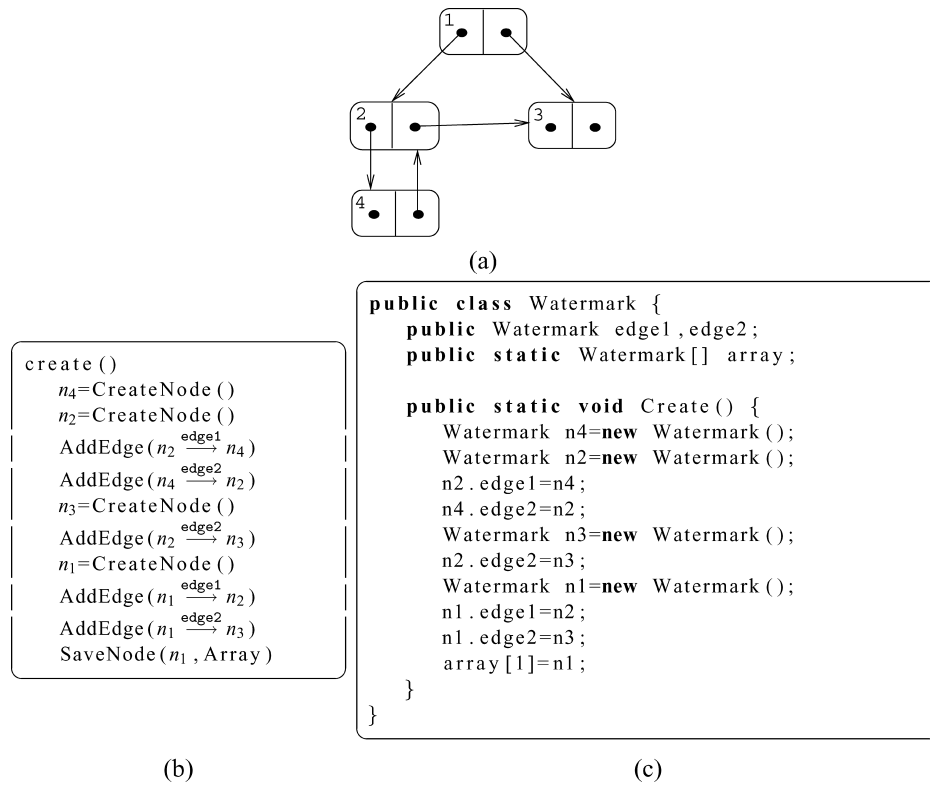


Fig. 9. (a) Shows a fingerprint graph, (b) the corresponding intermediate code, and (c) the resulting Java code.

for optimizing transformations. Similarly, our intermediate representation provides

- (1) retargetability, to allow future generation of code for other languages; and
- (2) transformability, that is, the ability to optimize or otherwise transform the intermediate code prior to generating Java code.

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

Given the graph in Figure 9(a) we would generate the intermediate code in Figure 9(b). Nodes are named  $n_1, n_2$ , etc. The  $n = \text{CreateNode}()$  instruction creates a new node  $n$ . The  $\text{AddEdge}(n \xrightarrow{\text{edge}} m)$  instruction adds an edge from node  $n$  to node  $m$ . Since the graphs are multi-graphs the out-edges are named  $\text{edge1}, \text{edge2}$ , etc. The instruction  $\text{SaveNode}(n, L)$  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  $m = \text{CreateNode}()$  and  $n = \text{CreateNode}()$  have both been generated.

The complete set of intermediate code instructions is given in Table I. 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 straightforward. We use the BCEL library to generate a bytecode class `Watermark`. From the intermediate code in Figure 9(b) we would generate the Java code in Figure 9(c).

The chosen `mark()`-location is replaced by a call to `Watermark.Create()`. A final obfuscation pass over the fingerprinted application will inline the call, rename the fingerprint 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

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

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

### 3.7 Extraction

To extract the fingerprint, the fingerprinted application is run as a subprocess under debugging, again using Java's JDI debugging framework. The user enters their secret input sequence  $I_0, I_1, \dots$  exactly as they did during the tracing phase. This causes the method `Watermark.Create()` to be executed and the fingerprint 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 fingerprint 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, when the secret key input sequence has been entered, the root node of the fingerprint graph will be one of the very last objects to have been added to the heap. Hence, a good strategy is to examine the heap objects in reverse allocation order. Of course, an adversary who is able to guess the correct key input (or even a partial key) when running our watermarked program can use the same strategy and look for a plausible root node near the



top of the heap. As noted in Section 1.5, we cannot expect to have a stealthy watermarking system unless the adversary is initially very uncertain about the keys and watermarks we are using. Under this assumption, the adversary will not learn very much more about the keys and watermarks from each program run they observe.

There is no support as yet, in JDI, for examining the heap after a program run. This makes it more difficult for us to build a watermark recognizer, and our recognizer is somewhat inefficient, but it also causes our adversary to spend more time when attacking our watermarks.

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

```

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. The additional 4-byte overhead for every allocated object might also be prohibitive under some circumstances.

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 10. The downside is a fairly substantial slowdown due to the overhead incurred by handling the breakpoints.

To extract the fingerprint graph, we consider each object on the circular buffer in reverse allocation order, extracting the reachable subgraph. Since every fingerprint graph has a root from which every node is reachable we are guaranteed to eventually find the graph. From the graph, the fingerprint number is extracted by the algorithm in Figure 11.

#### 4. IMPROVING RESILIENCE

To properly design and evaluate a fingerprinting algorithm it is essential to define a precise model of a realistic attack. This has been a failing of most previous work on software fingerprinting. In this article, we assume that the fingerprinted 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 fingerprint code. Rather, we want to protect the fingerprinted

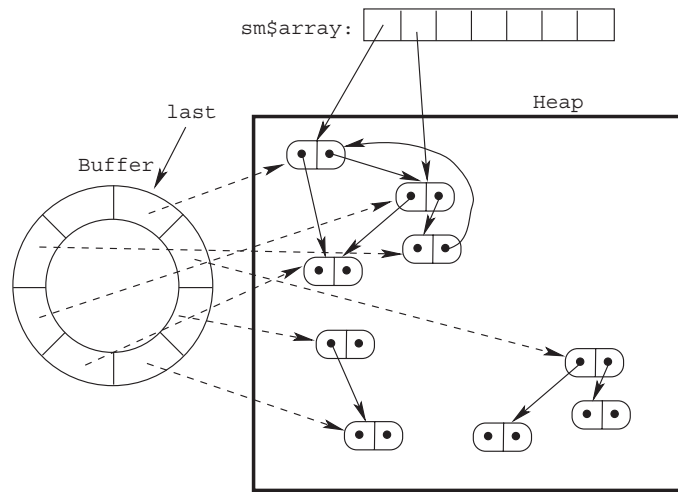


Fig. 10. A view of memory during extraction. A circular linked buffer holds the last allocated objects. The extractor examines the objects in reverse allocation order and extracts the subgraph reachable from each object. This is decoded into the fingerprint.

