The Department of Computer Science
The University of Auckland
New Zealand

Threading Software Watermarks

Jasvir Nagra

February 2007

Supervisor:

Clark Thomborson

A thesis submitted in partial fulfillment of the requirements of Doctor of Philosophy in Computer Science

The University of Auckland

Thesis Consent Form

This thesis may be consulted for the purpose of research or private study provided that due acknowledgement is made where appropriate and that the author's permission is obtained before any material from the thesis is published.

I agree that the University of Auckland Library may make a copy of this thesis for supply to the collection of another prescribed library on request from that Library; and

1. I agree that this thesis may be photocopied for supply to any person in accordance with the provisions of Section 56 of the Copyright Act 1994.

   Or

2. ~~This thesis may not be photocopied other than to supply a copy for the collection of another prescribed library.~~

   (*Strike out 1 or 2*)

Signed: .....

Date: .....

Last updated: February 14, 2007

Abstract

In this thesis, we introduce thread-based watermarks - a new technique for embedding robust software watermarks into a software program using thread contention. Our technique can help protect the intellectual property that exists in software programs from reverse engineering and piracy.

We build thread-based watermarks by adding new threads of execution to single-threaded portions of the program. The locations of these widgets are chosen carefully, such that when given particular input, the dynamic behavior of the threads is distinctive and encodes a watermark. An attacker challenged with attacking such a watermark must first analyze an application sufficiently that he is able to remove or distort the watermark while preserving the program's semantics. However, the embedding of our proposed watermark also increases the complexity of the program and the cost of subsequent analysis.

Our technique is built using opaque predicates proposed by Collberg et al. We show our technique to be resilient to many semantic-preserving transformations that existing proposals are susceptible to. This thesis also introduces a novel use for opaque predicates to merge two different pieces of code such that they appear identical under static analysis.

We describe the technique for encoding the watermark as a bit string and a scheme for embedding and recognizing the watermark using thread contention. Our approach consists of three stages:

1. Tracing where the program is run with a key input.

2. Embedding where the program is transformed such that code that was executed by a single thread is now executed by multiple threads. The pattern in which these threads execute is distinctive and encodes a watermark.

3. Recognition where the program is run with the key input and the value of the embedded watermark is extracted from the execution trace

We prove the correctness of this watermarking technique and give a theoretical basis to show that the technique is difficult to attack using static analysis. Furthermore, we clearly identify the limitations of our solution and the assumptions made in the development of our thread-based watermarks. Finally, in implementing our prototype, we identify and resolve the problem of pattern matching attacks.

Acknowledgement

The writing of this thesis has been akin to navigating a maze of twisty passages. I could not have managed it without the guidance and support of many people who deserve my deepest special thanks.

First I am deeply indebted to Clark Thomborson, my supervisor who in the last five years has taught me a great deal about the care, patience and rigour required in science.

To my adviser and friend Christian Collberg without whose encouragement, advice, inspiration and insight this thesis would simply have not been possible and without whose sense of humor and cheer the journey would have been far more arduous.

I am thankful to Vanish Pattni, Cameron Skinner, Damien Duff and Mike Stay for the many conversations we had roughing out ideas, finding flaws and fixing them. I owe Ginger Myles and Kelly Heffner for inspiring me to keep on going when all seemed hopeless. Thank you Ailish Roughan, Jubal John, Brett Lomas and Jonothan Teutenburg, all of whom let me rely on them then pushed me onward. In the most difficult moments, I repeatedly turned to Shwetha Shankar and to the entire Shankar family for encouragement. Their faith in me made me stronger and made finishing this thesis possible. Thank you for believing in me.

I owe a debt of gratitude to the entire Department of Computer Science for their encouragement. I reserve a special thanks for Penny Barry and Anita Lai for the many administrative acrobatics they performed to mitigate the results of my absent-mindedness.

To my family go my greatest gratitude. Thank you Sonal and Sheetal for always having a kind word for their wayward brother. To ma and dad for nurturing me, for urging me to pursue my studies and coaxing me on when things seemed most bleak. They have instilled in me a sense of curiosity and aspiration for which I will always be grateful.

Contents

1 Introduction **13**

1.1 Introduction . . . 13
1.2 Software Watermarking . . . 14
1.3 Static vs Dynamic Analysis . . . 15
1.4 Using Complexity . . . 16
1.5 Solution Overview . . . 16
1.6 Dissertation Organization . . . 18

2 The Software Watermarking Problem **19**

2.1 Software Watermarking . . . 19
2.2 Categorical Properties . . . 20
2.2.1 Fragile and Robust Watermarks . . . 20
2.2.2 Visible and Invisible Watermarks . . . 20
2.2.3 Static and Dynamic Watermarks . . . 21
2.2.4 Informed and Blind Watermarks . . . 21
2.2.5 Focused and Spread Spectrum Watermarks . . . 22
2.3 Usability Properties . . . 22
2.3.1 Correctness . . . 22
2.3.2 Developer Cost . . . 22
2.3.3 Machine Cost . . . 23
2.3.4 Platform Independence . . . 23
2.3.5 Performance . . . 23
2.4 Security Properties . . . 24
2.4.1 Resilience . . . 24
2.4.2 Credibility . . . 24
2.4.3 Stealth . . . 25
2.5 Attack Model . . . 25
2.5.1 General Attack . . . 26
2.5.2 Targeted Attack . . . 26
2.6 Summary . . . 27

3 Threading Software Watermarks 28
3.1 Introduction . . . 29
3.2 Multi-threading . . . 31
3.2.1 Formal Model of Multi-threading . . . 33
3.2.2 Semantic-Preserving Transformations . . . 35
3.2.3 Static Analysis . . . 35
3.3 Thread-Based Watermarking . . . 37
3.3.1 Tracing . . . 37
3.3.2 Embedding . . . 38
3.3.3 Embedding a Single Bit . . . 38
3.4 Obfuscation . . . 43
3.4.1 Opaque Predicates . . . 43
3.5 Tamper-proofing . . . 44
3.5.1 Proof of Correctness . . . 45
3.5.2 Recognition . . . 50
3.6 Summary . . . 51

4 Theoretical Security Analysis 53
4.1 Introduction . . . 53
4.2 Static Analysis of Programs . . . 54
4.3 Security Against Targeted Attacks . . . 56
4.3.1 Proving Sequentiality . . . 57
4.3.2 Distinguishing Watermarks and Tamper-proofing . . . 58
4.3.3 Limitations on Security Against Targeted Attacks . . . 60

5 Implementation 61
5.1 Implementation Platform . . . 62
5.1.1 Target Platform . . . 62
5.1.2 Challenges posed by Java . . . 63
5.1.3 Development Platform . . . 65
5.2 Watermarking Java Bytecode . . . 66
5.2.1 Encoding . . . 66
5.2.2 Tracing . . . 67
5.2.3 Embedding . . . 70
5.2.4 Closures . . . 71
5.2.5 Opaque Predicates . . . 72
5.2.6 Thread Pools . . . 75
5.2.7 Tamper-proofing Widgets . . . 77
5.2.8 Recognition . . . 80

5.3 Discussion . . . 84
5.4 Summary . . . 85

6 Empirical Evaluation 87
6.1 Security Evaluation . . . 88
6.1.1 Resilience . . . 89
6.1.2 Credibility . . . 97
6.1.3 Stealth . . . 98
6.1.4 Discussion . . . 101
6.2 Performance . . . 102
6.2.1 Performance Overhead of Watermarking widgets . . . 102
6.2.2 Maximum Watermarking Data Rate . . . 107
6.3 Summary . . . 109

7 Related Work 110
7.1 Software Watermarking . . . 110
7.2 Simple Robust Software Watermarking . . . 111
7.2.1 Moskowitz and Cooperman . . . 111
7.2.2 Davidson and Myhrvold . . . 112
7.2.3 Akito Monden . . . 112
7.2.4 Stern et al. . . . 112
7.2.5 Comparison . . . 113
7.3 Static Watermarking based on Graphs . . . 114
7.4 Other Static Watermarks . . . 115
7.4.1 Cousot and Cousot . . . 115
7.5 Dynamic Watermarks . . . 115
7.5.1 CT watermark . . . 116
7.5.2 Path-based Watermarking . . . 116
7.5.3 Comparison . . . 117
7.6 Software Analysis . . . 117
7.6.1 Abstract Interpretation . . . 118
7.6.2 Theorem Provers . . . 118

8 Conclusion 120
8.1 Future Work . . . 122
8.1.1 Dynamic Analysis . . . 122
8.1.2 Truly Opaque Predicates . . . 123
8.2 Final Remarks . . . 123

1 Introduction

THIS dissertation describes the design, implementation, and analysis of an approach to software watermarking using multi-threading. This approach is novel in that it is difficult to read the watermark or attack it using static analysis. In contrast to other attempts to implement a truly dynamic watermark, the design described in this thesis pays careful attention to pattern matching. We show that without such careful attention to pattern matching, an implementation of a seemingly dynamic watermark is reduced to a static one.

Another attractive property of the proposed solution is that embedding the watermark simultaneously increases the threading complexity of the program which in turn makes reversing the process difficult. The technique described is platform independent, easy to automate and has low performance costs. Furthermore it is the first software watermarking technique which solves the open problem of using threads to embed information in programs.

1.1 Introduction

A software program is an implementation of an idea. Software is written after the expenditure of effort on the design of algorithms, user interface, choice of data structures and trade-offs in the selection of implementation details. These decisions are the result of creativity and give rise to the notion of intellectual property.

Intellectual property is the intangible product that is the result of creativity. We recognize creative effort in art, music, design and many other human endeavors. In software, intellectual property is particularly susceptible to misappropriation because digital media make it so easy to generate perfect copies, in part or as a whole. Computers further provide a suitable environment for reverse engineering whereby an attacker can expend resources to extract from a program sensitive information and trade secrets employed in its design.

In general, software developers desire to protect the intellectual effort they invest into writing software. There are some circumstances when developers may not wish to keep their software intellectual property secret such as when the software has been written to be fully open to inspection for security assurance purposes. Some software is developed for the common good, and is freely donated by its authors under ethical or moral principles such as those espoused by the Free Software Foundation [8]. Moreover, in some circumstances it may be undesirable or even illegal to prevent one's competitors from reverse engineering an application, when this reverse engineering is necessary to create compatible or competing products. However, we believe there is a sufficiently large base of developers who desire to protect their software to justify research into technological tools to assist this desire.

While it is generally believed that complete protection of intellectual property in software using technological tools is an unattainable goal, some existing research avenues provide some degree of protection. The three major research areas in software protection have been obfuscation, watermarking and tamper-proofing. We will define these terms precisely later in the thesis, however, for the moment we discuss them informally. To *obfuscate* a program means to make it hard to understand and thus hard to reverse engineer. To *tamper-proof* a program means to make it difficult to alter. To *watermark* a programs means to embed information into a program to make it or its author identifiable.

The aim of any of these techniques is more modest than to be completely invulnerable. If an attack against a software protective technique is expensive and difficult to perform, then that attack becomes unattractive.

1.2 Software Watermarking

The need for software watermarking is best motivated by introducing three actors, Alice, Bob and Charles as shown in Figure 1.1. Alice is the author of a software program P_ω which she sells to her customer Charles. Bob is a pirate who attempts to illegally copy and resell Alice's entire program or perhaps some part of it. If Bob distorts Alice's program, using for example by applying some transformation function q, then it may be difficult for Alice to prove that Bob pirated P_ω. Function q may be an *obfuscation function*

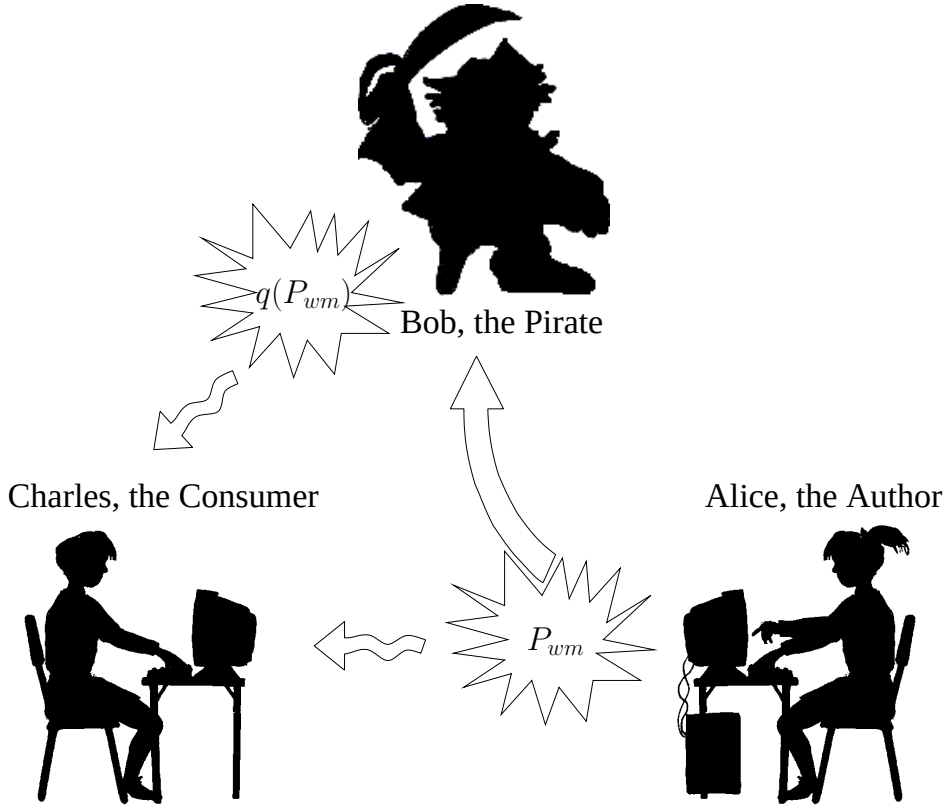


Figure 1.1: Overview of watermarking

or a *cropping function*. An obfuscation function produces code which is semantically equivalent to the original, however, is harder to understand. A cropping function removes some section of the original program such that the resulting program remains executable but with possibly less functionality.

Our aim in this thesis is to build a resilient watermarking technique that Alice can use to prove that Bob's program contains her watermark and thus must contain at least some part of her software. Such a watermarking technique could serve to give Alice approximate information on the nature and extent of Bob's piracy and be a part of the evidence required to prove Bob's guilt in a court of law.

1.3 Static vs Dynamic Analysis

Alice will want to use a watermarking technique that is *resilient*. We mean the watermark persists in spite of Bob's deliberate attempts to remove it. Bob has at his disposal a range of tools which he can use to analyze P_{wm} to discover and remove the watermark. Examples of such tools include automatic obfuscaters, decompilers, program slicers, optimizers, debuggers and interpreters. Some amount of time, cost and level of expertise is required for Bob to use each of these tools. What is more there is a trade-off in the amount of

knowledge Bob can gain versus the expense of learning to use the tool, the cost of the tool and the time taken for it to perform its analysis. A good watermark is one which is as costly for Bob to remove as it would be for him to develop the program himself.

Most analysis tools that are currently usable are based on deducing properties of a program without executing it. Static analysis tools are used by compilers, and occasionally by reverse engineers, and have been well studied. An alternative approach to analysis, dynamic analysis, executes a program and traces its execution. Although some dynamic analysis tools such as debuggers are easy to use and are readily accessible, dynamic analysis tools that do sophisticated analysis of programs are difficult to produce and have been prototyped but not fully implemented. Furthermore, because dynamic analysis depends on input to the program being analysed, in general it reveals information about only one path through the program at a time. Using dynamic analysis to gather enough information to attack a long running program with many possible inputs would take an attacker many runs of the program and potentially a long time.

For these reasons, in this thesis we limit Bob to using solely static analysis tools when attacking our author Alice's watermark.

1.4 Using Complexity

The aim of an attacker attempting to statically analyze a program is to arm himself with sufficient knowledge of the program, and the watermark, to allow him to remove the watermark without destroying the correctness of the program. To hinder an attacker we can decrease his ability to analyze the program by increasing the program's complexity.

In this thesis we propose a technique to increase the complexity of the program by introducing contending threads. The altered program is semantically equivalent to the original.

The introduced threads serve a dual purpose: watermarking and obfuscation. When the watermarked program is executed with a key input, the dynamic pattern of execution of threads encodes a distinctive watermark. As a result, a watermark is encoded into a program that is semantically equivalent to the original, but more difficult for an attacker to analyze and thus attack.

1.5 Solution Overview

The initial insight of this thesis can be summarized as follows: To remove a watermark from an application, an attacker must analyze an application sufficiently to be able to apply a semantic-preserving transformation that makes the watermark unrecognizable. Our proposed solution is to develop a watermarking technique which in addition to embedding

information efficiently increases the complexity of the software and the cost of subsequent analysis.

Our idea is to embed new threads into single-threaded portions of the program. The locations of the multi-threaded portions are selected carefully, such that when given key input the dynamic behavior of the threads is distinctive and encodes a watermark. The locations in a program where a watermark is embedded are chosen by tracing the execution of the program on the user selected input.

The dynamic nature of the watermark recognition gives our technique good resilience against static attacks. The two most potent attacks that we defend against are listed below.

1. An attacker attempts to remove unnecessary threads from a program thus disrupts recognition.

2. An attacker uses statistics or other means to pattern match the watermark and remove it.

The main result of this thesis is a method for watermarking software with a small cost in performance and demonstrable resilience to static attacks under reasonable assumptions. The most severe shortcoming in security of our proposed solution is its heavy reliance on *opaque predicates*. Informally, opaque predicates are expressions whose value is known to the author of the predicate but is difficult for anyone else to deduce. Collberg [9] and others have suggested manufacturing opaque predicates based on factoring numbers, pointer aliasing and other hard problems. While the predicates manufactured by these methods resist some analysis, a method of manufacturing strong and stealthy opaque predicates remains an open problem. The technique outlined in this thesis joins a list of several watermarking, obfuscation and tamper-proofing techniques built using opaque predicates which may be susceptible until this open problem is solved.

We summarise our reliance on the existence of opaque predicates as a limitation on our solution in the following way:

- Limitation 1 No attacker can perform a static analysis which will distinguish with 100% reliability between an opaque true and an opaque false predicate.

In designing and evaluating our watermark, we introduce the following two additional limitations:

- Limitation 2 An attacker is unable to guess the key input sequence.

Limitation 2 means the input space is sufficiently that an attacker is unable to guess the key that the watermark was embedded with. For large GUI programs which we are targeting, it is common for an application to require a large number of mouse clicks, mouse motion and keyboard input. In this case, Limitation 2 is not unreasonable.

- Limitation 3 The attacker is unable to construct a copy of the TBW recognition code table.

During recognition, we build a random code lookup table using a secret seed. Limitation 3 means an attacker is unable to compromise the cryptographic security of our random number generator and compute a seed given a sequence of random numbers.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows: In Chapter 2, we describe in detail the problem of software watermarking and the properties that must be possessed by a watermarking system for it to be useful. In Chapter 3, we describe the proposed thread-based watermarking (TBW) solution and present a proof of its correctness. In Chapter 4, we examine the theoretical strength of TBW. We give a thorough description of our implementation of a TBW prototype in Chapter 5 and in Chapter 6 present the results of a set of empirical experiments on this implementation. In Chapter 7 , we place this work in context by examining related work. Finally, in Chatper 8, we conclude the dissertation with a summary of the contributions of this work and future directions.

2

The Software Watermarking Problem

In this chapter we formally introduce the problem of software watermarking and identify three sets of characteristics that are desirable in our software watermarking system. We describe the categorical properties, usability properties and security properties that Alice will use to evaluate the effectiveness of a watermarking scheme. Finally we summarize the threat model and the design objectives that are desirable in our software watermark.

2.1 Software Watermarking

The fundamental purpose of a software watermark is to embed information in an application. We give the following formal expressions to relevant concepts:

Let P be a computer program that is available for manipulation and I_{key} be some valid input sequence to P. Let an integer ω be the watermark we wish to embed in P.

The function $\mathcal{E}$ is called a *watermark embedding function* and has the property that

$$\mathcal{E}(P, I_{key}, \omega) = P_\omega$$

where the output of $\mathcal{E}$ is a *watermarked program*, P_ω.

The function $\mathcal{R}$ is called a *watermark recognition function* and has the property that

$$\forall P_\omega : \mathcal{R}(P_\omega, I_{key}) = \omega$$

A watermarking algorithm, $A = (\mathcal{E}, \mathcal{R})$, is a combination of an embedding function $\mathcal{E}$ with its corresponding recognition function $\mathcal{R}$.

2.2 Categorical Properties

For a taxonomy of watermarks and their uses, the reader is referred to Nagra [1]. In this section, we identify and categorize some properties that we can choose when selecting or designing our watermark algorithm. As mentioned in Chapter 1, the purpose of our watermark is to identify Alice's authorship in the event her program is stolen by Bob, modified and resold. Such a watermark must be robust and invisible, in the sense defined below. Furthermore, in this thesis we are designing a dynamic watermark with focused embedding and blind detection, see Section 2.2.3, Section 2.2.3 and Section 2.2.5.

2.2.1 Fragile and Robust Watermarks

The first categorical property of a watermark is robustness. In some applications, robustness of a watermark is desirable and should be maximised, whereas in other applications the converse property of fragility is required. Thus there are two types of watermarking algorithms when categorized by robustness: fragile and robust.

Fragile watermarks are designed to become illegible if an attacker applies any prohibited transformation to a program. They are useful for proving the authenticity of a program. Fragile software watermarks are analogous to security watermarks on currency. These are designed so that they do get reproduced easily, for example when currency is photocopied.

In contrast, *robust watermarks* are designed to embed information in such a way that it is difficult for a pirate to distort or remove. A robust watermark will resist attacks and continue to exist in spite of attempts by an attacker to destroy it. Such information can then act as an identifier of authorship or ownership and be used to dissuade piracy. For example, by embedding the identity of the customer who purchases a piece of software, we can increase the probability of an illegal distributor getting caught.

To meet her need to identify the pirate of her software, Alice requires a robust watermark.

2.2.2 Visible and Invisible Watermarks

A second categorical property of a watermark is its visibility. A *visible watermark* describes a watermarking algorithm (defined above as $A = (\mathcal{E}, \mathcal{R})$) in which the recognition function $\mathcal{R}$ is public knowledge. Thus everyone is able to read a visible watermark.

In contrast an *invisible watermarking* algorithm is one in which the recognition function (or some critical component thereof, such as an encryption key or the "location" of the mark) is *not* public knowledge. Invisible marks are intended to be recognizable only by the watermarker.

The purpose of Alice's watermark is to identify to her and possibly to a court of law, who the pirate of her software is. To meet Alice's intent it is not necessary for her customers to be able to identify her watermark. In fact, making her watermark public would help the pirate since he would be able to tell once his distortion of Alice's software had removed her watermark. Thus Alice requires an invisible watermark.

2.2.3 Static and Dynamic Watermarks

A third categorical property of a watermark is how it is implemented. As described in the previous chapter there are two types of watermarks: static and dynamic. A static watermark is embedded in either the data or code section of a program. It does not require the program to be executed for the watermark to be recognized. In other words, the recognizer of a static watermark $\mathcal{R}$ takes only one parameter P_ω. A dynamic watermark on the other hand must be executed by $\mathcal{R}$ for the embedded watermark to be recognized, thus in general an additional parameter (suitable input to the program) must also be provided.

As discussed in Chapter 1, our design goal is to build a dynamic watermark.

2.2.4 Informed and Blind Watermarks

A fourth categorical property is the information needed to identify the watermark. To extract a watermark, both informed and blind watermark recognizers require the watermarked program and possibly a key. In contrast to a blind recognizer, an informed recognizer also requires the unwatermarked version of the program, the embedded watermark, or both.

While it is possible for Alice in our scenario to use either informed or blind watermarks to protect her watermark, blind watermarks have the added advantage that the original unwatermarked program does not need to be maintained or produced during recognition. This advantage can be significant if Alice needs to prove her ownership in an untrusted environment where she cannot be assured that her unwatermarked program will not be copied. Losing the original unwatermarked copy is catastrophic for Alice since a pirate can watermark it and claim it as his own leaving Alice with little recourse. For this reason we chose to build a blind watermark.

2.2.5 Focused and Spread Spectrum Watermarks

A fifth categorical property is how the watermark is embedded. Spread spectrum watermarking for software is based on a similar technique in image watermarking. An watermark embedder may be *focused* and make changes to a small sections of a program (which we will call the *location* of the watermark) or it may be *spread spectrum* and change the entire application.

Alice's needs can be met by both focused and spread-spectrum watermarks. In this thesis, however, we will be designing our watermark using thread-based widgets which lend themselves more easily to building focused watermarks.

2.3 Usability Properties

When designing a new kind of watermark, we are interested in evaluating its usability. For a watermark to be useful it must have a small impact on the execution time of application it is embedded in, maintain the correctness of the application and be easy to embed. We are interested in knowing how much data can be embedded and finally, we would like the embedding of such a watermark to be easy to automate to minimize the cost of embedding.

2.3.1 Correctness

For a program P that is being watermarked to remain useful after it has been watermarked, the watermarked program P_ω must exhibit the same observable behavior as P.

In other words, for all input I to P if running P with I terminates and produces a result x then running P_ω with I must also terminate and produce result x.

2.3.2 Developer Cost

Before a watermark can be embedded into software, the software may need to be "prepared" in some way. This generally involves annotating the source code to indicate the locations where the watermark should reside, the parts of the source code that it should avoid (for example, highly optimised or extremely fragile sections of code) and the information that should be embedded. Depending on the watermarking scheme, the cost of this preparation phase may vary.

Furthermore, some watermark schemes may require extensive interaction with the end developer, either during development or during the watermarking phase. Such interaction with the developer is expensive.

Conversely, some watermarking technologies lend themselves easily to being automated. Such watermarking schemes are cheaper to adopt and easier to integrate with an

application's development process.

We are not aware of any quantitative research on the developer time costs of various techniques for software watermarking. However we would expect to see a tradeoff between resilience and developer time.

