Threading Software Watermarks

Jasvir Nagra and Clark Thomborson

Department of Computer Science, University of Auckland
Auckland, New Zealand
{jas,cthombor}@cs.auckland.ac.nz

Abstract. We introduce a new dynamic technique for embedding robust software watermarks into a software program using thread contention. We show the technique to be resilient to many semantic-preserving transformations that most existing proposals are susceptible to. We describe the technique for encoding the watermark as a bit string and a scheme for embedding and recognizing the watermark using thread contention. Experimental results with Java bytecode indicate that thread based watermarks have small impact on the size of applications and only a modest effect on their speed.

1 Introduction

Software watermarking is a technique for embedding an identifier into a piece of software in order to encode some identifying information about it. This identifying information can be used to demonstrate ownership; and in cases of piracy, may make it possible to trace software to the source of its illegal distribution. Watermarking has received an increasing amount of interest from the research community which has resulted in increasingly resilient techniques. However, no single watermarking algorithm has emerged that is effective against all existing and known attacks. In fact, it is generally agreed that it is not possible to devise a watermark that some sufficiently determined attacker would not be able to defeat. As a result, the goal of the watermarking community is to develop techniques that are sufficiently robust that the resources required to defeat the watermark are too expensive to be worth the attackers while.

In this paper, we propose a new technique for software watermarking, *thread-based watermarking*, for embedding and detecting a watermark using thread contention. Our premise is that multithreaded programs are inherently more difficult to analyse and the difficulty of analysis increases with the number of threads that are "live" concurrently [18].

Software watermarks can be used for different purposes and their desirable properties vary depending on their use [15]. For software piracy the two properties that interest us are "robustness" and "invisibility". "Robustness" ensures that the watermark is difficult for an attacker to remove and thus the watermark can act as a software intellectual property identifier. "Invisibility" means that the watermarks are designed to be non-apparent to the end-user and thus do not interfere with legitimate use of the program.

Our proposed technique embeds the watermark in the order and choice of threads which execute different parts of an application. The embedding is a

two step process. Firstly, we increase the number of possible paths through the program by creating multiple threads of execution. The semantics of the old program are maintained by introducing locks. Secondly, we add other locks to ensure that only a small subset of the possible paths are in fact executed by the watermarked program. The particular paths that are executed encode the watermark.

The rest of the paper is organized as follows. In Section 2, we give an overview of the state-of-the-art in watermarking literature and other related work. In Section 3, we give an overview of the basic idea behind thread based watermarks. Section 4 describes how thread based watermarks can be implemented for Java bytecode. Section 5 gives experimental evaluation of our technique. Finally, Section 6 gives future directions and conclusions.

2 Related Work

There are several other published techniques for doing software watermarking, static watermarks and dynamic watermarks. The earliest software watermarks were static watermarks where the watermark was embedded in either the code section (eg. in variable names, order of exectuable statements) or in the static data sections (eg. in the strings, images, headers) of a program [8]. Moskowitz [14] describes such a scheme in which the watermark is embedded in an image or other digital media using any known media watermarking scheme. The image is in turn embedded in the static data section of the program, and the watermark is extracted at runtime. This fragile watermark is necessary for program correctness.

A more advanced kind of static code watermark was introduced by Davidson and Myhrvold [7]. The technique involved statically encoding the watermark in the ordering of basic blocks that constitute program. Another code watermark was introduced by Monden [13] which involved injecting dummy unexecuted methods into the program. These dummy methods contain an encoding of the watermark in the choice of opcodes and numerical operands. A comparable spread spectrum technique was introduced by Stern et al. [22] for embedding a watermark by modifying the frequencies of instructions in the program.

Instead of watermarking the code or data sections of a program, Sander and Tschudin [20] introduce a technique for watermarking a function by embedding information statically in the I/O interface between the client and the server.

Static watermarks are particularly susceptible to obfuscation attacks. Two such attacks described by Collberg et al. [4] involve breaking and scattering all strings and other static data around the program and/or replacing this static data with code that generates the same data at runtime. Both these attacks are extremely effective in making watermark detection impractical.

Perhaps the strongest known static watermark was introduced by Venkatesan et al. [23] which involves modifying a program so that its control flow graph encodes the watermark represented as a graph.

Dynamic data structure watermarks were introduced by Collberg and Thomborson [2]. These watermarks alter the original program so that a data structure that represents the watermark gets built whenever the program is run with the correct input. In order to implement these dynamic data structure watermarks, a system called SandMark [5] was designed jointly at the University of Auckland and 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.