```

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  $[e_1, e_2, \dots, e_k]$ ;
    for each pair  $(e_i, e_j)$  of edges in  $[e_1, e_2, \dots, e_k]$  do
       $G' =$  subgraph of  $G$  with edges labeled  $(e_i, e_j)$ ;
      if fingerprint decoding  $w = C(G')$  succeeds then
        yield  $w$ 

```

Fig. 11. Fingerprint 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 subgraphs 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.

program against *class attacks*—the construction of automated methods of destroying the fingerprint. For example, if a particular code transformation can be shown to destroy a fingerprint, then it is an easy task for an adversary to construct an attack that will destroy *every* fingerprint in *every* program.

An appealing 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 that can obfuscate dynamic data, particularly for languages with typed object code, like Java. There are four types of obfuscating transformations that are dangerous to a dynamic software fingerprint. To confuse the extractor, an adversary can

- (1) add extra pointers to the nodes of linked structures (Figure 12(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 12(b));

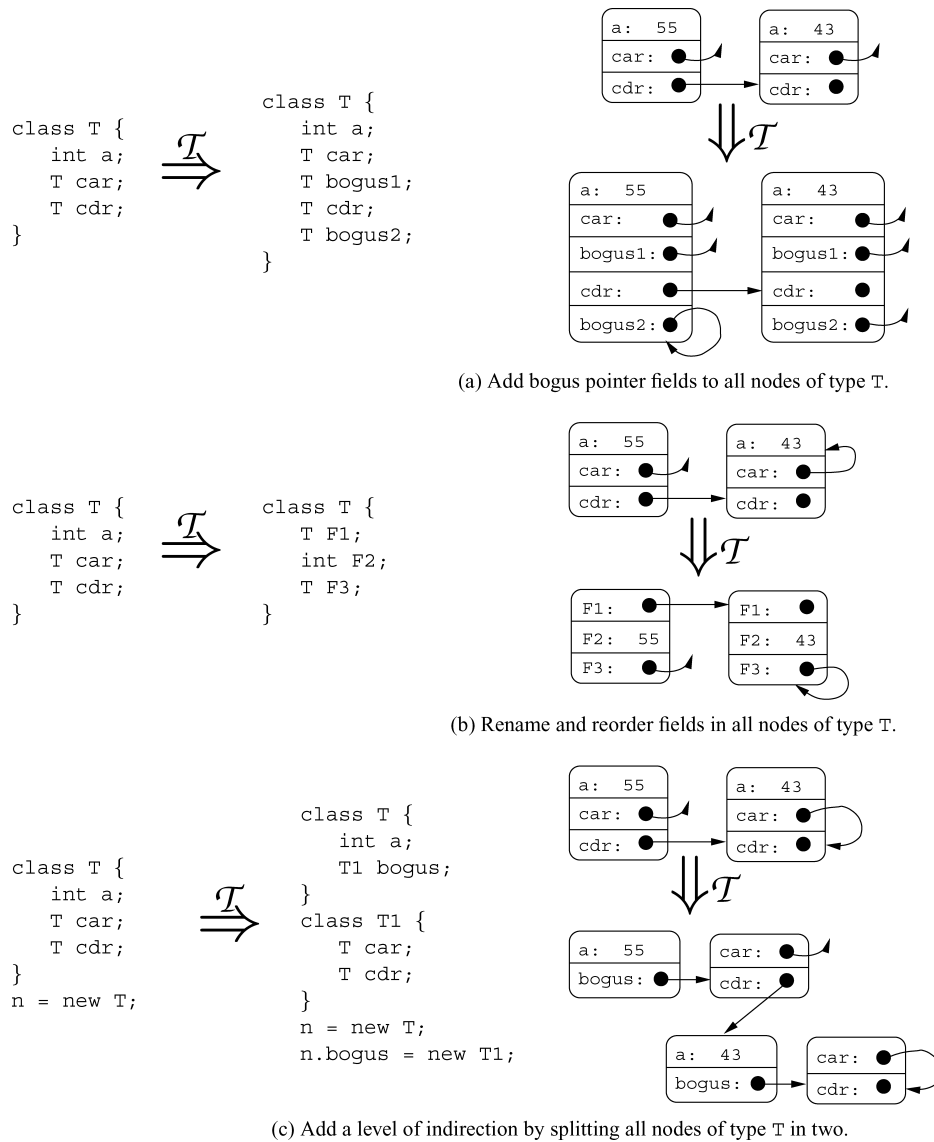


Fig. 12. Obfuscation attacks against graph-based fingerprints.

- (3) add levels of indirection, for example by splitting nodes into several linked parts (Figure 12(c));
- (4) add extra bogus nodes pointing into our graph, preventing us from finding the root.

Figure 13 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 defingerprinted program. For example, splitting a node costs 12 bytes per allocated

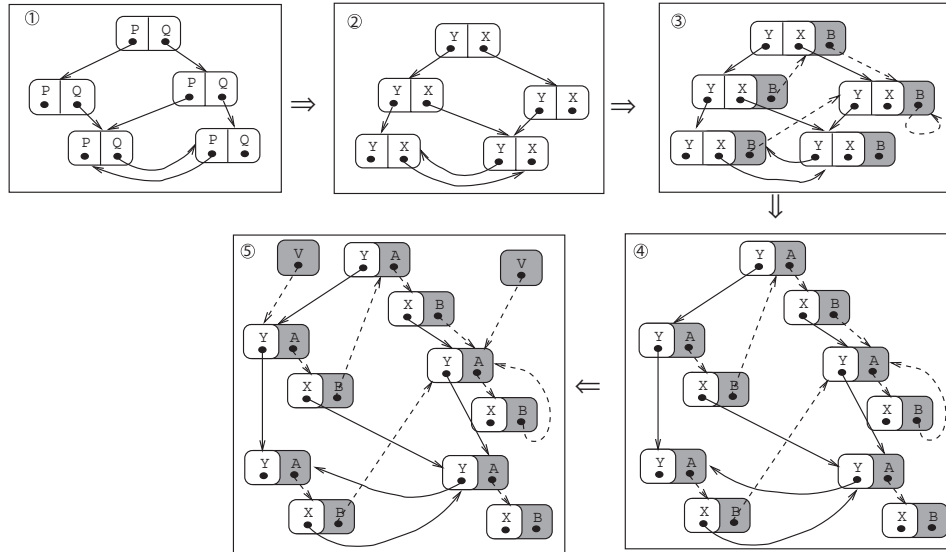


Fig. 13. Example obfuscation attack against the fingerprint graph in ①. The adversary renames and reorders node pointer fields (②), adds a bogus pointer field B (③), and splits nodes by adding a bogus pointer field A (④). Finally, in ⑤ bogus pointers into the graph obscure the root node.

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 fingerprint is hidden, he is going to have to apply the transformations uniformly over the *entire* program in order to be certain the fingerprint has been obliterated. In other words, programs with high allocation rate of dynamic storage are likely to be resilient to these types of attacks, since the de-fingerprinted program will have a much higher memory requirement than the original one.

#### 4.1 De-Fingerprinting by Field Reordering

A simple way to protect against field reordering is to consider every order of the fields during fingerprint extraction. This could, however, lead to an increased false positive rate, since a user graph might, under some reordering transformation, appear to have the same structure as our fingerprint graphs. A better approach is to choose an unlabeled class of graphs for which the order of outgoing edges do not affect the fingerprint 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 8(c) shows a *reducible permutation graph* (RPG) [Collberg et al. 2003a]. 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.

*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  $w$ th 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  $0.375 \lg n - 0.62$  bits per node, plus or minus approximately  $0.125 \lg n$  bits per node depending on the size of the preamble required for the RPG corresponding to the fingerprint  $w$ . 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 fingerprint 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 fingerprint  $w' \neq w$ .

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 8(e) shows an example of the class  $G_n$  of *planted plane cubic trees* (PPCT) on  $n$  leaf nodes  $v_1, v_2, \dots, v_n$ . PPCT graphs are enumerated in Goulden and Jackson [1983]. 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-addressable 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, efficiently, by tamper-detection code. The idea is to test the PPCT's planarity locally, for any internal node  $x$ , by confirming that the leftmost child of  $x$ 's right subtree is l-linked to the rightmost 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 fingerprint) 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 fingerprinted program.

Any tamper-response mechanisms invoked by a tamper-detection code will introduce a stealthiness vulnerability, as will the tamper-detection code itself. This vulnerability can be mitigated by introducing additional code that is very similar to the watermarking code but which is required for program correctness, as suggested by Nagra [2006].

#### 4.2 De-Fingerprinting by Node Splitting

Node-Splitting is an effective attack against our fingerprint graphs but one which will have serious detrimental effect on the performance of the de-fingerprinted program. Even so, Figure 14 shows another representation that, at the expense of a lower data rate, will increase a graph fingerprint'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 fingerprint 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 15.

#### 4.3 De-Fingerprinting 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 fingerprint against many types of attack, including the addition of bogus fields in the graph nodes. Assume that

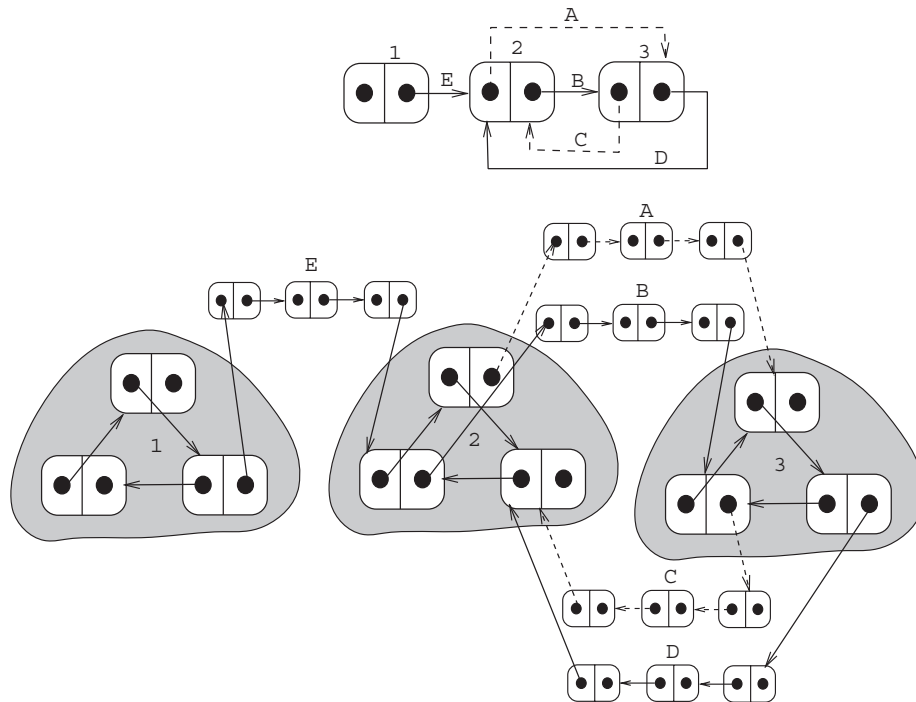


Fig. 14. The top-most graph shows a radix encoding of the value 3. The bottom-most graph shows how this encoding is made more resilient against node-split attacks by turning each node into a 3-cycle and each edge into a path of length three. For identification purposes, nodes have been labeled 1–3 and edges A–E in this figure. Edges on the spine are shown solid and edges encoding a radix digit are shown dashed.

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();
```

Unfortunately, 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 fingerprint pointers through reflection. For example, rather than `o.car=V`, we let car be represented by the first relevant pointer in

```

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;

```

Fig. 15. Algorithm to contract edges and cycles for cycled graphs.

