Dynamic Path-Based Software Watermarking *

C. Collberg E. Carter S. Debray A. Huntwork J. Kececioglu C. Linn M. Stepp
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{collberg,ecarter,debray,ash,kece,linnc,steppm}@cs.arizona.edu

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection—*Proprietary rights*

General Terms

Languages; Legal aspects, Security

Keywords

Software piracy, Software protection, Watermarking

ABSTRACT

Software watermarks — bitstrings encoding some sort of identifying information that are embedded into executable programs — are an important tool against software piracy. Most existing proposals for software watermarking have the shortcoming that they can be destroyed via fairly straightforward semantics-preserving code transformations. This paper introduces path-based watermarking, a new approach to software watermarking based on the dynamic branching behavior of programs. We show how error-correcting and tamper-proofing techniques can be used to make path-based watermarks resilient against a wide variety of attacks. Experimental results, using both Java bytecode and IA-32 native code, indicate that even relatively large watermarks can be embedded into programs at modest cost.

1. MOTIVATION

It is estimated that fully 40% of the software in use around the world is pirated, with retail value of over $13 billion in 2002 [1]. It is therefore crucially important to be able to protect software intellectual property rights. This means that the intellectual property owner of a piece of software should be able to demonstrate that ownership if called upon to do so; and in case of suspected piracy, it should be possible to trace a piece of software back to the person who originally obtained it prior to illegal distribution. Both of these goals can be met using *software watermarks*.

A software watermark is, in essence, an identifier that is embedded into a piece of software in order to encode some identifying information about it. The utility of a software watermark depends on its resilience against semantics-preserving code transformations: if it is easy to destroy a watermark via simple transformations, e.g., by renaming identifiers in the program, then it has relatively limited utility. In general, it is not possible to devise watermarks that are immune to all conceivable attacks; it is generally agreed that a sufficiently determined attacker will eventually be able to defeat any watermark. The goal, then, is to come up with watermarking techniques that are "expensive enough" to break—in time, effort, or resources—that for most attackers, breaking it isn't worth the trouble.

While a number of researchers have proposed schemes for software watermarking [2, 3, 4, 10, 13], most of these are not very difficult to break (see Section 6). For example, watermarks that rely on static code properties, such as basic block ordering [4] or register interference [10] can be defeated using straightforward binary optimizers [5, 11]; watermarks that use the topology of dynamically constructed data [2] can be foiled via code obfuscations that modify the pointer topology of such structures.

The primary contribution of this paper is to describe a new approach to software watermarking, *path-based watermarking*, which embeds the watermark in the runtime branching structure of the program. The idea is based on the intuition that the (forward) branches executed by a program are an essential aspect of its computation and part of what makes the program unique. However, an obvious apparent drawback with using the branch structure of a program to encode information is that, in principle, the branch structure of a program can be modified quite extensively without affecting program semantics, using well-known transformations such as basic block reordering, branch chaining (where the target of a branch instruction is itself a branch to some other location), loop unrolling, etc. In order for a path-based watermark to be resilient against such transformations, we must be able to either cause any such transformation to change the program semantics, or devise embedding techniques such that the watermark can survive such transformations. As we will see, path-based watermarking

*The work of C. Collberg, E. Carter, A. Huntwork, and M. Stepp was supported in part by the National Science Foundation under grant CCR-0073483 and the Air Force Research Lab under contract F33615-02-C-1146. The work of S. Debray and C. Linn was supported in part by the National Science Foundation under grants EIA-0080123 and CCR-0113633.

PLDI'04, June 9–11, 2004, Washington, DC, USA.

lends itself well to error-correction and tamper-proofing—significantly more so than most watermarking schemes proposed in the literature. Finally, since branches are ubiquitous in real programs, path-based watermarks are less likely to be susceptible to statistical attacks.

The remainder of this paper is organized as follows. Section 2 gives some background on software watermarking and describes the basic idea behind path-based watermarking. Section 3 discusses how path-based watermarking can be applied to Java bytecode, and how error correcting codes can be made to protect against attacks on such watermarks. Section 4 discusses the application of path-based watermarking to native code executables using a very different approach called branch functions, and describes how the resulting code can be tamper-proofed. Section 5 gives experimental evaluations of our approach. Section 6 discusses related work. Finally, Section 7 concludes.

2. PATH-BASED WATERMARKS

Software watermarking protocols are chiefly classified along four axes:

static/dynamic: In a static algorithm the watermark recognizer directly examines the code or data segments of the executable program. A recognizer for a dynamic algorithm will execute the watermarked program on a particular (secret) input sequence and then extract the watermark from the state of the program at this point.

watermark/fingerprint: In a pure watermarking algorithm the recognizer returns a value representing the likelihood that the mark is present. In a fingerprinting algorithm the recognizer returns the mark itself (a number) which can be different for every distributed copy of the program.

blind/informed: In a blind watermarking algorithm the recognizer is given the watermarked program and the watermark key as input. In an informed algorithm the recognizer also has access to the unwatermarked program and/or the watermark itself.

embedding technique: Typical software watermarking algorithms embed the marks by

1. reordering parts of the code where such reordering can be shown to be semantics preserving;
2. inserting new (non-functional or never executed) code that encodes a watermark number;
3. manipulating instruction frequencies.

There are several possible attack scenarios:

distortive attacks: The watermarked program can be subjected to a series of semantics-preserving transformations such as code optimization, binary translation, code compression, and code obfuscation, in the hopes that these will render the watermark unrecognizable.

additive attacks: New marks can be added to an already watermarked application, such that it is impossible to determine which is the original mark.

subtractive attacks: If the location of the watermark can be determined (for example by a statistical analysis of the code), it can be manually cropped out of the program.

```
int main(int argc) {
  if(argc == 3)
    printf("secret\n")
  return 0;
}
```

(a) Original program

```
int main(int argc) {
    int a = 1,b = 0;
    if (argc == 3) {
        if (b == 0) a = 0;
        if (b != 0) a = 0;
        if (b == 0) a = 0;
        if (b != 0) a = 0;
        if (b == 0) a = 0;
        if (b != 0) a = 0;
        if (b != 0) a = 0;
        if (b == 0) a = 0;
        printf("secret\n")
    }
    return 0;
}
```

(b) A trivial embedding of the bitstring 01010110.

Figure 1: A simple example of path-based watermark embedding

collusive attacks: Different fingerprinted programs can be compared in order to determine the location of the marks.