2.3.3 Machine Cost

Once developers have annotated the software, there is a time cost involved in running the watermarking algorithm. The two watermarking operations of embedding and recognition occur at different times.

In most applications, a time consuming embedding method would be acceptable in exchange for other beneficial properties. This is because most software is produced slowly. However, for other applications, such as livestream video or audio, a fast embedding method would be critical.

Similarly, the need for fast recognition of watermarks vary from application to application.

For some applications, it may in fact be desirable to have a recogniser that works slowly in order to stall oracle attacks. An *oracle attack* is where an adversary has access to the recogniser and makes small changes to the software until the watermark recogniser fails. Such a system would be particularly beneficial for watermarks that would need to be recognised only occasionally.

In our scenario, where Alice would like to detect attempts by Bob to alter and resell her software, we are willing to accept some slowdown in embedding and recognition times.

2.3.4 Platform Independence

For maximum effectiveness, a watermarking technique should rely on minimum specialized hardware. Furthermore, it ought to be easy to port to new platforms.

2.3.5 Performance

The runtime cost of a watermark is the increased computing resources consumed by a watermarked software as compared to the original. Furthermore inserting software watermarks can result in software that runs more slowly than the unwatermarked version.

To evaluate a watermark, Alice is interested in the increase in size of a program, the increased CPU time and the increased elapsed (wall clock) time a watermarked program requires as compared to the same software without watermarks.

2.4 Security Properties

Once we have selected and built a watermark with adequate performance we evaluate how well the watermark performs in achieving its security objectives. In particular, we evaluate the resilience of a watermark, its credibility and its stealth.

2.4.1 Resilience

We are interested in how the recognition function will operate on watermarked objects that undergo various transformations such as those occurring in data compression, decompilation or obfuscation. *Resilience* is a measure of the *false negative rate* of a watermark, that is, the chance that a watermark becomes "illegible" once some semantic-preserving transformations have been applied. We formalize our definition of "legible".

We say that a watermarking algorithm is *legible* after a semantics-preserving transformation $T : P_\omega \rightarrow P'_\omega$ if

$$\forall P_\omega : \mathcal{R}(T(P_\omega), I_{key}) = \mathcal{R}(P_\omega, I_{key})$$

The type, complexity and cost of the set of transformations $\mathcal{T}$ that make a watermarking algorithm illegible is a measure of its resilience. Alice needs a watermark algorithm that remains legible unless the cost and complexity of T is very high. In the terminology of decision theory, a resilient watermark has a low false negative error rate under adversarial conditions.

2.4.2 Credibility

The *credibility* of a watermark is a measure of its *false positive rate*, that is, the chance that an unwatermarked program appears to contain a watermark. There are two types of false positives: non-malicious and attacked. A *non-malicious false positive* is a unattacked unwatermarked program which appears to contain a watermark. For a watermark to be reliable as a good indicator of ownership, Alice needs the non-malicious false positive rate to be low. Unless the non-malicious false positive rate is low, Alice will recognize her watermark in a large number of programs which are not hers and will have difficulty arguing in a court of law that finding her watermark in Bob's program implies that he pirated her software.

An *attacked false positive* is an attacked unwatermarked program which appears to contain our watermark using either his own or our key sequence. A low attacked false positive rate is important to prevent an attacker from falsely authenticating his own software as having being written by someone else. In our scenario, Alice is interested in proving that Bob has pirated at least some part of her software. In this situation, the attacked false positive rate may not be relevant to Alice.

2.4.3 Stealth

Stealth quantifies the difference between the types of instructions used to embed a watermark and those used for other program computations [10]. It is a measure of how similar the properties of the watermark are in comparison to the software it is embedded in. A stealthy watermark is more difficult to detect than an unstealthy one and thus may be harder to remove.

Unfortunately, stealth is highly subjective and good metrics are difficult to devise. Identifiable properties of the watermark such as the frequency of using specific instruction opcodes differ depending on the programming language and platform.

We say a program is *statically stealthy* if an automated inspection by a static analysis tool is unable to distinguish a watermarked program from an unwatermarked one.

We say a program is *dynamically stealthy* with respect to a given input sequence I_0 if an automated inspection by a dynamic analysis tool is unable to distinguish a watermarked program executing on I_0 from an unwatermarked one executing on I_0. Dynamic analysis tools such as debuggers and profilers pose a significant threat to our watermarking technology.

We say a program is *generally stealthy* if an expert with knowledge of the watermarking technique armed with any number of tools is unable to distinguish between a watermarked and unwatermarked program.

2.5 Attack Model

A successful attack by Bob against Alice's program P_ω causes Alice's watermark detector to make an error. As noted in Section 2.4.1 and 2.4.2, Alice need only defend her detector against false negative attacks, that is, against attacks which prevent her detector from recognizing her watermark. Furthermore, she does not have to be very concerned about attacks by Bob which change the semantics of her watermarked program P_ω, for such attacks will leave Bob with a damaged program. We assume that Bob has access to the watermarking algorithm A that Alice used to watermark her program. Bob does not have access to either Alice's watermark, knowledge of the key input or information about the exact location of the watermark in Alice's program.

While Alice may wish to protect a variety of programs, including programs that are very short or simple, we will not aim to protect such small applications (¡ 10,000 lines). This is because some of the security of a watermarking technique derives from obscurely embedding an identifier and increasingly the complexity of analysis required to identify and remove it. Small applications do not provide adequate opportunities to embed such complexity stealthily. Similarly, programs that take no input or who execution is not strongly dependent on input have a small execution space. Such programs also lack the

complexity we require to embed a robust watermark stealthily.

In contrast, large GUI programs which provides varied functionality, not all of which is executed during a typical run of the program are ideal targets for embedding a watermark. The large input space and optional functionality means that the watermark will not be expressed on every run of the program and input may be used as a key for the watermark. For this reason, we will design our watermark algorithm primarily for use with large GUI programs.

Finally, Bob only has access to Alice's compiled program and not to her source. In our scenario, Bob acquires access to Alice's program only after she has sold it to Charles. This may not prevent Bob from reconstructing some parts of the source by analysing the binary, especially if the program is distributed using an expressive target language such as Java bytecode [11]. We note, however, that complete reconstruction of the source from its compiled form is in general uncomputable [12].

There are two general attack models we are concerned with in this thesis: general attack and targeted attack.

2.5.1 General Attack

In a general attack, the attacker applies semantic-preserving transformations uniformly over P_ω. This attack is applied without detailed information about the location of a watermark in an application.

There are two types of general attack depending on the intent of the attacker. In a general distortive attack, an attacker applies transformations that may obfuscate, optimize and reorder the code in order to distort the watermark and confuse the recognizer. Alternatively, with an general additive attack, an attacker introduces spurious structures into a program that a recognizer falsely believes to be a part of the programs watermark. As a result, the recognizer recognizes a different watermark from the one that was embedded.

2.5.2 Targeted Attack

In a targeted attack, the attacker starts by attacking the watermarks stealth. If this attack is successful, the watermark has been located and the attacker then proceeds to crop or distort it such that the recognizer fails to detect it. An attack on stealth usually requires more detailed analysis of the program than a general attack, and in this respect it is more difficult to perform.

2.6 Summary

The purpose of our software watermark is to allow our software author, to identify Bob, the pirate of her software. The watermark must allow Alice to achieve this purpose in spite of the Bob's efforts to distort the program. Such a watermark must be robust, invisible and dynamic.

In order for a watermark to be usable, Alice must consider the impact of watermarking on program correctness, and the cost in developer time, machine execution time and reduced runtime performance of her software. Ideally for the watermark to be usable in a large number of settings the watermarking technique should be platform independent.

When evaluating the security properties of a watermark, Alice is interested in the resilience, credibility and stealth of a watermarking technique.

The pirate Bob knows the watermarking algorithm that Alice uses and will attempt to transform either the entire application, or attempt to target the watermarked locations in the program by analyzing the application.

In the next chapter, we describe a novel watermarking technique that can meet Alice's purpose. To achieve this purpose, the watermarking algorithm must first robustly embed a watermark into a program with a small impact on performance and second make static analysis of the resulting program difficult.

3

Threading Software Watermarks

His locks are the key to his strength.
– Delilah of Sorek

This chapter presents the design of *thread-based software watermarks*. Some parts of this chapter have been published previously in conference proceedings [3] and as patent [5, 6]. The core of thread-based software watermarking is a semantics-preserving transformation of programs which introduces new threads and locks into a program in such a way that the resulting program has a distinctive pattern of thread execution. The particular dynamic pattern that is embedded into a program encodes a watermark.

Our thread-based watermark has two major goals as discussed the previous chapter. The first goal is to robustly embed a watermark into a program with a small impact on performance. The second goal is to make static analysis of the resulting program sufficiently difficult such that a successful attack against the embedded watermark becomes computationally expensive.

The design of thread-based watermarks is based on the assertion that program-specific information is needed to successfully tamper with or remove an embedded software watermark. This assertion is implicit in the design of many proposed watermarks in literature and was explicitly asserted by Wang [13]. Wang argues in the context of obfuscation that in order to "mount a successful intelligent tampering or impersonation attack", the

adversary must "acquire information on program semantics". She further argues that by obstructing program analysis, we can prevent an attacker from gaining sufficient information about a program's semantics to launch a successful attack. Wang limits her attention to obstructing static analysis of programs which are the easier, better researched and more common type of analysis that is performed. In this thesis, we also limit our attention to static analysis of programs.

For software watermarking, a similar argument can be made on the utility of obstructing program analysis in preventing a successful attack. Collberg [9] suggests that without knowledge of location of the watermark an attacker is limited to "applying semantics-preserving transformations uniformly" over a program. He states that if "an adversary can locate the code that builds the watermark...he can easily destroy it...". Our attack model of Section 2.5 reflects both of these cases. In either case to successfully apply semantics-preserving transformations, or to locate and remove a watermark without destroying the semantics of a program, an attacker will require analysis of the original program.

Thread-based watermarks make it expensive to perform the program analysis required to successfully extract program-specific information by increasing the number of contending and interacting threads in a program. The current literature on analysis of such multi-threaded programs suggests that static analysis will produce incomplete or imprecise information about a programs behavior. There is an awareness that multi-threading significantly complicates program analysis [14, 15, 16, 17].

The solution outlined in this chapter describes the design of thread-based watermarks and the set of semantics preserving transformations that embed a thread-based watermark while increasing the complexity of the program. We describe how to embed a thread-based watermark into a program, and how to recognize it. We also outline how the watermark can be obfuscated to make the watermark that is being embedded difficult to decipher. Finally we describe how to tamper-proof a thread-based watermark such that it cannot be easily removed.

3.1 Introduction

The idea behind *thread-based watermarking* is to embed a watermark in the threading behavior of the program. The technique relies on introducing new threads into single-threaded sections of a program.

Software programs are formed from a set of basic blocks of program code. Informally a basic block is a "sequence of instructions that can be entered only at the first of them and exited only from the last of them" [18, pp. 73]. A formal definition appears on page 33. When a computer program executes, there will be a sequence of basic blocks that are

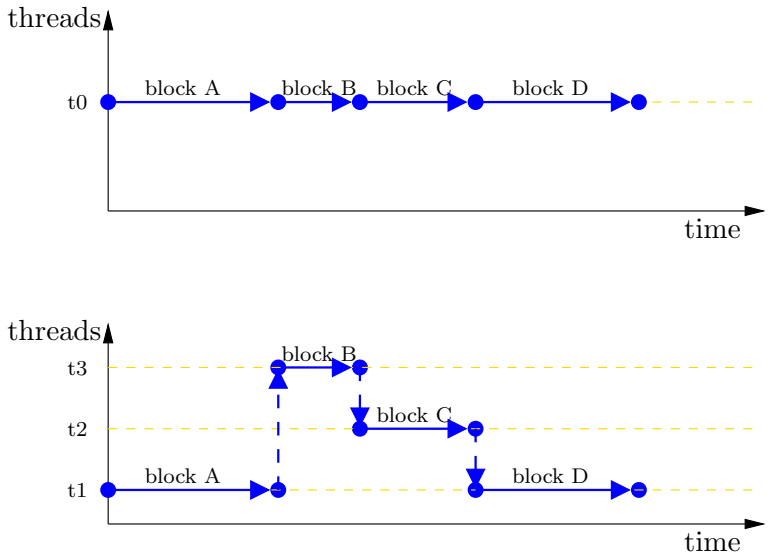


Figure 3.1: Executing two semantically equivalent programs. The top half of the figure shows a single thread $t0$ executing four blocks $blockA$, $blockB$, $blockC$ and $blockD$ sequentially. The bottom half of the figure shows the same four blocks being executed by four different threads $t1$, $t2$, $t3$ and $t4$. Note that the subsequent blocks may rely on computations performed in earlier blocks thus the different threads may be forced to run sequentially.

executed forming a path or thread through the software program. The particular path that is chosen may vary depending on the particular data that is input.

In fact there could be more than one path or thread through a software program. For such a multi-threaded program to be equivalent, in general, to a single-threaded one, we need to ensure that although different threads are executing different parts of the program, the threads are synchronized or coordinated with each other such that the same blocks of code get executed in the same order as in the original. This is illustrated in Figure 3.1. The top half of the figure illustrates a single-threaded program that executes four blocks. The lower half of the figure illustrates a semantically equivalent program with three distinct threads executing the four blocks. The dotted lines illustrate control flow constructs which will be discussed below.

We note that it is not strictly necessary for the blocks in the multi-threaded version of the program to execute in exactly the same order as in the original program for them to be semantically equivalent. If a result computed in a block of code does not depend on another block then these two blocks can be executed in arbitrary order with respect to each other in the multi-threaded program. However, these blocks could also have been reordered in this fashion in the single-threaded version.

In an unsynchronized multi-threaded program, two or more threads may try to read or write to the same area of memory or try to use resources simultaneously. This results in a *race condition* - a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing and scheduling of

threads.

One technique that allows threads to share resources in a controlled manner is using a *mutual exclusion* object often called a mutex. A mutex has two states, *locked* and *unlocked*. Before a thread can use a shared resource, it must lock the corresponding mutex. Other threads attempting to lock a locked mutex will block and wait until the original thread unlocks it. Once the mutex is unlocked, the queued threads contend to acquire the lock on the mutex. The thread that wins this contention is decided by priority, order of execution or by some other algorithm. However, due to the nature of multi-threaded execution and the number of factors that can affect the timing of thread execution, if there are multiple threads in the queue the particular thread that acquires the lock is difficult to predict and appears to be largely random [19].

We take advantage of the fact that by carefully controlling the locks in a program, we can force a partial ordering on the order in which some parts of the program are executed and thus embed some information into our program. For example, in Figure 3.1, the first statement of `block B` might be a wait on a mutex that was locked until the last statement of `block A`.

Figure 3.2 illustrates the idea of embedding watermarks using the threading behaviour of a program. The central panel in Figure 3.2 illustrates the original program segment, P which consists of three blocks executed in sequence, `A`, `B` and `C`. The panels on the left and right side of Figure 3.2 consist of two alternate multi-threaded semantically equivalent versions of P. In the left hand panel, different threads execute each of the blocks `A`, `B` and `C`. In the right hand panel, the same thread executes `A` and `C` and a different thread executes `B`.

3.2 Multi-threading

Our model of multi-threaded computation is an extension of the abstraction of multi-threading described by Leiserson and Prokop [20].

In their model, a thread is the smallest schedulable unit and defines a path of execution within a program. Leiserson and Prokop illustrate their model with a multi-threaded procedure for calculating Fibonacci numbers similar to the one shown in Algorithm 1.

The Fibonacci algorithm uses three thread control functions named **spawn**, **join** and **fetch•from**. The **spawn** function on line 5 takes, as input, a list of procedures. It starts a new thread for each procedure, and returns a list of references to these threads. In our example the function $fib(n-1)$ will execute in parallel with the procedure $fib(n-2)$ as well as the parent procedure $fib(n)$ itself. Unlike an ordinary function call, however, where the parent is not resumed until after its child returns, in the case of a **spawn**, the parent can continue to execute in parallel with the child. In general the parent can

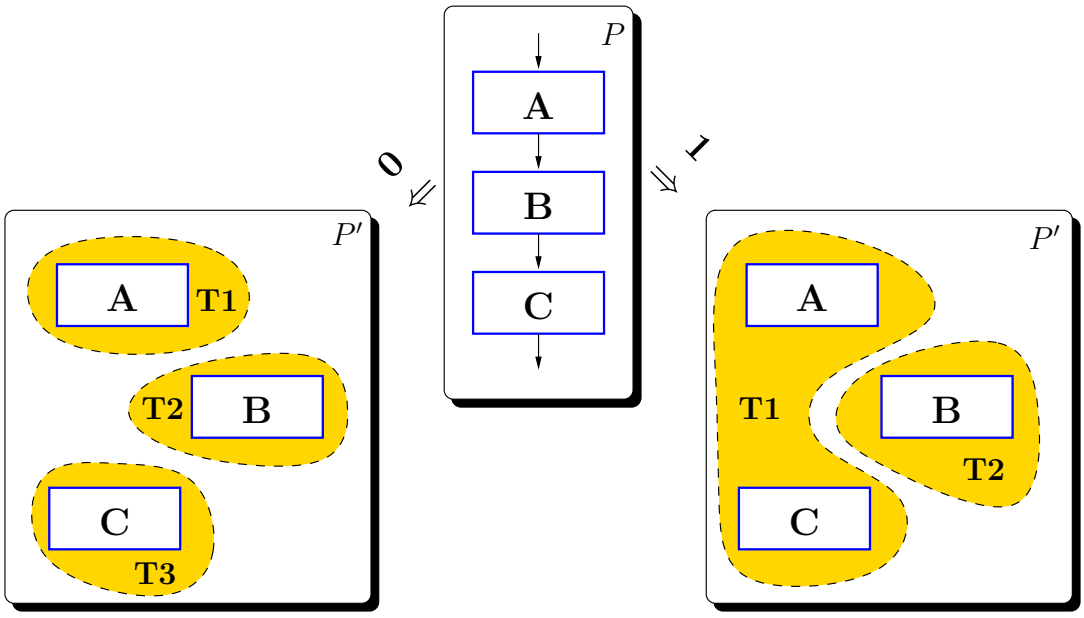


Figure 3.2: Overview of thread-based watermarking. The central panel shows a program P which contains three basic blocks, `A`, `B` and `C` which are executed sequentially. The left and right panels show two alternate semantically equivalent programs P'. The left hand panel has three different threads execute `A`, `B` and `C`. The right hand panel has the same thread execute `A` and `C` while a different thread executes `B`. This behavior is dynamically distinguishable and thus can be used to encode a watermarking bit.

Algorithm 1 Fibonacci number generator

```
1:  function  fib ( integer n ) → integer
2:  if n < 2 then
3:     return n
4:  else
5:     (Thread_1, Thread_2) ← spawn fib(n - 1), fib(n - 2)
6:           // Threads 1 and 2 execute in
7:           // parallel with parent thread
8:     join Thread_1, Thread_2
9:     f_{n-1} ← fetch•from Thread_1
10:    f_{n-2} ← fetch•from Thread_2
11:    return f_{n-1} + f_{n-2}
12: end if
```

continue to spawn children, producing a high degree of parallelism.

Every **spawn** function is paired with a corresponding **join** function. In our example, the parent executes a **join** instruction on line 11. The **join** keyword takes as its argument a list of threads and waits for all the threads to complete executing. When all of its children return, execution of the block resumes at the point immediately following the **join** statement. In our example, execution of the parent thread will block until $fib(n-1)$ and $fib(n-2)$ return.

Finally, a thread uses the **fetch•from** keyword to access the return value of a child

thread. A program cannot safely use the return values of the children it has spawned until it executes a **join** statement with the threads whose values it needs to use. In our example, the **join** statement in line 8 is required before the return statement in line 7 to avoid the anomaly that would occur if f_{n-1} and f_{n-2} were summed before each had been computed.

For our purposes, we introduce an additional keyword which is not illustrated in Leiserson and Prokop's example. The **is•alive** keyword performs a boolean test on whether a thread is still executing. In our example calling **is•alive** $Thread_1$ or **is•alive** $Thread_2$ after the threads are spawned on line 5 but before the **join** on line 7 may return either **True** or **False**, depending on whether the thread has completed its computation at the time **is•alive** is called. After the **join**, **is•alive** is guaranteed to return **False**.

3.2.1 Formal Model of Multi-threading

Before we describe our thread-based watermarking technique, we formalize our notions of a program and semantic-preserving transformations.

A program is divided into a set of procedures. We do not make the distinction between those procedures which return values (sometimes called functions) and those that do not. Each procedure consists of sequence of instructions. These instructions may be either machine instructions, bytecode instructions or more generally, minimal statements for the machine being worked on. The instructions of a procedure can be represented in a control flow graph (CFG), consisting of nodes of basic blocks and edges representing potential flow of control.

Definition 1 (Basic block)**.** A *basic block* B is a sequence of instructions $\langle b_0, b_1, ..., b_n \rangle$ where $\forall 1 \leq i \leq n-1$, b_i is not a spawn/join instruction, a call/return statement, a lock/unlock statement or a branch statement or target. The first statement b_0 is called the leader and it maybe the target of a branch. The final instruction b_n in every basic block is either a spawn/join instruction, a branch instruction, a call instruction, a return instruction or an exit instruction. If b_n is a branch instruction it is either an unconditional branch which has a single target, or a conditional branch which has several potential targets.

The instructions $\langle b_0, b_1, ..., b_{n-1} \rangle$ of a basic block include variable declarations and assignment statements.

The spawn and join instructions are considered special and are the only instruction in their basic blocks.

A procedure is divided by branches into basic blocks. Control flow enters B at the beginning and exits at the end with no possibility of branching except at the end of the basic block. [21]

Definition 2 (Control flow graph)**.** Let $CFG = \langle V, E, b_{start}, b_{end} \rangle$ be the *control flow graph* of a procedure f. The CFG consists of a V, the set of basic blocks in f and $E \subseteq V \times V$ a set of directed edges where $(u, v) \in E$ iff u contains a branch instruction with a target v. The node $b_{start} \in V$ has no predecessor and $b_{end} \in V$ has no successor. All other nodes in V have at least one predecessor and at least one successor.

Definition 3 (Predecessors and Successors)**.** The *successor* function $succ : V \to V$ is defined by $succ(v) = \{u | (v, u) \in E\}$ and the *predecessor* function $pred : V \to V$ is defined by $pred(v) = \{u | (u, v) \in E\}$.

Definition 4 (Path)**.** A *path* is a sequence of nodes $\langle v_0, ..., v_n \rangle$ such that $\forall_{0<i<n}(v_i, v_{i+1}) \in E$. Given two nodes u and v, let $[[u, v]]$ be the set of all paths with first node u and last node v.

A node v is *reachable* from a node u if $[[u, v]] \neq \emptyset$. We say a node v is *between* a node u and a node w if $\exists p \in [[u, w]]$ such that $v \in p$.

A node v *dominates* a node w iff $\forall p \in [[b_{start}, w]]$, p contains v. A node v is *post-dominated* by a node w if $\forall p \in [[v, b_{end}]]$, p contains w.

The sections of code on which we will be applying our transformations and the resulting code will be restricted to control flow graphs which contain no loops. We call such control flow graphs cycle-free.

Definition 5 (Cycle-free Path)**.** A path in a CFG, $p = [[u, v]]$ is *cycle-free* if every node $w_x \in p$ occurs just once ie. the only path from u to v is the trivial one.

On a cycle-free path if a node v dominates another node w then v executes before w. Furthermore, if a node v post-dominates a node w, node w executes after v. To capture these semantics, we introduce a set of temporal dependence edges $\varepsilon \subseteq V \times V$. An edge $(v, w) \in \varepsilon$ is called a temporal edge and implies that if a node v is executed at time t_v then w is executed at time $t_v + j$ for $j > 0$.

A thread T_x is a single sequential flow of control within a program. Each thread is associated with a control flow graph $CFG_T = \langle V_T, E_T \rangle$.

A concurrent program $\langle \zeta, \varepsilon \rangle$ is composed of a set $\zeta = \{T_0, T_1, ..., T_x\}$ of such threads, each with its own control flow $CFG_{T_x} = \langle V_{T_x}, E_{T_x} \rangle$; and a set of temporal dependencies ε. A temporal edge $(u_{T_i}, v_{T_j}) \in \varepsilon$ is a directed edge between a node $u \in T_i$ and the $v \in T_j$. and means that node u *must* be executed *before* v. There are two instructions that introduce such temporal dependencies between programs, spawn and join.

1. There is a temporal edge in ε between every spawn node and the corresponding b_{start} of the thread spawned.

2. There is a temporal edge in ε between every b_{end} and the corresponding join node.

3.2.2 Semantic-Preserving Transformations

We would like our watermarking transformations to be semantics preserving. The intuitive meaning of semantics-preserving we would like to capture is "when a user cannot distinguish whether a transformation has been applied to an application except by measuring performance".

We adopt our definition from traditional compiler parlance, where *semantics-preserving transformations* preserve the input-output behaviour of the program. This means that the program produces the exact same results if given the same input [18]. Unfortunately, in the case of some event-driven and multi-threaded programs the input-output relation can be ill defined. The existence of race conditions can result in a program whose behavior is highly dependent on the particular timing, order and number of instructions executed and thus seemingly semantic-preserving transformations will nevertheless change the output of the program.

Our definition of semantic-preserving accounts for this possibility by defining the execution of a program on a given input to result in a set of possible outputs that depend on the timing or other non-predictable behavior.

Definition 6 (Program execution)**.** Let P be a program and $\mathcal{I}$ be the set of inputs accepted by P. Then the result of executing P with input I, $P(I)$ is the set of all possible outputs.

Definition 7 (Program Transformation)**.** Let $\mathcal{P}$ be the set of programs. A transformation $\mathcal{T}$ of a program $P \in \mathcal{P}$ is a mapping from $P \xrightarrow{\mathcal{T}} P'$ where $P' \in \mathcal{P}$.