Historically, watermarking has not been the only technique used for protection of intellectual property of software. Other techniques include the use of a registration database [9] [21], hardware cryptography [17], obfuscation [4] and tamper-proofing [1]. Furthermore, research has been conducted into using "software birthmarks", which are preexisting properties of a piece of software, to establish the authorship of a program [8] [11].

In Collberg et al. [5] the authors suggest using thread contention, but as a possible technique for obfuscating the execution of a program, not for watermarking. This paper gives the first practical method for software watermarking using thread contention.

3 Thread Based Watermarks

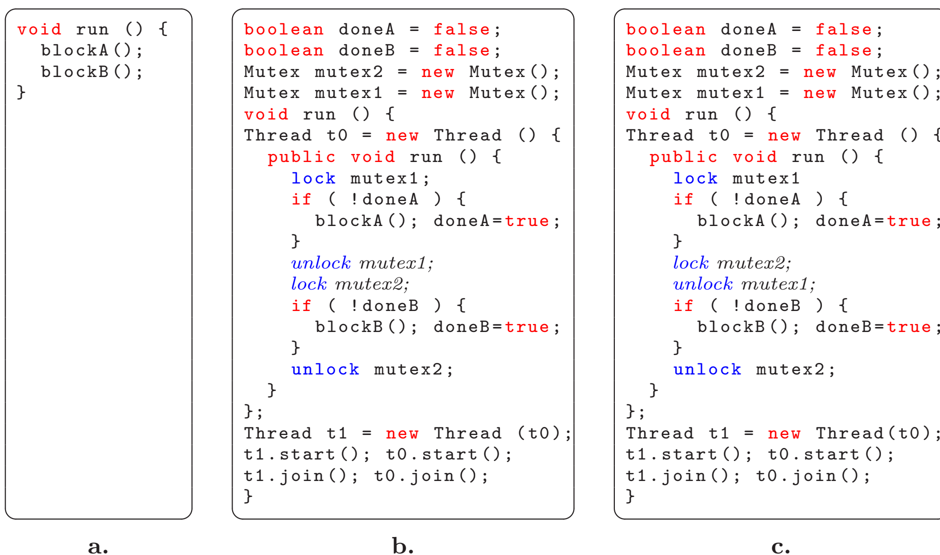

Fig. 1. In **a.** we have the original program. **b.** shows a multithreaded but unconstrained version of the original program. There are four different correct paths through this program, all of which may be executed. **c.** shows a multithreaded and constrained version of the original program. In this version, although both threads contend to execute `blockA` which ever thread executes the first block also executes the second one because of the order of locks.

We describe a new watermarking algorithm, *thread based watermarking*, where the basic idea is to embed the mark in the threading behavior of the program. Our proposed technique relies on introducing new threads into single threaded sections of a program. In an unsynchronized multithreaded 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 of how the threads are scheduled.

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 multithreaded execution and the number of factors that can affect the timing of thread execution, the particular thread that acquires the lock is difficult to predict and appears to be largely random [18].

In order to embed our watermark, we take advantage of the fact that although thread contention appears to be random, 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.

For example consider Figure 1a which shows a simple snippet of a program with a `run()` method that calls other methods `blockA()` and `blockB()`. We could introduce new threads into the program to execute each of the statements as show in Figure 1b. This version of the program remains correct and semantically equivalent to the original, however, there a several paths of execution with either `t0` or `t1` executing `blockA()` followed by either `t0` or `t1` executing `blockB()`. In order to embed information into a program, we manipulate the locks so that only a given subset of paths through the code is taken. In Figure 1c, we show one example of such manipulation. In this example, although the two new threads race to acquire a lock on `mutex1` like before, in this case whichever thread locks this mutex is also guaranteed to lock `mutex2` and thus executes both `blockA()` and `blockB()`. We can detect this scenario as distinct from the case where different threads execute `blockA()` and `blockB()` and thus we can use it to embed a bit of information.

The advantage of allowing some thread contention to remain is that although it allows a bit to be embedded, the actual path of execution still changes every time the program is executed. This makes the attackers task of determining which exact sequence embeds the mark more difficult. We discuss this resilience to attack more in the Section 5.