the node 0:

```

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.

#### 4.4 Future Work: Manipulating Graphs

One of our main motivations for using dynamically built graph structures to represent the fingerprint is that the code that builds the fingerprint will be difficult to analyze. The reason is the inherent hardness of alias analysis. However, current alias analysis algorithms do very well with noncircular 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 [Ghiya and Hendren 1996].

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

### 5. INCREASING FINGERPRINT SIZE

While the operations by which the fingerprint graph is built (pointer assignments and the “new” bytecode for dynamic memory allocation) are stealthy in themselves, a large number of such operations concentrated to one place in the code would be likely to arouse suspicion. Furthermore, our fingerprint becomes more input-specific if it is built at several points along an execution path with input dependencies, rather than at a single point. For these reasons, we



Table I. Intermediate Code Instructions.

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

therefore split the graph  $G$  into several components  $G_0, G_1, \dots$  whose code is spread along the special execution path. There are several issues to consider:

- (1) The subgraphs should be of roughly equal size.<sup>1</sup>
- (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 by Kundu and Misra [1997] to partition the fingerprint graphs. Other algorithms which split the graph in connected and rooted components of equal size would do equally well. Given the graph on the left in Figure 16(a) our implementation would produce the two graph components  $G_2$  and  $G_4$  on the right. Figure 16(b) shows the generated intermediate code. The main intermediate code instructions are given in Table I. 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  $G_0, \dots, G_{i-1}$ . If there is an inter-subgraph edge  $m \xrightarrow{\text{edge}} n$  from  $G_k$  to  $G_i$  (i.e.,  $m$  is a node in  $G_k$  and  $n$  is in  $G_i$ ), we generate

- (1) 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  $n$ .

<sup>1</sup>For stealth reasons, it might be better if the components are of random size. The current implementation, however, splits in equal-size pieces.

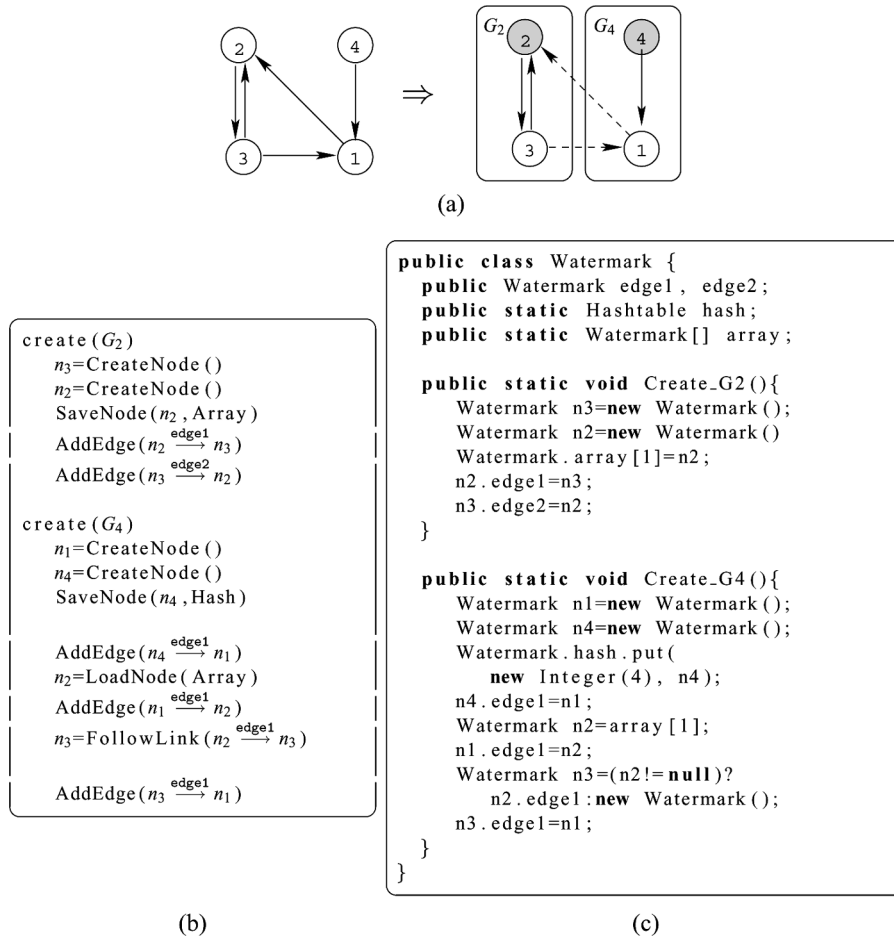


Fig. 16. (a) Shows a fingerprint graph that has been split into two components. Root nodes have been shaded and inter-component edges have been dashed. The graphs are identified by their root nodes. (b) Shows the intermediate code for each component and (c) the corresponding Java code.

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 II lists possible translations of the intermediate instructions and Figure 16(c) shows the Java class generated from the intermediate code in Figure 16(b). Method `Create_G2` builds subgraph  $G_2$  and `Create_G4` subgraph  $G_4$ . Note, 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 an extraction run (i.e., the input sequence is *not* exactly  $I_0, I_1, \dots$ ) then  $G_2$  may not have been created, and  $n_2$  may be `null`. Table II shows several ways to protect against null-pointer exceptions. It is useful to have a

Table II. Translation from Intermediate Code Instructions to Java

INSTRUCTION	JAVA
AddEdge( $n \xrightarrow{\text{edge}} m$ )	Generate $n.\text{edge} = m$
CreateNode( $n$ )	Generate Watermark $n = \text{new Watermark}()$
CreateStorage( $G, S$ )	Generate one of (1) <code>static java.util.Hashtable hash = new java.util.Hashtable();</code> (2) <code>static Watermark arr = new Watermark[m];</code> (3) <code>static java.util.Vector vec = new java.util.Vector(m); vec.setSize(m);</code> (4) <code>static Watermark n1,n2,...;</code> where $m$ is the number of nodes in the graph and $n1,n2,\dots$ are the root nodes of the subgraphs.
FollowLink( $n \xrightarrow{\text{edge}} m$ )	Generate <code>Watermark m = n.edge</code> or if $n$ can be null 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>
LoadNode( $n, S$ )	Generate one of (1) <code>Watermark n = (Watermark) Watermark.hash.get(new java.lang.Integer(k));</code> (2) <code>Watermark n = Watermark.arr[k - 1];</code> (3) <code>Watermark n = (Watermark) Watermark.vec.get(k - 1);</code> (4) <code>Watermark n = Watermark.nk</code> depending on how $n$ is stored. $k$ is $n$ 's node number.
SaveNode( $n, L$ )	Generate one of (1) <code>hash.put(new java.lang.Integer(k), n);</code> (2) <code>Watermark.arr[k - 1] = n;</code> (3) <code>Watermark.vec.set(k - 1, n);</code> (4) <code>Watermark.nk = n</code> depending on how $n$ is stored. $k$ is $n$ 's node number.

whole library of such protection mechanisms to prevent attacks by pattern matching.

## 5.2 Future Work: Splitting the Fingerprint Number

There is an alternative approach to increasing the size of the fingerprint that can be inserted in a given program. Rather than splitting the graph, we split the fingerprint number  $n$  into several (smaller) numbers  $n_0, n_1, \dots, n_k$  using the Chinese Remaindering theorem [Muratani 2001]. The numbers are encoded into graphs  $G_0, G_1, \dots, G_k$  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 fingerprint from automatic means of destruction, we would also like to protect it against manual attack whenever possible. Furthermore, we would also like our fingerprints to be as stealthy as possible, so that an adversary is unable to detect their presence with any degree of accuracy.

The goal of stealth is often linked to the goal of resilience, because an adversary's unkeyed detector for a dynamic software fingerprint may operate by pattern-matching or other analysis of the static representation of the fingerprinted code. Consider an adversary who develops an automated tool that identifies, with 90% assurance, a small amount (say 10%) of a fingerprinted code as the portion that contains fingerprint-building constructs. Such an adversary may then be enabled to manually search these regions of the decompiled program, correctly locating some or all of the fingerprint code, and successfully modifying or deleting the fingerprint.

Thus, it is very important to improve the stealth of a software fingerprint, even when one's main goal is to increase its resilience. In this section, we will consider three techniques for improving the stealth of our graph fingerprints.

### 6.1 Avoiding Global Variables

We have so far assumed that the roots of subgraphs are stored in static fields. In Figure 16(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 17 shows the trace forest generated from the trace points in Figure 7. 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, 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 fingerprinting code that the CT algorithm inserts into a program has approximately the distribution shown in Table III. 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 that—directly or indirectly—overrides a method in the Java standard library. For example, we cannot change the signature of `actionPerformed` in Figure 7. 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

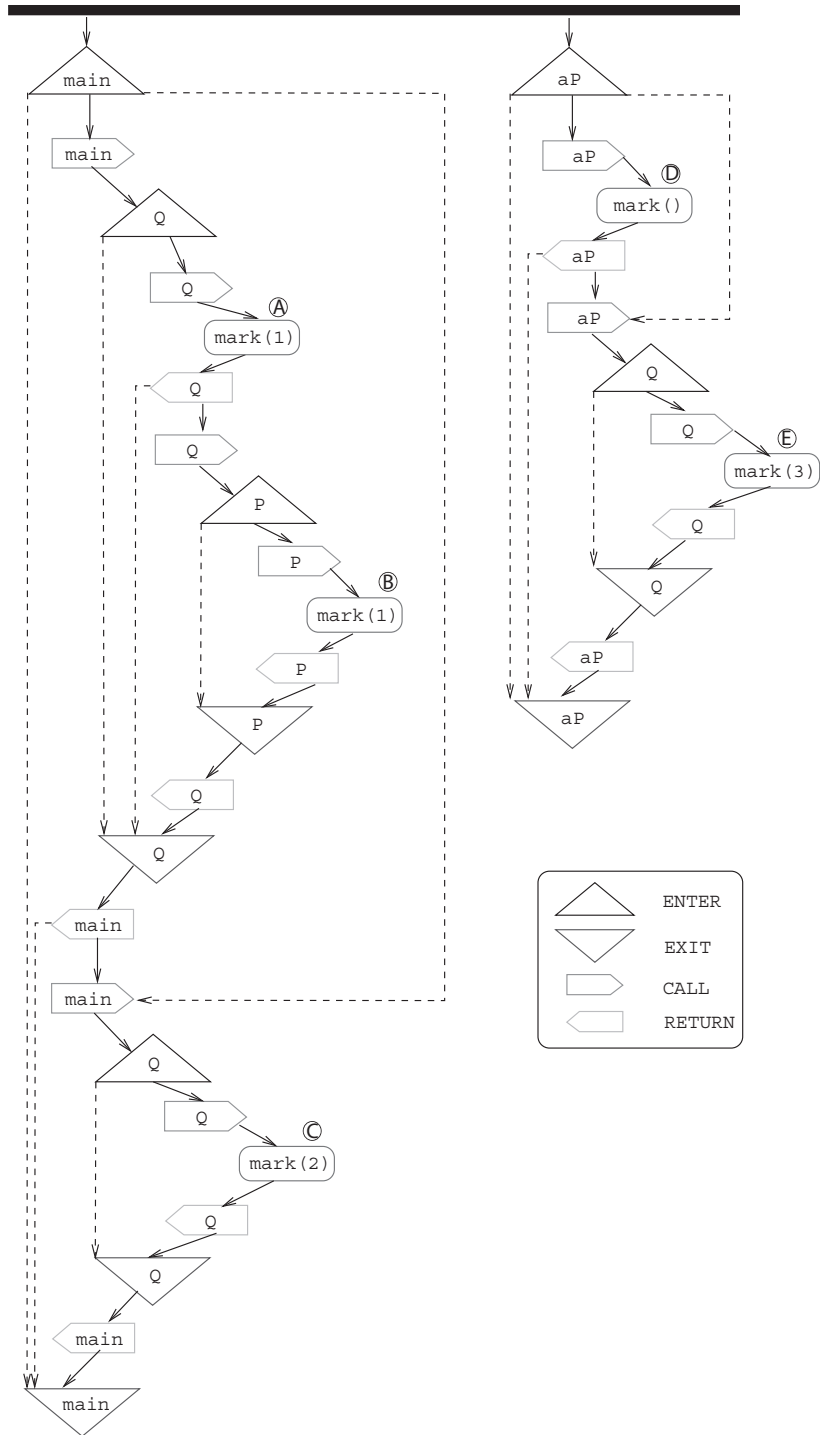


Fig. 17. The trace forest generated from the trace points in Figure 7.

Table III. The Distribution of Java Bytecode Instructions in Generated Graph-Building Code

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%

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 Static Collusive Attacks

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

SandMark also contains several *obfuscation executives* [Heffner and Collberg 2004], 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),

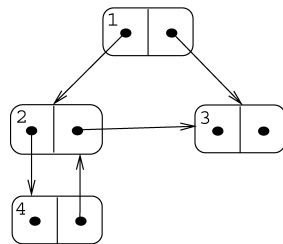
and a seed that initializes a random number generator. This allows us to obfuscate each differently fingerprinted version of a program using a different sequence of obfuscating transformations. As a result, collusive attacks become more difficult.

Against particularly powerful collusive adversaries, the obfuscation techniques of Chow et al. [2001] and Wang [2000] are likely to be very effective. These are both advanced forms of obfuscation by interpretation. In both techniques, the control flow of the program is encoded into a finite state machine, where, in Wang’s construction, the next-state function is computed dynamically. In Chow’s finite state machine, each block is either *functional* (i.e., originating from the original program) or a *glue* block which permutes register numbers. The idea is that many clones of functional blocks are strung together during interpretation by the glue blocks that ensure that each block manipulates the appropriate input and output registers. This obfuscation should effectively frustrate a static pattern-matching attack on local stealth, if each basic block in the original code appears in more than one of the functional blocks.

### 6.3 Hiding the Watermark Class

Figure 4 shows how a special class `Watermark` has been generated that is used to create nodes in the fingerprint graph. Obviously, this is not very stealthy. Instead, we search the application to be fingerprinted 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.

In situations where there is no appropriate user class, and adding new classes would be unstealthy, it may be possible to reuse classes from the Java library. For example, the graph from Figure 9 can be built using Java’s `LinkedList` class (which can build up arbitrary lists of lists), the `Event` class (which has two public `Object` fields `target` and `arg`), arrays of length two of `Object` type, or even combinations of such types:



```
import java.util.LinkedList;
import java.awt.Event;
...
LinkedList n4 = new LinkedList();
n4.add(null);
LinkedList n2 = new LinkedList();
n2.add(n4);
n4.add(n2);
Event n3 = new Event(null, 0, null);
n2.add(n3);
Object[] n1 = {n2, n3};
```

To be useable, a class  $C$  in the Java library needs to have at least two fields  $f_i$  of reference type  $t_i$  such that  $C$  is a subclass of  $t_i$  and the  $f_i$  are either public



or have both *setter* and *getter* methods. Unfortunately, in Java 1.4.2, we know of only 21 classes that fulfill these criteria, and, as a result, this method of increasing stealth is of limited usefulness.

#### 6.4 Future Work: Protecting against Dynamic Collusive Attacks

So far, research into software fingerprinting has focused on protecting against attacks based on static analysis. Much more powerful attacks are possible if we allow the adversary to execute the marked program. This gain in power is particularly apparent in collusive attacks.

Consider, for example, the following attack scenario. Bob buys two programs  $P_1$  and  $P_2$  from Alice, knowing that they carry different CT-style fingerprints. The programs have been heavily obfuscated, so a static pattern-matching attack yields Bob no information as argued in Section 6.3. Bob might then engage in a dynamic attack in which he runs both programs simultaneously, on the same input, synchronizing them by stopping whenever the programs execute a system call. He then forces  $P_1$  and  $P_2$  to perform garbage collections, and compares the remaining linked structures on the heaps. Bob will naturally assume that  $P_1$  and  $P_2$  will execute the same sequence of system calls, in the same order. When stopped at a particular call, these two fingerprinted programs should be in the same state. Thus, when Bob compares the two heaps, any CT-like structure that doesn't match a structure on the other heap is very likely to be a part of a CT-style fingerprint. Bob could then trace back through the code to discover where this structure was built. If this attack is successful, the fingerprint is not locally stealthy.

This scenario represents a very serious attack that will be difficult to defend against. One possible defense is for Alice to insert random bogus system calls that will prevent Bob from synchronizing the execution of the two programs. This will be hard to do for a typical Java program, since many calls have undesirable and difficult-to-cancel effects such as opening a window or writing to a file. Bob can choose any set of system calls for his dynamic analysis, so his list would probably include graphics and IO primitives, but exclude the collection classes and any other system call which have side-effects he thinks Alice might know how to cancel. So we are not optimistic that Alice can prevent Bob from synchronizing his executions of fingerprinted code at many system call sites.

However, we believe it should be possible for Alice to prevent Bob from learning very much from his synchronization points. The idea is to ensure that on most every path through the program bogus CT-style graph pieces are built. If there is a “sufficiently large number” of bogus live pieces on the heaps wherever Bob stops the two programs, it will be difficult for him to distinguish Alice's real from her bogus fingerprint structures. Unfortunately, for a fingerprinter to add such graph-building code automatically to an arbitrary program is non-trivial—it has to ensure that any bogus pieces are live long enough that Bob's garbage collection won't remove them, but not so long that the heap becomes too full. This obfuscation can be seen as advanced form of Palsberg's tamper-proofing [Palsberg et al. 2000], and has been explored in some of our other publications. See, for example Thomborson et al. [2004].

## 6.5 Future Work: Avoiding Weak Cuts

An important part of Venkatesan’s fingerprinting algorithm [Venkatesan et al. 2001] is to bind the fingerprint 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 fingerprint 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. [2001] 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 fingerprint 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 fingerprint graph as input to compute the value 3, perhaps as a function of the path length from the root to a leaf [Thomborson et al. 2004].

## 7. EVALUATION

There is no widely accepted method for measuring the strength of a software fingerprinting 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 straightforward, measuring stealth and resilience is much harder, since this requires a *model* of how an adversary might measure and transform the fingerprinted program. Future work in this area will have to develop and validate such models.

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. Our main definition was sketched in the Introduction of this article, and is stated formally in Eq. (2) below. Informally, we say a fingerprinting scheme has a high degree of *steganographic stealth* if, when given access to a fingerprinting algorithm  $A$  and a fingerprinted program  $P_w$ , an adversary cannot determine if  $P_w$  has been fingerprinted with  $A$  or not.

We developed an alternative definition of stealth informally in the previous section, when we discussed the interaction of stealth with resilience. We formalize this definition in Eq. (1) below. Informally, an algorithm exhibits a high degree of *local stealth* if, given access to a fingerprinting algorithm  $A$  and a program  $P_w$  known to be fingerprinted with  $A$ , an adversary cannot determine the location of  $w$  within  $P_w$ .

Further refinements are conceivable. We might speak of *static local stealth* if the adversary cannot identify the static code that generates a dynamic software fingerprint. *Dynamic local stealth* might imply that the adversary is unable to distinguish a program’s unfingerprinted data structures from the fingerprinted data structures it builds when it is exhibiting its fingerprint. However, we doubt that this distinction will be helpful in practice, as either possible form of local unstealthiness probably implies the other form, if the adversary is at all competent. For the sake of clarity and simplicity, then, the phrase “location of  $w$ ” in our definition of local stealth should be understood to mean the “location of the code that builds  $w$ ” in the case of a dynamic software fingerprint.

Any measure of stealth must be conditioned on, at least implicitly, on some universe  $U$  of unfingerprinted programs. Any adversary must make some assumptions about  $U$  in order to distinguish some fingerprinted version of a program  $P \in U$  from the unfingerprinted original version of this program  $P$ . Furthermore, any well-designed application of a fingerprinting scheme will adjust the embedding algorithm so that it is as stealthy as possible within the range  $U$  of its intended application. For example, a fingerprinting algorithm which encodes the fingerprint in the number of xor instructions in the program is likely to be unstealthy for most host programs expressed in typical executable languages, since xor instructions are unusual in common code. So this algorithm is inappropriate for general  $U$ ; however, it could be quite stealthy when applied to a range  $U'$  of programs with cryptographic or bitmapped-graphic primitives.

As noted before in this article, stealth interacts with data rate as well as with resilience. For example, a fingerprinting scheme  $A$  might allow a stealthy embedding of a 4-bit fingerprint but not a 40-bit fingerprint, in a given program  $P$ . We parameterize this dependence of stealth on fingerprint size in our formal definitions.

*Definition LOCAL STEALTH.* Let  $A$  be a watermarking algorithm,  $L$  be an adversary’s locator function,  $m$  the length (in bits) of a watermark,  $w$  a watermark chosen uniformly at random from  $0..2^m - 1$ ,  $U$  a universe of benchmark unwatermarked programs,  $P$  a program chosen at random from  $U$ , and  $\lambda$  a real-valued constant in  $[0..1]$ . The locator function  $L : X \rightarrow \{0, 1\}^{|X|}$  is intended to flag the instructions and constant data in a watermarked program  $X$ . Let  $\hat{L}$  be an error-free locator function, such that  $\hat{L}(X)[i] = 1$ , if the  $i$ th instruction of  $X$  builds, modifies or encodes a watermark, and  $\hat{L}(X)[i] = 0$  otherwise. We say  $A$  is locally  $\lambda$ -stealthy for  $L$  if either the false-positive or false-negative error rate is at least  $\lambda$  on an instruction  $j$  chosen uniformly at random from a random input  $X = A(P, w)$ :