Definition 8 (Well Behaved)**.** An execution of a program P on input I is well behaved iff $card(P(I)) = 1$. In other words, on input I the program P has exactly one output. We denote this output O.

Definition 9 (Semantic-Preserving)**.** Given a program P that takes a set of input $\mathcal{I}$, a transformation $\mathcal{T} : P \xrightarrow{\mathrm{T}} P'$ is *semantic-preserving* if for all well-behaved $I \in \mathcal{I}$, $P(I) = P'(I)$.

3.2.3 Static Analysis

Static analysis refers to techniques designed to extract information from a static image of a computer program. In contrast, dynamic analysis extracts information by tracing the execution of a program. In general, Collberg [9] suggests that static analysis is more efficient than analyses performed dynamically.

Traditionally, static analysis has often been used by compilers for the purpose of code optimization and debuggers for the purpose of proving program correctness. In the

context of software-protection, static analysis could yield useful information for reversing obfuscation and removing watermarks from protected software.

According to existing informal taxonomies of software watermarking techniques [9, 22, 1], one can distinguish between static and dynamic watermarking.

Static watermarking stores the watermark in the program source either as data or code. As a result the watermark can be extracted from the text of the program without any need for execution. Static analysis is limited to those aspects of a program that can be deduced solely using the program source or object code.

Many static analysis problems are known to be intractable if exact results are required [23]. Heuristic methods are feasible but deliver inexact or incomplete analyses for some programs. In particular, all static analysis algorithms are either inexact or require exponential time when analyzing concurrent program execution paths [24]. There are two main techniques for statically analyzing concurrent programs: syntax-based static analysis and symbolic execution.

In syntax-based static analysis, a flow graph is constructed for each concurrent component of the program. This flow graph is very simplified, containing only synchronization and branch behavior. Using these flow graphs, a master graph of the concurrency states of the program can be built. These states contain synchronization information but no data values.

Syntax-based analysis is useful for examining some types of thread interaction. For example, syntax-based analysis can be used to find some concurrency errors such as deadlocks by examining a program's master graph. However, the approximations made by ignoring data values often results in the master graph containing states that are actually unreachable.

Another type of static analysis is symbolic execution. Symbolic execution builds flow graphs that preserve as much data value information as possible, in order to compute branch predicates. Data races can be detected in the resulting master graph. However, because it has to consider all possible data values, the exponential explosion is tremendous for typical programs, greatly limiting the applicability of symbolic execution.

The presence of pointers or heap references in a program causes great loss of accuracy since static analysis cannot always identify the identity of objects if they are subscripted by variable expressions or referred to through a chain of references [25].

Our dynamic watermarking method is designed to exploit the limitations described above. In Chapter 4 we argue that no static analysis method with these limitations can remove or modify our watermark.

3.3 Thread-Based Watermarking

Thread-based watermarking is a dynamic watermarking technique. This means that in order to recognize a thread-based watermark, a watermarked program P_w has to be executed in order for a program to be recognized. Furthermore, on execution the program is provided with an input $I = I_0, I_1, ...$ This input is called the *key input*, and would ideally be kept secret from an adversary.

The process of embedding a thread-based watermark involves four steps. Firstly the original program is annotated for tracing and executed with the key input I that the user selects. Secondly the user selects a watermark string and encodes it using some encoding scheme. Thirdly watermark code is inserted into the original program.

The fourth and final stage occurs some time later once a possible attacker has had an opportunity to attack the code. The watermarked program is executed with the key input sequence I and the resulting trace is collected. This trace is decoded to extract the watermark.

3.3.1 Tracing

We begin the tracing phase by performing control flow analysis on the input program to build up a control flow graph. This graph represents the possible paths through a program. The nodes of the graph represent basic blocks while the directed edges represent jumps from one node to another.

The trace consists of a series of basic blocks that are executed when the program is run with key input I. This series of basic blocks constitute the potential basic blocks in which the bits of a watermark can be embedded.

Algorithm 2 Tracing a program

1: Perform control flow analysis of the program P to get a control flow graph G.

2: Annotate every basic block in the control flow graph such that when executed, it stores a tuple (b_i, T_i), where b_i is the id of the basic block and T_i is the id of the thread that executes it.

3: Execute the program P with key input I, retaining a record of its trace $C = [(b_1, T_1), (b_2, T_2), ...]$.

A user selects input I that is well behaved on P. She does so by inspecting the sequence of basic blocks executed by the thread on multiple runs of the program with input I. Ideally, some basic blocks in the sequence are input dependent to make the value of the expressed watermark vary with I.

The tuples on the trace are temporally ordered, however temporal ordering can be problematic to determine when the program is executed on a multiprocessor. In this case, the trace can be collected when the program is running on a single processor.

The program trace serves two purposes. Primarily, the program trace is used to find the basic blocks that are executed by the input program when given the chosen input. These basic blocks are potential blocks to embed bits of the watermark. As a secondary purpose, the program trace counts how often each basic block gets executed and therefore helps identify tight loops, recursion and other program hotspots. There is a computational and thread switching run time cost associated with inserting new threads into the program. In view of this run time cost, it is preferable to avoid inserting watermarks into these hotspots.

The input I acts as the key and the watermark will be expressed when this input is entered. Thus the detection algorithm relies on this trace being reproducible given the input I.

Keeping this input a secret impedes an attacker who gains access to the recognizer from mounting an oracle attack [26]. An oracle attack is possible if an attacker has access to the recognizer and makes small changes to the software until the watermark recognizer fails. In this was the attacker can construct a non-watermarked version of the program.

3.3.2 Embedding

The embedding phase modifies the input code so that the watermark W can be extracted from a trace of the basic blocks executed on the input sequence I. The trace of the program P with the key input I selects a path through the program. This path through the program is an ordered sequence of basic blocks that are executed when the program is run.

To embed a watermark into a program we first encode the watermark as a sequence of bits. A watermarker can choose to encode a watermark in a sparse space to increase its credibility or to armor the watermark with an error-correcting code to increase its resilience. The result of the encoding step is an N bit encoded watermark. To embed these N bits along the path selected by I above, we select a subset of N basic blocks in the trace and apply a bit embedding transformation on them as described in Section 3.3.3.

3.3.3 Embedding a Single Bit

In order to embed a single watermarking bit in a basic block, we divide the basic block into three pieces called *thunks* as shown in Listing 4. The thunks are called `piece1`, `piece2` and `piece3` respectively. We replace the original code with the code shown in Listing 5. This new code executing with the original thread T_{orig} locks $mutex_{orig}$ then forks of three new

Algorithm 3 Embedding a watermark

1: Encode the watermark W as a sequence of bits $< w_1, w_2, ..., w_N >$.

2: Select N unique basic blocks from the trace $< b_1, b_2, ..., b_N >$.

3: **for** $i = 0$ to N **do**

4: Apply bit embedding transformation on b_i such that it embeds bit w_i

5: **end for**

identical threads T_0, T_1 and T_2. The new threads then execute `bitEncoder` in parallel. The original thread, T_{orig} waits for these threads to terminate. The call to `bitEncoder` is a macro. The macro expands to two very similar watermarking widgets - one which embeds a 0 bit and one which embeds a 1 bit. In the next section we will show how to use opaque predicates to merge the static differences between these two macros.

Listing 4 Original Program

```
⋮
1: piece1 ()
2: piece2 ()
3: piece3 ()
⋮
```

Listing 5 Embedding a single bit

Code in the original thread

```
⋮
1: doneA ← False
2: doneB ← False
3: doneC ← False
4: doneD ← False
5: lock mutex_orig
6: (T0, T1, T2) ←spawn bitEncoder(), bitEncoder(), bitEncoder()
7: while is•alive T0 and is•alive T1 and is•alive T2 do
8:    // Yield this thread
9: end while
10: unlock mutex_orig
11: join threads T0, T1 and T2
⋮
```

```
macro bitEncoder ()
 1: lock mutex_0
 2: if not doneA then
 3:    piece1 ()
 4:    doneA ← not doneA
 5:    lock mutex_1
 6:    unlock mutex_0
 7:    lock mutex_orig
 8:    unlock mutex_orig
 9: end if
10: if not doneB then
11:    piece2 ()
12:    doneB ← not doneB
13:    unlock mutex_0
14:    lock mutex_1
15: end if
16: if doneC or doneD ① then
17:    doneC ← not doneC
18:    if doneD then
19:       unlock mutex_1
20:    else
21:       doneD ← not doneD
22:       unlock mutex_1 ②
23:    end if
24: else
25:    piece3 ()
26:    doneC ← not doneC
27:    unlock mutex_0 ③
28: end if
```

Figure 3.3: Embedding bit 0

```
macro bitEncoder ()
 1: lock mutex_0
 2: if not doneA then
 3:    piece1 ()
 4:    doneA ← not doneA
 5:    lock mutex_1
 6:    unlock mutex_0
 7:    lock mutex_orig
 8:    unlock mutex_orig
 9: end if
10: if not doneB then
11:    piece2 ()
12:    doneB ← not doneB
13:    unlock mutex_0
14:    lock mutex_1
15: end if
16: if not doneC ① then
17:    doneC ← not doneC
18:    if doneD then
19:       unlock mutex_1
20:    else
21:       doneD ← not doneD
22:       unlock mutex_0 ②
23:    end if
24: else
25:    piece3 ()
26:    doneC ← not doneC
27:    unlock mutex_1 ③
28: end if
```

Figure 3.4: Embedding bit 1

In Listing 3.3 we embed a bit 0. The three new threads contend for $mutex_0$ and the winner proceeds to execute line 2 as shown in Figure 3.5. This causes `piece1 ()` to be executed by the winner while the other threads wait.

The body of the threads are identical and the cases are symmetric, we can assume without loss of generality that T_0 wins the lock. T_0 proceeds to execute line 2 and lock $mutex_1$, unlock $mutex_0$ then blocks waiting for $mutex_{orig}$ which is owned by T_{orig}. Threads T_1 and T_2 now contend for the freed $mutex_0$ and one of them wins the lock.

Once again the cases are symmetric and we assume T_1 locks $mutex_0$. T_1 now executes line 10 and thus T_1 executes `piece2 ()`, unlocks $mutex_0$ and blocks waiting for $mutex_1$ owned by T_0. At this point T_0 is still waiting on $mutex_{orig}$. Finally, T_2 locks $mutex_0$, executes `piece3 ()` unlocks $mutex_0$ and exits. At this point, T_{orig} is able to wake and unlock $mutex_{orig}$ allowing either T_0 or T_1 to wake up, release their locks and exit. Finally, T_{orig} waits until all three threads T_0, T_1 and T_2 have exited before continuing execution.

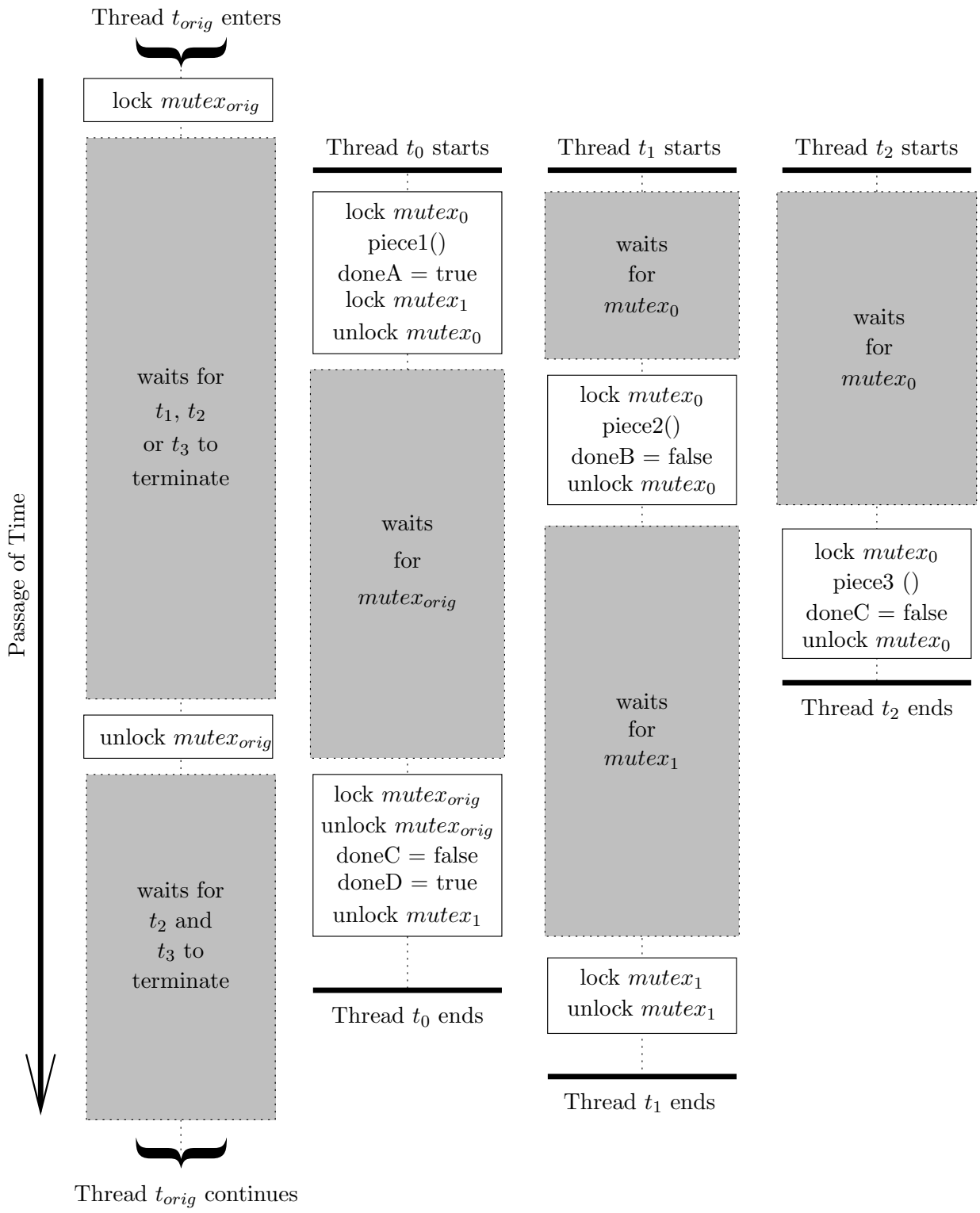


Figure 3.5: The dynamic behavior of watermarking code that embeds a bit 0. Different threads execute the thunks `piece1`, `piece2` and `piece3`. Note that all lock acquisitions and releases are well paired and no dead-lock occurs.

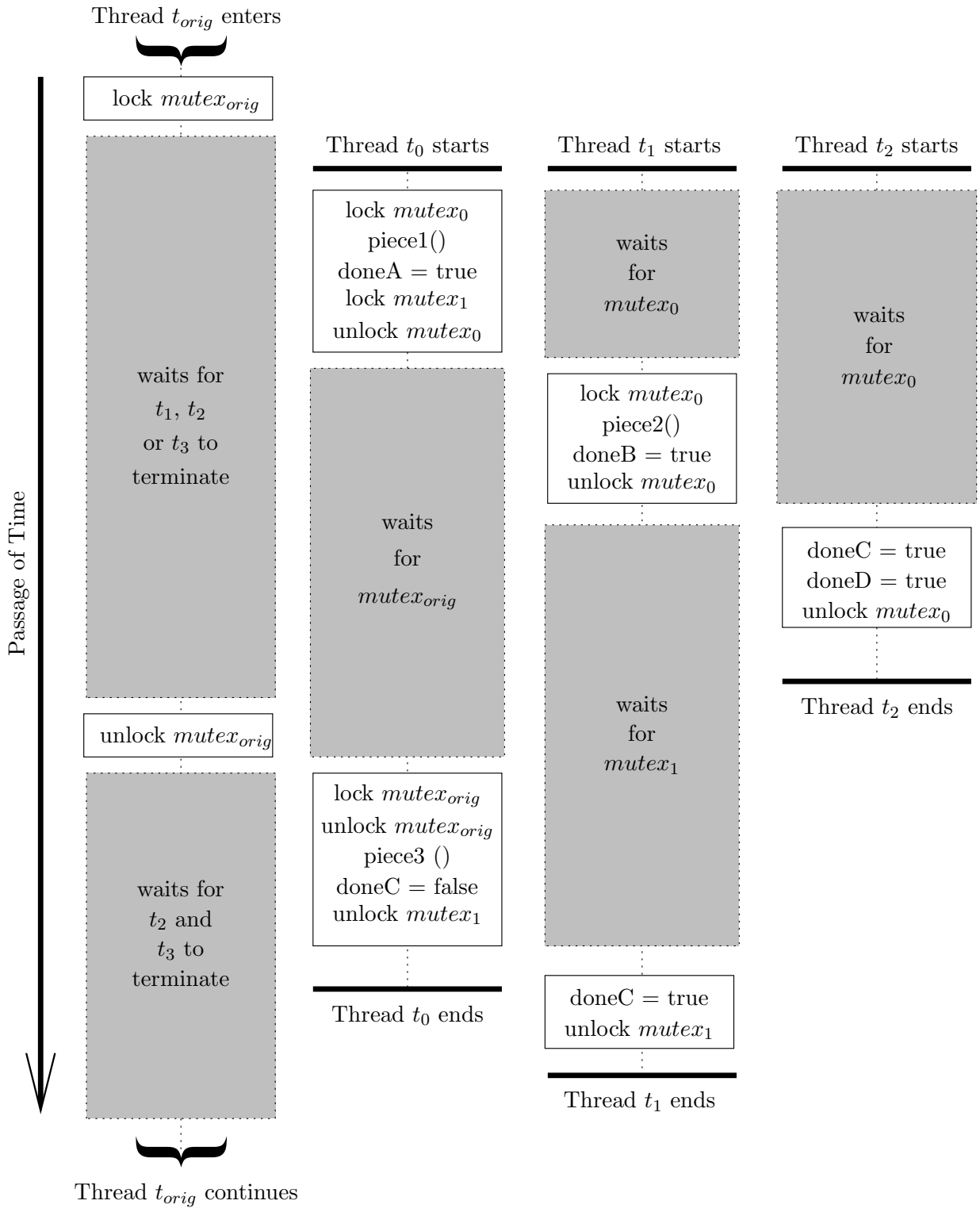


Figure 3.6: The dynamic behavior of watermarking code that embeds a bit 1. The same thread which executes the thunk `piece1` is guaranteed to execute `piece3` while a different thread executes thunk `piece3`. Note that all lock acquisitions and releases are well paired and no dead-lock occurs.

As a result of this execution, three distinct threads have executed the three pieces. We declare this pattern of threads to be embedding a bit 0.

In Listing 3.4 we embed a bit 1. The behavior of the threads is identical to embedding bit 0 until T_2 evaluates the third conditional. In this case, T_2 skips evaluating `piece3 ()` and instead unlocks $mutex_0$ and exits. As a result, T_{orig} unlocks $mutex_{orig}$ and T_0 acquires it. T_0 then executes `piece3 ()` and exits allowing T_1 to also release its locks and exit. As a result of this execution, the same thread executes `piece1 ()` and `piece3` while a different one executes `piece2 ()`. We declare this pattern of threads to be embedding a bit 1.

We will prove the correctness of these transformations and show freedom from deadlock later in this section.

3.4 Obfuscation

The thread-based watermarking technique outlined above successfully embeds a given watermark W. However, to ensure that it is truly a dynamic watermark, we need to ensure that the value of the watermark cannot be deciphered or removed with static analysis.

Note that because of the dynamic nature of tracing and the method of selecting a path using a user given input I, we already have some defense against static analysis. Even if an attacker is able to determine which basic blocks contain bits of a watermark and the value of individual bits of a watermark, with static analysis at best he is still unable to distinguish which of the $N!$ permutations is the value of the watermark.

We can gain better strength in hiding the value of the watermark by ensuring the individual bits of the watermark are not distinguishable statically. Note that the watermarking widgets have been carefully constructed such that the only differences between the widgets for embedding a 0 bit and a 1 bit are numbered ①, ② and ③ in Listing 3.3 and Listing 3.4. These differences consist of the identity of the particular locks that a thread uses and one conditional.

3.4.1 Opaque Predicates

One of the tools we use to mask the identity of the predicates and different conditionals is using *opaque predicates*. An opaque predicate [27] is an expression whose value is known to the watermarker at time of watermarking but which is not possible for the attacker to deduce.

Definition 10 (Opaque Predicate)**.** A predicate P is *opaque* at a point p in a program, if its outcome is known *a priori* by the watermarking algorithm but is difficult to deduce

by statical analysis.

We say a predicate is opaquely true, P^{true} if P always evaluates to **True** and opaquely **False**, P^{false} if P always evaluates to **False**.

Collberg et al. describe several different possible techniques for manufacturing opaque predicates [27]. These are based on pointer aliasing, algebraic properties and on concurrent threads. Although currently there are no implementations for generating a large variety of such opaque predicates, if such a library of opaque predicates were available they could be used to obscure the differences between our two bit watermarking widgets.

Using these opaque predicates we produce a new listing of our final watermarking widget as shown in Listing 6. This widget unifies the differences between a 0 bit embedding widget and a 1 bit embedding widget into a single block of code. The only difference between the two pieces of code is the value of the opaque predicates highlighted in Listing 6.

3.5 Tamper-proofing

The techniques described in Section 3.4.1 make it difficult for an attacker to use static pattern matching attacks to distinguish between watermarking code for embedding 0 or 1.

However, the code for embedding either bit is very distinctive and an attacker can easily identify those sections of code. Furthermore, he can use static pattern matching to extract out the useful sections of code, namely `piece1`, `piece2` and `piece3`. Finally the attacker can replace the identified watermarking code with a concatenation of the extracted chunks and thus reconstruct the original unwatermarked program. He can manage all of this without evaluating any opaque predicates or performing any thread analysis.

In order to thwart this attack we introduce a third widget. This widget is shown in Widget 7 and is statically indistinguishable from the two watermarking widgets introduced earlier. In fact, this tamper-proofing widget differs only in the locations identified as ①, ② and ③ in Listing 7 and 8.

As a result of the introduced code, none of the thunks `piece1`, `piece2` or `piece3` are executed in this arrangement. We can thus introduce incorrect code into these thunks confident that they will not be executed. However, since an attacker will be unable to distinguish these tamper-proofing widgets from watermark embedding widgets, he will be unable to transform them.

Listing 6 Watermarking Widget

macro bitEncoder ()

```
 1: lock mutex_0
 2: if not doneA then
 3:    piece1 ()
 4:    doneA ← not doneA
 5:    lock mutex_1
 6:    unlock mutex_0
 7:    lock mutex_orig
 8:    unlock mutex_orig
 9: end if
10: if not doneB then
11:    piece2 ()
12:    doneB ← not doneB
13:    unlock mutex_0
14:    lock mutex_1
15: end if
16: if ( Opaque Predicate and doneC or doneD ) or
    (not Opaque Predicate and not doneC) then
17:    doneC ← not doneC
18:    if doneD then
19:       unlock mutex_1
20:    else
21:       doneD ← not doneD
22:       if Opaque Predicate then
23:          unlock mutex_1
24:       else
25:          unlock mutex_0
26:       end if
27:    end if
28: else
29:    piece3 ()
30:    doneC ← not doneC
31:    if Opaque Predicate then
32:       unlock mutex_0
33:    else
34:       unlock mutex_1
35:    end if
36: end if
```

3.5.1 Proof of Correctness

A variety of methods have been proposed for reasoning about concurrent programs. However, these methods are not necessary when reasoning about our particular set of transformations. This is because although statically the resulting program appears to be

Listing 7 Tamper-proofing code in the Original Thread

```
1: doneA ← opaquelyTrue ①
2: doneB ← opaquelyTrue ①
3: doneC ← False
4: doneD ← False
5: lock mutex_orig
6: (T_0, T_1, T_2) ←spawn bitEncoder(), bitEncoder(), bitEncoder()
7: while is•alive T_0 and is•alive T_1 and is•alive T_2 do
8:    // Yield this thread
9: end while
10: unlock mutex_orig
11: join threads T_0, T_1 and T_2
```

multi-threaded there is minimal actual contention between threads.

In order to show that our transformations are correct, we need to show that the trace produced by the multi-threaded version of the program is equivalent to the trace produced by the single-threaded version in every case. Furthermore, we need to show that the resulting program is free of any introduced deadlocks.

Definition 11. Let the first thread to win the lock contention be called t_x.

Theorem 3.5.1. In the altered program P', the first thunk executed is `piece1` ().

Proof. The critical sections controlled by $mutex_0$ have varying lengths depending on which path is taken by the various threads. Once t_x wins the lock contention, the other two threads to wait on $mutex_0$. Thus t_x will see doneA is **False** and execute `piece1`. □

Corollary 3.5.2. While executing `piece1` (), t_x is the only thread that is not waiting on a lock or in a busy loop.

Lemma 3.5.3. The thunk `piece1` will execute at most once.

Proof. By Theorem 3.5.1, `piece1` is in a critical section such that exactly one thread, t_x executes it. This thread will also toggle `doneA` making `doneA` **False**. No other line of code alters the value of `doneA`, thus `doneA` will stay **False** and no other thread will execute `piece1` again. □

Lemma 3.5.4. The thread that executes `piece1` will acquire a lock on $mutex_1$ and block waiting on $mutex_{orig}$.