We say that an attack is successful if the watermark cannot be reliably recognized in an attacked application, and if the performance of the attacked program is such that it still has value to the attacker.

In this paper we will describe a new approach to software watermarking, *path-based watermarking*, where the basic idea is to embed the mark in the branching structure of the program. This has several interesting consequences. First, the forward branches that a program takes are an essential part of what makes the program unique. This makes branches inherently difficult to change or remove, making path-based watermarks resilient to many distortive attacks. Second, branches are inherently binary in nature (they are either taken or not taken), making it easy to embed a bitstring. Third, as we will see, path-based watermarking lends itself well to error-correction and tamper-proofing. Fourth, branches are ubiquitous in real programs, hopefully making path-based marks insusceptible to statistical attacks.

We will present two realizations of this basic idea, one for Java bytecode and one for x86 native executables. For native code we tamper-proof the watermark branches, and for typed architecture-neutral codes (for which code introspection is not possible) we use error-correcting codes to increase resilience to attack. Both our implementations are *dynamic blind fingerprinting* techniques. That is, (1) every distributed copy of a program encodes a unique integer, (2) only the watermarked program itself is used during recongition, and (3) during recognition the program is run with a special input sequence, a trace of the branches executed is extracted, and the trace sequence (taken/not taken) is examined for the watermark.

Figure 1 illustrates the basic path-based watermarking technique. Figure 1(a) shows the original program, and Figure 1(b) the code after bogus branches have been inserted to embed the watermark $w = 01010110$. The secret input to the program, in this case, is the number of arguments it is given: if it is invoked with three arguments ($\texttt{argc} = 3$), the watermark code is executed. This is obviously a very simple example embedding that is trivial to break: a simple attack would be, for example, to randomly change the tests so that the branch and fall-through cases are flipped. We can address the issue of resilience against semantics-preserving code transformations in a number of different ways, including more sophisticated embedding techniques and tamper-proofing of the code. These are discussed in more detail in the next two sections.

3. WATERMARKING JAVA BYTECODE

We have implemented execution path watermarking for Java bytecode. Because it is impossible to prevent semantics-preserving Java bytecode transformations, our implementation is designed to survive such attacks. In particular, using subroutines to implement the branch function scheme presented in Section 4 is not possible because return addresses are a primitive type in bytecode and cannot be used in any computation. Furthermore, the Java bytecode verifier requires that subroutines have a unique entry point and exit point. Our implementation is robust to certain types of semantics-preserving attacks because it is a dynamic watermark and because we duplicate the watermark and use error correction.

Our implementation consists of three phases. In the tracing phase, the dynamic behavior of the program is determined by tracing its execution path on a particular input sequence. In the embedding phase, a watermark number is embedded in the input code by modifying the sequence of branches taken and not taken on the secret input sequence. In the recognition phase, the program is traced again, and the branch sequence is checked for the watermark.

Figure 2 illustrates the watermarking process. In Ⓐ we split the watermark number into two pieces. In Ⓑ each piece is run through a block cipher. In Ⓒ the original program is executed with a special input sequence (the watermark key) in tracing mode. In Ⓓ watermark code is inserted into the original program in the form of added branches. When the watermarked program is executed with the special input sequence (Ⓔ), the resulting trace will contain the watermark pieces, in the form of branches that are taken and not taken in a particular pattern.

3.1 Tracing

In the tracing phase, we instrument the input program to write to a file the sequence of basic blocks it executes, together with the value of every local variable in the method executing and every static and instance field of the containing class. We then execute the instrumented program with the secret input sequence $I = I_0, I_1, \ldots$. The secret input sequence need not consist of characters read through standard input. This sequence may also consist of packets sent through the network or other sources of input. I may even be an empty sequence. The only restriction on the secret input sequence is that it be reproducible during recognition. The trace information collected from a program that computes the greatest common divisor of 25 and 10 is shown in Figure 2.

This trace aids code generation and is used to find appropriate insertion positions in the embedding phase. In the recognition phase, the branch sequence revealed by this trace is decoded into a bitstring that encodes the watermark. There are many possible ways of doing this. The binary representation of the address of the first instruction in every basic block could be written down. However, an attacker could change many of the bits in the resulting string simply by adding no-ops to the watermarked application. The bitstring can be formed by looking at every branch instruction of the form "if P goto Q else goto R" and writing down 0 if P is true and 1 otherwise. However, an attacker can toggle bits by negating the associated predicate and exchanging Q and R.

In order to survive these and other attacks that modify static program characteristics, we have chosen an algorithm that uses the dynamic behavior of branches to generate bits. For each conditional branch instruction i that occurs in the trace, we find its first occurrence, and find the block j that immediately follows that occurrence in the trace. Then we decode the trace into a string of bits by scanning the trace from beginning to end and writing down a 0 whenever a conditional branch is immediately followed by the same instruction by which it was first followed, and a 1 otherwise.

The resulting bitstring does not change if the input code is reordered, if branch senses are inverted, or if instructions other than conditional branches are inserted or deleted. Addition and removal of branches has only local effects on the resulting bitstring.

3.2 Embedding

The embedding phase adds branches to the input code so that the watermark number W can be extracted from a trace of the basic blocks executed on the input sequence I, as described in Section 3.3. The embedding phase consists of two steps. In the first step, the watermark value is split into a set of values to be embedded into the program. In the second step, this set of values is embedded into the program in the form of additional branch instructions and other code.

The process of turning W into a set of values to be embedded into the program consists of several steps, illustrated with an example in Figure 3:

1. Let $p_1, \ldots, p_r$ be pairwise relatively prime, where $W < \prod_{k=1}^{r} p_k$. W is split into up to $\frac{r(r-1)}{2}$ pieces, each piece being of the form $W \equiv x_k \bmod p_{i_k} p_{j_k}$, where $0 \le x_k < p_{i_k} p_{j_k}$. This is step Ⓐ.

2. Each statement $W \equiv x_k \bmod p_{i_k} p_{j_k}$ is turned into a single integer by an enumeration scheme. In our scheme,
$$w_k = x_k + \sum_{n=1}^{i_k-1} \sum_{m=n+1}^{r} p_n p_m + \sum_{m=1}^{j_k-1} p_{i_k} p_m.$$
This is step Ⓑ.

3. In step Ⓒ, each piece w_k is put through a block cipher. This step enables us to make randomness assumptions about any corrupted data when decoding.