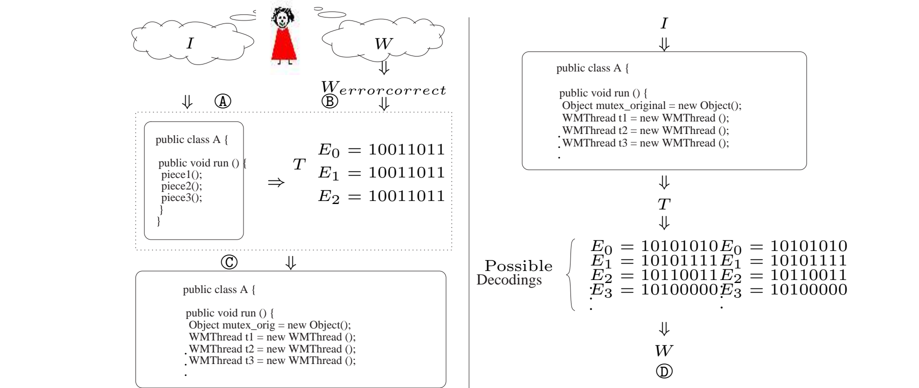

Fig. 2. Overview of thread based watermarking

4 Watermarking Java Bytecode

We have implemented thread based watermarking for Java bytecode. This implementation consists of three stages. In the tracing phase, the dynamic behaviour of the program is captured by tracing its execution on a secret input, I. In the embedding phase, the watermark number W is selected by the user and embedded in the input code by modifying the behavior of the program on the secret input I. Finally, in the recognition phase, the program is traced again with input I, and the watermark is extracted from the trace.

Figure 2 illustrates the watermarking process. In Ⓐ the original program is annotated for tracing and executed with the secret input I that the user selects. In Ⓑ the user selects a watermark string and encodes it using some encoding scheme. In Ⓒ watermark code is inserted into the original program. When the watermarked program is executed with the special input sequence in Ⓓ, the resulting trace will contain the watermark.

4.1 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. A basic block is a piece of straight line code without any jumps or jump targets. We instrument the input program to write a program trace to a file and execute the program with the secret input I. The trace is a series of tuples (B_i, T_i) where B_i is the block id of every basic block executed and T_i is the id of the thread that executed B_i. The watermark is embedded in the execution behavior of input program and as such we select input I such that for a given thread T_n, the sequence $T =< B_0, B_1, ..., B_n >$ is reproducible on different runs.

The program trace serves two purposes. Primarily it 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. Secondarily, the program trace counts how often each basic block gets executed and thus helps identify tight loops, recursion and other program hotspots. There is a computational and thread switching runtime cost associated with inserting new threads into the program and to avoid excessive slow down, we avoid inserting watermarks in to these hotspots.

The secret input I acts as the key and the watermark will be expressed when this secret input is entered. Other inputs may express other watermarks. Keeping this input a secret impedes an attacker who gains access to the recognizer from mounting the so called Oracle attack [6] which can be used to create a non-watermarked program when a watermark recognizer is available.

4.2 Embedding

The embedding phase modifies the input code so that the watermark W can be extracted from a trace of basic blocks executed on the input sequence I, as described in Section 4.1.

In our prototype design, we encode a 24-bit watermark string W into a 48-bit string E, using a randomly chosen code. The extreme sparseness of this code gives us a strong error-detection property which we will use in our recognition step: if a 48-bit string is chosen uniformly at random from the set $\{0,1\}^{48}$, the probability of this string being a legal codeword is only 2^{-24}.

We split our 48-bit string into six 8-bit bytes $E = < E_0, E_1, ..., E_5 >$. Each byte is embedded separately. For each byte, we select a thread T_i at random and a subsequence of $T < (B_0, T_i), ..., (B_n, T_i) >$ - that is a set of n basic blocks executed by T_i in the order of execution. To simplify embedding we ensure that we select n distinct basic blocks - that is we select B_i such that $\forall i, j : i \neq j \rightarrow B_i \neq B_j$.

As mentioned earlier thread switching code is expensive in time. Basic blocks that are executed repeatedly are poor candidates for embedding as slowing them down will significantly deteriorate the overall performance of the program. Furthermore we select some of the basic blocks that are input dependent to make the value of the expressed watermark vary with I.

In order to embed our watermark we require our chosen thread to be able to execute an arbitrary piece of code that it is passed. Thus we first extend the java `Thread` class so that threads can be passed a closure to execute. A closure is a data structure that contains an expression and an environment of variable bindings in which the expression is to be evaluated. There is no direct support for closures in Java. However, several techniques for implementing closures in Java exist in literature. In particular, Pizza [16] describes two schemes for implementing closures in Java. In our implementation a closure is translated into a class that implements the `Runnable` interface. This interface contains a single `run()` method. The body of the closure is inserted into the `run()` method of

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

A closure allows the introduced threads to access and possibly alter the local variables used by the basic block. Unfortunately, formal parameters in Java are passed by value and we need some mechanism by which to pass updates out of the function body. In our implementation we construct a `Locals` class for every closure in which all variables used by the closure are captured. When the closure is instantiated we pass this environment to it.