Listing 8 Tamper-proofing code in the New Threads

```
macro bitEncoder ()
 1: lock mutex_0
 2: if not doneA then
 3:   piece1 ()
 4:   doneA ← not doneA
 5:   lock mutex_1
 6:   unlock mutex_0
 7:   lock mutex_orig
 8:   unlock mutex_orig
 9: end if
10: if not doneB then
11:   piece2 ()
12:   doneB ← not doneB
13:   unlock mutex_0
14:   lock mutex_1
15: end if
16: if Opaque True then
17:   doneC ← not doneC
18:   if doneD then
19:     unlock mutex_0 ③
20:   else
21:     doneD ← not doneD
22:     if Opaque False then
23:       unlock mutex_1
24:     else
25:       unlock mutex_0
26:     end if
27:   end if
28: else
29:   piece3 ()
30:   doneC ← not doneC
31:   if Opaque True then
32:     unlock mutex_0
33:   else
34:     unlock mutex_1
35:   end if
36: end if
```

Proof. It follows from Corollary 3.5.2 that the thread t_x will acquire a lock on $mutex_1$. Since t_x is the only thread not waiting on $mutex_0$ or in a busy loop, it is guaranteed to win the lock on $mutex_1$ because $mutex_1$ has no other contenders.

The thread t_x then releases $mutex_0$ and blocks waiting on $mutex_{orig}$ which is locked by thread t_{orig}. □

Definition 12. Let the second thread to win the lock contention be called t_y.

Definition 13. Let the remaining thread be called t_z.

Theorem 3.5.5. In the altered program P', the second thunk executed is `piece2 ()`.

Proof. It follows from Lemma 3.5.4 that the thread that executed `piece1` will block waiting on $mutex_{orig}$. It does this having relinquished the lock on $mutex_0$. Thus the only two threads that are not blocked or in a busy loop are free to execute. Both these threads are waiting on $mutex_0$ thus will race and one of them will win this race. It follows from Definition 12 and 13 that t_y will acquire a lock on $mutex_0$ while t_z will block waiting on $mutex_0$.

The only executing thread is t_y. It sees doneA is **True** and doneB is **False** and executes `piece2`. □

Corollary 3.5.6. While executing `piece2 ()`, t_y is the only thread that is not waiting on a lock or in a busy loop.

Lemma 3.5.7. The thunk `piece2` will execute at most once.

Proof. By Theorem 3.5.5, `piece2` is in a critical section such that exactly one thread, t_y executes it. This thread will also toggle `doneB` making `doneB` **False**. No other line of code alters the value of `doneB`, thus `doneB` will stay **False** and no other thread will execute `piece2` again. □

Lemma 3.5.8. The thread that executes `piece2` will block waiting on $mutex_1$.

Proof. It follows from Corollary 3.5.6 that the thread t_y will attempt to acquire a lock on $mutex_1$ having relinquished its lock on $mutex_0$. It is guaranteed that it will block waiting on $mutex_1$ since t_x is guaranteed to already hold this lock. □

Theorem 3.5.9. In the altered program P', the third thunk executed is `piece3 ()`.

Proof. It follows from Lemma 3.5.8 that t_z is the only thread not blocked or in a busy loop. The thread will acquire a lock on $mutex_0$, see doneA is **True** and doneB is **True**. At this point there is a difference in which predicate it will see depending on whether it is executing a bit0 macro or a bit1 macro:

Executing 0 embedding bitEncoder macro The thread t_z tests `(doneC or doneD)`. Neither `doneC` nor `doneD` has been altered since it was initialized to **False** hence `(doneC or doneD)` evaluates to **False** and t_z executes `piece3`.

Executing 1 embedding bitEncoder macro The thread t_z tests `(not doneC)`. The boolean `doneC` has not been altered since it was initialized to **False** hence `(not doneC)` evaluates to **True**. The thread t_z then toggles `doneC`, sees `doneD` to be **False**, toggles `doneD` and unlocks $mutex_0$ before exiting. At this point by Lemma 3.5.4 and 3.5.8, t_x is waiting on $mutex_{orig}$ and t_y is waiting on $mutex_1$. Once t_z terminates, **is•alive** t_z becomes **False** thus t_{orig} unlocks $mutex_{orig}$ and begins waiting on t_x and t_y to terminate.

The only thread that is in a state to execute is t_x which was waiting on $mutex_{orig}$. This thread sees `(not doneB)` to be **False** by Lemma 3.5.3. Furthermore t_z toggled `doneC` to **True** as shown earlier. Thus t_x executes `piece3`, toggled `doneC` to **False**, unlocks $mutex_1$ and exits.

In either of these cases, `piece3` gets executed. □

We have now established from Theorems 3.5.1-3.5.9 that the three pieces of the watermark widget are executed in order and that if `piece1` is executed then so are the remaining `piece2` and `piece3`.

Theorem 3.5.10. The program P' is deadlock free if P is deadlock free.

Proof. For deadlock to occur it is necessary for there to exist mutual exclusion, serial acquisition of locks, no preemption and a circular dependency among the set of locks. In P' the first three of these necessary conditions are met, however, from Theorems 3.5.1-3.5.9 it follows that no circular dependency exists among the locks $mutex_{orig}$, $mutex_0$ and $mutex_1$. Thus if a deadlock occurs a circular dependency exists in the set of locks in `piece1-piece3` and $mutex_{orig}$, $mutex_0$ and $mutex_1$.

However, the locks $mutex_{orig}$, $mutex_0$ and $mutex_1$ are newly introduced locks therefore are not used by the `piece1 - piece3`. From this it follows that no new circular

dependencies are introduced as a result of our transformation. Thus if P is deadlock free P' is also deadlock free. □

Theorem 3.5.11. The program P' terminates if `piece1`, `piece2` and `piece3` terminate.

Proof. The thread-based watermarking (TBW) transformation introduces no loops and from Theorem 3.5.10 it follows that there are no deadlocks introduced by the TBW transformation. Thus the only way for P' not to terminate is if one of `piece1`, `piece2` and `piece3` fail to terminate. □

Theorem 3.5.12. The altered program P' is semantically equivalent to the original program P.

Proof. The original program consisted of the sequential execution of thunks `piece1`, `piece2` and `piece3`. From Theorems 3.5.1-3.5.9 it holds that P' also executes the thunks in the same order. □

Theorem 3.5.13. There are 6 patterns of thread execution and lock acquisitions.

Proof. From 3.5.1 and 3.5.5 it follows that the only thread contention that occurs is in the naming of the threads. There are 3! ways of assigning t_0, t_1 and t_2 to t_x, t_y and t_z hence there are 6 possible patterns of execution. □

3.5.2 Recognition

Watermark recognition involves identifying our original watermark in a possibly tampered piece of code. As discussed in Section 3.3, in our scheme using dynamic watermarking, recognition involves replaying the watermarked program with key input and decoding the watermark from the threading behaviour of the application.

First, we extract information about the threading behaviour of the watermarked program. We begin by collecting a trace of its execution on key input I, using a technique similar to the one described in Section 5.2.2. We annotate every occurrence of the instruction `lock` and `unlock` such that it records the object being locked or unlocked and

Algorithm 9 Recognising a watermark

1: Annotate every `lock` call in the program such that when executed, it stores a tuple (LOCK, obj_i, T_i), where obj_i is the object being locked and T_i is the id of the thread that executes the call.

2: Annotate every `unlock` call in the program such that when executed, it stores a tuple (UNLOCK, obj_i, T_i), where obj_i is the object being unlocked and T_i is the id of the thread that executes the call.

3: Execute the program P with key input I, retaining a record of its trace $D = [(op_1, obj_1, T_1), (op_2, obj_2, T_2), ...]$ where op_i is either the string "LOCK" or "UNLOCK".

4: The watermark is decoded from D

the thread id, T_I which executes the call. We then execute the program with input I and collect the resulting trace, D.

A subset of the `lock` and `unlock` calls in D are from the embedding of the encoded watermark. We use the distinctive pattern of lock and unlock calls to avoid having to use explicitly named basic blocks. The search through the trace D for the encoded watermark is highly dependent on the choice of encoding. We will describe in detail one implementation of encoding and recognition of watermarks in Section 5.2.1 and 5.2.8.

3.6 Summary

In this chapter we showed the design of thread-based watermarks. We outlined and proved the correctness of a set of semantics preserving transformations that embed a single bit using threads. We illustrated how such a bit embedding widget could be used to embed a multi-bit watermark.

A naive implementation of thread-based watermarks would be easy to recognize and thus attack. We also showed how to obfuscate the bits of the watermark being embedded so that an attack could not decipher the value that was encoded by the watermark. This was done using opaque predicates to mask the differences between an embedding of a 0 bit

and a 1 bit in the program. Furthermore, we introduced a technique for tamper-proofing the watermark using alternate configurations of the watermarking widget. These tamper-proofing widgets are statically indistinguishable from watermarking widgets, however, contain incorrect code that is never executed. As we will argue in the next chapter, an attacker who is unable to distinguish between watermarking widgets and tamper-proofing widgets will be unable to use pattern matching to remove all the watermarking locks in a program.

4

Theoretical Security Analysis

In this chapter, we establish a theoretical basis for evaluating the strength of thread-based watermarking. We define static analysis and show that an attacker armed with static analysis is not powerful enough to remove thread-based watermarks without risking the correctness of the watermarked program.

4.1 Introduction

In Chapter 2, we showed that there are many levels of attack open to an attacker. In this thesis, we have restricted our attention to watermarks that resist static analysis attacks. The use of aliasing and multi-threading increases the difficulty and reduces the precision of static data-flow analysis. We believe this difficulty and reduced precision of analysis can

be exploited to build a robust watermark which can be expected to resist static attacks.

In the previous chapter, we described a set of transformations that embed watermarking and tamper-proofing widgets. These also hinder static analysis using a combination of multi-threading and opaque predicates. In this chapter, we show that the result of embedding these widgets is to push the effort required to analyze and place them outside the domain of standard static analysis.

4.2 Static Analysis of Programs

We have previously described informally static analysis as analysis that extracts information from a static image of a program without executing it. This is distinct from dynamic analysis which has complete access to the inputs to the program and program traces. In reality, there are levels of abstraction between these two extremes where the analysis relies on partial execution of programs over sets of input. Such analysis of the sound approximation of the semantics a program is called abstract interpretation. For our needs, however, it is sufficient to define static analysis as analysis that relies only on information present in the control-flow of a program. More formally, we define static analysis as follows:

Definition 14 (Static Analysis)**.** Given a control flow graph CFG and a path $\alpha \in CFG$, static analysis is a theorem-prover of facts about P in CFG.

We call any analysis that cannot be performed using only the CFG, dynamic analysis.

It is important to note here that information about the actual predicates controling branches in the CFG is not used by the theorem-prover. Thus not all theorems about CFG are in fact theorems about the program. In other words, a static analyzer is not necessarily precise and some of the paths found by static analysis will not be executable paths in the program.

For example, Figure 4.1 shows a CFG for a program. The program has constraints such that when the first predicate is true, the second predicate is false. This path of

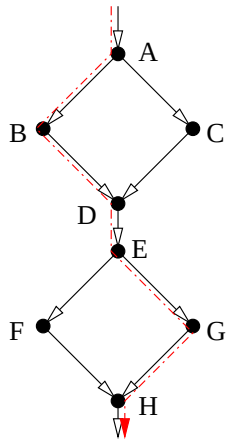


Figure 4.1: A simple control flow graph. The lines in black show possible flow of control as specified by the CFG. The dotted red line indicates the actual flow of execution. The actual flow of execution is a subset of control flow graph.

execution, $\langle A, B, D, E, G, H\rangle$, is shown as a dotted red line. However, static analysis will prove facts about all four paths of the CFG, namely $p_1 = \langle A, B, D, E, F, H\rangle$, $p_2 = \langle A, C, D, E, F, H\rangle$, $p_3 = \langle A, B, D, E, G, H\rangle$ and $p_4 = \langle A, C, D, E, G, H\rangle$. These four paths can be expressed in a regular language path expression as $p = A(B|C)DE(F|G)H$ where the actual execution path p_e is proven to be in the language p by static analysis.

In her thesis, Wang [28] argues that the aim of the attacker attempting to statically analyze a program is to arm himself with sufficient knowledge of the program to allow "intelligent tampering". As we discussed in Section 2.2.1, in our context the goal of the attacker is more specific, namely to modify the program such that it no longer exhibits our watermark.

To succeed in such an attempt, an attacker uses static analysis to:

1. Identify the location of code which builds the watermark.

2. Identify the pieces of code that depend on the watermark such that they can be replaced with semantically equivalent code without this dependence. The watermark building code can then be replaced.

This is our targetted attack of Section 2.5.2. A non-analytic "general attack" is also possible, as noted in Section 2.5.1. We will analyse this attack in Section 6.1.1.

To successfully thwart an attacker, our watermarking technique must foil one or both of these analyses. Earlier watermarking techniques relied on defeating the first of these analyses by keeping the location of the watermark a secret. If the attacker successfully identified the watermark, there was no tamper-proofing protecting it and the watermark could be trivially removed.

Unfortunately, code that builds watermarks is usually quite distinctive and easily revealed by statistical analysis and pattern matching. The approach taken by TBW is to accept that an attacker may be able to identify potential watermarking locations. Examination of the watermarking widget reveals a distinctive sequence of opcodes which an attacker could search for. Instead TBW inserts a large number of tamper-proofing widgets which are similar to the watermarking widgets but which if removed would cause the program to become incorrect. The security of the TBW technique thus crucially pivots on preventing the attacker from distinguishing between watermarking and tamper-proofing widgets.

4.3 Security Against Targeted Attacks

There are two major forms of targeted attacks that TBW is designed to prevent.

1. The attacker uses static analysis to prove that the named blocks in a watermark widget always execute sequentially. The attacker then replaces the watermark widget of Figure 6 with a basic block consisting of a concatenation of the individual pieces.

2. The attacker guesses (using statistical or exact patterns of code) the location of the watermark widgets, extracts the executed pieces using pattern matching and replaces the watermark widgets with the original code

We now show that these attacks are infeasible.

4.3.1 Proving Sequentiality

In the first attack, an attacker uses a static analysis tool to identify multi-threaded sections of code which execute sequentially, then replaces this code with its sequential equivalent.

We reiterate Limitation 1 about opaque predicates and static analysis.

Limitation 1. No attacker can perform a static analysis which will distinguish with 100% reliability between an opaque true and an opaque false predicate.

This limitation is required because constructing truly opaque predicates that are resilient to static analysis is an open problem and resolving this problem by constructing such a predicate is outside the scope of this thesis.

In order for an attacker to deduce sequentiality they need to be able to statically determine that on every execution of a watermarking widget, the three thunks named `piece1`, `piece2` and `piece3` execute sequentially. A thunk in this context is an environment and a block of code that takes no arguments.

Theorem 4.3.1. Static analysis will not reveal the sequentiality of `piece1`, `piece2` and `piece3` in a watermarking widget.

Proof. Consider the outline of a watermarking widget shown in Listing 10. This outline was obtained by replacing hiding lock manipulation and shows the possible paths through the watermarking widget.

The thunks `piece1`, `piece2` and `piece3` occur once each on lines 7, 11 and 17. In each case, they are guarded by an opaque predicate on line 2, 10 and 14 respectively. By Assumption 1 an opaque predicate cannot be statically determined to be true or false. From this it follows, that a static analyzer will deduce that the path expression of the code is $B_1(\epsilon|B_2)B_3(\epsilon|B_4)B_5(B_6|B_7)B_8$ which includes all paths with zero or one executions of each piece. Thus due to Assumption 1, static analysis is insufficiently precise to allow an attack. □

Theorem 4.3.1 formalizes the notion that an attacker cannot use a static analysis tool to prove that a watermarking widget is in fact merely straight line code. Without addi-

Algorithm 10 Outline of a watermarking widget

```
macro bitEncoder ()
 1: doneA = Opaque False
 2: doneB = Opaque False
 3: doneC = Opaque False
 4: doneD = Opaque False
 5: ⋮
 6: if Opaque Predicate then
 7:    piece1 ()
 8:    ⋮
 9: end if
10: if Opaque Predicate then
11:    piece2 ()
12:    ⋮
13: end if
14: if Opaque Predicate then
15:    ⋮
16: else
17:    piece3 ()
18:    ⋮
19: end if
```

tional information beyond what can be provided by static analysis (or by "cracking" an opaque predicate, thereby violating Assumption 1), the attacker cannot replace watermarking widgets with their equivalent code. However, an attacker may use some other method such as pattern matching to identify the watermarking widget. We address this case in the next section.

4.3.2 Distinguishing Watermarks and Tamper-proofing

In the second attack, the attacker uses some heuristic method, such as pattern matching, to guess the location of a watermarking widget and its thunks. The attacker then replaces the watermarking widget with a concatenation of these thunks.

For this attack to succeed, the attacker must be able to distinguish between a watermarking widget, which has correct code in the thunks, and a tamper-proofing widget, which has incorrect code.

Theorem 4.3.2. Static analysis cannot reliably distinguish between a watermarking wid-

Algorithm 11 Outline of a tamper-proofing widget

```
macro bitEncoder ()
 1: doneA = Opaque True
 2: doneB = Opaque True
 3: doneC = Opaque False
 4: doneD = Opaque False
 5: ⋮
 6: if Opaque Predicate then
 7:    piece1 ()
 8:    ⋮
 9: end if
10: if Opaque Predicate then
11:    piece2 ()
12:    ⋮
13: end if
14: if Opaque Predicate then
15:    ⋮
16: else
17:    piece3 ()
18:    ⋮
19: end if
```

get and a tamper-proofing widget.

Proof. Consider the outline of a tamper-proofing widget shown in Listing 11. This outline was obtained by replacing hiding lock manipulation and shows the possible paths through the tamper-proofing widget.

Note that the differences between a tamper-proofing and watermarking widget, shown in Listing 11 and 12, is limited to the values of the opaque predicates. Thus static analysis can distinguish a watermarking widget from a tamper-proofing widget if and only if static analysis can statically distinguish an **Opaque True** from an **Opaque False**. By Assumption 1 this is not possible and thus the theorem holds. □

Theorem 4.3.2 implies that in the absence of a reliable means of "cracking" a opaque predicate (Assumption 1), static analysis is insufficient.

The complexity of the analysis faced by the attacker can be made more difficult by increasing the number of tamper-proofing widgets, or by increasing the complexity of the opaque predicates.

4.3.3 Limitations on Security Against Targeted Attacks

The proof of the two theorems given in this chapter rely heavily on opaque predicates remaining inviolate in the face of static analysis. This is the weakest aspect of TBW and the most likely basis of a successful attack against it.

No currently known opaque predicate satisfies Assumption 1: With some effort and appropriate tools an attacker can statically decipher the value of such weak opaque predicates. Once the attacker successfully manages this, the remaining attack outlined above is simple to implement.

With current technology, a watermarker may at best use a variety of opaque predicates based on different problems such as pointer aliasing [9], algebraic equalities [29] and hash functions which will require the attacker to resolve each opaque predicate with a different set of tools.

5

Implementation

Any programming language is at its best
before it is implemented and used.
– /usr/bin/fortune

In this chapter we describe the implementation of our prototype of the Thread-Based Watermarking (TBW) system. Our prototype is called **tbwer** and the version described in this thesis was completed in July 2005. The **tbwer** is implemented in Java and embeds watermarks in Java applications. The code transformations were implemented at the Java byte-code level using the analysis framework provided by Sandmark [30], a software protection tool developed by Christian Collberg at the University of Arizona.

The **tbwer** tool requires the user to select a watermark, a random seed and a key input sequence to the program. The input sequence is kept secret and is used during embedding and recognition to find basic blocks into which a watermark is embedded. The tool also inserts additional tamper-proofing code into the target program to make it more difficult for an attacker to identify or remove watermarking code. There are two types of tamper-proofing code inserted.

In this chapter, we describe the implementation of TBW transformation and the reasoning behind the specific design choices that we have made.

5.1 Implementation Platform

In implementing TBW, there are two different platforms that have to be considered. The *target platform* is the programming language and environment we choose to watermark. The *development platform* is the programming language and environment in which we implement our TBW technique.

5.1.1 Target Platform

For the TBW prototype, we chose Java bytecode as the target language for our watermarking transformations. The implementation operates on single threaded and multi-threaded programs. It performs the watermarking transformations directly on compiled Java bytecode and produces watermarked Java bytecode.

Java bytecode provides a suitable target for an implementation of a thread-based watermarking technique. The design of Java bytecode makes decompilation possible and more piracy prone [31] and thus stands to benefit from watermarking. Multi-threading is also integral to Java and the language makes it relatively simple to learn and easy to use.

Before proceeding further, however, it should be noted although the techniques described in this chapter use some specific threading features provided by Java, these techniques could be implemented on any other multi-threading platform that supports the creation of new threads, shared variables and synchronization support between threads,

In fact, as described next, Java presents some challenges that are peculiar to it and are not found in other multi-threading platforms.

5.1.2 Challenges posed by Java

In this thesis, when discussing an implementation in Java, we distinguish between the high level programming language (Java) and its virtual machine (JVM). In this thesis, we use the term Java to refer to the high level language and the term JVM to refer to the virtual machine.

The Java language provides support for complex multi-threading with its elegant approach to synchronization. Each thread has its own private stack. This stack contains primitives and object references that no other thread can access. The actual objects referred to by the object references are stored on the *heap*. This heap is shared by all threads.

Concurrent access to objects on the heap is managed using *monitors*. Every Java object has a single associated lock which can be owned by only one thread at a time. Every synchronized block has a monitor object. When a thread executes a synchronized block, it must first acquire the associated lock of the block's monitor object. Only one thread may have this lock at a time. Other threads which try to acquire the lock are put into the monitor's *wait set*. When the initial thread releases the lock on the monitor, the threads in the monitor's wait set will contend for the lock.

At the language level, Java offers two mechanisms for managing JVM monitors: the `synchronized` statement and the `synchronized` method modifier. A `synchronized` statement, shown in Figure 5.1(a), takes a object and a block as parameters. It acquires the object's lock on behalf of the executing thread, executes the block, then releases the lock.

A `synchronized` method, shown in Figure 5.1(b), is similar to a `synchronized` statement in that a lock is acquired before the method executes, and released after the method returns. The object on which a `synchronized` method locks is variable. It depends on

```
synchronized ( this ) {
  if ( a == 0 )
    a = 1;
}
```

(a) A synchronized statement

```
synchronized void foo ( ) {
  if ( a == 0 )
    a = 1;
}
```

(b) A synchronized method

Figure 5.1: Examples of the two types of synchronized code in Java

whether the method is static or virtual. Static synchronized methods lock on the `Class` object for the class that the method is in, while virtual synchronized methods lock on the object for which the method was invoked. This object is referred to as `this` in Java.

The `synchronized` construct is the only mechanism in the Java language for manipulating monitors. The syntax of this construct described above ensure two properties.

Property 1. Every lock that is acquired in a method is released before the method returns.

Property 2. All lock acquisitions and releases are properly nested.

At the virtual machine level, the JVM provide two mechanisms for implementing monitor management, mirroring the two high-level `synchronized` constructs in Java. A synchronized method is normally implemented by setting ACC_SYNCHRONIZED flag on the method. This flag is checked for by the method invocation instructions. To implement the `synchronized` statement, the JVM provides two instructions `monitorenter` and `monitorexit`, which acquire a lock and release a lock on an object respectively. The Java compiler ensures that these instructions are properly paired, such that every path through a lock call is followed by an unlock call.

Once a compiler translates high-level Java into JVM bytecode, there are no longer any syntactical restrictions required to preserve Property 1 and 2. Instead, the JVM specification defines a set of static properties which legal bytecode must possess. These properties are designed to be easy to verify and minimize the number of runtime checks that need to be performed to run Java correctly. For a complete list of these properties, the reader is referred to the JVM specification [32].

When JVM bytecode is executed, a theorem prover called a *verifier* is run first. The

verifier performs compile-time checks of the above properties.

Of interest to us are the following two properties concerning threads and locks as stated by the JVM specification.

Let T be a thread and L be a lock. Then:

Property 3. The number of lock operations performed by T on L during a method invocation must equal the number of unlock operations performed by T on L during the method invocation whether the method invocation completes normally or abruptly.

Property 4. At no point during a method invocation may the number of unlock operations performed by T on L since the method invocation exceed the number of lock operations performed by T on L since the method invocation.

By observation it can be seen these Properties 3 and 4 are equivalent to Property 1. Furthermore, at JVM level there are no equivalent constraints to maintain Property 2. The absence of such a constraint is necessary to build TBW.

The JVM specification suggests that implementations of the Java virtual machine are "permitted but not required" to enforce Property 3 and Property 4. We believe most existing implementations of the JVM do enforce these two constraints and although this feature can be selectively turned off, doing so diminishes the trust that a user may have in a piece of code. Accordingly, we designed our transformations to preserve these properties.

5.1.3 Development Platform

The TBW algorithm was implemented using the Sandmark framework for software protection, developed at the University of Arizona. The tool is implemented in Java and operates on Java bytecode. Currently, it contains implementations of ten watermarking algorithms and twenty-four obfuscation algorithms, as well as some tools to perform static analysis on Java applications. An attractive feature of the Sandmark architecture is that it allows new watermarking and obfuscation algorithms to be easily implemented and tested.

In watermarking an application, Sandmark does not use any intermediate representation. Instead it relies on the Byte Code Editing Library (BCEL) to directly edit the application bytecode. Each obfuscation and watermarking algorithm that is to be applied is free to perform arbitrary analysis and transformation. Such a loose coupling between the passes of different algorithms has the advantage of modularity, allowing new software protection algorithms to be designed and introduced into the framework easily without many dependencies on existing algorithms. Unfortunately, as discussed in the Chapter 8, the lack of a intermediate representation also means that some transformations result in a loss of precision in subsequent analysis, so subsequent transformations can be difficult or even impossible to apply.

5.2 Watermarking Java Bytecode

In this section we describe the Java implementation of the encoding, tracing, embedding and recognition stages described earlier in Chapter 3. We also give detailed descriptions of the watermarking and tamper-proofing widgets that we build during embedding to make detection and removal of the watermark more difficult.

Although the transformations were implemented at Java bytecode level, wherever possible we represent these transformations in Java for clearer exposition in this thesis. Bytecode transformations which cannot be expressed directly in Java, for example, improperly nested synchronized statements, are presented here as macros in a simplified language.