We next insert code that causes the decoded trace to contain each piece of the watermark. Several manners of generating the inserted code and choosing its location exist. For example, code could be located so as to supplement bitstrings already occurring in the program.

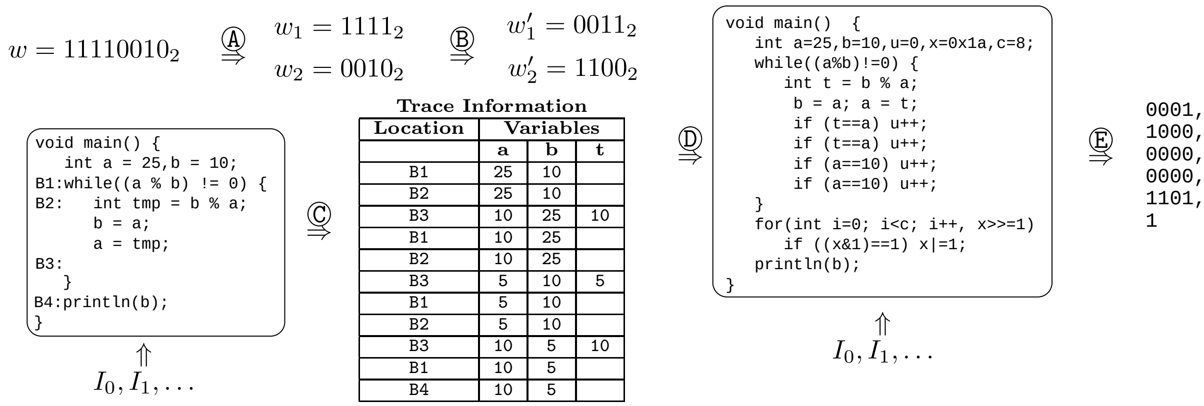


Figure 2: Embedding w into a program.

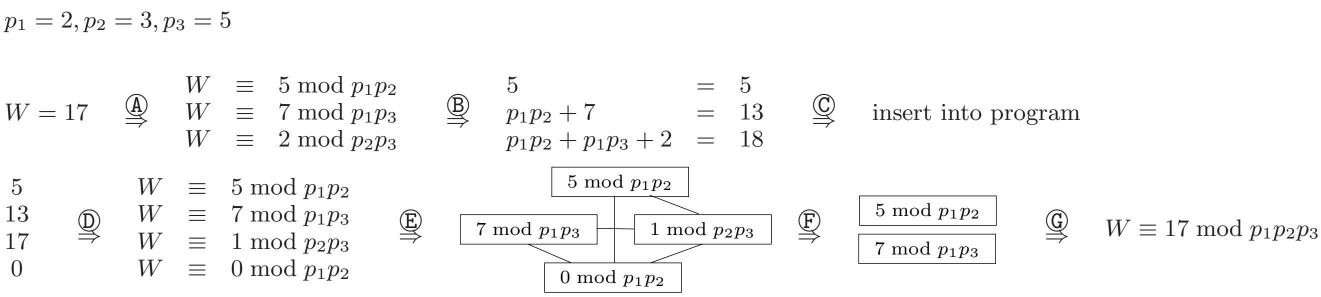


Figure 3: Splitting and re-combining the watermark value via the Generalized Chinese Remainder Theorem.

For simplicity, we insert code that generates an entire watermark piece. We insert code for each piece in a random location weighted inversely with respect to its frequency in the trace. Thus, code is less likely to be inserted in program hotspots than in infrequently executed code.

We generate two types of code to insert bits into the input code's bitstring. The first is based on loops, and the second is based on simple tests using variable values collected as part of the trace.

3.2.1 Loop Code Generation

Given a watermark piece w_k, this code generation technique generates a loop with a body that contains a conditional branch. The code generator generates a prologue to the loop and loop body code that causes the inner branch to succeed and fail in the order of the bits of w_k. For example, the following code is generated to insert the bits 0101 into the input code:

```
int bits = 0xa;
int counter = 5;
int j = 0;
for(int i = 0; i < counter; i++,bits >>= 1)
  if((bits & 1) == 1)
    j++;
if(P^F)
  live_var += j;
```

Because the least significant bit of $0xa$ is 0, the test inside the loop fails, thus indicating that future failures of this test will indicate a 0 and future successes of this test will indicate a 1. The remaining bits of $0xa$ in order from least to most sigificant are $0, 1, 0, 1$, which cause the test inside the loop to fail, then succeed, then fail, and finally succeed. This test result pattern generates the desired bits 0101. This code generator inserts approximately 60 bytes of instructions per 64 bits of input.

3.2.2 Condition Code Generation

This code generation technique inserts sequences of predicates and branches at locations that are executed multiple times on the secret input sequence. The first execution of the inserted code on the input sequence identifies which branch direction should generate which bit, and the remaining executions generate sequences of bits. Our embedder generates code so that at least one of the executions after the "priming" execution generates the desired bitstring.

Ideally, we would like the conditional branches we insert to look inconspicuous so that they are less obvious targets to an attacker. Thus, we base our predicates on existing program variables so that we do not have to insert new ones that may look suspicious to an attacker. This is the purpose of tracing variable values in addition to the sequence of basic blocks executed. By examining the values of variables in the program, we can determine predicates that will be true or false at a particular time in the execution of the program with the secret input sequence. In addition, for any set of predicates we determine to be true at a particular point in the program, we can logically AND them together to produce compound conditions that will also be true. Thus we can construct arbitrarily complex conditional statements using existing program variables, so an attacker cannot know that these statements are safe to remove while preserving correctness.

It is possible that some of the variable values may be environment dependent. For example, the value of a variable could represent the current time or the process ID. In this case, the generated code would not generate the desired branching pattern during recognition. We accept that possibility and assume that it will not happen frequently enough to destroy the watermark. The frequency of this kind of fault is likely to be less than the frequency of attacks against the watermark pieces.

As an example, a trace may indicate that a certain location is executed twice on the secret input sequence, that just before the first execution, $a = b$ and $c = d$, and that just before the second execution, $a = b$ and $c \neq d$. The following code would be generated to produce the string 1010:

```
int tmp = 0;
if(c == d)
  tmp++;
if(a == b)
  tmp++;
if(c == d)
  tmp++;
if(a == b)
  tmp++;
if(P^F)
  live_var += tmp;
```

The variable *live_var* shown in the code listings represents a variable that is live at the point of insertion. The expression P^F represents an opaquely false predicate chosen from SandMark's Opaque Predicate Library (OPL). The OPL contains aliasing, arithmetic, and threading opaque predicates like those described in [Christian's paper on OP's]. This opaque predicate makes it difficult to determine through static analysis that the generated loop has no semantic effect on the program.