We insert into each basic block B_i code that causes the threads to switch in such a way as to encode E_i. A simple implementation is shown in Figures 3 and 4.2.

In our implementation, a bit 0 is encoded as a sequence of three basic blocks executed by three different threads. A bit 1 is encoded as a sequence of three basic blocks, where the first and third basic blocks are executed by the same thread and the second basic block is executed by a different thread. The advantage of such an encoding scheme over one that explicitly uses named blocks and threads is that it is more resilient to renaming attacks.

We use Java monitors to control the ordering of locks. The only mechanism in the Java language for manipulating monitors is the `synchronized` statement which acquires a lock on an object before executing a block and then releasing it. The synchronized statement requires all lock and unlock calls to be fully nested and is not sufficiently expressive for our purposes. Thus to we use the macros `monitor_enter()` and `monitor_exit()` in the source code of our examples. These expand to `monitor_enter` and `monitor_exit` calls in Java bytecode, and have the advantage that they cannot be decompiled to synchronized statements in Java source. This provides some defense against decompilation attacks.

```
WMThread t1;
WMThread t2;
WMThread t3;
int[] wm = { 1,0,1,1,1,0,1,0 };
...
for ( int i=0; i < wm.length; i++ ) {
  embedBit_macro ( t1, t2, t3,
  Bit0_Closure );
} else {
  embedBit_macro ( t1, t2, t3,
  Bit1_Closure );
}
```

a.

```
embedBit_macro ( t1, t2, t3, body ) {
  Object mutex_orig = new Object ();
  t1.setBody ( body );
  t2.setBody ( body );
  t3.setBody ( body );
   monitor_enter ( mutex_orig );
  t1.start(); t2.start(); t3.start();
  while ( t1.isAlive() &&
          t2.isAlive() &&
    t3.isAlive() )
  { Thread.yield() }
  monitor_exit ( mutex_orig );
  t1.join(); t2.join(); t3.join();
}
```

b.

Fig. 3. Part a shows the code inserted to embed the bits 10111010. The `embed_bit_macro` call is the macro that expands as shown in Part b. The `setBody` method takes a closure as its argument.

The problem with the simple implementation of Figures 3 and 4.2 is that the inserted threads do not in fact perform any computation and as such are conspicuous as well as easily removed. In order to tamperproof the watermark we use the new threads to perform the computation that was originally occur-

```
boolean doneA, doneB, doneC, doneD;
doneA=doneB=doneC=doneD=false;
Object mutex0 = new Object ();
Object mutex1 = new Object ();
monitor_enter ( mutex0 );
if ( !doneA ) {
  doneA = !doneA;
  monitor_enter ( mutex1 );
  monitor_exit ( mutex0 );
  monitor_enter ( mutex_orig );
  monitor_exit ( mutex_orig );
}
if ( !doneB ) {
  doneB = !doneB;
  monitor_exit ( mutex0 );
  monitor_enter ( mutex1 );
  monitor_enter ( mutex_orig );
  monitor_exit ( mutex_orig );
}
if ((!doneC && opaque_true) ||
   (( doneC && opaque_false) ||
   (doneD && opaque_false )) ) {
  doneC = !doneC;
  if ( doneD )
    monitor_exit ( mutex1 );
  else {
    monitor_exit ( mutex1 );
    doneD = !doneD;
  }
} else {
  doneC = !doneC;
  monitor_exit ( mutex0 );
}
```

```
boolean doneA, doneB, doneC, doneD;
doneA=doneB=doneC=doneD=false;
Object mutex0 = new Object ();
Object mutex1 = new Object ();
monitor_enter ( mutex0 );
if ( !doneA ) {
  doneA = !doneA;
  monitor_enter ( mutex1 );
  monitor_exit ( mutex0 );
  monitor_enter ( mutex_orig );
  monitor_exit ( mutex_orig );
}
if ( !doneB ) {
  doneB = !doneB;
  monitor_exit ( mutex0 );
  monitor_enter ( mutex1 );
  monitor_enter ( mutex_orig );
  monitor_exit ( mutex_orig );
}
if ((!doneC && opaque_false) ||
   ((doneC && opaque_true) ||
   (doneD && opaque_true) ) {
  doneC = !doneC;
  if ( doneD )
    monitor_exit ( mutex1 );
  else {
    monitor_exit ( mutex0 );
    doneD = !doneD;
  }
} else {
  doneC = !doneC;
  monitor_exit ( mutex1 );
}
```

Fig. 4. Implementation of `Bit0_Closure`(left) and `Bit1_Closure`(right). The only differences between the implementations have been highlighted.

ring in the basic block. Firstly we divide the selected basic block into three pieces, `piece1()`, `piece2()` and `piece3()` with each piece containing zero or more instructions and construct a closure around them. We then pass these new closures along with those that implement the watermarks to the new threads for execution as shown in our final implementation at Figure 5 and Figure 6.