$$\max \left\{ \begin{array}{l} \text{Prob} (\hat{L}(X)[j] = 0 \wedge L(X)[j] = 1), \\ \text{Prob} (\hat{L}(X)[j] = 1 \wedge L(X)[j] = 0) \end{array} \right\} \geq \lambda \quad (1)$$

*Definition STEGANOGRAPHIC STEALTH.* Let  $S$  be an adversarial detection method mapping programs onto 0-1 (where 1 indicates the presence of a watermark) and let  $\sigma$  be a real-valued constant in  $[0..1]$ . If  $A$ ,  $U$ ,  $P$ ,  $m$ ,  $w$  are defined as above, we say  $A$  is steganographically  $\sigma$ -stealthy for  $S$  if either the

false-positive or false-negative error rate of  $S$  is at least  $\sigma$ :

$$\max\{\text{Prob}(S(P) = 1), \text{Prob}(S(A(P, w)) = 0)\} \geq \sigma \quad (2)$$

Any given fingerprinting algorithm will score differently on a stealth metric based on either of the definitions above, depending on the specific adversary that is chosen. We can, for example, imagine a worst-case scenario where a “local-stealth” adversary decompiles, reads, and learns to understand an entire program in order to locate redundant fingerprint code. Alternatively, we can imagine a much less demanding “steganographic-stealth” scenario, where an adversary constructs a class attack which computes static statistics of a program, from which a simple heuristic determines if the program contains a fingerprint or not. In this article we make an initial exploration of the latter scenario, deferring other evaluations of stealth to future work.

**7.1.1 Experimental Setup.** We collected a universe of 622 Java jar-files from the Internet in July 2003. The programs range in size from 6 to 40858 methods. The total number of instructions in each program ranged from 21 to 508,806; with an average of 13,500 instructions and a median of 3,676 instructions. The 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. We thus assert that they form an interesting random sampling of real Java programs.

For each jar-file a set of *instruction windows* was computed by sweeping a peephole or window of size  $n$  ( $n = 1, 2, 3, 4$ ) instructions over the instruction stream. The frequency of each  $n$ -gram (the number of times it occurred in each jar-file) was recorded.

Prior to computing the  $n$ -grams, similar instructions were put into equivalence classes. This prevents anomalies resulting from different applications being compiled with different compilers, using different code-generation strategies. Our use of equivalence classes, and our restriction to short window lengths, has the further advantage of making our simple stealth analysis reasonably predictive of actual stealth (relative to a rudimentary adversary) after an obfuscation. Note that any adversary who relies on long windows, or specific choices of instructions, would be easily fooled by simple obfuscations that reorder instructions and substitute short sequences of instructions by synonyms.

For example, to push the value 2, one compiler (or one obfuscator) might generate `iconst_2`, while another might generate `iconst 2`. With this in mind, we put all `iconst` instructions into an `ICONST` class, all `iload` instructions into an `ILOAD` class, all integer arithmetic instructions into an `IARITH` class, all branch instructions into an `IF` class, etc.

We generated Fingerprint classes (as in Figure 4)  $W_{4,3}$ ,  $W_{16,3}$ ,  $W_{32,3}$ , and  $W_{64,3}$  for uncycled Radix Graph fingerprints of size 4, 16, 32, and 64 bits. The second subscript indicates that the fingerprint graphs were split into three components. We analyzed the  $n$ -grams in these classes by the algorithm sketched above: first we classified the opcodes into their equivalence class (e.g., `ILOAD`), and then we computed instruction window frequencies. Figure 18 plots our data

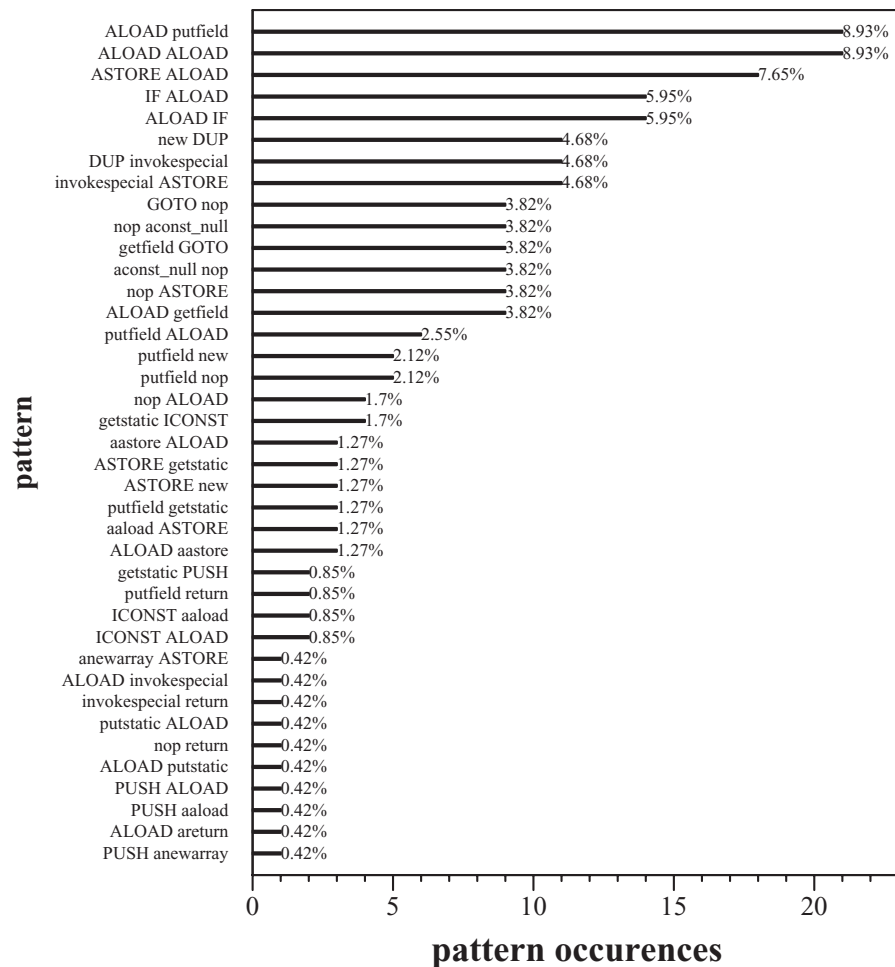


Fig. 18. Digram frequencies generated for a 32-bit CT fingerprint.

for windows of size  $n = 2$ , revealing that digrams involving `aload` and `astore` instructions are very common in the graph building code  $W_{32,3}$ .

In our steganographic stealth analysis below, we ignore the frequency of occurrence. Instead, we assume our adversary computes the set of  $n$ -grams that occur in a jar-file, for  $n \leq 4$ . Specifically, we reveal to our adversary (or assume they somehow learn) the sets  $W_{m,n,3}$  of  $n$ -gram occurrences in our graph-building codes  $W_{m,3}$  for  $m = \{4, 16, 32, 64\}$  for  $n \leq 4$ . Our adversary will compute a measure of similarity between this occurrence vector, and the  $n$ -gram occurrences observed in some possibly fingerprinted code  $X$ . If almost all of the  $n$ -grams in  $W_{m,n,3}$  also occur in  $X$ , the adversary will conclude that  $X$  is likely to be fingerprinted. In future work, we will consider more powerful adversaries who possess  $n$ -gram frequency information on our fingerprints.

In order to evaluate stealth against our adversary, we construct a large number of datasets  $P_{m,n,i}$  of the  $n$ -gram occurrences in fingerprinted Java

bytecode code. Each of these  $4 \times 4 \times 622 = 9952$  datasets is a union of the  $n$ -gram occurrences in one of our four fingerprint classes  $W_{m,3}$ , with the  $n$ -gram occurrences of one of our 622 applications  $P_i$ , for some window size  $n = 1, 2, 3, 4$ . Note that this is an approximation to the true  $n$ -gram occurrences that would be obtained if we actually embedded fingerprints in these applications  $P_i$ . It would be impractical for us to produce a more accurate set of  $n$ -grams for a dataset of this size. As noted earlier in this article, the CT embedding algorithm in SandMark is not completely automatic, for the user must annotate the code and then run it with a secret input sequence to produce a trace.

We have identified two sources of error in our simulation. First of all, in an actual embedding, some windows in the application would be broken up by the inserted fingerprinting code, so a few digrams in our simulated fingerprinting may not be present at all in actual fingerprinted code. Also, before and/or after an actual embedding, we would want to apply code obfuscation to increase stealth. This would certainly affect  $n$ -gram frequencies, and it may also have an effect on the  $n$ -gram occurrences we use in our stealth analysis. Both effects are small, and would tend to decrease the accuracy of our adversary’s detector in any practical setting, so they will cause us to make a slight overestimation of the stealth of the CT algorithm.

*7.1.2 Steganographic Stealth Analysis.* We compute steganographic stealth, relative to an adversarial detector  $B$ . This detector is a one-sided test of similarity, the value of which is maximized when the code  $X$  under test contains all the  $n$ -grams occurring in a reference set  $W_{m,n,3}$ .

We define our adversary formally as follows.

$$B_{m,n,\delta}(X) = \begin{cases} 1, & \text{if } \frac{|\text{window types that occur in } W_{m,n,3} \text{ but not in } X|}{|\text{window types that occur in } W_{m,n,3}|} < \delta \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

Our adversarial detector  $B$  is parameterized on the length  $m$  of the fingerprint ( $m = \{4, 16, 32, 64\}$ ), the length  $n$  of the  $n$ -grams ( $n \leq 4$ ), and a sensitivity parameter  $\delta$  ( $0 < \delta < 1$ ). We would expect an adversary to choose an  $n$  that maximizes the accuracy of their detector. We assume the adversary will choose  $m$  accurately, based on some prior knowledge of the fingerprinting scheme. The adversary will choose an appropriate  $\delta$  based on their tolerance for false-positive and false-negative errors. For  $\delta$  near 1, the detector will suffer from false-positive errors: it will report some unfingerprinted  $X$  as being fingerprinted. For  $\delta$  near 0, the detector will suffer from false-negative errors: it will report some fingerprinted  $X$  as being not fingerprinted.

We assume the adversary will be intolerant of false-negative errors, because it can be very expensive to be “caught out” distributing code bearing someone else’s fingerprint. By contrast, false-positive errors are likely to be inexpensive to the adversary. In response to a false-positive, an adversary would conduct further attacks on  $X$ , until an  $X'$  is constructed for which  $B(X') = 0$ . We assume an adversary will be unable to evaluate false-positive error rates accurately, for this will require excellent knowledge of many parameters that are difficult

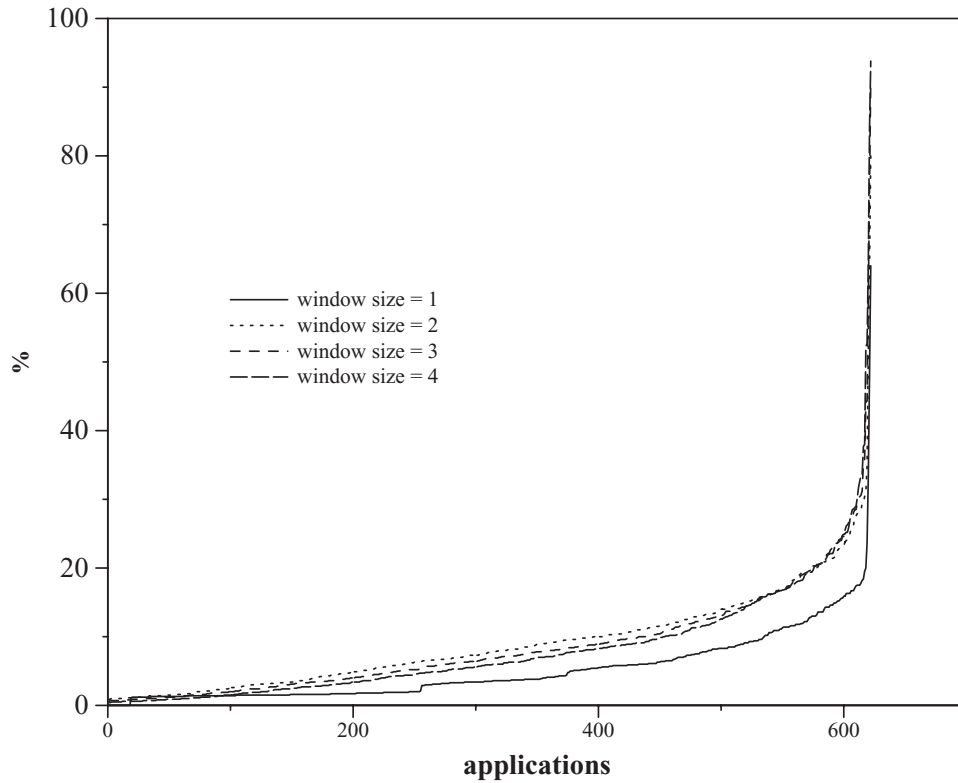


Fig. 19. Steganographic stealth evaluation. For each of the 622 applications in our benchmark universe, the graph shows the fraction of  $n$ -grams which occur in the embedded 32-bit CT fingerprint code, but which do not occur in the application itself.