5.2.1 Encoding

The thread-based watermarking process is begun by encoding the watermark. The purpose of this phase is to represent the watermark as a bit string and armor it with an error-detecting code. In the current implementation, a watermark is a 32-bit number and we use a random lookup table to encode the watermark. The encoding algorithm is as follows:

1. Selection of a watermark W

   - The user selects a watermark bit string W which is of size N bits where N is a multiple of 16. In our implementation, $N = 32$.

2. Selection of random seed S

   - The user selects a random seed S which is used to initialize a random number generator. This seed is kept secret and used during recognition.

3. Building the code table

   - A random code table K_{code} is built that maps from 16-bits to 64-bits. This table consists of all 2^{16} key strings. Each key string maps to a unique random 64-bit value string.

4. Armoring the watermark

   - The watermark W is divided into blocks of 16-bit pieces and each piece is encoded replacing it with the corresponding value in the lookup table. The resulting bit string $W_{armored}$ is an armored bit string of length $N_{armored} = N \times \frac{64}{16} = 128$ which we will embed into the application.

The sparseness of this code gives us a strong error-detection property which we will use in our recognition step to reduce the chance of a false positive watermark: if a string is chosen uniformly at random from the set $\{0,1\}^{64}$, the probability of this string being in the code table and thus being a legal codeword is only $\frac{2^{16}}{2^{64}} = 2^{-48}$. This property of the code allows us to identify spurious bits during recognition. Furthermore, by keeping the random seed secret, it would be difficult for an attacker to rebuild the code table and successfully insert additional watermark bits into the trace.

5.2.2 Tracing

During tracing, first an input sequence I is chosen. This is the key input which causes the watermarked program P to execute the trace which reveals its watermark.

A trace file is collected using the following algorithm:

1. Instrumentation of P

   - A set of control flow graphs (CFGs) is built, one for every method in every class in P. The CFG building algorithm assigns a unique integer, **block ID** to every basic block that is discovered.
   - For every basic block in the set of control flow graphs, bytecode instructions illustrated in Figure 5.2 are added to beginning of each block. These instructions output the following information into a trace file:
     - **block ID** - an integer that is unique for each basic block across the program. The code marked `BLOCK_ID` in Figure 5.2(a) is a compile-time constant which is different for every basic block. Every time this code is prepended to a basic block, this constant is incremented.
     - **thread ID** - the ID of the thread which executes the block. As shown in Figure 5.2, this is found by querying the JVM at runtime.
   - As described in Chapter 3, the trace, C_{orig} is a sequence of tuples (**block ID**, **thread ID**) which are ordered temporally. This means that if ($block_x$, $thread$) occurs before ($block_y$, $thread$) in a trace then $thread$ executed $block_x$ before $block_y$.

2. Selecting the key thread

   - The program is run with the key input sequence I and C_{orig} is collected.
   - The programmer selects a thread ID, T_{main} in the trace. This thread is called the *key thread* and is the thread in which our watermark will be embedded.
   - A filtered trace called a *key trace* is generated using the key thread and trace. The key trace C_{key} is such that $C_{key} = \{(b,t)|(b,T_{main}) \in C_{orig}\}$

3. Verifying the reproducibility of the trace

```
ldc2_w BLOCK_ID
invokestatic java/lang/Thread/currentThread()Ljava/lang/Thread;
invokevirtual java/lang/Thread/getId()J
invokestatic mark/logToFile(JJ)V
```

(a) Bytecode prepended to every basic block in P

```
public class mark {
  File logFile;
  public static void logToFile ( long blockID, long threadID ) {
    if ( logFile == null )
      logFile = openLogFile ();
    logFile.println ( "Block: " + blockID + ", Thread: " + threadID );
  }
}
```

(b) Method that logs the trace

Figure 5.2: Code for logging the execution trace to a file for embedding

- Step 2 above is repeated several times using the same input sequence I and key thread T_{main} to ensure that an identical sequence C_{key} results each time. This step helps ensure the watermark's reproducibility.
- If the trace is not reproducible, the programmer selects a different T_{main} and retries Step 2. If no such thread ID is found, the programmer selects a new input I and reruns the algorithm.

The key trace consists of tuples of the form (b_x, T_{main}) where is b_x is a potential watermark bit carrying basic block. A sequence of $N_{armored}$ basic blocks is selected at random from the key trace. For reasons of performance, it may be undesirable to the user for the watermark to be embedded in some blocks. The current implementation supports this by allowing the user to edit the key trace file to remove undesirable target blocks before selection.

- Select $N_{armored}$ distinct blocks $B = \langle B_1, B_2, ..., B_n \rangle$ from the key trace file where the probability of a block being selected is inversely proportional to the number of times the block occurs in the trace.

The blocks selected in such a way are less likely to occur in tight loops, recursion and other program hotspots. However, the information in a trace is only representative of running the program with I as input. Future enhancements of this implementation could

try to identify more general hotspots by collecting traces from other executions of the program or by static analysis of the program.

5.2.3 Embedding

To embed the watermark we embed each bit of $W_{armored}$ into the selected $N_{armored}$ blocks. As mentioned previously, in our implementation $|W_{armored}| = 128$. For every selected block:

- Split the basic block into three pieces: `piece1`, `piece2` and `piece3`. If the basic block is too small to be split in this way, one or two of these pieces are empty.

- Replace the selected basic block in P with code as shown in Figure 5.3. The code shown uses two macros. The first macro, `buildClosure`, expands to code that constructs a closure as described in Section 5.2.4 below. A second macro, `getThread`, expands to code to fetch a thread from a thread pool as described in Section 5.2.6. This thread is assigned to the task constructed by `buildClosure`. Initially, the thread is halted. Later, it is started by a `start` method. It dies when it reaches the end of its task; this status can be checked from the main thread, by an `isAlive` method call.

- The call to `yield` causes the currently executing thread to pause and allow other threads to execute. Its purpose to minimize the amount of time spent in a busy loop for efficiency reasons.

- The call to the `join` method of each thread causes the thread executing `join` to wait for the thread it is called on to die.

- The `buildClosure` macro expands to two different pieces of code, depending on whether `wmBit`, the bit to be embedded, is 0 or 1. The code expansion of `buildClosure` when `wmBit` = 0 bit is shown in Figure 5.4. Figure 5.5 shows the expansion of `buildClosure` when `wmBit` = 1.

```
Closure task = buildClosure ( wmBit, piece1, piece2, piece3 );
Thread t1 = getThread ( task );
Thread t2 = getThread ( task );
Thread t3 = getThread ( task );
Object mutex_orig = new Object ();
synchronized ( mutex_orig ) {
  t1.start(); t2.start(); t3.start();
  while ( t1.isAlive() && t2.isAlive() && t3.isAlive() ) {
    Thread.yield()
  }
}
t1.join(); t2.join(); t3.join();
```

Figure 5.3: Code which replaces the original basic block.

5.2.4 Closures

A *closure* is an anonymous function that "closes" over its surrounding scope. This means when the function is executed, it has access to all the local variables that were in scope when it was created. The closure is a data structure that contains a block and an environment of variable bindings in which the block is to be evaluated. Closures are especially useful for callbacks because they provide flexibility in deciding what function will be called at runtime.

We use closures to allow an arbitrary thread from our pool to execute different parts of a program. Java provides a restricted form of closures using the mechanism of inner classes. The Java Language Specification [33] specifies this limitation as "Any local variable, formal method parameter or exception handler parameter used but not declared in an inner class must be declared final, and must be definitely assigned before the body of the inner class". This limitation means that only final variables are passed into Java "closures".

There have been several proposals to work around these limitations used to implement closures in Java [34, 35, 36]. In our implementation, a closure is a class that implements the `Runnable` interface which is part of the Java language API. Implementing this interface is the strategy recommended by Sun for constructing new threads. This interface contains a single `run()` method. The body of the closure is inserted into the `run()` method of the

```
boolean doneA = opaqueFalse;  boolean doneB = opaqueFalse;
boolean doneC = opaqueFalse;  boolean doneD = opaqueFalse;
Object mutex0 = new Object(); Object mutex1 = new Object();
Object mutex2 = new Object();
monitorenter ( mutex0 );
if ( !doneA ) {
  piece1;  doneA = !doneA;
  monitorenter ( mutex1 );
  monitorexit ( mutex0 );
  monitorenter ( mutex_orig );
  monitorexit ( mutex_orig );
}
if ( !doneB ) {
  piece2;  doneB = !doneB;
  monitorexit ( mutex0 );
  monitorenter ( mutex1 );
}
if ( ( ( doneC || doneD ) && opaqueTrue  ) ||
     ( (    ! doneC     ) && opaqueFalse ) ||
     opaqueFalse ) {
  doneC = !doneC;
  if ( doneD )
    monitorexit ( mutex1 );
  else {
    doneD = !doneD;
    monitorenter ( mutex2 );
    monitorexit  ( opaqueFalse ? mutex0 : mutex2 );
    monitorexit  ( opaqueTrue  ? mutex1 : mutex2 );
  }
} else {
  piece3;  doneC = !doneC;
  monitorexit ( opaqueTrue ? mutex0 : mutex1 );
}
```

Figure 5.4: Code that embeds a bit 0.

new class while the call location is replaced with an instantiation of the new class and an invocation of the run() method.

5.2.5 Opaque Predicates

The differences between a widget embedding of 0 bit and a 1 bit can be used to statically distinguish between them. We minimize such static occurrences whenever possible to make pattern matching more difficult. For example, in Figure 5.4 and Figure 5.5, we use

```
boolean doneA = opaqueFalse;  boolean doneB = opaqueFalse;
boolean doneC = opaqueFalse;  boolean doneD = opaqueFalse;
Object mutex0 = new Object(); Object mutex1 = new Object ();
Object mutex2 = new Object();
monitorenter ( mutex0 );
if ( !doneA ) {
  piece1;  doneA = !doneA;
  monitorenter ( mutex1 );
  monitorexit ( mutex0 );
  monitorenter ( mutex_orig );
  monitorexit ( mutex_orig );
}
if ( !doneB ) {
  piece2;  doneB = !doneB;
  monitorexit ( mutex0 );
  monitorenter ( mutex1 );
}
if ( ( ( doneC || doneD ) && opaqueFalse  ) ||
     ( (    ! doneC     ) && opaqueTrue   ) ||
     opaqueFalse ) {
  doneC = !doneC;
  if ( doneD )
    monitorexit ( mutex1 );
  else {
    doneD = !doneD;
    monitorenter ( mutex2 );
    monitorexit  ( opaqueTrue  ? mutex0 : mutex2 );
    monitorexit  ( opaqueFalse ? mutex1 : mutex2 );
  }
} else {
  piece3;  doneC = !doneC;
  monitorexit ( opaqueFalse ? mutex0 : mutex1 );
}
```

Figure 5.5: Code that embeds a bit 1.

`doneA = !doneA` to toggle the value of `doneA` rather than to set it explicitly.

In Chapter 3.4.1, we suggest that opaque predicates can be used to minimize the apparent differences between the two watermarking widgets. In this section, we will describe how such opaque predicates can be implemented and used.

There are several suggestions on sources of opaque predicates in literature. Collberg and others [27] suggest a comprehensive list of sources of opaque predicates. Ideally an implementation will have access to a large variety of such opaque predicates at its disposal, such that the opaquely true and opaquely false predicates closely resemble each other; then

an adversary cannot use pattern matching attacks to statically read, eliminate or modify our watermarks. Furthermore, predicates should be chosen which have instructions similar to the others in the vicinity of the predicate. This will increase its stealth. An adversary will not be able to distinguish an opaque predicate (which needs to be attacked) from a predicate in the unwatermarked code (which is presumably for program correctness, and therefore should not be modified in an attack). Although no good library of such predicates exists, the Sandmark project is currently developing one.

One source of opaque predicates is based on the pointer aliasing problem [37] which has been shown to be NP-hard. However, generating probably-hard instances of these problems has not be shown to be feasible. Furthermore, we believe current techniques for generating instances of opaque predicates could be deobfuscated using simple pattern matching attacks. The problem of generating good opaque predicates that are resilient to pattern matching attacks falls outside the scope of this thesis. In our prototype design of a thread-based watermark, we use a single type of opaque predicate. In the next chapter we show empirically that it partially fulfils Limitation 1, in that it is resilient to current analysis tools, but may succumb to future analysis tools designed specifically to defeat it.

The opaque predicate we build is based on pointer aliasing and multi-threading. The predicate is built as follows:

1. Generate data structure with known properties

   - Generate a rooted, singly-linked cycle of k nodes where the last node points to the first node as shown in Figure 5.6(a).

2. Construct true or false predicate

   - Construct two pointers α and β that refer to nodes in this cycle.
     - For an opaquely false predicate, α and β should initially refer to different nodes. This case is shown in Figure 5.6(a) by variables `a'` and `b'`.
     - For an opaquely true predicate, α and β should initially refer to the same node. This case is shown in Figure 5.6(a) by variables `a` and `b`.

3. Update while maintaining the invariant

   - Generate a new thread which asynchronously and atomically updates these pointers, such that each time α is updated to point to its next node in the cycle, β is also updated to point to its next node in the cycle. The code for this thread is given in Figure 5.6(b).

The pointers `a'` and `b'` initially refer to the same node in the cycle, while `a` and `b` refer to different nodes. The pointers are advanced asynchronously around the cycle, maintaining the invariant, either `a' == b'` or `a != b` depending on whether a opaque true or opaque false is being called.

5.2.6 Thread Pools

Three threads (in addition to the original) are required for every bit that is embedded in an application. We could construct three new threads every time and successfully manage to embed and recognize our TBW. However, this constitutes some unnecessary overhead which could be avoided by reusing existing threads.

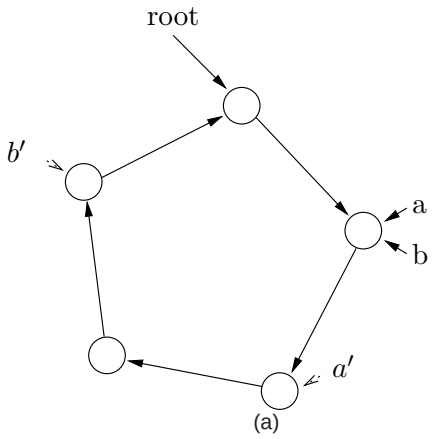
In our implementation, as shown in Figure 5.3 (see Section 5.2.3), we use a pool of threads to minimize this overhead.

1. Initialize the pool

   - A `Pool` class is constructed, and all watermark widgets which require a thread to execute, instead pass a `Closure` structure to the `Pool` for execution.
   - The first time the `Pool` is given a task, an empty array of `MAX_POOL_SIZE` is initialized. A single thread is constructed and assigned the `Closure`.

2. Emulate a single use thread

   - A thread in the `Pool` only starts executing a given task when its `start` method is called.



(a)

```
public class Race extends Thread {

  public Race ( Node a,
                Node b,
                Object lock ) {
    this.a = a;
    this.b = b;
    this.lock = lock;
    start();
  }

  public getA () {
    return a;
  }

  public getB () {
    return b;
  }

  public void run () {
    while ( true ) {
      synchronized ( lock ) {
        a = a.next;
        b = b.next;
      }
    }
  }
}
```

```
int size = getRandomBetween ( 2, 10 );
Node cycle = createCycle ( root, size );
int x = getRandomBetween ( 2, 10 );
int y = 0;
while ( x == y )
  y = getRandomBetween ( 2, 10 );
Node a = getNth ( root, x );
Node b = getNth ( root, x );
// a  == b  is transparently true

Node c = getNth ( root, x );
Node d = getNth ( root, y );
// c == d is transparently false

Object lock = new Object ();
Race race1 = new Race ( a, b, lock );
Race race2 = new Race ( c, d, lock );
```

```
macro opaqueTrue      // opaquely true
  synchronized ( lock ) {
    race1.getA () == race1.getB();
  }
```

```
macro opaqueFalse     // opaquely false
  synchronized ( lock ) {
    race2.getA () == race2.getB();
  }
```

(b)

Figure 5.6: In (a) a randomly sized cycle graph data structure that gets built. There are pointers a and b (or a' and b') which point to the same (or different) nodes in the cycle. The pointers are advanced asynchronously around the cycle. However, whenever a (or a') is advanced so is b (or b').

- Calling the `start` method marks a thread busy. Once a thread completes executing its closure, it marks itself not busy.

- The `isAlive` method for threads in the `Pool` is overridden such that it returns whether a thread is busy, and not whether it has died.

- Each time the `Pool` is passed a `Closure`, it first checks if there are any threads

in the array that are marked not busy. If there exists one, it is marked busy and assigned the `Closure`. If there all threads are busy, a new thread is constructed and added to the array. This thread is assigned the `Closure`.

Using a thread pool has a further advantage that the same pool of threads that are used to embed watermarks can also be used to perform other program tasks thus more tightly binding the watermark to the application.

One avenue to explore in future work would be to use the thread pool to execute existing tasks in a program which use new threads and to exploit the thread pool when introducing concurrency into independent parts of the program which execute sequentially.

5.2.7 Tamper-proofing Widgets

The TBW implementation also builds non-watermark embedding widgets which statically appear the same as watermark embedding widgets. These differ from the watermarking widget only in the objects on which the execution locks, and in the initial value of some of the variables. However, these differences are sufficient to alter the execution such that none of the blocks `piece1`, `piece2` or `piece3` execute.

The algorithm for embedding a tamper-proofing widget is very similar to embedding a watermarking widget.

1. Selection of locations for tamper-proofing

   - Select N_{tamper} blocks from the trace. N_{tamper} is a user-configured number of tamper-proofing widgets which will be inserted into the trace. The tamper-proofing widgets will be inserted following each selected block.

2. Embed the tamper-proofing code

   - For every selected basic block:

     - Generate three new basic blocks of stealthy, but incorrect code called `piece1`, `piece2` and `piece3`. There are many possible methods for generating these basic blocks including assigning random values to live variables,

using a basic block from elsewhere in the method that uses only live variables, or executing code that has user observable side-effects.

- Our implementation selects a random variable from the selected basic block and assigns to it either a null if the variable is of object type, or zero if it is a numerical primitive.
- Append after the selected basic block in P the code as shown in Figure 5.3 as in the watermarking widget. However `wmBit` is set to -1, causing `buildClosure` to expand to the code shown in Figure 5.7.

The current implementation of tamper-proofing basic blocks successfully introduces non-executed incorrect blocks into the code. However, the technique for generating them is trivial and it is possible for an attacker to distinguish a tamper-proofing widget from a watermarking widget by pattern matching. However, there are a large number of possible ways to generate random code.

The critical criteria for a watermarker generating random code is whether the random code is stealthy. An attacker may perform statistical analysis on suspected tamper-proofing code to evaluated whether it varies significantly from surrounding code. For example, an attacker may search for assignments to dead variables, unusual error message strings or use of data structures in a short piece code which are not used anywhere else in the program. One simple source of stealthy code is other sections of code in a program. For purposes of stealth, data flow analysis is required to ensure that all variables used in the chosen snippet of code exist and have been initialized. Furthermore, code in a tamper-proofing widget must have some side-effect to be effective. For example, the chosen snippet may change the value of some live variable.

In TBW, a possible choice for random code in a tamper-proofing widget is the corresponding piece in an earlier watermarking widget. For example, the code in `piece1` of our tamper-proofing widget can be a copy of `piece1` of an earlier watermarking widget. Alternatively, expressions in the neighborhood of the tamper-proofing widgets with slight alterations would also exhibit many of the statistical properties of genuine statements.

```
boolean doneA = opaqueTrue;      boolean doneB = opaqueTrue;
boolean doneC = opaqueFalse;     boolean doneD = opaqueFalse;
Object mutex0 = new Object ();   Object mutex1 = new Object ();
Object mutex2 = new Object();
monitorenter ( mutex0 );
if ( !doneA ) {
  piece1;  doneA = !doneA;
  monitorenter ( mutex1 );
  monitorexit ( mutex0 );
  monitorenter ( mutex_orig );
  monitorexit ( mutex_orig );
}
if ( !doneB ) {
  piece2;  doneB = !doneB;
  monitorexit ( mutex0 );
  monitorenter ( mutex1 );
}
if ( ( ( doneC || doneD ) && opaqueFalse  ) ||
     ( (    ! doneC     ) && opaqueTrue   ) ||
     opaqueTrue ) {
  doneC = !doneC;
  if ( doneD )
    monitorexit ( opaqueTrue ? mutex0 : mutex1 );
  else {
    doneD = !doneD;
    monitorenter ( mutex2 );
    monitorexit  ( opaqueTrue  ? mutex0 : mutex2 );
    monitorexit  ( opaqueFalse ? mutex1 : mutex2 );
  }
} else {
  piece3;  doneC = !doneC;
  monitorexit ( mutex1 );
}
```

Figure 5.7: Tamper-proofing code which embeds neither 0 nor 1. The code in piece1, piece2 and piece3 contain incorrect code and are never executed.

For example, negating boolean assignment statements and converting inclusive comparisons to exclusive comparisons and vice versa, introduce subtle off-by-one errors which would course the attacked program to behavior incorrectly.

This far the TBW algorithm has only altered the basic blocks on the path of P when run with I. In addition, the current implementation also inserts a user selected number of random watermarking widgets and tamper-proofing widgets into blocks which do not occur in the trace. Thus on input sequences other than I the watermarked program

produces traces which also contain thread locking behaviour.

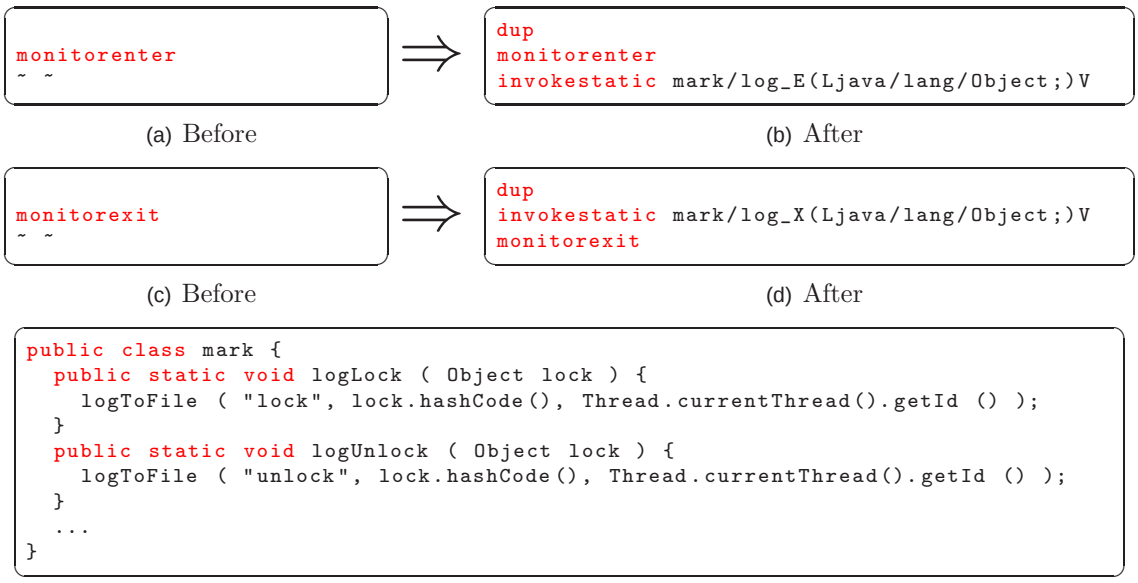
5.2.8 Recognition

The recognition process involves detecting the threads which execute `piece1`, `piece2` and `piece3`. We need to do this without relying on the naming the blocks or the threads as an attacker may obfuscate this information statically.

It is sufficient for our detector to be sensitive to the order in which threads acquire and release locks. Using this information the extraction process can deduce the identity of the executed blocks. Thus our watermark recognition relies solely on the dynamically distinctive pattern of lock acquisitions and releases performed by our watermarking widget.

The first step in recognizing our watermark is to gather a trace file similar to the one that was gathered during tracing. However, we do not require as much detailed analysis as was required to perform embedding. In particular, we do not perform control flow analysis of the target program nor build a control flow graph. This is advantageous as an attacker may have applied control flow obfuscation algorithms (such as described by Wang [28]) or other attacks that make building a precise and accurate CFG difficult. Instead we statically replace every call to `monitorenter` and `monitorexit` in the target program with code which reports the monitor which has successfully acquired or released and the thread that acquired the lock:

- For every method in every class in the target application:
    - Substitute `monitorenter` and `monitorexit` as shown in Figure 5.8 such that the following information is output to a trace file:.
        - **locked object** - the argument to the `monitorenter` or `monitorexit` call. This is the object on the top of the stack when the monitor instruction is called.
        - **lock or unlock** - whether the object was being locked or unlocked ie. whether the object is the argument to a `monitorenter` or `monitorexit`



(a) Before (b) After

(c) Before (d) After

(e) Method that logs the trace

Figure 5.8: Code for logging the execution trace to a file for recognition

call.

- **thread ID** - the ID of the thread which executes the block. This is found by querying the JVM at runtime.

- Run the program with key input sequence I and collect trace.

The next step during recognition is to extract the armored watermark bits from the trace. We present two recognition algorithms for TBW, one being an extension of the other.

Simple recognizer

As identified in Section , our watermark can be extracted by pattern matching on lock acquisitions. The extraction algorithm is described informally below.

1. Rebuild the code table

   - The user enters the same random seed S which was used during embedding to initialize the random number generator. This seed is used to rebuild the random code table which maps from 16-bits to 64-bits. The code table is

inverted using a hash table such that the table can be used to look up 64-bit 'armored' watermarks to discover the corresponding 16-bit code.

2. Convert the trace to a string

   - Filter the trace by removing all "unlock" calls. Convert the remaining trace of (lock, locked object, thread ID) tuples into a string of letters where each distinct thread ID maps to a distinct letter. Thus we convert our trace of tuples into a string of letters.