In Figure 5 we embed a bit 0. The original thread T_{orig} locks $mutex_{orig}$ then forks of three new threads T_0, T_1 and T_2 which are executing identical closures. It then waits for these threads to terminate. The three new threads contend for $mutex_0$ and the winner proceeds to execute LA1 as shown in Figure 5. This causes `piece1()` to be executed by the winner while the other threads wait. The body of the threads are identical and because the cases are symmetric, let us assume T_0 wins the lock. T_0 proceeds to execute LA1 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 LB1 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_1 or T_2 to wake

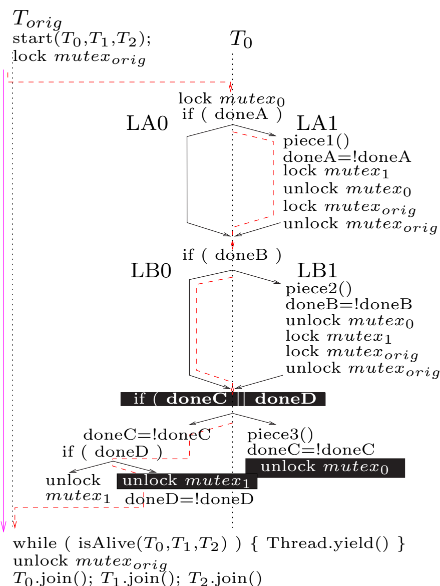

Fig. 5. Embedding a bit 0: A control flow diagram of T_{orig} and the three threads T_0, T_1 and T_2 executing an identical body. The threads T_1 and T_2 are identical to T_0 and are not shown. One possible path of execution of these threads is that T_0 executes `LA1` and `LB0`; T_1 executes `LA0` and `LB1`; and T_2 executes `LA0` and `LB0`. Continuing this path T_2 will execute `piece3()`.

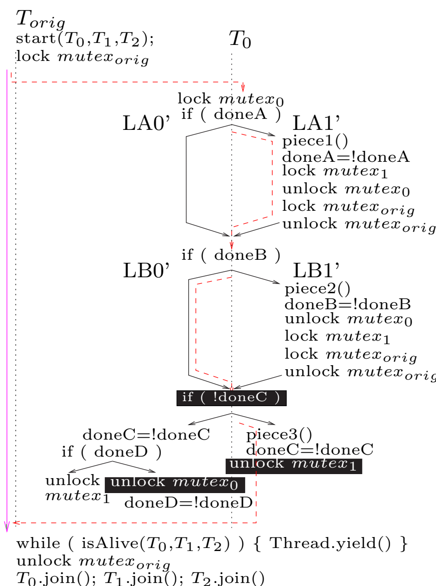

Fig. 6. Embedding a bit 1: A control flow diagram of T_{orig} and the three new threads T_0, T_1 and T_2. To execute LA1' and LB0', T_1 executes LA0' and T_2 executes LA0' and LB0' Continuing this path T_0 will execute `piece3()`. This figure is identical to Figure 5 except where shown in **reverseface**.

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. As a result of this execution, three distinct threads have executed the three `pieces` thus embedding a bit 0.

In Figure 6 we embed a bit 1. The behavior of the threads is identical to embedding bit 0 until T_2 evaluates the third conditional marked ⌜`!doneC`⌝. 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()` thus embedding a bit 1.

The introduced code is carefully constructed so that the only differences between the embedding of bit 0 and bit 1 are the arguments to `unlock` and the third conditional as shown in Figure 6.

The first of these differences, the arguments to `unlock` is obscure to an attacker because in Java `monitor_enter` and `monitor_exit` are stack operations. Thus it is not possible to statically pattern match on the code to determine if a 0 or a 1 bit is being embedded. Furthermore, it is difficult given the stack

operations to determine purely statically which object $mutex_0$ or $mutex_1$ will be on top of the stack when `unlock` is called.

The second of these differences may allow an attacker to pattern match on the conditional statements (⌜`!doneC`⌝ versus ⌜`(doneC || doneD)`⌝ to distinguish between an embedding of 0 and an embedding of 1. To prevent this, we use *opaque predicates* to fold the two different expressions into one. An opaque predicate [3,4] is an expression whose value is known to the watermarker at time of watermarking but which is difficult for the attacker to deduce.