to estimate, such as the universe  $U_1$  of codes that are likely to be input to a fingerprint embedder. In our stealth evaluation, we assume that the adversarial detector has a low sensitivity  $\delta = 0.1$ , so that it can have a low false-negative rate.

Note that  $B$  has the correct endpoint behavior, for any sensitivity in the range  $0 < \delta < 1$ . It will report that  $X$  is fingerprinted ( $B_{m,\delta}(X) = 1$ ), if all  $n$ -grams in  $W_{m,3}$  also occur in  $X$ . However, if none of the  $n$ -grams in  $W_{(m,3)}$  also occur in  $X$ , then  $B_{m,\delta}(X) = 0$ .

Figure 19 shows that at sensitivity  $\delta = 0.1$  (plotted on the y-axis), a 32-bit fingerprint is steganographically stealthy in 395 out of the 622 applications in our benchmark universe  $U$ . We conclude that the CT fingerprint, in its present form, is suitable for use against adversary  $B$  in about half of the Java programs that have been published to the web recently.

This is the first published evaluation of stealth for any software fingerprinting technique. Improvements to the stealth analysis are certainly possible. For example, we could evaluate stealth against stronger adversaries who compute  $\chi^2$  measures of similarity to  $n$ -gram frequency vectors of fingerprinting code. We could also evaluate stealth against adversaries who construct signatures,

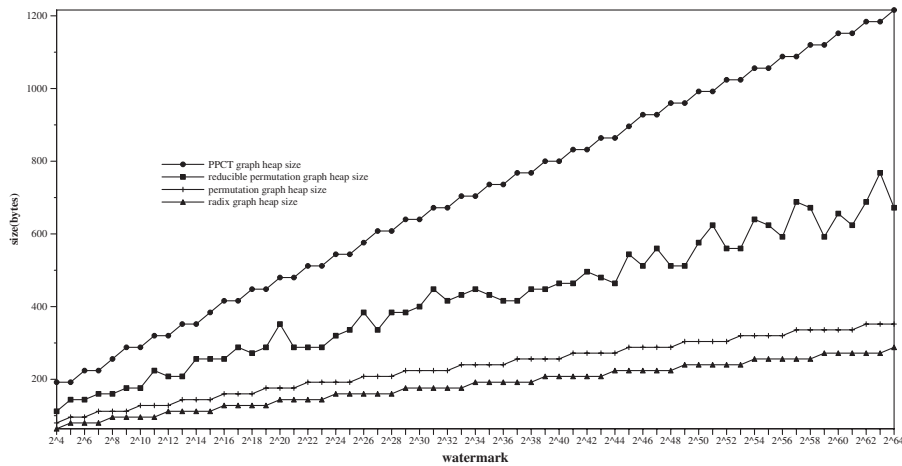


Fig. 20. Sizes of the graph as a function of the size of the fingerprint.

similar to those used in heuristic scans for computer viruses in email, of the  $n$ -grams that are most characteristic of fingerprinted code.

In addition to improving the offensive power of the adversary, we can also improve the defensive stealth of any fingerprinting method against that adversary. Indeed, in Section 6 we have already discussed several methods for improving stealthiness against various types of adversaries.

## 7.2 Data Rate

Figure 20 shows the runtime size of the different graph structures as a function of the size of the fingerprint.

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 fingerprint integer  $w$ , that is,  $m = \lceil \lg(w + 1) \rceil$  when  $w$  is 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 fingerprint. This was reflected in our experimental data, however the non-linearity 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 ( $R^2 = 0.970$ ) of unexplained variance. A tiny amount of this unexplained variance is the expected nonlinearity, but the majority is clearly visible as “noise” in Figure 20. We believe this apparent noise is due to the dependence of the size of the preamble of an RPG, on the value of the fingerprint  $w$  being encoded as an RPG: two fingerprints  $w$  with the same number of bits can have differing sizes of RPGs. Our asymptotic analysis indicated that the RPG size would vary by  $\pm 33\%$  for any given number of bits  $m$ ; however, this was an asymptotic upper bound. 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  $R^2 = 0.991$  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 fingerprint.

Figure 21 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 fingerprinting methods. As expected, the static bitrates are in proportion to the dynamic bitrates: 21 codebytes per fingerprint 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.

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 nonmonotonic nature of the radix graph curve in Figure 21.

Figure 22 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 fingerprint bit of the Radix Graph method becomes approximately 6 codebytes per fingerprint 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.

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

Overall, our experimentation has confirmed that the dynamic and static overhead of dynamic data fingerprinting is acceptably small for most applications.

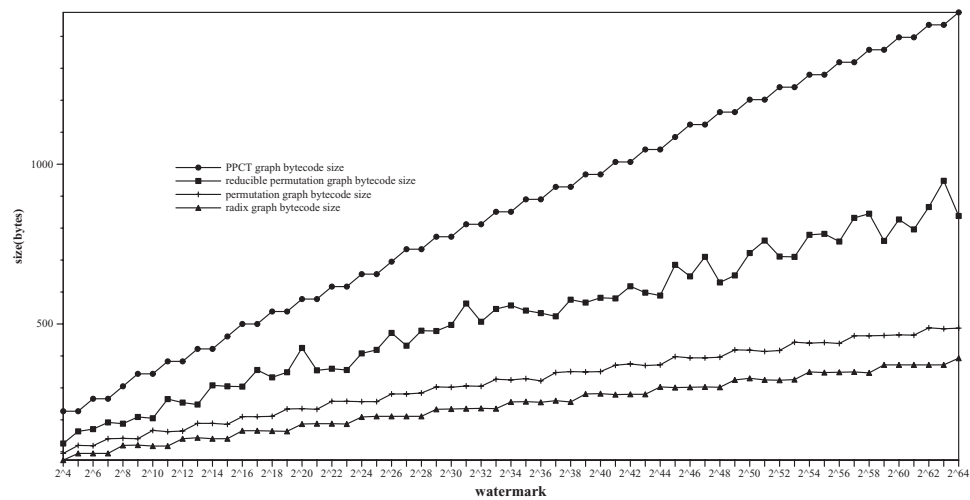


Fig. 21. Sizes of the graph building bytecode as a function of the size of the fingerprint.

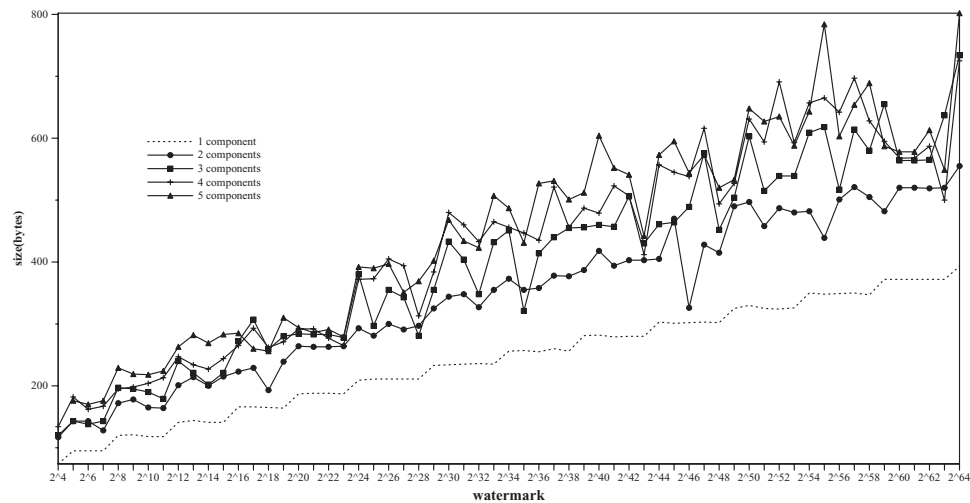


Fig. 22. Size of the bytecode for building a Radix Graph, as a function of the number of components into which the graph is split.

### 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 fingerprint 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 23, this resilience comes at a significant cost: the dynamic data rate is reduced by a factor of almost two.

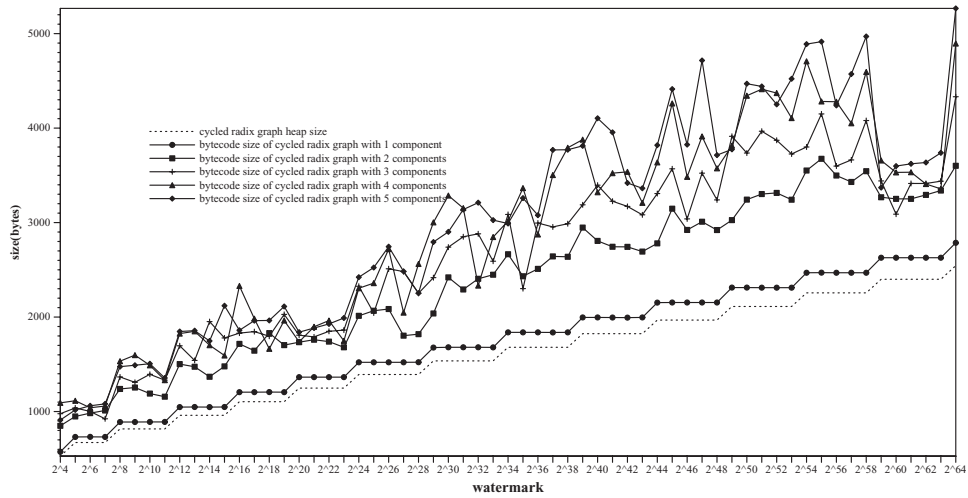


Fig. 23. Bytecode size, and runtime heap size, for cycled radix graphs.

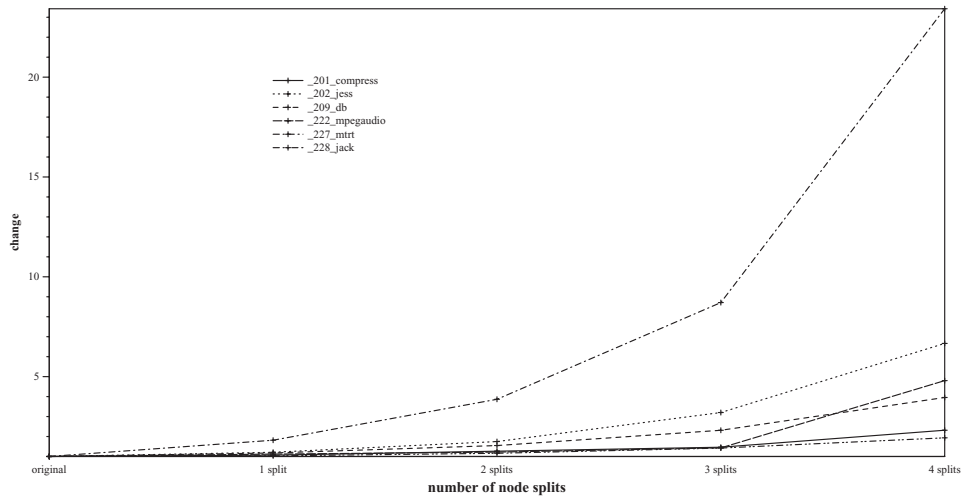


Fig. 24. Overhead of applying sequence of node-splittings to the SpecJVM benchmark suite. The javac benchmark was excluded since it currently fails on the NODESPLITTER obfuscation.

Figure 24 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-splittings without having to worry about performance overhead. The variance is high, however. Applying two node-splittings 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.

The BLOAT Nystrom [2004] bytecode optimizer is included in the SandMark system. It also has no effect on the success of fingerprint extraction. In general, unless a compiler discovers (through a static shape analysis [Ghiya and

Hendren 1996], a dead code analysis, etc.) that our inserted code is redundant, and removes it, typical compiler transformations are ineffective attacks against our fingerprints.

## 8. DISCUSSION

Because software fingerprinting 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 article, we have identified several types of threats:

- (1) Distortive attacks on the resilience of the fingerprint, using semantics-preserving transformations such as translation, optimization, and obfuscation.
- (2) Statistical attacks on the local stealth of the fingerprint, which attempt to locate a fingerprint by identifying anomalies in the distribution of instructions or computations.
- (3) Collusive attacks on the local stealth of a fingerprint, attempting to locate it by comparing several differently fingerprinted copies of a program.
- (4) Cropping attacks on the resilience of an unstealthy fingerprint, which remove a located fingerprint or extract an individual module from a fingerprinted application.
- (5) Additive attacks on the resilience of a fingerprint, which insert new bogus fingerprints into an already fingerprinted program.

None of the methods we have presented are immune to all types of attacks. Easter egg fingerprints and dynamic graph fingerprints are highly resilient against distortive attacks, but, by their very nature, they fingerprint *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 fingerprints, 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 fingerprints are highly susceptible to distortive attacks.

Whether a statistical attack is successful or not will depend on the nature of the fingerprint, and the nature of the application. Dynamic graph fingerprints 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; and our analysis in Section 7.1 confirms that our prototype implementation of the CT fingerprinting method is unstealthy in approximately half of all Java applications currently published to the web. Davidson's [Davidson and Myhrvold 1996b] 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 suboptimal.

It is interesting to note that the problems we face in software fingerprinting are often quite different from those that arise in fingerprinting 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 fingerprints 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 fingerprints than to software fingerprints. 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 Legal and Practical Concerns

Deploying a software fingerprinting system in a practical setting raises many issues not covered in this article. We briefly discuss some of these legal and technical problems here.

First of all, what, if any, legal protection does software fingerprinting provide? If a program marked with Bob's identifier is discovered in the possession of Charles, will Alice be able to successfully prosecute Charles and/or Bob? At the present time we are aware of no case law that answers this question. The problem is complicated by the fact that Bob might have bought his copy of the program using false credentials (such as a stolen credit card) and that Charles might be an end user from whom Alice is unlikely to extract financial relief. It is our belief that software fingerprinting will be one of many pieces of evidence needed in a legal proceeding against a software pirate.