3.3 Recognition

The recognition phase collects a trace of the input program's execution on secret input I and decodes it into a string of bits $b_0 b_1 \dots b_n$ as described in Section 3.1. The decoding algorithm is composed of three steps, illustrated with an example in figure 3. First, the bitstring $b_0 b_1 \dots b_n$ is split into a set of fixed-size blocks $B_0 = b_0 \dots b_{63}, B_1 = b_1 \dots b_{64}, \dots$. These blocks are decrypted using the same cryptosystem as in the embedding process. Finally, the resulting 64-bit blocks $B_0^d = d_k(B_0), B_1^d = d_k(B_1) \dots$ are passed to an algorithm that attempts to find a group of these blocks that agree on the watermark as follows.

In step Ⓓ, we invert our enumeration scheme to turn these 64-bit blocks into statements about W of the form $W \equiv x_k \bmod p_{i_k} p_{j_k}$.

First, in order to reduce the number of statements to consider, we hold a vote on the value of $W \bmod p_i$ for each i. If there is a clear winner (which we define as the first-place vote-getter being strictly greater than twice second-place), this winner is presumed to be the value of $W \bmod p_i$, and all statements contradicting this are removed from consideration.

Among the various pairs of statements, some are inconsistent, some are consistent because the x's agree $\bmod p_i$ for some i, and some are consistent by the Chinese Remainder Theorem since the 4 p_i's referred to are all distinct and the p_i's are pairwise relatively prime.

Let $V = \{v_0, v_1, \dots, v_m\}$ be the set of statements on W we are given. In step Ⓔ, we construct two graphs, G and H, on V. (The figure only shows G.) Two vertices are adjacent in G iff they are inconsistent. Two vertices are adjacent in H iff they are consistent because the x's agree $\bmod p_i$ for some i, not if they are consistent by the Chinese Remainder Theorem. For step Ⓕ, we initialize $U := \emptyset$ and repeat the following steps until G is a coclique:

1. Let v be some vertex in the set $V - U$ of maximum degree in H. This vertex is presumed to be a true statement about W.

2. Let S be the set of v's neighbors in G. Set $G := G - S$, $H := H - S$, and $U := U \cup \{v\}$.

Once G is a coclique, we have a set of statements about W that are consistent and can be combined by the Generalized Chinese Remainder Theorem [8] in step Ⓖ.

4. WATERMARKING NATIVE EXECUTABLES

Native code executables offer significantly greater flexibility, compared to Java bytecode, in terms of the transformations that can be applied during watermarking. This makes it possible to use techniques that cannot be used in the case of bytecode. Here we discuss one such technique for implementing path-based watermarking, using a device called *branch functions*.

4.1 Branch Functions: An Overview

A branch function is a function that is called in the normal manner, but which manipulates its return address such that, when it returns, control may be transferred to an address different from the original call site [9]. Consider a program containing a particular set of unconditional jumps of interest, at locations $a_1, \dots, a_n$, with targets $b_1, \dots, b_n$ respectively, i.e., the instruction at location a_i is

a_i : `jmp` b_i $\qquad 1 \leq i \leq n$

With branch functions, we replace each of these jumps by a call to the branch function f, resulting in code of the form:

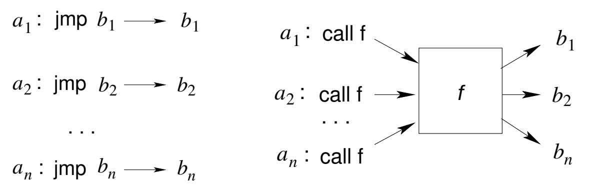
a_i : `call` f $\qquad 1 \leq i \leq n$

The function f uses the return address to figure out the location $a_i (1 \leq i \leq n)$ it was called from, then uses this information to change its return address to the value b_i. When it subsequently executes a `ret` instruction, therefore, control is transferred to the original target b_i. The situation is illustrated in Figure 4.

The implementation of branch functions can be illustrated by first considering a very simple-minded variation of the idea above, where the call at location a_i passes the offset to its target address as an argument. The branch function then simply adds its argument to the return address, then returns. The corresponding code, on the Intel IA-32 architecture, has the form:

```
xchg %eax, 0(%esp)    #I1
add  %eax, 4(%esp)    #I2
pop  %eax             #I3
ret                   #I4
```

Instruction I1 exchanges the contents of register `%eax` with the word at the top of the stack, effectively saving the contents of `%eax` and at the same time loading the return address into `%eax`. Instruction I2 then has the effect of adding



(a) Original code (b) Code using a branch function

Figure 4: Branch functions

the displacement to the target (passed as an argument on the stack) to the return address; the sum—which is in fact the target address b_i—is now written to the location `4(%esp)`. I3 restores the previously saved value of `%eax`, leaving the address of the target on top of the stack. I4 then has the effect of branching to the address computed by the function.

The straightforward implementation described above has two shortcomings. The first is that it is not difficult to detect a function that modifies its own return address. On architectures such as the Intel IA-32, the return address is passed at some fixed offset from the stack pointer; on RISC architectures, the standard calling convention passes the return address in some fixed register. In either case, an observant attacker can detect when the location containing the return address happens to be the destination of an arithmetic (or *move*) instruction. The second shortcoming is that this simple scheme uses just a single straightforward arithmetic operation for the return address modification, and so is not as robust against reverse engineering as we would like.

We can address the first problem by using "helper" functions with the branch function. The idea is as follows. The branch function f does not itself do any tampering with its return address: instead, it calls a helper function f_1, which might itself call another helper function f_2, etc. The helper function calls cause the original return address to be saved on the stack regardless of whether the calling convention passes the return address in a register or on the stack. Moreover, because the chain of helper functions $f \to f_1 \to \cdots \to f_m$ is fixed in a given implementation of branch functions, we know (based on knowledge of the stack frame sizes for the helper functions $f_1, \ldots, f_m$—note that these stack frame sizes can be chosen randomly by the implementation) how deep in the stack the original return address is located. The last helper function then "reaches into" the stack to modify the original return address.