An opaque false predicate is an opaque predicate which is always false whilst an opaque true predicate is one which is always true. We are able to construct a single expression of the form: ⌜`(!doneC &&` X_{opaque} `) || ((doneC &&` Y_{opaque}`) || (doneD &&` Z_{opaque}`))`⌝

To embed bit 0 as shown in Figure 4a, we set X to be opaquely true and Y and Z to be opaquely false, thus reducing the expression to ⌜`(!doneC)`⌝. Alternately, to embed bit 1 as shown in Figure 4b, we set X to be opaquely false and Y and Z to be opaquely true thus reducing the expression to (doneC || doneD) as required.

The opaque predicates can be selected from a large library of opaque predicates such as described by Collberg et. al. [4] which makes pattern matching or static analysis of this expression useless in distinguishing between an embedding of bit 0 or bit 1.

4.3 Recognition

Watermark recognition involves identifying our original watermark in a possibly tampered piece of code. As discussed in Section 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.

Watermark recognizers can be broadly classified as "detectors" - those that merely report the presence of a watermark and "extractors" - those that return the encoded value of the watermark. We can build a detector for our watermark by the following method.

First, we extract information about the threading behaviour of the watermarked program. We begin by collecting a trace of its execution on secret input I, using a technique similar the one described in Section 4.1. During detection, we are only interested in the transition from one thread to the next. Therefore given two consecutive tuples in the trace, $(B_i, T_i), (B_{i+1}, T_j)$, we only record a thread ID, T_i if $i \neq j$. This results in $T =< T_0, T_1, ..., T_m >$ which is a list of thread IDs in basic block execution order.

We select every combination of three distinct thread IDs that occur in T and form a subsequence with just these threads. Note that if there are 4 thread IDs, then we form $\binom{4}{3} = 4$ subsequences. In general, we form $\binom{m}{3} = O(k^3)$ subsequences from a trace T containing m thread IDs. From these subsequences, we then reconstruct all possible 8-bit watermark bytes by extracting all thread-transition sequences of length 24; recall that we embed each bit of an 8-bit

watermark code byte as a sequence of three thread-transitions. Our final step is to construct all possible 6-byte sequences, testing whether each of these is a legal codeword. We can quickly test whether each codeword is valid by hash-lookup of a 48-bit possibly-valid code in a table with approximately 16 million (2^{24}) valid 48-bit codes E. The appropriate 24-bit watermark signal W is stored with each valid code.

Two of our three benchmarks are single-threaded, so our extraction process is quite straightforward. During the extraction process on these benchmarks, almost all thread-transitions are due to our watermark, so our error-detection code is not heavily used. Our JFig benchmark is multi-threaded, however, with 7 threads and 47 thread-transitions when it is run on our secret input before watermarking. After watermarking, JFig has a total of 25 threads, because we add three threads for each byte in our 6-byte encoded watermark E. There are $\binom{25}{3} = 2300$ different ways to select three threads from twenty-five threads; only six of these thread-choices will reveal a valid byte from our encoded watermark. All other choices will give spurious signals, and most of these signals cannot be properly sequenced with five other bytes E_i to form a 48-bit possibly-correct codeword E. In our preliminary experimentation (although we are not confident of the correctness of our implementation) our reconstruction process generates less than 100 possible 48-bit codewords E for our watermarked JFig. This is well within the error-detection capacity of our encoding process: we'd estimate a false-extraction error rate of less than $100/2^{24}$ under the reasonable assumption, which has yet to be experimentally verified, that the spurious codewords are uncorrelated with our randomly-chosen encoding scheme.

4.4 Experimental Results

Experiments were performed on three pieces of software: `TTT`, a trivial tic-tac-toe program; `JFig`, a figure editor; and `SciMark`, a java benchmark. This latter benchmark is a composite benchmark consisting of five computational kernels used for measuring the performance of numerical codes occurring in scientific and engineering applications. The programs were selected for experimentation because they categorize different types of Java programs that may be watermarked. `TTT` is a small GUI program (64 lines) with one major loop and all but 4 of the lines in the program are executed on our sample input. `JFig` is a much larger GUI program ($\approx$ 23000 lines) with most lines of code never being executed. The `SciMark` benchmark ($\approx$ 1300 lines) is a non-GUI application that consists of many tight loops optimized for numerical computations. A significant number of lines (5%) are run more than $50,000$ times.

The two GUI programs have no bounds on running time and for our experiments were run for a fixed input. For `TTT` this consisted of two games of tic-tac-toe while for `JFig` it was the time taken to draw a simple figure. Table 1 summarizes the characteristics of these programs.