Software fingerprinting has many applications in addition to tracing software pirates. For example, Alice may want to distribute early versions of her software to a few select collaborating companies, allowing them the opportunity to integrate it with their own products, perform compatibility analysis, etc. Fingerprinting each copy with the collaborators' identities will allow her to seek relief from any company that allows these early versions to escape into the public domain. In this case, prosecution will be simpler, since Alice is dealing with a small number of possible culprits, all of whom can be held financially responsible.

Fingerprinting can also be used in a government and military setting. Several cases have come to light where security-sensitive software has escaped into the hands of an adversarial government, either by careless handling of portable devices or by deliberate sale by a traitor. Marking each distributed copy with the owner's signature can, at the very least, serve as a deterrent against carelessness and treachery.

Where and when does the actual fingerprinting of a program take place? At the present time, most software is sold "shrink-wrapped," that is, hard-coded on a CD-ROM. This distribution channel does not seem to lend itself to fingerprinting, since every distributed CD-ROM contains the same image.

Embedding the user's identifier could be done at install time (possibly sending information back to the manufacturer's site), but this provides a convenient attack: modify the installer so that the wrong identification is embedded. We believe that in the future, all software will be distributed in "soft" form, as network downloads. In this scenario, it is straight-forward for Alice to verify the customer's credentials (their credit card number, for example) and embed a unique identifier into the binary program before it is downloaded.

A major concern is debugging and quality assurance of fingerprinted and obfuscated programs. We expect every differently marked program to undergo a different set of obfuscating transformations in order to prevent collusive attacks. Our SandMark tool contains several *obfuscation executives* [Heffner and Collberg 2004] that can generate random but legal sequences of obfuscations, seeded by a secret key. In order to effectively field bug-reports it will be necessary for Alice to store, for ever sold copy, the keys used to fingerprint and obfuscate it. This will allow her to recreate the exact copy distributed to an individual user, should this copy need to be debugged.

## 8.2 SandMark and JavaWiz

The first implementation based on the CT algorithm was JavaWiz Palsberg et al. [2000]. 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 fingerprint extraction. User annotation of the source program is required to make the fingerprint input-dependent.

Both implementations embed an arbitrary bit string as the fingerprint, 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 fingerprint 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) 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 fingerprint.

Both implementations generate relatively straight-forward code under the assumption that it will be subsequently obfuscated.

## 9. SUMMARY

Software fingerprinting embeds an identifying value in a program. The ideal fingerprint 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 fingerprints are static: They are applied to, and detected in, an executable binary file. In this article, we have described the CT algorithm, where a dynamic fingerprint 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 fingerprint is much harder to detect than a static fingerprint and is also much more robust against transformations such as obfuscation and optimization.

The CT fingerprinting 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 fingerprint, 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 fingerprinting as part of SandMark allows obfuscation after modification.

We have explored trade-offs among graph encodings and fingerprint 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 fingerprint and found that the code introduced by the CT algorithm is stealthy for a large fraction of real Java programs.

We tested the CT fingerprint's resilience against SandMark's 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 greatest vulnerability of our implementation of CT watermarking is due to the limited stealthiness of the watermarking code. We have demonstrated, in Section 7.1, that CT watermarks are stealthy against a statistically-naïve attacker. A somewhat more clever attacker will eventually discover an accurate way to detect a CT watermark, by carefully analyzing they obtain a large number of programs that are known to be watermarked. An attacker could construct such a collection, fairly easily, by using the CT watermark embedder in the SandMark package, available for download at [sandmark.cs.arizona.edu](http://sandmark.cs.arizona.edu).

One, very ambitious, goal for future research would be to develop a highly stealthy watermark. The embedder in such a system can not leave any signature which will be obvious to the attacker. This suggests that its output must be highly randomized and yet still resemble unwatermarked code; otherwise, the attacker will be able to detect the presence of the watermark. A more realistic goal for future research is to construct a highly resilient CT watermark that is locally stealthy but not steganographically stealthy. Jasvir Nagra has recently shown how to construct a thread-based watermark with these security properties, under two assumptions: that the attacker is unable to guess the secret key and tables used in recognition, and that the attacker is unable to crack the

opaque predicates used in tamperproofing Nagra [2006]. It is an open problem to extend Nagra's approach to the development of a CT watermarking method that is highly resilient and locally stealthy.

#### ACKNOWLEDGMENTS

We thank Edward Carter, Andrew Huntwork, Kamlesh Kantilal, and Jasvir Nagra for implementation assistance. William Zhu provided some valuable insight into our theoretical models. Our anonymous referees made many suggestions and criticisms which pointed the way to many improvements.

#### REFERENCES

- BCEL. 2004. [jakarta.apache.org/bcel](http://jakarta.apache.org/bcel).
- DynamicJava. 2004. [koala.ilog.fr/djava](http://koala.ilog.fr/djava).
- AHPAH. 2005. [Sourceagain.ahpah.com](http://Sourceagain.ahpah.com).
- ALBERT, D. AND MORSE, S. 1982. Combating software piracy by encryption and key management. *IEEE Comput.* 17, 4 (Apr.), 68–73.
- ANDERSON, R. J. AND PETICOLAS, F. A. 1998. On the limits of steganography. *IEEE J-SAC* 16, 4 (May).
- ARBOIT, G. 2002. A method for watermarking Java programs via opaque predicates (extended abstract). In *Proceedings of the 5th International Conference on Electronic Commerce Research (ICECR-5)*. [citeseer.nj.nec.com/arboit02method.html](http://citeseer.nj.nec.com/arboit02method.html).
- BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv.* 26, 4 (Dec.), 345–420.
- BAKER, B. S. AND MANBER, U. 1998. Deducing similarities in Java sources from bytecodes. In *Proceedings of the USENIX Annual Technical Conference*.
- BENDER, W., GRUHL, D., MORIMOTO, N., AND LU, A. 1996. Techniques for data hiding. *IBM Syst. J.* 35, 3&4, 313–336.
- CHANG, H. AND ATALLAH, M. 2001. Protecting software code by guards. In *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001* (Philadelphia, PA), Lecture Notes in Computer Science, Vol. 2320. Springer Verlag, New York.
- CHOW, S., GU, Y., JOHNSON, H., AND ZAKHAROV, V. 2001. An approach to the obfuscation of control-flow of sequential computer programs. In *Information Security: Fourth International Conference (ISC 2001)*, Davida and Frankl, Eds. Lecture Notes in Computer Science, vol. 2200. Springer Verlag, 144–155.
- COLLBERG, C., CARTER, E., KOBOUROV, S., AND THOMBORSON, C. 2003a. Error-correcting graphs. In *Proceedings of the Workshop on Graphs in Computer Science (WG'2003)*.
- COLLBERG, C., MYLES, G., AND HUNTWORK, A. 2003b. SANDMARK—A tool for software protection research. *IEEE Magazine of Security and Privacy* 1, (Aug.).
- COLLBERG, C. AND THOMBORSON, C. 1999. Software watermarking: Models and dynamic embeddings. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, TX). ACM, New York.
- COLLBERG, C., THOMBORSON, C., AND LOW, D. 1997. A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Science, University of Auckland. July. [citeseer.nj.nec.com/collberg97taxonomy.html](http://citeseer.nj.nec.com/collberg97taxonomy.html).
- COLLBERG, C., THOMBORSON, C., AND LOW, D. 1998a. Breaking abstractions and unstructuring data structures. In *Proceeding of the IEEE International Conference on Computer Languages, ICCL'98*. (Chicago, IL), IEEE Computer Society Press, Los Alamitos, CA.