We address the second problem using perfect hashing [7]. Given the control flow mapping $\varphi = \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ we want the branch function to implement, we create a perfect hash function $h_\varphi : \{a_1, \ldots, a_n\} \mapsto \{1, \ldots, n\}$. We then construct a table T in the data section of the binary, that contains the exclusive or of each (a_i, b_i) pair,[1] as follows:

$$T[h_\varphi(a_i)] \leftarrow a_i \oplus b_i.$$

Upon invocation, the branch function proceeds as follows. It saves the appropriate registers; applies the perfect hash function h_φ to its return address a to compute a hash value $h_\varphi(a)$; uses the table T to obtain the value $T[h_\varphi(a)]$; `xors` this value into the return address, similarly to the scheme described earlier; and finally, restores the saved registers and returns. Figure 5 shows an example of branch function code, taken from the SPECINT-2000 benchmark program *parser* for a 512-bit watermark.

[1]This is done so that the data section of the binary does not contain a sequence of text section addresses, since such a sequence of entries could be conspicuous to an attacker.

```
  pushf                  # save flags
  push %edx              # register save
  push %ecx              # register save
  push %eax              # register save
  mov 0x10(%esp,1),%edx
  mov %edx,%eax
   # begin perfect hash computation
  shl $0xc,%eax
  and $0x7ff,%edx
  shr $0x15,%eax
  movzwl 0x80d2bb0(%edx,%edx,1),%ecx
  xor %ecx,%eax
   # begin return address fix
  imul $0xc,%eax,%eax
  mov %eax,%edx
  mov 0x80c3c04(%eax),%eax
  xor %eax,0x10(%esp,1)
   # begin tamper-proofing
  lea 0x80c3c08(%edx),%eax
  cmpl $0x0,(%eax)
  je cleanup
  mov 0x4(%eax),%edx
  xor %edx,(%eax)
  movl $0x0,0x4(%eax)
cleanup:
  pop %eax               # register restore
  pop %ecx               # restore restore
  pop %edx               # restore restore
  popf                   # restore flags
  ret
```

Figure 5: An example of branch function code

4.2 Using Branch Functions for Software Watermarking

In order to use branch functions for software watermarking, we have to specify how they should be used to encode the bits in the watermark, how a watermark is to be embedded within an executable, and how it can be extracted. This section discusses these issues in more detail.

4.2.1 Bit Encoding

As discussed above, a branch function implements a control flow mapping $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$. We can choose any subset—not necessarily proper—of the pairs in this map to encode the watermark: i.e., the branch function implementing the watermark can also be used to obfuscate other control transfers, elsewhere in the program, that have nothing to do with the watermark itself [9]. For simplicity of exposition, however, we will assume, in the discussion that follows, that all of the pairs in the branch function are used for encoding the watermark.

Each pair of addresses $a_i \mapsto b_i$ in the branch function map encodes a single bit of the watermark. In principle, we can use any property of these pairs that we want: for example, we could, if we wished, use the parity of a_i and b_i, using the predicate '$parity(a_i) = parity(b_i)$.' Our implementation uses a forward jump (i.e., $a_i < b_i$) to encode a '1' and a backward jump (i.e., $a_i > b_i$) to encode a '0.'

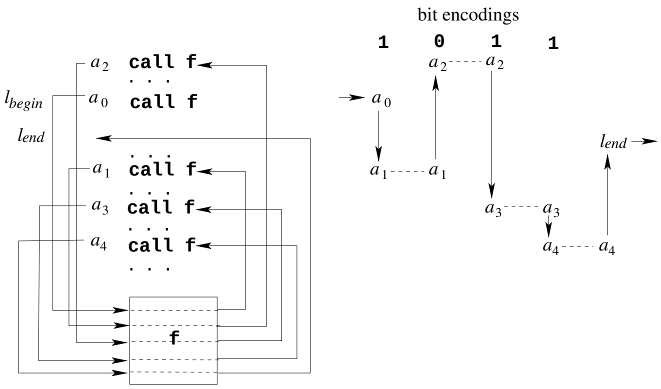


Figure 6: An example of branch-function-based watermarking (watermark = `1011`)

4.2.2 Watermark Embedding

To embed a k-bit watermark $w = w_0 w_1 \cdots w_{k-1}$ into an executable, we start with an unconditional control flow edge $\ell_{begin} \to \ell_{end}$; we will split this edge and insert the watermark code between ℓ_{begin} and ℓ_{end}. We first construct a list of $k+1$ branch function calls $(a_0, a_1, \ldots, a_k)$ such that the following hold:

1. for each a_i, $0 < i \leq k$, the instruction immediately before a_i is an unconditional jump, i.e., execution cannot fall through to a_i; and

2. the addresses of adjacent pairs of instructions (a_i, a_{i+1}), $0 \leq i < k$, encode bit w_i of the watermark:

$$\begin{array}{ll} addr(a_i) < addr(a_{i+1}) & \text{if } w_i = 1 \\ addr(a_i) > addr(a_{i+1}) & \text{if } w_i = 0. \end{array}$$

To construct the list, a_0 is inserted at address ℓ_{begin}. We then iteratively construct a_{i+1} from the last instruction a_i added to the list: we use the value of w_i, the i^{th} bit of the watermark w, to scan either forward (if $w_i = 1$, i.e., a_i need to jump forward) or backward (if $w_i = 0$, i.e., we need to jump backward), until we find a location that satisfies the first condition above, and insert a call instruction at that location. This is repeated until all of the instructions $a_0, \ldots, a_k$ have been constructed. The last branch function call, a_k, has ℓ_{end} as its target.

Once these branch function calls have been inserted into the instruction stream, the address of each such instruction is determined. Let $\widehat{a}_i$ denote the address of the instruction a_i, then the control transfer mapping for the branch function is $\{\widehat{a}_0 \mapsto \widehat{a}_1, \widehat{a}_1 \mapsto \widehat{a}_2, \cdots, \widehat{a}_{k-1} \mapsto \widehat{a}_k, \widehat{a}_k \mapsto \widehat{\ell}_{end}\}$.

Figure 6 illustrates an example of a branch-function-based embedding of the bitstring `1011` into a program. Starting at a_0, the first bit is 1, and is encoded by the forward branch $a_0 \to a_1$, which is realized via a call to the branch function `f()`; the next bit, 0, is encoded by the backward branch $a_1 \to a_2$; the third bit is a 1, and is encoded by the forward branch $a_2 \to a_3$; and the last bit, which is again a 1, is encoded by another forward branch, $a_3 \to a_4$. Finally, control returns from a_4 to the end point ℓ_{end} of the watermark.