We measured the impact of embedding bits of a watermark on the running time of an application. `SciMark` performs no IO operations after it was started, hence it required no special timing harness.

	Execution Frequency	Average Running Time
TTT	4 lines never run	30s
(64 lines)	34 lines run 1 time	
	0 lines run $>$ 100 times	
	0 lines run $>$ 50,000 times	
JFig	18678 lines never run	600s
(22,779 lines)	0 lines run 1 time	
	0 lines run $>$ 100 time	
	0 lines run $>$ 50,000 times	
SciMark	224 lines never run	26s
(1,279 lines)	105 lines run 1 time	
	146 lines run $>$ 100 times	
	61 lines run $>$ 50,000 times	

Table 1. Characteristics of benchmark programs measured on a Pentium(R) 4 - M CPU 2.40GHz running GNU/Linux Java HotSpot(TM) Client VM (build 1.4.2-beta-b19, mixed mode)

For the two GUI applications, we used `xnee`, an X event recorder to record the X events sent to an application. After watermarking the application we replayed the X events and timed the entire procedure.

The original applications were timed 10 times and averaged to calculate initial speed. Following this they were watermarked and run ten times again to record how much they slowed down. The left-hand plot of Figure 7 shows the average slow down that results from embedding a watermark. In each of our ten timed tests the location at which the watermarks are embedded is selected randomly from the basic block trace which is produced during the trace step. It should be noted that although inserting a 48-bit watermark in SciMark results in a very significant slow down with a factor of ≈ 8, real world applications like TTT and JFig which have a GUI and wait for user interaction were observed to have very few time critical loops. For these applications, the resulting slow down was much less noticeable.

We also measured the size overhead of embedding our thread based watermark. The most significant contribution to the increased size of the application was the creation of closures. The right-hand plot of Figure 7 shows that thread based watermarks have a significant impact on the size of the small input application. Each embedding of a watermark bit caused the code size to increase by about 1.2 kilobytes.

5 Attacks

A software pirate attempting to steal a watermarked program may carry out several different attacks to prevent a watermark from remaining recognizable. To evaluate the resilience of our watermarking scheme we must know how resilient it is to these attacks.

5.1 Obfuscation Attacks

The simplest static attack that may remove a watermark is obfuscations that rename all variables and methods in a program, reorder blocks of code, or restructure data [2]. A more advanced obfuscation technique which attempts to

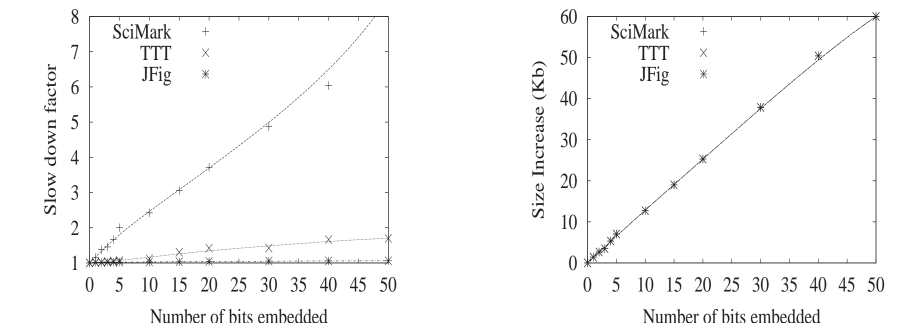

Fig. 7. Slow down of program execution, and Increase in the code size (in kilobytes), as a function of number of watermark bits embedded.

obscure the identity of variables or methods is "inlining" or "outlining". Inlining is a common compiler optimization technique that involves replacing a method call with an instance of the method's body. Similarly, outlining is where a set of instructions is replaced with a call to a method containing those instructions.

Our proposed technique is completely resilient to all of these attacks. This is because the recognition relies on the executed behavior of the program and not on its static structure - and this executed behaviour is preserved by these static attacks.

5.2 Decompilation/Recompilation Attack

An advanced 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. In particular, it was found that of the three decompilers tried Jad [10], Homebrew [19] and Dava [12], only Dava successfully handled unnested monitor calls correctly. It uses a library that emulated Java monitors in pure Java. Unfortunately, other errors prevented it from correctly decompiling our watermarked program. Even if an attacker is given a decompiler able to handle unnested monitors, we believe the proposed technique will survive a decompilation attack because the watermark is embedded in the order of execution of threads. This will be maintained by any semantic preserving decompile-recompile transformation. The decompilation attack can be made even more difficult by obfuscating the watermarked program using additional thread switches that are not used for watermark encoding, but which are necessary for program correctness. This can be easily done by introducing straight-line code where one of the two threads executes a subtly different and buggy version of each statement in the original code.