- COLLBERG, C., THOMBORSON, C., AND LOW, D. 1998b. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the Principles of Programming Languages (POPL'98)* (San Diego, CA), ACM, New York.
- COMPAQ. 2004. FreePort Express. [hwww.support.compaq.com/amt/tools/migrate-cover.html](http://hwww.support.compaq.com/amt/tools/migrate-cover.html).
- COUSOT, P. AND COUSOT, R. 2004. An abstract interpretation-based framework for software watermarking. In *Proceedings of the ACM Principles of Programming Languages*. ACM, New York.



- CRAVER, S., MEMON, N., YEO, B.-L., AND YEUNG, M. M. 1998. Resolving rightful ownerships with invisible watermarking techniques: limitations, attacks, and implications. *IEEE J. Select. Areas Commun.* 16, 4 (May), 573–586.
- DAVIDSON, R. AND MYHRVOLD, N. 1996a. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, Assignee: Microsoft Corporation. [www.delphion.com/details?pn=US055598844](http://www.delphion.com/details?pn=US055598844).
- DAVIDSON, R. L. AND MYHRVOLD, N. 1996b. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884. Assignee: Microsoft Corporation.
- DEBRAY, S., EVANS, W., MUTH, R., AND SUTTER, B. D. 2000. Compiler techniques for code compaction. *ACM Trans. Prog. Lang. Syst.* 22, 2 (Mar.), 378–415.
- DEBRAY, S., MUTH, R., WATTERSON, S., AND BOSSCHERE, K. D. 2001a. ALTO: A link-time optimizer for the Compaq Alpha. *Softw.—Pract. Exp.* 31, 67–101.
- DEBRAY, S., SCHWARZ, B., ANDREWS, G., AND LEGENDRE, M. 2001b. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proceedings of the 2001 Workshop on Binary Rewriting (WBT:2001)*.
- GHIYA, R. AND HENDREN, L. J. 1996. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'96)* (St. Petersburg Beach, FL). ACM, New York, 1–15.
- GOULDEN, I. P. AND JACKSON, D. M. 1983. *Combinatorial Enumeration*. Wiley, New York.
- HALSTEAD, M. H. 1977. *Elements of Software Science*. Elsevier North-Holland. Amsterdam, The Netherlands.
- HARARY, F. AND PALMER, E. 1973. *Graphical Enumeration*. Academic Press, New York.
- HARRISON, W. A. AND MAGEL, K. I. 1981. A complexity measure based on nesting level. *SIGPLAN Notices* 16, 3, 63–74.
- HAUSER, R. C. 1995. 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*. Vol. 1. (Honolulu, Hawaii), 199–204.
- HEFFNER, K. AND COLBERG, C. S. 2004. The obfuscation executive. In *Information Security, 7th International Conference*. Lecture Notes in Computer Science, Vol. 3225. Springer Verlag, New York, 428–440.
- HENRY, S. AND KAFURA, D. 1981. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.* 7, 5 (Sept.), 510–518.
- HERZBERG, A. AND KARMI, G. 1984. On software protection. In *Proceedings of the 4th Jerusalem Conference on Information Technology*. Jerusalem, Israel.
- HERZBERG, A. AND PINTER, S. S. 1987. Public protection of software. *ACM Trans. Comput. Syst.* 5, 4 (Nov.), 371–393.
- HORNE, B., MATHESON, L., SHEEHAN, C., AND TARJAN, R. E. 2001. Dynamic self-checking techniques for improved tamper resistance. In *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop (DRM 2001)*. (Philadelphia, PA). Lecture Notes in Computer Science, vol. 2320, Springer Verlag, New York.
- INTERNATIONAL PLANNING AND RESEARCH CORPORATION. 2003. Eighth annual BSA global software piracy study. [Global.bsa.org/globalstudy](http://Global.bsa.org/globalstudy).
- KNUTH, D. E. 1997. *Fundamental Algorithms*, Third ed. *The Art of Computer Programming*, vol. 1. Addison-Wesley, Reading, MA.
- KUNDU, S. AND MISRA, J. 1997. A linear tree partitioning algorithm. *SIAM J. Comput.* 6, 1 (Mar.), 151–154.
- MADOU, M., ANCKAERT, B., SUTTER, B. D., AND BOSSCHERE, K. D. 2005. Hybrid static-dynamic attacks against software protection mechanisms. In *DRM '05: Proceedings of the 5th ACM Workshop on Digital Rights Management*. ACM, New York, 75–82.
- MALHOTRA, Y. 1994. Controlling copyright infringements of intellectual property: the case of computer software. *J. Syst. Manage.* 45, 6 (June), 32–35. part 1, part 2: No 7, Jul. pp. 12–17.
- MAUDE, T. AND MAUDE, D. 1984. Hardware protection against software piracy. *Commun. ACM* 27, 9 (Sept.), 950–959.
- MCCABE, T. J. 1976. A complexity measure. *IEEE Trans. Softw. Eng.* 2, 4 (Dec.), 308–320.

- MONDEN, A., IIDA, H., ICHI MATSUMOTO, K., TORII, K., AND ICHISUGI, Y. 1998. Watermarking method for computer programs. In *Proceedings of the 1998 Symposium on Cryptography and Information Security (SCIS'98 - 9.2A)*. (In Japanese).
- MONDEN, A., IIDA, H., MATSUMOTO, K., INOUE, K., AND TORII, K. 2000. A practical method for watermarking Java programs. In *Proceedings of the 24th Computer Software and Applications Conference*.
- MORI, R. AND KAWAHARA, M. 1990. Superdistribution: The concept and the architecture. *The Transactions of the IEICE* 73, 7 (July). [www.virtualschool.edu/mon/ElectronicProperty/MoriSuperdist.html](http://www.virtualschool.edu/mon/ElectronicProperty/MoriSuperdist.html).
- MOSKOWITZ, S. A. AND COOPERMAN, M. 1996. Method for stega-cipher protection of computer code. US Patent 5,745,569. Assignee: The Dice Company.
- MUNSON, J. C. AND KOHSHGOFTAAR, T. M. 1993. Measurement of data structure complexity. *J. Syst. Softw.* 20, 217–225.
- MURATANI, H. 2001. A collusion-secure fingerprinting code reduced by Chinese remaindering and its random-error resilience. In *Information Hiding: 4th International Workshop (IHW 2001)*. (Pittsburgh, PA), 303–315.
- MYLES, G. AND COLLBERG, C. 2003. Software watermarking through register allocation: Implementation, analysis, and attacks. In *Proceedings of the International Conference on Information Security and Cryptology*.
- MYRVOLD, W. AND RUSKEY, F. 2001. Ranking and unranking permutations in linear time. *Inf. Proc. Lett.* 79, 6 (Sept.), 281–284.
- NAGRA, J. 2006. Threading software watermarks. Ph.D. dissertation. University of Auckland, Auckland, New Zealand.
- NAGY-FARKAS, D. 2004. The Easter egg archive. [www.eeggs.com](http://www.eeggs.com).
- NYSTROM, N. 2004. *BLOAT—The Bytecode-Level Optimizer and Analysis Tool*. [www.cs.purdue.edu/s3/projects/bloat](http://www.cs.purdue.edu/s3/projects/bloat).
- OVIEDO, E. I. 1980. Control flow, data flow, and program complexity. In *Proceedings of IEEE COMPSAC*. 146–152.
- PALSBERG, J., KRISHNASWAMY, S., KWON, M., MA, D., SHAO, Q., AND ZHANG, Y. 2000. Experience with software watermarking. In *Proceedings of ACSAC'00, 16th Annual Computer Security Applications Conference*. 308–316. [citeseer.nj.nec.com/323325.html](http://citeseer.nj.nec.com/323325.html).
- PETICOLAS, F. A., ANDERSON, R. J., AND KUHN, M. G. 1998. Attacks on copyright marking systems. In *Proceedings of the 2nd Workshop on Information Hiding* (Portland, OR).
- PETITCOLAS, F. A. P. 2004. Stirmark 3.1. [www.cl.cam.ac.uk/~fapp2/watermarking/stirmark](http://www.cl.cam.ac.uk/~fapp2/watermarking/stirmark).
- PIEPRZYK, J. 1999. Fingerprints for copyright software protection. In *Proceedings of the 2nd International Workshop on Information Security (ISW'99)*, Lecture Notes in Computer Science, vol. 1729, Springer Verlag, pp. 178.
- PROEBSTING, T. A. AND WATTERSON, S. A. 1997. Krakatoa: Decompilation in Java (Does bytecode reveal source?). In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*.
- QU, G. AND POTKONJAK, M. 1998. Analysis of watermarking techniques for graph coloring problem. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*. ACM, New York 190–193.
- RAMALINGAM, G. 1994. The undecidability of aliasing. *ACM Trans. Prog. Lang. Syst.* 16, 5 (Sept.), 1467–1471.
- SAHOO, T. AND COLLBERG, C. 2004. Software watermarking in the frequency domain: Implementation, analysis, and attacks. Tech. Rep. TR04-07, Department of Computer Science, University of Arizona. Mar.
- SIMMEL, S. S. AND GODARD, I. 1994. 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. Interactive Multimedia Association, John F. Kennedy School of Government, MIT, Program on Digital Open High-Resolution Systems, 81–110.
- STERN, J. P., HACHEZ, G., KOEUNE, F., AND QUISQUATER, J.-J. 1999. Robust object watermarking: Application to code. In *Information Hiding*. 368–378.

- THOMBORSON, C., NAGRA, J., SOMARAJU, R., AND HE, C. 2004. Tamper-proofing software watermarks. In *Proceedings of the 2nd Australasian Information Security Workshop (AISW2004)*, P. Montague and C. Steketee, Eds. Number 32 in CRPIT. ACS, 27–36.
- VENKATESAN, R., VAZIRANI, V., AND SINHA, S. 2001. A graph theoretic approach to software watermarking. In *Proceedings of the 4th International Information Hiding Workshop* (Pittsburgh, PA).
- WANG, C. 2000. A security architecture for survivability mechanisms. Ph.D. dissertation, University of Virginia, School of Engineering and Applied Science. [www.cs.virginia.edu/~survive/pub/wangthesis.pdf](http://www.cs.virginia.edu/~survive/pub/wangthesis.pdf).

Received April 2004; revised July 2005 and November 2006; accepted January 2007