3. Search for the watermark in the trace

   - Initialize an output watermark W to the empty string.
   - For every combination of 3 letters, x, y and z which occur in the trace string:
     - Search through S for the regular expression: "aabc(a|c)ab". This regular expression can be written using fixed length negative lookahead. Our first atom "a" can be any letter x, y or z ie. "`[xyz]`". Our second atom "b" can be any letter x, or z but not "a" ie. `(?!\\1)[xyz]`. Similarly our third atom "c" can be any letter x, y or z but not "a" or "b" ie. `(?!\1)(?!\2)[xyz]`. Thus we are searching S for the following regular expression (in Perl Regular Expression format [38]:

```
([xyz])\1(?!\1)([xyz])(?!\1)(?!\2)([xyz])(\1|\3)\1\2
```

     - If the fifth letter in the matched string is "a", append a '0' to our watermark string W. If the fifth letter is "c", append a '1' to W.
     - The method described above is slightly simplified. Our actual implementation is in Java, and it adds "wildcards" to the pattern. These match threads other than the "a", "b" and "c" of the regular expression above. Other threads may obtain locks at any time during our watermark sequence. The regular expression used is illustrated in Figure 5.10 using the Java regular expression API.

- Finally the resulting watermark string W is divided into 64-bit pieces and decoded using the code table. This results in a 32-bit watermark, if W is 128-bits, and if both its 64-bit pieces are in the code table. Otherwise the extraction routine reports "no watermark found".

```
String traceString = convertToString ( trace );
char[] uids = uniqueThreadIDs ( traceString );
int[] wm_prime = new int [ traceString.length ];

for ( int i=0; i < wm_prime.length; i++ )
  wm_prime[i] = -1;

for ( int t_0 = 0; t_0 < uids.length; t_0++ )
  for ( int t_1 = t_0+1; t_1 < uids.length; t_1++ )
    for ( int t_2 = t_1+1; t_2 < uids.length; t_2++ )
      wm_prime = search ( traceString, wm_prime,
                  uids[t_0], uids[t_1], uids[t_2] );

Integer watermark = lookupCodeTable ( wm_prime );
if ( watermark == null )
  System.out.println ( "No watermark found" );
else
  System.out.println ( "Watermark: " + watermark );
```

Figure 5.9: Search for a watermark

Advanced recognizer

We also build a more complete recognizer which recognizes the complete dynamic signature left by the watermarking widget. The only difference between this advanced recognizer and a simple recognizer described previously is that the advanced recognizer searches for the complete pattern of lock acquisitions and releases and includes the objects which were locked and the threads that performed the locks. The advantage of such a recognizer over the simpler one described earlier is it further reduces the chance of a spurious recognition that results from a random occurrence of our watermark pattern in a program. A pattern of sixteen lock and unlock calls on a set of four objects is much less likely to occur at random than a pattern of seven lock and unlock calls.

```
int[] search ( String trace, int[] wm, char x, char y, char z ) {
    String xyz = "([" + x + y + z + "])";
    String not_xyz = "(?:[^" + x + y + z + "]*)";

    Pattern bit0pattern = Pattern.compile (
      "%a %_ \\1 %_ %b %_ %c %_ \\3 %_ \\1 %_ \\2"
         .replaceAll ( "%a", xyz )
          .replaceAll ( "%b", "(?!\1)" + xyz )
           .replaceAll ( "%c", "(?!\1)(?!\2)" + xyz )
            .replaceAll ( "%_", not_xyz )
             .replaceAll ( " ", "" ) );
    Pattern bit1pattern = Pattern.compile (
      "%a %_ \\1 %_ %b %_ %c %_ \\1 %_ \\1 %_ \\2"
         .replaceAll ( "%a", xyz )
          .replaceAll ( "%b", "(?!\1)" + xyz )
           .replaceAll ( "%c", "(?!\1)(?!\2)" + xyz )
            .replaceAll ( "%_", not_xyz )
             .replaceAll ( " ", "" ) );

    Matcher m0 = bit0pattern.matcher ( trace );
    Matcher m1 = bit1pattern.matcher ( trace );
    int searchStartIndex = 0;
    while ( true )
        if ( m0.find ( searchStartIndex ) ) {
            wm[ m0.start() ] = 0;
            searchStartIndex = m0.end();
        } else if ( m1.find ( searchStartIndex ) ) {
            wm[ m1.start() ] = 1;
            searchStartIndex = m1.end();
        } else
              break;
    return wm;
}
```

Figure 5.10: Method that searches for a watermark bit given the current thread set

The pattern of execution for embedding a 0 bit and a 1 bit is shown in Figure 5.11. The regular expressions for recognizing these patterns are shown in Figure 5.12.

5.3 Discussion

The **tbwer** has the following limitation in code that can be watermarked.

Implementation Limitation 1. No exceptions are thrown by the blocks in which the watermark bits are to be embedded.

lock	mutex_w	T_M
lock	mutex_x	T_A
lock	mutex_y	T_A
unlock	mutex_x	T_A
lock	mutex_x	T_B
unlock	mutex_x	T_B
lock	mutex_x	T_C
unlock	mutex_x	T_C
unlock	mutex_w	T_M
lock	mutex_w	T_A
unlock	mutex_w	T_A
lock	mutex_z	T_A
unlock	mutex_z	T_A
unlock	mutex_y	T_A
lock	mutex_y	T_B
unlock	mutex_y	T_B

(a) Bit 0 dynamic pattern

lock	mutex_w	T_M
lock	mutex_x	T_A
lock	mutex_y	T_A
unlock	mutex_x	T_A
lock	mutex_x	T_B
unlock	mutex_x	T_B
lock	mutex_x	T_C
lock	mutex_z	T_C
unlock	mutex_x	T_C
unlock	mutex_z	T_C
unlock	mutex_w	T_M
lock	mutex_w	T_A
unlock	mutex_w	T_A
unlock	mutex_y	T_A
lock	mutex_y	T_B
unlock	mutex_y	T_B

(b) Bit 1 dynamic pattern

Figure 5.11: Dynamic pattern of locks.

This is not a limitation of the algorithm but rather of the current prototype. Some instructions in Java throw an exception. This causes the normal flow of execution of Java to be interrupted and the control passes to a corresponding exception handler. The correct handling of exceptions will require each block to be surrounded by exception catchers that correctly release locked monitors and rethrow the exception. In this implementation, we chose to avoid this additional complexity.

5.4 Summary

In this chapter we described our prototype implementation of thread-based watermarking in Java. The prototype watermarks Java byte-code and uses monitors to implement the locking strategy required by TBW. We described the details of encoding, embedding a TBW and two types of recognizers. We also described an implementation of opaque predicates which are used to implement tamper-proofing widgets. We described the implementation of threadpools which optimize thread-based watermarks by reducing the overhead cost of constructing new threads.

In the next chapter we evaluate this implementation in terms of the criteria outlined in Chapter 2.

```
String mabc = "[" + m + a + b + c + "]";
Pattern bit0 = Pattern.compile (
    "L(.)(" + mabc + ")"
  + "L(?!\\1)(.)(?!\\2)(" + mabc + ")"
  + "L(?!\\1)(?!\\3)(.)\\4"
  + "U\\3\\4"
  + "L\\3(?!\\2)(?!\\4)(" + mabc + ")"
  + "U\\3\\6"
  + "L\\3(?!\\2)(?!\\4)(?!\\6)(" + mabc + ")"
  + "U\\3\\7"
  + "U\\1\\2"
  + "L\\1\\4"
  + "U\\1\\4"
  + "L(?!\\1)(?!\\3)(?!\\5)(.)\\4"
  + "U\\8\\4"
  + "U\\5\\4"
  + "L\\5\\6"
  + "U\\5\\6" );
```

(a) Regular expression which matches a 0 bit

```
String mabc = "[" + m + a + b + c + "]";
Pattern bit1 = Pattern.compile (
    "L(.)(" + mabc + ")"
  + "L(?!\\1)(.)(?!\\2)(" + mabc + ")"
  + "L(?!\\1)(?!\\3)(.)\\4"
  + "U\\3\\4"
  + "L\\3(?!\\2)(?!\\4)(" + mabc + ")"
  + "U\\3\\6"
  + "L\\3(?!\\2)(?!\\4)(?!\\6)(" + mabc + ")"
  + "L(?!\\1)(?!\\3)(?!\\5)(.)\\7"
  + "U\\3\\7"
  + "U\\8\\7"
  + "U\\1\\2"
  + "L\\1\\4"
  + "U\\1\\4"
  + "U\\5\\4"
  + "L\\5\\6"
  + "U\\5\\6" );
```

(b) Regular expression which matches a 1 bit

Figure 5.12: Regular expression for matching different bit patterns.

6

Empirical Evaluation

You can't evaluate a man by logic alone.

– McCoy, "I, Mudd", stardate 4513.3

In this chapter we present an evaluation of the watermarking transformation described in the previous chapters. The evaluation is presented in two parts. Firstly we present the empirical results for a some of attacks against TBW. These attacks are using existing software analysis tools and are performed on the implementation as described in the previous section. This part is supported with a discussion of the practical difficulty of analyzing multi-threaded programs. Secondly we present the results of performance impact of TBW watermarked programs. For this section, we used a variety of Java applications and two benchmark suites.

6.1 Security Evaluation

The TBW prototype was evaluated based on the criteria given in Section 2.4. The three security properties can be summarized as:

Resilience The resilience of TBW is a measure of how well it remains legible after semantic-preserving transformations. These transformations can fall under either general or targeted attacks, using our taxonomy of Section 2.5. To evaluate resilience, we record whether a recognizer is able to recognize a TBW after a watermarked program has been subjected to a number of code transformations.

Credibility The credibility of TBW is a measure of the chance that an unwatermarked program appears to contain a TBW watermark. The pool of unwatermarked Java programs is very large and it is difficult to select a meaningful random sample. To measure the non-malicious false positive rate of TBW for this evaluation, we measure the sparsity of the space of watermark strings in our implementation of TBW.

Stealth The stealth of TBW is a measure of how easily it can be distinguished from the rest of an application and thereby be removed from the application. We limit our attention in this evaluation to the static stealth of TBW, and to the question of automatically distinguishing watermarking widgets from tamper-proofing widgets in an application.

The target programs that were used in these experiments consist of Scimark - four numerically intensive and highly optimized for scientific computations; JTidy - a HTML syntax checker and pretty printer; TTT - a simple TicTacToe game; SandMark - a framework for obfuscation and watermarking; and JFig - a graphical editor used to draw the figures used in this thesis. The characteristics of these test programs are summarized in Table 6.1.

The platform for the experiments is a 2.40 GHz Intel Pentium with 1 GB of RAM, running Linux 2.4.24. Tests were performed using Sun JDK 1.4.2.

Program	Version	Description	Size (bytes)
TTT	15 Sep, 1998	A simple game	16,286
Scimark	Unknown	A numerical benchmark	17,975
JTidy	20 Dec, 2000	HTML syntax checker	272,430
JFig	07 Jan, 2002	A diagram editor	1,071,524
SandMark	08 Jun, 2004	A obfuscation and watermarking tool	3,856,474

Table 6.1: Test programs (bytecode before transformations)

6.1.1 Resilience

Collberg [22, 39] describes two types of attacks against resilience, "distortive" and "subtractive". These correspond to our notions of general and targeted attacks, respectively. As described in Section 2.5.2, to mount a targeted attack, an adversary must first successfully attack a watermarks stealth. Attacks against stealth are discussed below in Section 6.1.3.

In this section, we consider general attacks on resilience. We tested the ability of the TBW recognizer to recognize the watermark after four types of general attacks.

1. Obfuscation attack
2. Encryption attack
3. Recompilation attack
4. Noise addition attack

Obfuscation Attack

The SandMark suite implements thirty-three different static obfuscations. These range from very simple transformations which an attacker may use such as renaming all variables and methods in a program and reordering blocks of code to much more far-ranging transformations such as merging methods, splitting classes, splitting arrays, changing the signature of methods and box and unboxing of scalars. For a complete description of obfuscations implemented by SandMark refer to [30].

The experimental design was as follows: Each of the six application listed in Table 6.1 was watermarked. A fixed watermark was embedded in each application. The applications

were then obfuscated by applying each of the 33 obfuscators singly. The TBW recognizer was then run on the resulting obfuscated application.

Possible outcomes of this attack were "fail" if the same watermark that was embedded was recognized; "error" if the obfuscator failed to produce an obfuscated application; "crash" if the obfuscated application failed to execute, crashed or produced output observably different from the original; and "success" if a different watermark was recognized than was embedded or no watermark was recognized; . "Success" is only outcome that results in an attacker having an unwatermarked, semantically equivalent program. The results of this experiment is shown in Table 6.2.

From the final column of Table 6.2, we see that SandMark is unable to obfuscate itself. This is due to a fundamental design limitation of the SandMark suite.

The four rows of "error" indicate that four obfuscators, "add interprocedural opaque predicates", "apply dynamic inliner", "mark blocks" and "split variables" did not produce an obfuscated version of the tested applications. This seems to be the result of errors in their implementation in Sandmark. The "rename identifiers" obfuscator incorrectly transform JFig and the resulting application was unrunnable; the sole "crash" in Table 6.2.

Summarising, in all our tests the obfuscation attack was unsuccessful, either because the obfuscation itself failed ("error", "crash") or because the obfuscation was ineffective at stopping recognition of the watermark ("fail"). This is an encouraging result that suggests TBW is resilient to these existing obfuscation techniques.

Encryption Attack

In an encryption attack, every class file in an application is replaced by an encrypted form of itself.

The experimental design was as follows: All test applications in Table 6.1 were watermarked with TBW with a fixed watermark. To mount the attack, every class in each test application was encrypted with a fixed key. The `main` method of the application was altered to introduce a new application class loader which decrypted any class that was requested before loading and executing it. The TBW recognizer was then run on the

Obfuscator	Obfuscation Attack				
	Scimark	JTidy	TTT	JFIG	SandMark
Add aliases to parameters	Fail	Fail	Fail	Fail	Error
Add bogus fields	Fail	Fail	Fail	Fail	Error
Add bogus predicates	Fail	Fail	Fail	Fail	Error
Add branches	Fail	Fail	Fail	Fail	Error
Add buggy code	Fail	Fail	Fail	Fail	Error
Add inter-procedural opaque predicates	Error	Error	Error	Error	Error
Add opaque predicates	Fail	Fail	Fail	Fail	Error
Append bogus code	Fail	Fail	Fail	Fail	Error
Apply dynamic inliner	Error	Error	Error	Error	Error
Apply static inliner	Fail	Fail	Fail	Fail	Error
Apply static outliner	Fail	Fail	Fail	Fail	Error
Degrade classes	Fail	Fail	Fail	Fail	Error
Encode strings	Fail	Fail	Fail	Fail	Error
Interleave methods	Fail	Fail	Fail	Fail	Error
Make CFG irreducible	Fail	Fail	Fail	Fail	Error
Make fields public	Fail	Fail	Fail	Fail	Error
Mark blocks	Error	Error	Error	Error	Error
Merge methods	Fail	Fail	Fail	Fail	Error
Merge scalars	Fail	Fail	Fail	Fail	Error
Perform method overloading	Fail	Fail	Fail	Fail	Error
Promote locals to objects	Fail	Fail	Fail	Fail	Error
Promote primitives to objects	Fail	Fail	Fail	Fail	Error
Rearrange local variable table	Fail	Fail	Fail	Fail	Error
Rename identifiers	Fail	Fail	Fail	Crash	Error
Reorder instructions	Fail	Fail	Fail	Fail	Error
Reorder parameters	Fail	Fail	Fail	Fail	Error
Reorder variable tables	Fail	Fail	Fail	Fail	Error
Reverse if-else statements	Fail	Fail	Fail	Fail	Error
Split CFG nodes	Fail	Fail	Fail	Fail	Error
Split arrays	Fail	Fail	Fail	Fail	Error
Split booleans	Fail	Fail	Fail	Fail	Error
Split classes	Fail	Fail	Fail	Fail	Error
Split variables	Error	Error	Error	Error	Error

Table 6.2: Recognition after obfuscation attack. A `Fail` indicates the original watermark was recognized; a `Error` indicates an error in the obfuscator; a `Crash` indicates an error in the watermarked application. The obfuscations shown did not successfully remove the TBW hence there are no instances of `Success` in this table.

resulting application. The two possible outcomes of this attack were "successful" if same watermark that was embedded was recognized and "failed"if the correct watermark was not recognized.

The result of this experiment, shown in Table 6.3, indicates the encryption attack

succeeds.

Application	Encryption Attack
Scimark	`Success`
JTidy	`Success`
TTT	`Success`
JFIG	`Success`
SandMark	`Success`

Table 6.3: Recognition after encryption attack. In all five cases, an encryption attack succeeded in preventing correct recognition.

This attack successfully foiled the TBW recognizer in every tested instance. It is an interesting attack because it has no effect on the threading behavior of the application nevertheless successfully thwarts the recognizer by preventing bytecode instrumentation during tracing.

An encryption attack can be generalized to a more generic class of *translation attacks*. In this attack, every class in an application is replaced with a translated form of itself and by suitably altering the class loader to reverse the translation before load and execution.

Although successful, the attack reveals a flaw in the implementation of the detector and not in the embedding. Ultimately, the JVM necessarily must have access to the decrypted bytecode during execution. To counter such an attack (or more generally a class translation attack), the TBW implementation must be built to instrument the JVM, use a tracing JVM or the Java debugging and profiling interface to annotate a program. In this way, a trace could still be successfully collected. Thereafter, the remainder of the recognition would perform correctly.

Recompilation Attack

A recompilation attack is one where the watermarked program is decompiled then recompiled. Decompilation of programs that contain our watermark is difficult because although the watermarked code is legal Java bytecode, the improperly nested monitor calls mean that it cannot be directly expressed in the Java language.

The design of the experiment was as follows: All test applications in Table 6.1 were watermarked with TBW with a fixed watermark. To mount the attack, the test applications

were decompiled then recompiled using each of the following decompilers:

1. Dava (part of Soot version 2.0.1)
2. Homebrew version 0.2.3
3. Jad version 1.5.8e
4. JODE version 1.1.2-pre1
5. SourceAgain Professional version 1.10k

If the application was successfully recompiled, the TBW recognizer was run on the resulting application. The observed outcomes of this attack, shown in Table 6.4, were "error" if the application failed to decompile or recompile and "crash" if the recompiled application ran incorrectly.

Decompiler	Recompilation Attack				
	Scimark	JTidy	TTT	JFIG	SandMark
Dava	`Error`	`Error`	`Error`	`Error`	`Error`
Homebrew	`Error`	`Error`	`Error`	`Error`	`Error`
Jad	`Error`	`Error`	`Error`	`Error`	`Error`
JODE	`Error`	`Error`	`Error`	`Error`	`Error`
SourceAgain	`Crash`	`Crash`	`Crash`	`Crash`	`Crash`

Table 6.4: Recognition after recompilation attack. A `Error` indicates that recompilation of the program failed while a `Crash` indicates that the recompiled program ran incorrectly.

In no case did we observe a successful attack, producing either a false negative non-detection or a reported extraction of an incorrect watermark.

The row of "Crash" shows that only SourceAgain successfully decompiled our watermarked applications such that they could be recompiled without errors. However, the code that it generated removed all instances of synchronization in the watermarking and tamper-proofing widgets. As a result, the recompiled applications ran incorrectly.

Of the remaining decompilers, Homebrew and Dava failed to produce any usable decompiled code while both JODE and Jad produced syntactically incorrect source code when decompiling the unusually nested monitor calls in our watermarking and tamper-proofing widgets. The JODE decompiler was used to decompile a watermarked copy of

the TTT and the resulting source code was corrected by hand. A thread library was used to emulate Java monitors in pure Java as described by Miecznikowski [40]. The resulting source code was recompiled. The TBW recognizer successfully recognized the original watermark on the recompiled application.

Noise Addition Attacks

An attacker uses a noise addition attack to attempt to obscure the original watermark by inserting extraneous noise into the watermarking signal. For TBW, this attack consists of inserting extra threads and locks. Furthermore, he can re-watermark an application which adds additional watermarking bits to an application. The following experiments were run to test the resilience of our implementation to additive attacks:

1. Forcing thread switches using existing threads
2. Inserting new threads and locks into an application
3. Re-watermarking the application

New Thread Switches Attack

An attacker attempts to confuse the TBW recognizer by causing an application to perform a large number of thread switches, for example by running many other Java threads concurrently. In our experiment, this was simulated by inserting calls to `Thread.yield()` randomly through the program. This has no effect on the logic of the program while requesting the JVM to perform a thread switch.

The only observed result of this attack was that it increased the size of the trace file produced during recognition. It had no observable impact on the accuracy of TBW recognition.

New Threads and Locks Attack

An attack may also increase the number of threads and locks in an application, for example by executing some basic blocks in new threads.

In this experiment, twenty basic blocks were selected randomly from the sequence of basic blocks that are executed in the JFig application. For each basic block, a new thread was introduced in that basic block. The basic block was replaced with a `start()` call to the corresponding thread and a `join()` call that will cause the original thread to suspend until the thread completed execution. The transformation was manually inspected and edited to ensure that the transformation did not introduce deadlocks.

This attack was unsuccessful: it had no impact on the accuracy of recognizing a TBW watermark in JFig.

Re-watermarking Attack

Another noise-adding general attack an attacker may apply is re-watermarking to overwrite or obscure part of the original watermark by . For a dynamic watermark there are two cases:

1. The attacker uses the same input sequence as the original watermark
2. The attacker uses a different input sequence

In this experiment, our test applications were TBW watermarked using a fixed key input I_1 and with a fixed string W_1. Each application was then re-watermarked using TBW with a different watermark string W_2. This was done twice, once using the key input I_1 and a second time using a new input I_2. The attack "fails" if it recognizes the original watermark, W_1 and "succeeds" otherwise.

The Table 6.5 shows the results of recognizing the original watermark after such an attack. As shown, the re-watermarking attack is a potent attack in all five test applications if the original key input is known and used by the attacker. This is primarily because both watermarks W_1 and W_2 are expressed during recognition and the bits of the watermark intermingle.

In the case where a new watermark sequence was used by the attacker, the TBW recognizer continued to find the original W_1 in the case of JFig and SandMark, however, failed for the remaining three simpler applications. The likely reason for this discrepancy

is that Scimark, JTidy and TTT have a very small state space and for most input, execute largely similar parts of the program. As a result, the trace during key input and the trace during a new input sequence shared many basic blocks and the bits of both watermarks were expressed during recognition.

On the other hand, JFig and SandMark have a much larger state space and each input executes only a small part of the application. On the inputs that were selected, distinct parts of the program were executed and thus the watermarks did not intermingle.

For more comprehensive results, a more rigorous set of experiments are required to determine the effect of varying the key input sequence on the trace that is selected through a program. Among the questions that we would like to answer are "Does varying the input alter the path of execution sufficiently that an embedded watermark is no longer expressed?", "For a given watermarked application, how difficult is it an attacker to produce a input that causes the watermark to be expressed?" and "For a given watermarked application, that is re-watermarked with a 'random' key input, how likely is it that the traces through the application will share basic blocks?" These questions and experiments constitute a starting point for future research.

As a consequence, we introduce the following limitation:

Limitation 2. An attacker is unable to guess the key input sequence.

Program	Re-watermarking Attack	
	Using key input sequence	Using a new input sequence
Scimark	`Succeeds`	`Succeeds`
JTidy	`Succeeds`	`Succeeds`
TTT	`Succeeds`	`Succeeds`
JFig	`Succeeds`	`Fails`
SandMark	`Succeeds`	`Fails`

Table 6.5: Recognition after re-watermarking attack. For Scimark, JTidy and TTT, re-watermarking, either with the original key input or with a different input, the attack successfully destroyed the original watermark. In the case of JFig and SandMark, the attack was only successful when the original key input was used during the re-watermarking attack.

6.1.2 Credibility

In Section 2.4.2, we defined credibility as a measure of a watermarks non-malicious and attacked false positive rate.

It is difficult to measure the true non-malicious false positive rate of TBW because there does not exist an adequate sample of "real world" Java programs and relatively little is known about the statistical properties of such programs. Without this data, we do not have a control that can be used to compare against watermarked programs.

Instead, we measure the credibility of one stage of recognition. The TBW recognizer relies on a distinctive locking pattern to extract potential watermark strings and a code lookup table to decode the watermark. In this experiment we measure how often these potential watermark strings may occur and how often the recognizer falsely identifies a watermark.

We generated $100,000,000$ random 64 bit strings and attempt to decode these using the random code table described in Section 5.2.1. None of these $100,000,000$ random strings were identified by the decoding process as a valid watermark string. This suggests that if the locking behaviour of programs is random, the false positive rate will be very low.

In actuality, the locking behaviour of programs is far from random and is usually limited to very simple properly nested lock and unlock calls. These are even less likely to result in a pattern that is falsely identified as a 64-bit potential watermark string.

An advantage of such a sparse encoding for our watermark is that it helps prevent malicious false positives. A *malicious false positive* occurs if the attacker manages to insert their watermark into the watermarked program either by reverse engineering our key sequence or using their own key sequence. Given the sparseness of the code table and unlikelihood of a random 64-bit string being recognized as a valid watermark, we are confident that an attacker would be unable to maliciously insert false positive watermarks without access to the seed for the error-detection algorithm.

We are unable to construct a persuasive experimental validation of this expectation,

however, because its validity in practice will depend on the cryptographic security of our random number generator, and on the "black box" confidentiality requirements in the lookup tables in the watermark recognition system. Both of these concerns are outside the scope of our thesis, hence we introduce the following limitations:

Limitation 3. The attacker is unable to construct a copy of the TBW recognition code table.

6.1.3 Stealth

Targeted attacks against stealth are the most potent tool in an attackers arsenal. If an attacker is able to automatically distinguish watermarking blocks, he may then be able to remove (or "subtract") it with high reliability and low cost. Such attacks are called "subtractive" in Collberg's taxonomy.

In this experiment, the efficiency of a static analysis tool in distinguishing a watermarking widget from the rest of the code is examined.

There are a very small number of publicly available static analysis tools for Java. Among these only two support analysis of multi-threading: JLint and BDDBBDB.