4.2.3 Watermark Extraction

To extract a watermark, we start with the pair of addresses $(\ell_{begin}, \ell_{end})$ bracketing the watermark (currently, these are supplied manually; however, we expect to augment the implementation in the near future to use a framing scheme that would allow these addresses to be identified automatically). We use a *tracer* tool that uses hardware single-stepping to obtain a dynamic trace of the instructions executed between the time control reaches ℓ_{begin} and when it subsequently reaches ℓ_{end}. This trace is then analyzed to identify the branch function f_w, by observing functions that do not return to the instruction following the call instruction. Once the branch function has been identified, we obtain, from the trace, the list of locations from which f_w is called, and for each such location a_i the corresponding location b_i to which control returns from that call. By comparing the values a_i and b_i, we can determine whether it corresponds to a forward or backward jump, and thereby extract the corresponding bit of the watermark. This is repeated until all instructions in the trace have been processed; this corresponds to having execution return to ℓ_{end}.

4.3 Tamper-proofing Branch Functions

An important property of a software watermark is its robustness under semantics-preserving code transformations. Since a branch function synthesizes a mapping between pairs of absolute addresses, any transformation that causes code addresses to change, but which does not at the same time update the mapping implemented by the branch function, will cause the resulting program to break. Moreover, the perfect hash functions used to compute these mappings tend to be quite cryptic and difficult to reverse engineer (e.g., see Figure 5). For this reason, we believe that branch-function-based watermarks are resilient against code transformations that cause addresses within the text section to change; in particular, this includes additive and distortive attacks.

To guard against subtractive attacks, our basic idea is to have the branch function carry out a computation that is essential for the subsequent execution of the program. Recall that the branch function is entered starting at a location ℓ_{begin}. We begin by taking an unconditional branch at a location $\ell_\star$ such that ℓ_{begin} dominates $\ell_\star$. We then transform the branch instruction at $\ell_\star$ to an indirect branch through a memory location M, such that M contains the correct target address if and only if the branch function has been executed. For this, M is initialized to some random text section address, and code is added within the branch function to update the contents of M to the correct target address. In general, this update can occur incrementally, such that each time the branch function executes, some set of bits of the target address are computed. This is done for multiple jump instructions: in our current implementation, when embedding a k-bit watermark we attempt to find up to k candidate branches that can be tamper-proofed in this manner; each branch function call updates a different such candidate (a branch is considered to be a candidate if it occurs in an infrequently executed portion of the code and is not part of a loop; the last requirement is to avoid excessive performance degradation on inputs that may cause different execution characteristics than the training inputs). With this, if the branch function is identified and "snipped out" of the execution by an attacker, the control flow behavior of the program will no longer be correct.

5. EXPERIMENTAL RESULTS

5.1 Java Bytecode

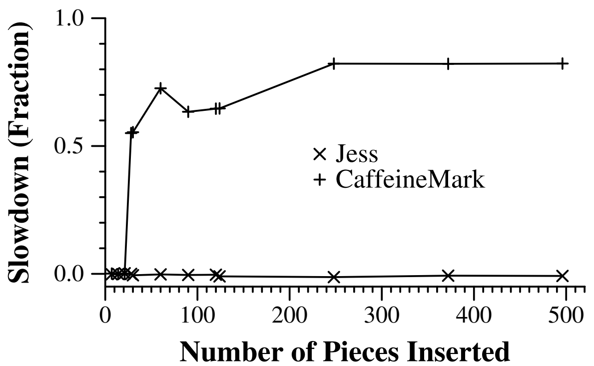


Figure 7: CaffeineMark and Jess slowdown resulting from the insertion of different numbers of watermark pieces

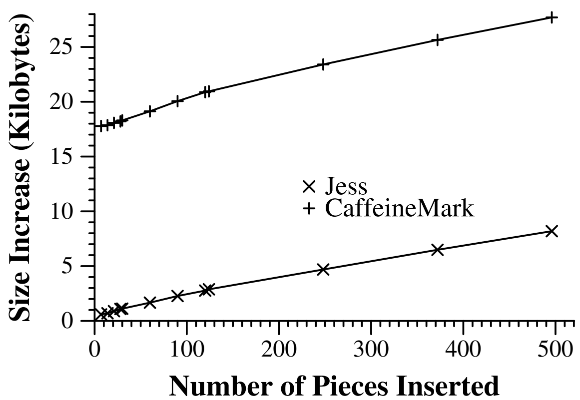


Figure 8: CaffeineMark and Jess size increases resulting from the insertion of different numbers of watermark pieces

CaffeineMark is a collection of micro-benchmarks, so a high percentage of the instructions are executed frequently. Figure 7 shows that embedding the path-based watermark in such an application can have dramatic performance consequences. It is likely that performance does not suffer when few watermark pieces are inserted because the weighted random location choice described in Section 3.2 selects infrequently executed locations as insertion points. As more pieces are inserted, the probability that some frequently executed location will be chosen increases; when this eventually occurs, there is a dramatic performance degradation. As many more pieces are inserted, more are inserted in "hot" locations, resulting in further performance degradation.

In contrast, Jess contains more code (300KB as opposed to 9KB) and a lower percentage of frequently executed code. It appears that our random insertion position algorithm successfully avoided the frequently executed portions of Jess, and therefore caused an insignificant slowdown.

Figure 8 shows that the path-based watermark has little size impact on the input application. Even for a very large watermark, the equivalent of at least 200 and at most 2000 lines of code are added to the application.

5.1.1 Attacks

The major forms of attack against software watermarking systems are additive, subtractive, distortive, and collusive attacks. The SandMark library implements many such attacks, most of which were unsuccessful in destroying the embedded watermark. One code transformation that was successful was encrypting classfiles.

In this attack, every classfile in an application is replaced with an encrypted version of itself. The startup code for the application is then replaced by a new program that decodes and runs the encrypted classes. After performing this obfuscation on a watermarked jarfile, the encrypted classfiles will contain the branch pattern needed to reveal the watermark at runtime. However, during recognition the jarfile must be instrumented to produce a trace of the execution of the program. This cannot be done in the normal fashion because the encrypted classfiles are not recognized as classfiles, and hence are not instrumented by the recognizer. Therefore, when the program is executed in order to produce a trace, all flow of execution through the methods of the encrypted classfiles produces no trace at all. Since the watermark is embedded in the methods of the encrypted classfiles, this means that the watermark branch pattern will not appear in the trace.