5.3 Additive attacks

The most potent attack against the proposed technique is one where the attacker succeeds in inserting random thread switches within a watermark piece. Note it is not enough for the attacker to simply insert new threads, or for him to insert new basic blocks such that an existing thread executes it. These types of errors are successfully corrected during our decoding process.

For an attacker to successfully destroy the watermark, they will need to cause at least two of the three threads involved in embedding a bit in a piece to switch. Such an attack need not be stealthy and thus can be achieved simply by inserting a `Thread.yield()` inside a basic block. However, the attacker cannot modify a large number of basic blocks in this way, because this may result in a large slowdown of the program. Alternately, unless an attacker can identify which thread switches are encoding watermarks, they will not know where to insert thread switches.

6 Conclusion

This paper has shown a novel technique for embedding watermarks using multiple threads, locks and thread contention. In particular, we showed how to encode the watermark in preparation for embedding, how to embed a single-bit and multi-bit watermark, and how to recognize the watermark.

Experimental results using an implementation to watermark Java bytecode indicate that the cost of watermarking is relatively small for real world applications. In addition, we looked at several classes of attacks against thread based watermarks, and we have proposed techniques for minimizing the effectiveness of these attacks.

References

1. Ross J. Anderson, editor. *Tamper Resistant Software: An Implementation*, Cambridge, U.K., May 1996. Springer-Verlag. Lecture Notes in Computer Science, Vol. 1174.
2. Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Symposium on Principles of Programming Languages*, pages 311–324, 1999.
3. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
4. Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 1998, POPL'98*, pages 184–196, San Diego, CA (USA), January 1998.
5. Christian Collberg and Gregg Townsend. Sandmark: Software watermarking for java, 2001. http://www.cs.arizona.edu/sandmark/.

6. Ingemar Cox and Jean-Paul Linnartz. Some general methods for tampering with watermarks. *IEEE Journal on Selected Areas in Communications*, 16(4):587–593, May 1998.
7. Robert L. Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent number 5,559,884, September 24 1996.
8. Derrick Grover. *The Protection of Computer Software: Its Technology and Applications*, chapter Program Identification. The British Computer Society Monographs in Informatics. Cambridge University Press, second edition, 1992.
9. Keith Holmes. Computer software protection. US Patent number 5,287,407 , Assignee: International Business Machine, February 1994.
10. Pavel Kouznetsov. Jad - the fast java decompiler, version 1158e for Linux on Intel platform. Avaliable `http://kpdus.tripod.com/jad.html`, 5 august, 2001.
11. Ivan Krsul. Authorship analysis: Identifying the author of a program. Technical Report CSD-TR-94-030, Computer Science Deparment, Purdue University, 1994.
12. Jerome Miecznikowski. Dava decompiler, part of SOOT, a Java optimization framework, version 7.1.09. Avaliable `http://www.sable.mcgill.ca/software/soot/` 17 December, 2003.
13. Akito Monden, Hajimu Iida, 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, January 1998.
14. Scott A. Moskowitz and Marc Cooperman. Method for stega-cipher protection of computer code. US Patent number 5,745,569, April 28 1998.
15. Jasvir Nagra, Clark Thomborson, and Christian Collberg. Software watermarking: Protective terminology. In *Proceedings of the ACSC 2002*, 2002.
16. 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.
17. Rafail Ostrovsky and Oded Goldreich. Comprehensive software system protection. US Patent number 5,123,045, June 16 1992.
18. John K. Ousterhout. Why threads are a bad idea (for most purposes). Invited Talk at 1996 Usenix Technical Conference, 1996. Slides avaliable at `http://www.sunlabs.com/~ouster/`.
19. Peter Ryland. Homebrew decompiler, version 0.2.4. Avaliable `http://www.pdr.cx/projects/hbd/`, 15 february, 2003. Avaliable `http://www.pdr.cx/projects/hbd/`.
20. Tomas Sander and Chrisitan F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, pages 44–60, 1998. Springer-Verlag, Lecture Notes in Computer Science 1419.
21. Narayanan Shivakumar and Héctor García-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of the First ACM International Conference on Digital Libraries DL'96*, Bethesda, MD (USA), 1996.
22. Julien P. Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. Robust object watermarking: Application to code. In Andreas Pfitzmann, editor, *Information Hiding (Proceedings of the Third International Workshop, IH'99), LNCS 1768*, Germany, 2000. Springer.
23. Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, Pittsburgh, PA, April 2001.