JLint

JLint [41] is a static analysis tool for Java that is designed to analyze synchronization, inheritance and data flow characteristic of a program. In particular, in order to detect potential deadlock the tool statically analyzes the order in which locks are acquired.

In this experiment we test the ability of JLint to statically determine the pattern of lock acquisitions and releases of a watermarked program. If JLint is successful in distinguishing between watermarking widgets and tamper-proofing widgets, then an attacker can simply remove the widgets and replace it with equivalent code.

During analysis JLint constructs an oriented graph G of monitor lock dependencies. This graph gives the relative order in which monitor locks are acquired and released. Each monitor is associated with an object. In the program documentation, the author

of JLint asserts that statically determining this object "cannot be done". As a result, he approximates the identity of a monitor lock using the class type of its associated object.

For this experiment, two sets of our five test programs were prepared. In our first set, Set A, a 16-bit watermark was embedded in each program. This resulted in 128 basic blocks being selected in each program and modified to contain a watermark widget. In our second set, Set B, the same set of 128 basic blocks were identified in a copy of the five test programs and each basic block modified to contain a tamper-proofing widget.

JLint was modified such that when run it outputs a copy of the dependency graph G that it builds for that program. JLint was run on each program in Set A and Set B and the output graphs were compared.

It was found that JLint built identical graphs for corresponding programs in Set A and B. In other words, JLint was unable to statically distinguish between programs containing solely watermark widgets and programs containing solely tamper-proofing widgets.

This result suggests that given a TBW watermarked program which contained a large number of tamper-proofing widgets, an attacker armed with a tool like JLint would not be able to distinguish between tamper-proofing widgets and watermarking widgets.

In a second experiment we replaced every opaque predicate in each program in Set A and Set B (as defined above) with its actual value. We then reran JLint and compared the dependency graphs that got built.

We found that in this case JLint build different dependency graphs for programs in Set A compared to the corresponding program in Set B. In other words, JLint was able to statically distinguish between programs containing watermark widgets from programs containing tamper-proofing widgets provided the opaque predicates were replaced with semantically equivalent but obvious predicates.

Although this experiment does not show that an attack is able to recognize watermarking widgets, the result suggests that if Limitation 1 holds (an attacker is unable to resolve the true values of opaque predicates), and if an attacker is able to use a static analyzer like JLint, then he may be able to distinguish between watermarking widgets and tamper-proofing widgets. In the next experiment we measure how difficult it is for

```
public class OpaqueTrue {
 public static void main ( ... ) {
  Node root = new Node ();
  createCycle ( root, 10 );
  int x = getRandomBetween ( 2, 10 );
  Node a = getNth ( root, x );
  Node b = getNth ( root, x );
  Object lock = new Object ();
  Race race = new Race ( a, b, lock );
  synchronized ( lock ) {
   if (
        race.getA () == race.getB();
      )
    System.out.println ( "true" );
  }
 }
}
```

(a) The expression on line 15 always returns true

```
public class OpaqueFalse {
 public static void main ( ... ) {
  Node root = new Node ();
  createCycle ( root, 10 );
  int x = getRandomBetween ( 2, 10 );
  int y = 0;
  while ( x == y )
    y = getRandomBetween ( 2, 10 );
  Node a = getNth ( root, x );
  Node b = getNth ( root, y );
  Object lock = new Object ();
  Race race = new Race ( a, b, lock );
  synchronized ( lock ) {
   if (
        race.getA () == race.getB();
      )
    System.out.println ( "true" );
  }
 }
}
```

(b) The expression on line 16 always returns false

Figure 6.1: Programs used to test JLint's ability to resolve a TBW opaque predicate

the attacker to resolve our thread-based opaque predicate.

BDDBDDB

BDD-Based Deductive DataBase (BDDBDDB) [42] is an implementation of Datalog, a declarative programming language similar to Prolog for specifying program analysis. In particular, it is able to efficiently solve the complex problem of context-sensitive pointer analysis for large programs.

The TBW implementation uses a thread based opaque predicate. It is based on updating two pointers on a cycle asynchronously. The opaque predicate then tests whether the two pointers point to the same object. The pointers are guarded by locks such that either the two pointers always point to the same node in the cycle (opaquely true) or always point to different nodes in the cycle (opaquely false).

Statically determining whether such a predicate is opaquely true or false then reduces to solving the "must-alias" problem. This has been shown to be NP-hard in general. However, a method of generating hard instances of this (or any other NP-hard) problem is not known.

In this experiment, with the assistance of a colleague Dr Antoine Monsifrot, we mea-

sured whether BDDBDDB is able to determine if our asynchronously updating pointers are aliased or not. To minimize the overhead of analysis, two trivial programs were written that utilize the opaque predicate described in Figure 5.6(b) (Chapter 5). These programs are illustrated in Figure 6.1. BDDBDDB was used to determine whether the expression on line 16 in either of the programs was true.

It was found that BDDBDDB analyzed the expression on line 16 to be "maybe true" for both the opaquely true and the opaquely false predicates. In other words, BDDBDDB was unable to distinguish between the two types of predicates.

These experiments are not conclusive. However, the ineffectiveness of both JLint and BDDBDDB in distinguishing between watermarking widgets and tamper-proofing widgets suggest that being armed with the current state of the art in static analysis would not be sufficient for an attacker to defeat thread-based watermarking.

6.1.4 Discussion

In this section we have discussed the tools that an attacker could use to analyse a watermarked program and attack our thread-based watermarks by comprising its resilience, stealth, or credibility. The effort required to mount these attacks depends on the attackers objective, technical expertise and the quality of tools available to him.

While the efficacy of the obfuscation attack discussed in Section 6.1.1 is low, it requires minimal effort and expertise for an attacker to use. The SandMark suite allows a user to automatically apply a large number of transformations. Although the current set of transformation were ineffective against TBW, the suite is under continuing development. Future versions may include attacks that do distort TBW and thus give an attacker with little expertise the ability to attack our watermark quickly. For example, the encryption attack discussed in Section 6.1.1 has been added in the most recent release of SandMark.

A successful attack against the stealth of TBW would require an attacker to have expertise in a wide range of areas including recognizing and analyzing opaque predicates, control and data flow analysis and familiarity with tools such as JLint and BDDBDDB.

The BDDBDDB tool is especially powerful for analysing multi-threaded programs and TBW widgets. However, using BDDBDDB requires a strong understanding of a Datalog - a programming language that is not widely used. What is more, the output of BDDBDDB needs to be interpreted with prior understanding of the program being analyzed. The example that was analyzed in Section 6.1.3 was a relatively short and simple program. Nevertheless, this program required over half an hour in computation time and several hours of analysing by a person familiar with the tool to interpret the results. Finally, tools like JLint and BDDBDDB are not mature yet and are rarely used to analyse the type of convoluted code that TBW produces. Our experiments revealed a large number of bugs in these tools which were time-consuming to identify and required the assistance of the original authors of the tool to fix.

These reasons lead us to conclude that while tools such as JLint and BDDBDDB are powerful and could be very useful in an attack against TBW, using these tools will require the expenditure of a lot of effort and time on the part of an attacker.

6.2 Performance

The TBW transformation significantly alters selected parts of the program increasing the number of threads and thread contention. This transformation may significantly impact the size and performance of a watermarked program.

There are two aspects of performance that we chose to consider when evaluating the TBW implementation. The *performance overhead* of the watermark widgets is the effect of embedding watermark widgets on the size, speed and responsiveness of a watermarked application. The *data rate* is a measure of the number of bits of watermark that can be embedded in an application.

6.2.1 Performance Overhead of Watermarking widgets

In considering the performance overhead of TBW, we divided our five test applications into two categories: *interactive* and *non- interactive* applications. Interactive applications

are those which have an event loop and regularly wait on user input. In these programs, the rate of input is primarily controlled by the user. Examples of interactive applications include graphical and network programs. The remaining programs are non-interactive applications. These either do not take any input or accept all input on the command-line or from an input stream. In these programs, the rate of input is primarily controlled by the program. Examples of non-interactive programs include benchmarks and compilers.

The reason for distinguishing between these two categories of programs is because while TBW may have a large impact the performance of a program, if this impact is negligible compared to the time a program spends idle, the performance impact may be tolerable.

The division of our test programs into "interactive" and "non-interactive" is given in Table 6.6.

Program	Interaction	Input source
Scimark	Non-interactive	No input
JTidy	Non-interactive	File input stream
TTT	Interactive	Mouse
JFig	Interactive	Mouse and Keyboard
SandMark	Interactive	Mouse and Keyboard

Table 6.6: Interactive vs non-interactive test programs

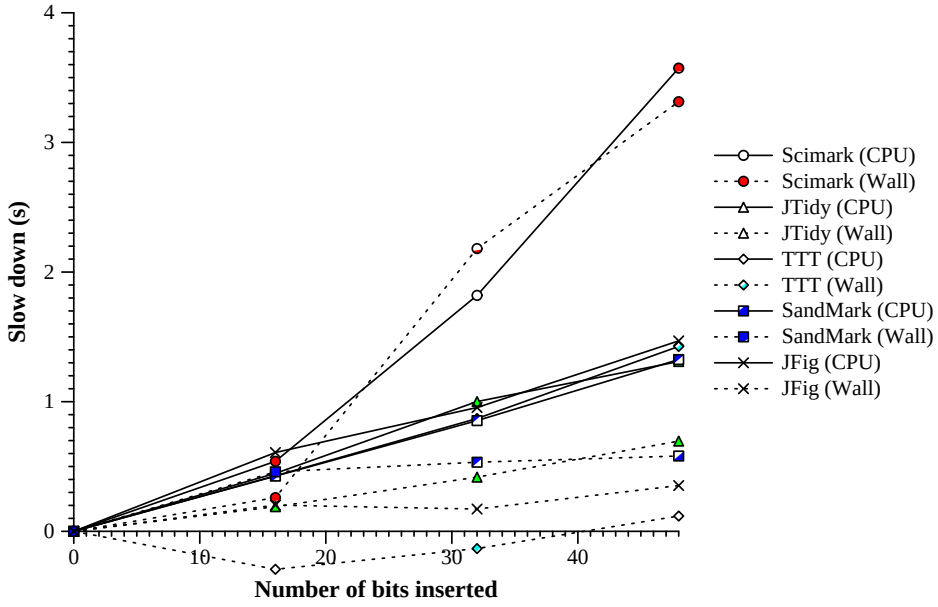
Impact of watermarking on overall runtime

The first set of experiments is designed to measure the slowdown that results from embedding a watermark. In these experiments, a fixed key input was used to embed a 16-bit, a 32-bit and a 48-bit watermark. No tamper-proofing widgets were added. For each program that was tested, the running time of original program and the watermarked program was measured when executed on key input.

The interactive programs were run using a Java `Robot` harness. This harness allowed the GUI events that constitute key input to be captured once for each application and replayed for each of the three different sized watermarks. To minimize the effect on speed due to the use of a harness, all measurements of the runtime of interactive applications

including the runtime of unwatermarked applications were taken using the same harness.

The slow down that resulted is shown in Figure 6.2. In each case, both the CPU time and wall-clock time for execution are given.



Program	Runtime (s)							
	No watermark		16-bit		32-bit		48-bit	
	CPU	Wall	CPU	Wall	CPU	Wall	CPU	Wall
Scimark	23.64	23.93	24.18	24.19	25.46	26.11	27.21	27.24
JTidy	0.43	1.24	0.87	1.43	1.43	1.65	1.74	1.93
TTT	0.49	9.08	0.92	8.78	1.36	8.94	1.91	9.19
JFig	3.30	55.10	3.91	55.30	4.26	55.27	4.77	55.45
SandMark	5.25	53.69	5.68	54.15	6.10	54.22	6.58	54.27

Figure 6.2: Slowdown due to embedding a watermark. The graph shows the slowdow that results from embedding a watermark.

As can be seen from Figure 6.2 the time take to execute a program increases with the size of the watermark. This increase is evident both in the CPU time and the wall time consumed by the execution of the test programs. The increase in running time for Scimark was significantly larger compared to the impact of TBW on the remaining test programs. This is principally because Scimark is a highly optimised and single threaded numeri-

cal benchmark. The introduced watermarking threads add significant and measurable overhead.

In comparison, the remaining programs saw significant increases only in the CPU time consumed. For example, the CPU time for JTidy and TTT increased by a factor of approximately 4 when a 48-bit watermark was embedded. Each of these programs spend relatively little time in computation compared with the time spent performing I/O. In this case, adding a watermarking widget adds a fixed amount of computational overhead. For example, in each of the programs shown, adding a 48-bit watermark added approximately 1.4s to the CPU time. The small variations in the execution time for different programs can be attributed to the number of variables that have to be copied into and out of closures that are built in the different programs.

In Scimark, the relative increase in wall time remains similar to the relative change in CPU time as the size of the watermark increases. On the other hand, in the remaining test programs, the wall time grows a lot slower compared to CPU time. As indicated earlier, the remaining programs are interactive and perform I/O. This time spent waiting for input from the user masks a large portion of the computational overhead of TBW by amortising it over the time the CPU is idle and awaiting input. This is especially noticeable in Sandmark, JFig and TTT all of which are GUI programs with a lot of user interaction.

From these results we can conclude that the embedding a thread-based watermark imposes slowdown which is linearly related to the size of the watermark. Our results are encouraging because they indicate that much of computational expense of embedding a TBW may be swallowed by the idle time of interactive programs. It also suggests that such graphical programs would be ideal candidates for TBW watermarking.

Impact of watermarking on response time

In a GUI application, the total running time is not the only indicator of the usability of the application. In this second set of experiments we measured the change in responsiveness of a GUI application as a result of watermarking.

Figure 6.3 shows the *queue-time* of GUI events in each of the test programs with no watermark, a 16-bit watermark, a 32-bit watermark and a 48-bit watermark. The queue-time is the time between a GUI event occurring and it being popped off the event queue to be processed.

A long queue time suggests that a program takes a long time between a user generating a GUI event, for example a mouse click, and the program responding to it. As a result a program that is executing with a long queue time will appear more sluggish and unresponsive than one with a shorter queue time.

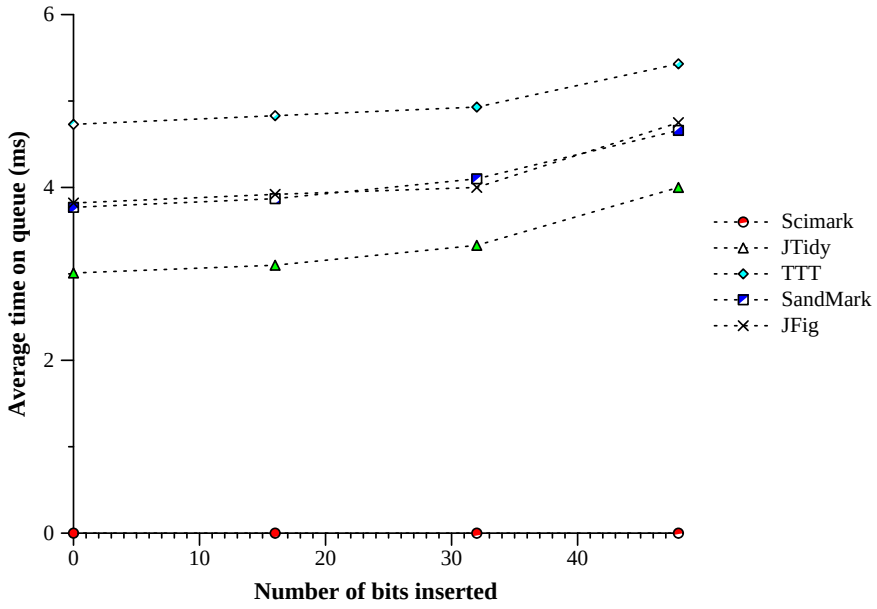


Figure 6.3: Responsiveness. The graph shows the average time that an events stays on the GUI event queue. Scimark being a non-GUI program has no GUI events.

As can be seen from Figure 6.3, the average response time for an event goes up slightly with an increasing size of watermark. This correlates well with the informal user experience using an application that contains a TBW watermark. Watermarked applications remain usable. The impact of watermarking is most noticeable with a 48-bit watermark in those applications that require a large number of mouse clicks. For example, when using JFig, there was occasionally a noticeable delay between a mouse click drawing a shape and that shape appearing.

Impact of watermarking on code size

The TBW algorithm adds a block of code for every watermark and tamper-proofing bit that is embedded in a program. There is also a fixed overhead contributed by the infrastructure code needed to support closures. Finally, for every variable that is closed over by the watermark widget, code is added that constructs a new local copy of the variable and updates the original after a watermarking widget is executed.

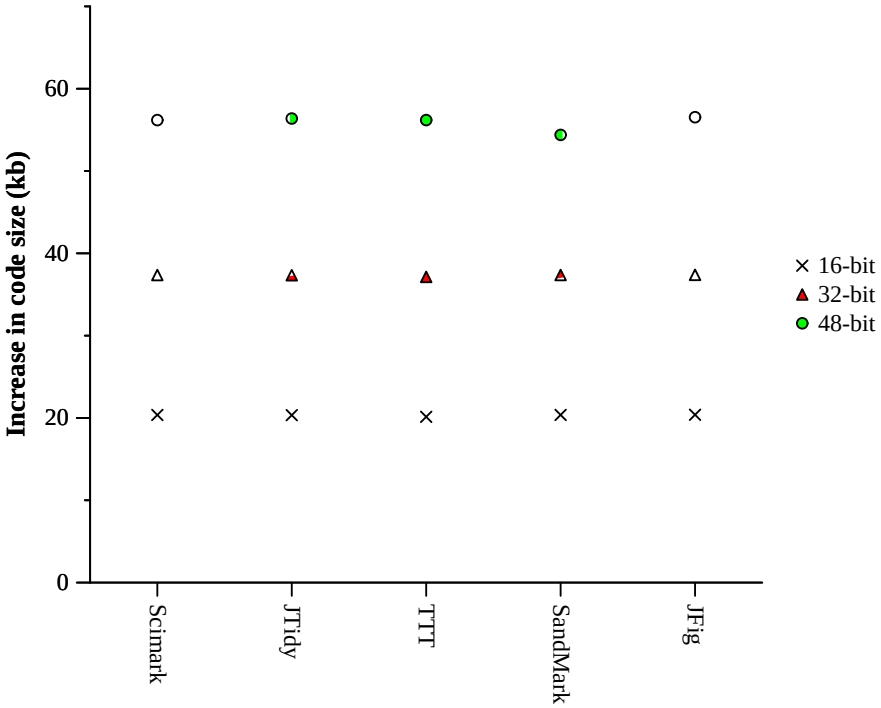


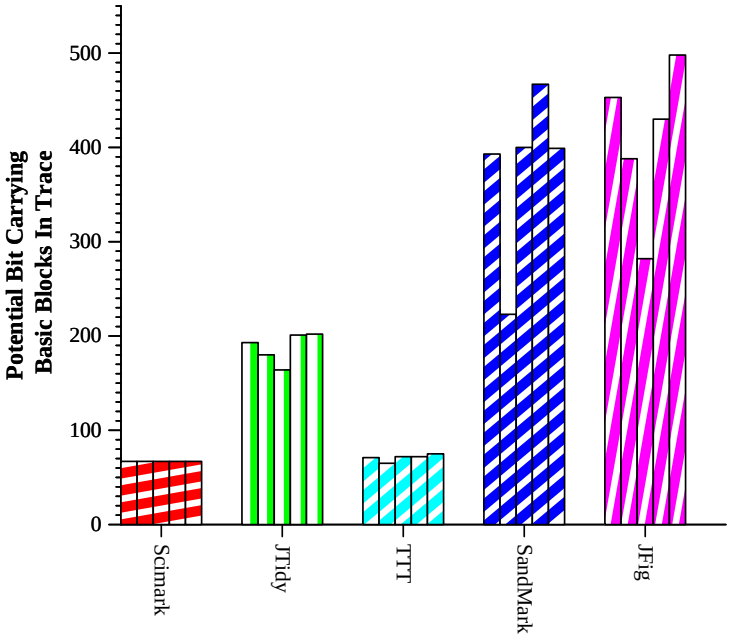
Figure 6.4: Size:The graph shows the increase in size as a result of embedding a watermark.

The Figure 6.4 shows the increase in size of each of the test applications as a result of embedding a 16-bit watermark, a 32-bit watermark and a 48-bit watermark. It can be seen that the size of the application grows approximately linearly with the size of watermark. The variation from linear growth is due to the number of variables in each of the closures that were built by the watermarking widgets.

6.2.2 Maximum Watermarking Data Rate

The TBW algorithm's data rate is dependent on the path through the program which has the largest number of unique basic blocks. This path is in general difficult to compute. In

this experiment we determine how many such basic blocks occur in our test programs on our test runs. This gives us an indication of how many bits may typically be embedded in a program without additional injection of code. Furthermore, we compute the total number of basic blocks in a program which gives us an upper-bound on the number of bits that can be embedded.



Program	Total blocks
Scimark	428
JTidy	6,183
TTT	75
JFig	44,119
SandMark	14,030

Figure 6.5: Maximum data rate. The graph shows the number of basic blocks that can have a watermarking widget embedded in it.

Figure 6.5 gives the maximum number of basic blocks in which a watermarking bit can be embedded on five different selected input sequences. Also shown is the total number of basic blocks in the application.

As can be seen, Scimark which takes no input has only one path through the program. This path has 67 unique basic blocks in which a TBW bit can be embedded which differs greatly from the 428 basic blocks which actually occur in the application. This is an intuitive result as most of the Scimark benchmark consists of tight loops which are unsuitable for TBW.

There is little variation in the number of bits that can be embedded in JTidy and TTT on different input sequences. In these programs, the path that is taken through the program has approximately the same length and repetition irrespective of the selected

input.

In the remaining two applications, JFig and SandMark only a very small portion of the program was executed on any input sequence. In our experiment, between 223 and 467 basic blocks of JFig and between 282 and 498 basic blocks of SandMark suitable to carry a watermark bit were recorded on our five inputs.

6.3 Summary

For most Java applications manual inspection is economically unfeasible. As a result analysis tools are required. Static analysis tools (which is all that is available given the current state of the art) do not perform well in executing *class attacks* aimed at recognizing and removing TBW. In particular, TBW was found to be resilient to the obfuscation and recompilation attacks launched against it.

While encryption (and the more general class of translation attacks) were successful against the current implementation, we suggest a technique to defend against this attack.

The most effective attack against TBW was the re-watermarking noise addition attack. This attack was 100% effective if the attacker had access to the key input sequence or the watermarked application had a small state space.

The credibility of TBW was difficult to evaluate experimentally. While not conclusive, the aspects of malicious and attacked false positives indicated that if the attacker can be prevented from building or reading the TBW code table, it would be difficult for an attacker to comprise TBW's credibility. Finally, attacks against its stealth failed using currently available multi-threaded analysis tools.

Performance-wise TBW has a small impact on the CPU time of a program. This impact is further mitigated when applied to highly interactive programs for example GUI programs requiring a lot of user input. The current implementation of TBW is not suited to CPU intensive and speed critical applications. The TBW has a small impact on the size of a program. The size of a watermarked application grows approximately linearly with the size of the watermark.

7

Related Work

A large amount of research has been carried out in the domain of software watermarking and more generally in software protection and obfuscation. In this chapter we examine the literature directly related to software watermarking and provide a taxonomy of existing watermarking techniques. Where possible we compare proposed schemes to our work. We also describe work in the area of software analysis which may eventually lead to tools to attack thread-based software watermarks.

7.1 Software Watermarking

When outlining existing literature on software watermarking it is useful to identify the types of software watermarking. As discussed in Section 2.2, a software watermark can be

categorized by its robustness ("robust" or "fragile"), visibility ("visible" or "invisible"), implementation ("static" or "dynamic"), its recognition ("blind" or "informed") and its embedding ("spread spectrum" or "focused").

In some sense, the oldest types of software watermarks are visible watermarks in the form of copyright notices. These visible watermarks are displayed as the program starts, as a splash screen, as part of the help system or in some other form easily accessible by the end user. More generally, the "look and feel" of an application that is familiar to users is also a visible watermark. Although highly prevalent in image (and other forms of media) watermarking literature, there have been no publications on visible software watermarking.

Fragile watermarking is intended to authenticate the integrity of software [1]. Similar to visible watermarks, there has been very little research into fragile watermarks for software. The most common type of fragile software watermarking is digitally signed applications such signed Java jars [43] and .NET applications [44]. These can be considered a type of visible, fragile watermarks that allow no changes to the application once they have been watermarked (ie. signed).

Most existing research in software watermarking has been in robust watermarking. In the next four sections, I outline the state of the art in static and dynamic, robust watermarking.

7.2 Simple Robust Software Watermarking

The early software watermarks were simple static watermarks. Grover [45] mapped out a large area of software protection including techniques for obfuscation, watermarking and tamper-proofing software.

7.2.1 Moskowitz and Cooperman

Moskowitz and Cooperman [46] proposed a technique that relies on identifying essential or commonly executed sections of an application. Image watermarking techniques are then

used to embed a watermark as well as these essential sections of an application into image or audio data contained in the program. The execution of the watermarked application relies on the extraction of the code sections from these watermarked images and audio data distributed with the watermarked program. This watermarking technique is static, blind and focused.

7.2.2 Davidson and Myhrvold

The Davidson-Myhrvold watermarking algorithm [47] embeds the watermark by reordering basic blocks in a program. The ordering of the basic blocks encodes the value of the watermark, while the control-flow graph is altered to maintain the semantics of the original program. The only known implementation of the Davidson-Myhrvold algorithm is by Myles et al. [48] for the Java platform. Davidson and Myhrvold's proposed technique is static, blind and spread spectrum.

7.2.3 Akito Monden

Akito Monden and colleagues [49, 50, 51] introduced a static, blind and focused watermarking technique which involved injecting dummy methods into the program. These dummy methods, which are never executed, contain an encoding of the watermark in the choice of opcodes and numerical operands. The introduction of a dummy method gives the watermarker a great deal of freedom during the embedding. The fact that these methods will not be executed means that the only major constraint on the choice of these instructions is to maintain syntactic correctness and type consistency.