The effectiveness of this attack is due to the particular method we use to gather a trace of execution at runtime. An alternative method would be to run the program through a bytecode interpreter to gather the trace. In this case, instrumenting the code would not be necessary, and thus the fact that the classfiles are encrypted would not hinder the tracing process.

Our implementation provides no protection against adding multiple watermarks to a program in an effort to obscure ownership, and is thus susceptible to additive attacks. Our implementation provides no protection against collusive attacks. However, collusive attacks can be prevented by obfuscating the program before it is watermarked, and thus producing a highly diverse program population. Any attempt to find the watermark code through comparison of multiple watermarked copies of the program would be thwarted by this defense because the differences between any two copies of the program would contain much more than just the watermark code.

One trivially implemented distortive attack against the path-based watermark is random branch insertion. If an attacker inserts a branch instruction at a random place in the program, he may cause random changes in the decoded bitstring. If he inserts many random branches into the program, he is likely to cause widespread random changes in the decoded bitstring. Because of the error correcting qualities of our watermark encoding scheme, our implementation can withstand a level of random branch insertion that varies with the number of watermark pieces embedded in the program and with the size of the watermark, as shown in Figure 9.

The performance penalty associated with this attack is likely to vary widely based on the code inserted to generate each branch. We have measured the performance penalty of this code (where x is an integer variable):

```
if(x * (x - 1) % 2 != 0)
   x++;
```

The resulting slowdown is shown in Figure 10. Thus, branch insertion is a viable attack against the path-based watermark, but with a cost of doubling the runtime of the watermarked program.

Subtractive attacks against this watermark are likely to resemble this distortive attack in that such attacks are likely to remove the watermark piece by piece. While there will be no accompanying slowdown, the number of watermark

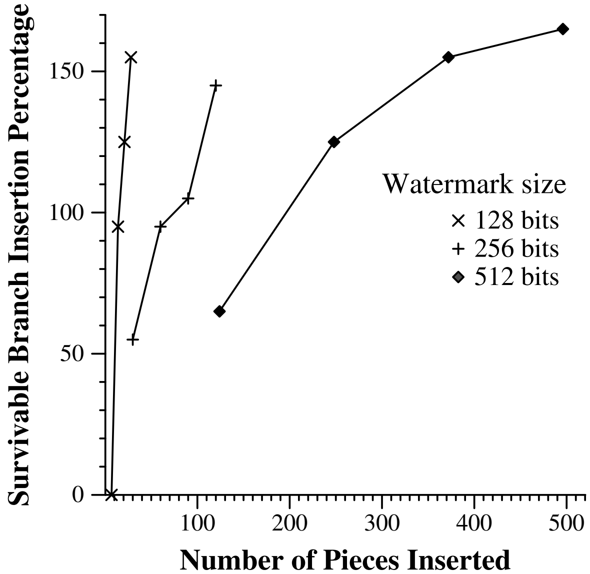


Figure 9: The path-based watermark can survive the addition of a percentage of branches that varies with the number of watermark pieces inserted and the size of the watermark.

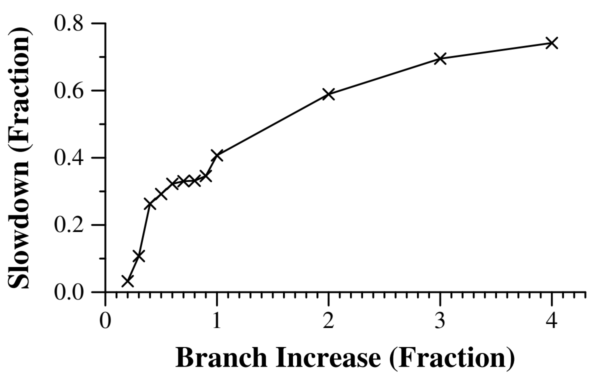


Figure 10: Adding branches to code causes a slowdown that varies with the number of branches added

pieces that can be removed without destroying the watermark will vary with the number of watermark pieces inserted and the watermark size in exactly the same way as in the above distortive attack.

5.2 Native Code

We evaluated the branch-function-based watermarking scheme, described in Section 4.2, using an implementation built on top of PLTO, a binary rewriting system developed for Intel IA-32 executables [6]. The system reads in statically linked executables, disassembles the input binary, and constructs a control flow graph, which can then either be instrumented to obtain execution profiles, or modified to have a given watermark embedded into it. We used ten programs in the SPECint-2000 benchmark suite for our experiments. Two benchmarks, *eon* and *perl* were omitted from our tests due to problems building the harness. Our experiments were run on an otherwise unloaded 2.4 GHz Pentium IV system with 1 GB of main memory running RedHat Linux 9.0. The programs were compiled with *gcc* version *3.2.2* at optimization level `-O3`. The programs were profiled using the SPEC training inputs and these profiles were used to identify any hot spots during our transformations. The final performance of the transformed programs were then evaluated using the SPEC reference inputs. Each execution time reported was derived by running five trials, discarding the highest and lowest run times so obtained, and computing the average of the remaining three times.

5.2.1 Cost

We evaluated the space and time cost of path-based watermarking using watermarks of three sizes: 128 bits, 256 bits, and 512 bits.

Figure 11 shows the relative increase in total size (text+data sections) incurred due to watermarking. All in all, the increases are fairly modest, ranging from about 5% for *crafty* to about 16% for *mcf*. The rate of growth in size is also fairly small. The mean increase in size ranges from 10.8%, for 128-bit watermarks, to 11.4% for 512-bit watermarks.

The runtime slowdowns experienced as a result of watermarking are shown in Figure 12. For most of the programs tested, the slowdown is quite small (less than 2%), and several of the programs actually speed up by about 2–3%, presumably due to cache effects. The mean slowdowns range from -0.65%, for 128-bit watermarks, to 0.85% for 512-bit watermarks.

5.2.2 Resilience

To evaluate the resilience of our watermarks against attacks, we subjected the watermarked programs to a number of code transformations of the sort that might be encountered in a standard binary manipulation tool. We tried the following transformations:

1. *No-op insertion.* This is intended to simulate a distortive attack where the attacker tries to inject additional code into the program, e.g., using a code obfuscator.

   As discussed in Section 4.3, the use of branch functions gives us a "lock-down" on a range of program addresses, such that a change to any of these addresses will cause the program to malfunction. The effect of such insertions is to change text addresses. Every one of our test programs breaks when even a single no-op is added to a watermarked binary.