7.2.4 Stern et al.

Stern et al. [52] describe how to use the classic spread spectrum technique used in image watermarking to watermark software. The technique embeds the watermark by modifying the frequency instructions in the application by replacing groups of instructions with other groups of instructions which are semantically equivalent but have different

statistical properties. The embedding of the watermark induces a large number of almost imperceptible variations in the signal. A watermark detector that knows the location and nature of these variations is able to combine these small modifications into an amplified signal.

This algorithm is further discussed by Hachez [53] and an implementation is described by Sahoo and Collberg [54].

7.2.5 Comparison

In comparison to the work described in this thesis, these early watermarking schemes were largely ad hoc and lacked any theoretic or empirical indication of resilience. The embedded watermarks were easy to identify and were relatively cheap to distort. In particular, global transformations on programs that replace sets of a small number of instructions with different but equivalent instructions would effectively remove the watermark in each of the above static watermarks.

The effectiveness of this simple attack illustrates the primary vulnerability of simple static watermarking schemes. The functionality of software is its dynamic nature whereas in static watermarking schemes, the watermark information is carried in the representation. However, the nature of software is that there are many representations for the same functionality. This makes it difficult to robustly store information in the representation of software unless the representation is directly incorporated into the behaviour of the program.

Our work bypasses this difficulty by embedding the watermark in the dynamic behaviour of the program and thus forcing the attacker to distinguish between the legitimate behaviour of the program and the dynamic behaviour due to the watermark. We have presented a security analysis which has both its strengths and weaknesses. Each weakness has been disclosed, as clearly as possible, in a stated limitation. See Limitation 1 on page 57. Limitation 2 on page 96 and Limitation 3 on page 98.

7.3 Static Watermarking based on Graphs

There are two static watermarking techniques that use the structure of program graphs to encode their watermark. These are the Venkatesan algorithm [55] and the QP algorithm [56].

The Venkatesan algorithm utilizes a program control flow graph and is claimed by some to be the strongest current static watermarking technique [22]. The technique works by modifying a program such that its watermark can be deduced from its control flow graph. This watermark has "minimal overhead, a high degree of stealthiness, and with relatively high bit-rate" [57]. The Venkatesan watermarking technique relies crucially on a method to enumerate basic blocks in such a way that is reliable in face of attack. The original paper suggests using "padded data" as a flag. Unfortunately, it is trivial to apply a semantic-preserving transformation to every basic block that has a high probability of altering this flag. No evidence is cited in the paper [57] to indicate that a resilient enumeration of basic blocks can be constructed.

The QP algorithm uses the interference graph of a program. An interference graph of a program consists of nodes for each variable and edges between each pair of nodes which are live at the same time. The interference graph is used to allocate registers when a program is compiled. The QP algorithm introduces artificial dependencies in the interference graph and thus results in a sub-optimal register allocation. The difference between a program's given register allocation and the optimal allocation encodes the watermark. The original QP algorithm contained errors that made it unrealizable. These errors have been addressed by Collberg [58] and Zhu [59].

From the description given above, it is clear that an effective attack against the QP watermark is for the attacker to reallocate registers optimally. Unless steps are taken to prevent such a reallocation, the QP watermark falls to such an attack.

7.4 Other Static Watermarks

In 2002, Arboit [29] proposed a technique for embedding watermarks based on the work of Collberg and Monden. The Arboit watermark uses the choice of opaque predicates embedded in a program to encode a watermark. The watermark is recognized by searching and heuristically identifying the opaque predicates that have been used.

Arboit's technique requires a large library of opaque predicates to be effective. What is more, this library must be held in secret by the watermark embedder to prevent the attacker from decoding or removing the watermark.

In comparison to the technique described in this thesis, Arboit watermarks are difficult to detect reliably since heuristic matching is required if the opaque predicates are tampered in any way.

7.4.1 Cousot and Cousot

A watermark embedding based on abstract interpretation was proposed by Cousot and Cousot [60]. Abstract interpretation is the use of static analysis to make a sound approximation of the semantics of a program. The proposed watermarking scheme embeds the watermark in a non-standard semantic interpretation. It can be extracted given the correct abstract semantics but difficult or impossible otherwise.

Cousot and Cousot note that "abstract watermarking is different from existing static and dynamic watermarking methods". In our framework, abstract interpretation watermarking is considered a static watermarking technique since the program does not require execution for the watermark to be extracted.

7.5 Dynamic Watermarks

Dynamic watermarks were first described by Collberg and Thomborson [61, 9]. Dynamic watermarks are embedded in the run time execution behavior of the program. Collberg suggests a number of dynamic behaviors which could potentially carry watermarks in-

cluding data structures, execution traces and easter eggs.

7.5.1 CT watermark

The oldest and best understood dynamic watermarks were first described by Collberg et al. [61] and are known as CT watermarks. CT watermarks alter the original program so that a data structure that represents the watermark gets built on execution of the program with the correct input.

In order to implement these dynamic data structure watermarks, a system called SandMark [30] was developed at the University of Arizona. Sandmark provides a framework to watermark Java programs by modifying the application bytecode to make it build a structure at runtime that encodes the watermark. This structure is recognized as the watermark by dumping and analyzing the Java heap.

In the Sandmark implementation of the CT watermark, once the watermark data structure has been identified, an attacker is able to remove the watermark by removing those instructions that build it. In a different implementation, Palsberg et al. [62] used a tamper-proofing technique to make cropping the watermark more difficult. Palsberg tamper-proofs the CT watermark by embedding a second data structure in the watermarked application. The application is altered to use this data structure to build opaque predicates. If an attacker is unable to distinguish between the true watermark and the tamper-proofing data structure, then altering or removing the tamper-proofing data structure would cause the application to become incorrect.

This idea was greatly extended by He [63] and Thomborson et al. [4] to use constants extracted from the watermarking data structure itself.

7.5.2 Path-based Watermarking

A second dynamic watermarking technique, path-based watermarking, was proposed and implemented by Collberg et al. [64]. In path-based watermarking the watermark is embedded in the runtime branch structure of the program. Collberg suggests that the “runtime

branching structure is an inherent aspect of the program" and thus difficult to hide. There are two known implementations of path-based watermarking for the Java bytecode and for native x86. These are described at length by Collberg et al. [64].

7.5.3 Comparison

CT watermarking and Path-based watermarking is comparable to the technique described in this thesis. All three techniques are dynamic, focused and blind. They demonstrate a common feature of dynamic watermarking schemes in that they require a runtime trace step prior to embedding a watermark that depends on a keyed input. This trace is used to identify a locations in the program where a watermark can be embedded.

In contrast to the implementation of CT watermarks and path-based watermarks described above, in the implementation of thread-based watermarks we describe in this thesis, we have concentrated on preventing pattern matching attacks. We have shown that even dynamic watermarking algorithms, if designed or implemented carelessly can result in a watermark that can be detected statically using pattern matching. The fact that a watermarking scheme uses a dynamic trace during recognition does not necessarily imply that a watermark can not be detected or even its value read statically using an alternate method.

7.6 Software Analysis

The development of new techniques in both static and dynamic analysis will have a significant impact on what software watermarking techniques remain practical. A survey of relevant techniques is outside the scope of this thesis. For a comprehensive list of papers on tools for attacking software watermarks, the reader is referred to the Annotated Re-engineering Bibliography [65].

The two approaches traditionally used to perform static analysis are *abstract interpretation* and *theorem provers* [66].

7.6.1 Abstract Interpretation

Abstract interpretation build up a model a program and calculate a fix point of properties using this model [67]. In this thesis, we used JLint [41] to attack our thread-based watermark. JLint works by building a call graph of synchronized and unsynchronized methods in a program. An extension proposed by Artho and Biere [66] allows JLint to successfully extend its analysis to handle the kind of synchronized blocks used by TBW by considering them to be pseudo-methods. Although tools such as JLint are not directly useful for removing watermarking widgets from TBW watermarked applications, they can help identify *potential* watermarking widgets.

The discussion by Artho and Biere also highlights that "JLint is in practice as good as any currently available program at checking multi-threading problems". In spite of this, they point out that existing cases where "JLint's model is too simple" or "JLint could not deduce enough context from the program source code" to distinguish false deadlocks from real ones.

7.6.2 Theorem Provers

The theorem proving approach to program analysis involves deducing proofs of conjectures about the properties of a program. Artho [66] claims that:

> This rigorous mathematical approach is sound and complete, but typically involves human interaction since the general problem of proving program correctness is undecidable. Tools to support mathematicians in finding and executing proof steps exist [68, 69, 70] but typically require a strong background in mathematics and considerable skill with such proofs [71].

The difficulty of using theorem provers for analysis of programs is further compounded when used to analyze programs that have been deliberately obfuscated.

Earlier in this thesis, we described how analysis of TBW was carried out using BDDB-DDB. BDDBDDB represents a program as a database of relations using binary decision

diagrams (BDDs). Datalog, a declarative programming language similar to Prolog is then used to specify program analysis.

The key advantage of this style of program analysis in attacking watermarks is the flexibility in specifying analysis such scheme affords. John Whaley, the creator of BDDBDDB, states it enables "non-specialists to easily develop their own program analyses". In our experiments, BDDBDDB was unable to distinguish a relatively trivial opaque false and a opaque true predicate. However, it remains an open area of research what predicates can be distinguished using techniques like BDDBDDB.

8

Conclusion

In this thesis we have presented a new dynamic technique for watermarking software using threads. Our aim in this thesis was to build a resilient watermark that a software developer could use to prove that some given program contained her watermark and thus must contain at least some part of her software.

The TBW was designed on the premise that analyzing heavily multi-threaded programs is inherently difficult and the difficulty of this analysis can be leveraged to build a robust, invisible, dynamic watermark. In Chapter 3 we showed how to devise such a scheme using thread locks. The key novelty of this part of the thesis is that thread locks can be used to partially order the execution of some basic blocks of the program they are embedded in and this partial ordering can encode a bit of a watermark.

The use of partial, as opposed to complete, ordering in the design of TBW. Ambiguity

in the total order of basic blocks in a watermark widget prevents an adversary from cracking the widget by building tools capable of answering questions such as "Does Thread X execute block A followed by block C?" Due to the unpredictable nature of thread scheduling in computer systems, for our threads the answer is always "maybe". Furthermore, two executions of the same watermarked program on the same input may result in very different thread traces that nevertheless encode the same watermark.

We also introduce a new use for opaque predicates in this thesis. Traditionally, opaque predicates are used to protect blocks of code, usually introduced by a obfuscator or watermarker, that either must or must not be executed. The TBW uses opaque predicates to make different widgets appear identical to analysis. To achieve this, whenever there are differences between two widgets, it surrounds the difference with a predicate that is opaquely true for one widget and opaquely false for the other.

A third key contribution was to identify and resolve a difficulty in implementing dynamic software watermarks. Pattern matching attacks are where an adversary identifies a sequence of instructions that distinguishes a watermark from the rest of the program and uses this knowledge to attack the watermark. Pattern matching attacks are relatively simple, cheap and can be very effective at identifying watermarks. Using statistical techniques could make pattern matching attacks even more potent. As a defense against pattern matching, the widgets used in TBW are byte code equivalent everywhere except at the opaque predicates. Furthermore, some of these widgets are tamper-proofing widgets. In this way, any attempt to use pattern matching to identify or attack watermarking widgets will also identify tamper-proofing widgets. If tamper-proofing widgets are removed in the same way watermarking widgets are, the program becomes incorrect.

This is a powerful technique and can be generalized to other dynamic watermarking schemes. For example, CT dynamic watermarks may be susceptible to pattern matching attacks. If in addition to data structures that encode the watermark, some similar data structures were built that were critical to program execution, attempts to remove the watermark by pattern matching would fail. Two such solutions specific to CT watermarks have in fact been proposed by Palsberg [62] and by Thomborson et al. [4].

We built a working prototype that can be used to automatically embed a thread-based watermark and tested it on a variety of command-line and GUI applications. Experiments using this prototype showed the proposed solution successfully embeds a watermark and simultaneously increases threading complexity of a program. Empirically, we have shown that TBW is feasible and has a small impact on the runtime performance of the protected application.

A fourth contribution has been clearly identifying the limitations of thread-based watermarks to guide future work in this area. The limitations were:

- Limitation 1 [page 57] No attacker can perform a static analysis which will distinguish with 100% reliability between an opaque true and an opaque false predicate.
- Limitation 2 [page 96] An attacker is unable to guess the key input sequence.
- Limitation 3 [page 98] The attacker is unable to construct a copy of the TBW recognition code table.

8.1 Future Work

The protection of software using watermarks is a vast field and there are several areas that require research attention. This is necessary to pull the field out of a constant race with attackers and into the realm where we can be certain that de-watermarking a protected applications is sufficiently expensive as to be completely infeasible.

8.1.1 Dynamic Analysis

The most important open area identified in this thesis is that of dynamic analysis with respect to reverse engineering software. When evaluating the strength of our watermark, we, along with most existing researchers in the field, have limited ourselves to static analysis. The existence of debuggers, profilers and dynamic analysis tools open up a powerful class of tools for an attacker that is difficult to address. Especially potent and

worthy of attention is the amount of information available to an attacker who combines data from both static and dynamic analysis.

In addition to new dynamic analysis techniques, the impact of general dynamic visualization techniques [72, 73, 74] and more specific tools such as those that exist in SandMark [2, 7] are an important but currently poorly understood area of research.

8.1.2 Truly Opaque Predicates

The technique described in this thesis joins a growing number of obfuscation and watermarking tools that rely on our ability to generate a large family of good opaque predicates.

Currently no such provably opaque class of predicates are known, nor is it clear how to generate difficult instances of hard problems which can be used as the basis for opaque predicates. For opaque predicates to be effective a sufficiently large family of predicates are required such that pattern matching is not an effective attack.

Finally, it is also an open problem to generate sets of predicates such that value of any one predicate is unpredictable but some given function of the set of predicates is always true or always false. The existence of such setwise opaque predicates could make simplify the construction of TBW widgets and be useful in other types of watermarking and obfuscation algorithms.

8.2 Final Remarks

In conclusion, the findings of this thesis can be summarized by the following three points:

- Given a limited attacker who is only able to perform static analysis of programs and given a method for manufacturing opaque-predicates, thread contention can to embed information resiliently.

- Thread-based watermarking slows down a program and increases its size. The slowdown in speed and increase in size are both linearly dependent on the number of bits embedded.

- Given a method for manufacturing opaque predicates, one method for minimizing the impact of pattern matching attacks against software watermarks is to use opaque predicates to merge the static differences between two pieces of code.

Bibliography

[1] J. Nagra, C. Thomborson, and C. Collberg, *A Functional Taxonomy for Software Watermarking*, in *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, edited by M. J. Oudshoorn, Melbourne, Australia, 2002, ACS.

[2] C. S. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler, *A System for Graph-Based Visualization of the Evolution of Software*, in *SOFTVIS 2003*, pages 77–86, 212–213, 2003.

[3] J. Nagra and C. Thomborson, *Threading Software Watermarks*, in *Proc. 6th International Workshop on Information Hiding*, edited by J. Fridrich, LNCS 3200, pages 208–233, Springer-Verlag, 2004.

[4] C. Thomborson, J. Nagra, R. Somaraju, and C. He, *Tamper-proofing software watermarks*, in *CRPIT '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 27–36, Darlinghurst, Australia, Australia, 2004, Australian Computer Society, Inc.

[5] J. Nagra and C. Thomborson, *Method of introducing digital signature into software*, US Patent Pending 20050262490A1, 2005, Avaliable at `http://www.delphion.com/details?pn=US25262490A1`.

[6] J. Nagra and C. Thomborson, *Method of introducing digital signature into software*, NZ Patent 533028, 2006.

[7] J. Nagra, C. Collberg, S. Kobourov, and K. Wampler, *Temporal Heap Visualization of Large Java Programs*, Submitted to Information Visualization, 2006.

[8] Free Software Foundation, *The GNU Manifesto*, Online publication, 1985, Available at `http://www.gnu.org/gnu/manifesto.html`.

[9] C. Collberg and C. Thomborson, *Software Watermarking: Models and Dynamic Embeddings*, in *Proceedings of Symposium on Principles of Programming Languages, POPL'99*, pages 311–324, 1999.

[10] G. Naumovich and N. Memon, *Preventing Piracy, Reverse Engineering, and Tampering*, Computer **36**, 64 (2003).

[11] W. S. A. Proesbsting, T. A., *Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)*, in *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, pages 185–197, 1997.

[12] C. Cifuentes and K. J. Gough, *Decompilation of Binary Programs*, Software - Practice & Experience , 811 (1995).

[13] C. Wang, *A security architecture for survivability mechanisms*, PhD thesis, 2001, Adviser-John Knight.

[14] S. Midkiff and D. Padua, *Issues in the optimization of parallel programs*, in *Proceedings of the 1990 International Conference on Parallel Processing*, pages 105–113, 1990.

[15] D. Grunwald and H. Srinivasan, *Data flow equations for explicitly parallel programs*, in *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[16] J. Knoop, B. Steffen, and J. Vollmer, *Parallelism for free: Efficient and optimal bit vector analysis for parallel programs*, in *ACM Transactions of Programming Languages and Systems*, volume 18, pages 268–299, 1996.

[17] R. Rugina and M. Rinard, *Pointer analysis for Multithreaded Programs*, in *SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–90, 1999.

[18] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[19] J. K. Ousterhout, *Why threads are a bad idea (for most purposes)*, Invited Talk at Usenix Technical Conference, 1996, Slides avaliable at `http://www.sunlabs.com/~ouster/`.

[20] C. Leiserson and H. Prokop, *A minicourse on multithreaded programming*, MIT OpenCourseWare lecture notes, 1998, Available from `http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-895Fall2003/BA739669-840E-4447-9C66-1781808F667E/0/minicourse.pdf`.

[21] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[22] C. S. Collberg and C. Thomborson, *Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection*, in *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, 2002.

[23] R. N. Taylor, *Complexity of analyzing the synchronization structure of concurrent programs*, Acta Informatica **19**, 57 (1983).

[24] M. Dwyer, *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*, PhD thesis, Amherst, MA, USA, 1995.

[25] D. L. Bruening, *Systematic Testing of Multithreaded Java Programs*, Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.

[26] I. Cox and J.-P. Linnartz, *Some General Methods for Tampering with Watermarks*, IEEE Journal on Selected Areas in Communications **16**, 587 (1998).

[27] C. Collberg, C. Thomborson, and D. Low, *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs*, in *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

[28] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, *Protection of Software-Based Survivability Mechanisms*, in *DSN01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 193–202, Washington, DC, USA, 2001, IEEE Computer Society.

[29] G. Arboit, *A Method for Watermarking Java Programs via Opaque Predicates*, in *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[30] C. Collberg, G. Myles, and A. Huntwork, *Sandmark–A Tool for Software Protection Research*, IEEE Security and Privacy **1**, 40 (2003).

[31] D. Curran, N. Hurley, and M. O. Cinneide, *Securing Java through Software Watermarking*, in *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 311–324, 2003.

[32] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, Addison-Wesley Longman Publishing Co., Inc., 1999.

[33] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification, Second Edition: The Java Series*, Addison-Wesley Longman Publishing Co., Inc., 2000.

[34] M. Odersky and P. Wadler, *Pizza into Java: Translating Theory into Practice*, in *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159, ACM Press, New York (NY), USA, 1997.

[35] M. Kizub, *Kiev language specification*, Online publication, 1998, Available from `http://www.forestro.com/kiev/`.

[36] N. Shaylor, *Java preprocessor (JPP)*, Online publication, 1996, Available from `http://www.geocities.com/CapeCanaveral/Hangar/4040/jpp.html`.

[37] W. Landi and B. G. Ryder, *Pointer-induced Aliasing: A Problem Classification*, in *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, New York, NY, USA, 1991, ACM Press.

[38] J. E. F. Friedl, *Mastering Regular Expressions*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[39] C. Collberg, C. Thomborson, and G. Townsend, *Dynamic Graph-Based Software Watermarking*, Technical report, Dept. of Computer Science, Univ. of Arizona, 2004.

[40] J. Miecznikowski and L. Hendren, *Decompiling Java using staged encapsulation*, in *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 368, IEEE Computer Society, 2001.

[41] K. Knizhnik, *JLint*, Online publication, 2004, Avaliable at `http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm`.

[42] J. Whaley and M. S. Lam, *Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams*, in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[43] S. Oaks, *Java security*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[44] A. Freeman and A. Jones, *Programming .NET Security*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[45] D. Grover, *The Protection of Computer Software: Its Technology and Applications*, The British Computer Society Monographs in Informatics, Cambridge University Press, second edition, 1992.

[46] S. A. Moskowitz and M. Cooperman, *Method for Stega-Cipher Protection of Computer Code*, US Patent number 5,745,569, 1996.

[47] R. L. Davidson and N. Myhrvold, *Method and System for Generating and Auditing a Signature for a Computer Program*, US Patent number 5,559,884, 1996.

[48] G. Myles, C. Collberg, Z. Heidepriem, and A. Navabi, *The Evaluation of Two Software Watermarking Algorithms*, Software: Practice and Experience **35**, 923 (2005).

[49] A. Monden et al., *A Watermarking Method for Computer Programs (in Japanese)*, in *Proceedings of the 1998 Symposium on Cryptography and Information Security, SCIS'98*, Institute of Electronics, Information and Communication Engineers, 1998.

[50] A. Monden, H. Iida, K. ichi Matsumoto, K. Inoue, and K. Torii, *Watermarking Java Programs*, in *International Symposium on Future Software Technology '99*, pages 119–124, 1999.

[51] A. Monden, H. Iida, and K. ichi Matsumoto, *A Practical Method for Watermarking Java Programs*, in *The 24th Computer Software and Applications Conference*, pages 191–197, 2000.

[52] J. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater, *Robust Object Watermarking: Application to Code*, in *Information Hiding Workshop '99*, pages 368–378, 1999.

[53] G. Hachez, *A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards*, PhD thesis, Universite Catholique de Louvain, 2003.

[54] T. R. Sahoo and C. Collberg, *Software Watermarking in the Frequency Domain*, Technical Report TR04-07, Department of Computer Science, University of Arizona, 2004.

[55] R. Venkatesan, V. V. Vazirani, and S. Sinha, *A Graph Theoretic Approach to Software Watermarking*, in *IHW '01: Proceedings of the 4th International Workshop on Information Hiding*, pages 157–168, London, UK, 2001, Springer-Verlag.

[56] G. Qu and M. Potkonjak, *Analysis of Watermarking Techniques for Graph Coloring Problem*, in *IEEE/ACM International Conference on Computer Aided Design*, pages 190–193, 1998.

[57] C. S. Collberg, A. Huntwork, E. Carter, and G. M. Townsend, *Graph Theoretic Software Watermarks: Implementation, Analysis, and Attacks.*, in *Information Hiding*, edited by J. J. Fridrich, volume 3200 of *Lecture Notes in Computer Science*, pages 192–207, Springer, 2004.

[58] G. Myles and C. Collberg, *Software Watermarking Through Register Allocation: Implementation, Analysis, and Attacks*, in *International Conference on Information Security and Cryptology*, 2003.

[59] W. Zhu and C. Thomborson, *Algorithms to Watermark Software through Register Allocation*, in *DRMTICS 2005*, Sydney, Australia, 2005, Springer.

[60] P. Cousot and R. Cousot, *An abstract interpretation-based framework for software watermarking*, in *Principles of Programming Languages 2003, POPL'03*, pages 311–324, 2003.

[61] C. Collberg and C. Thomborson, *On the Limits of Software Watermarking*, Technical Report 164, Department of Computer Science, Univ. of Auckland, 1998, Available from `http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson98e/`.

[62] J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, Q. Shao, and Y. Zhang, *Experience with software watermarking*, in *Proceedings of the 16th Annual Computer Security Applications Conference, ACSAC '00*, pages 308–316, IEEE, 2000.

[63] Y. He, *Tamperproofing a Software Watermark by Encoding Constants*, Master's thesis, University of Auckland, 2002.

[64] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp, *Dynamic Path-Based Software Watermarking*, in *SIGPLAN '04 Conference on Programming Language Design and Implementation*, 2004.

[65] R. Koschke, *Annotated Reengineering Bibliography*, Online publication, 2005, Available at `http://www.iste.uni-stuttgart.de/ps/reengineering/index.html`.

[66] C. Artho and A. Biere, *Applying Static Analysis to Large-Scale, Multi-threaded Java Programs*, in *Proc. 13th ASWEC*, edited by D. Grant, pages 68–75, IEEE Computer Society, 2001.

[67] P. Cousot and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, ACM, 1977.

[68] J.-R. Abrial, *The B-book: assigning programs to meanings*, Cambridge University Press, New York, NY, USA, 1996.

[69] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hhnle, W. Menzel, and P. Schmitt, *The KeY approach: Integrating object ori- ented design and formal verification*, in *Proc. 7th European Workshop on Logic in Artificial Inteligence (JELIA 2000)*, volume 1919, pages 21–36, Mlaga, Spain, 2000, Springer.

[70] J. Schumann, *Automated Theorem Proving in High-Quality Software Design*, in *Intellectics and Computational Logic*, pages 295–312, Deventer, The Netherlands, The Netherlands, 2000, Kluwer, B.V.

[71] D. A. Peled, D. Gries, and F. B. Schneider, editors, *Software reliability methods*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[72] B. A. Myers, *Taxonomies of Visual Programming and Program Visualization*, Journal of Visual Languages and Computing **1**, 97 (1990).

[73] B. A. Price, I. S. Small, and R. M. Baecker, *A Taxonomy of Software Visualization*, in *Proc. 25th Hawaii Int. Conf. System Sciences*, 1992.

[74] G.-C. Roman and K. C. Cox, *A Taxonomy of Program Visualization Systems*, IEEE Computer **26**, 11 (1993).