2. *Branch sense inversion.* This involves inverting the sense of conditional branches and rearranging basic blocks accordingly to maintain program semantics, so that the roles of the "branch-taken" and "branch-not-taken" targets get reversed. This is intended to simulate a distortive attack of the sort that might occur if an attacker applies code optimization or binary rewriting techniques to a watermarked binary. For the same reason as for no-op insertion, every one of our test programs breaks when branch senses are inverted.

3. *Double watermarking.* This involves taking a watermarked program and running it through the watermarker again. This simulates an additive attack where the attacker hopes to overwrite or obscure part or all of the original watermark by a second round of watermarking. As for the previous two attacks, this causes text addresses to change, and causes each of our test programs to break.

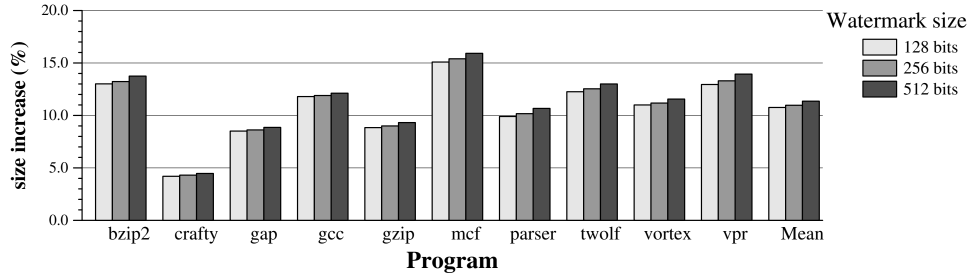


Figure 11: Space cost of watermarking native code

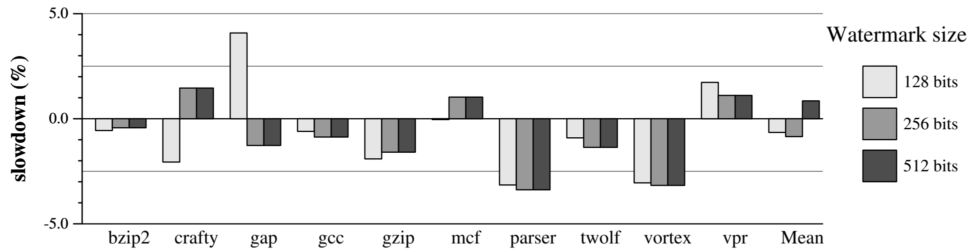


Figure 12: Time cost of watermarking native code

4. *Bypassing the branch function.* This involves overwriting some number of calls to the branch function with a *jump* instruction of exactly the same size (in bytes), so that there is no net change to any addresses; the target of this new jump instruction is the actual address that the branch function would transfer control to for that particular call. This has the effect of realizing the control transfer that the branch function would realize, but bypassing the actual branch function code. It simulates a subtractive attack.

   As discussed in Section 4.3, calls to the branch function also have the effect of updating the contents of memory locations that are used for indirect jumps. When the branch function is bypassed, therefore, some such locations are not properly updated, and therefore contain incorrect addresses. This causes execution to break.

5. *Rerouting Branch Function Entries.* This involves replacing a call to the branch function with a call to a different location [2] which then transfers control to the branch function. Consider the following transformation from the original branch function call in (a) to the sequence in (b), where *bf* is the branch function, and Y is the address of the end of the text.

```
(a)              (b)
   X: call bf       X: call bf
      ...              ...
                    Y: jmp bf
```

   This particular transformation allows the program to execute properly, since the branch function in (b) still sees X [3] as the hash input just as it would have in (a). A tracer which relies on looking at the address transferring control to the branch function to determine each $\widehat{a}_i$, such as our simple tracer, is disabled in (b) since it would deduce the $\widehat{a}_i$ to be Y instead of X. This attack can be obviated, however, simply by using a slightly more intelligent tracer that is constructed as follows.

   As explained above, one of the reasons that this particular attack works is that the return address, i.e., the hash input, remains unchanged and therefore maintains the integrity of the tamper-proofing. From this fact we can see that each time the branch function executes, it must still be using the address of the instruction just after the $\widehat{a}_i$ as it's hash input. By constructing a tracer that tracks the value of the hash input to the branch function each time it executes (as opposed to inspecting the address of the invoking instruction) , the original mapping $\{\widehat{a}_0 \mapsto \widehat{a}_1, \widehat{a}_1 \mapsto \widehat{a}_2, \cdots, \widehat{a}_{k-1} \mapsto \widehat{a}_k, \widehat{a}_k \mapsto \widehat{\ell}_{end}\}$. can be easily retrieved and the watermark can successfully be reconstructed.

6. RELATED WORK

Various other software watermarking schemes have been proposed. These schemes fall into two categories: static and dynamic. Static watermarking schemes are those that do not require the watermarked program or any part of it to be run or interpreted in order to embed or extract the watermark. Dynamic watermarking schemes, such as the one presented in this paper, are those that do require the watermarked program or some part of it to be executed.

Previously proposed static watermarking schemes include one by Qu and Potkonjak [10] embedding the watermark in register interference graphs, one by Venkatesan et al. [13] embedding the watermark in the control flow structure of

[2]This may require the header of the file to also be modified, but does not necessarily require any relocation to take place within the binary. If relocation was needed the attack becomes much more difficult because of known static analysis challenges with respect to native executables.

[3]The value it will see is actually $X + 5$ to account for the length of the call instruction

a designated piece of the program, one by Davidson and Myhrvold [4] embedding the watermark by reordering basic blocks, and one by Stern et al. [12] that embeds the watermark in the relative frequencies of instructions throughout the entire program using a spread spectrum technique. All of these schemes are vulnerable to relatively simple automated transformative attacks that do not have too great an impact on program performance.

Dynamic software watermarking was first proposed by Collberg and Thomborson [2]. Their scheme embeds the watermark in the topology of some data structure that is built on the heap at runtime given some secret input sequence to the program. This scheme is vulnerable to attacks that modify the pointer topology of the program's fundamental data types, which drastically changes the topology of any data structures built at runtime.

Another dynamic software watermarking scheme proposed by Cousot and Cousot [3] embeds the watermark in values assigned to designated integer local variables during program execution. These values can be determined by analyzing the program under the framework of abstract interpretation, enabling the watermark to be detected even if only part of the watermarked program is present. This scheme can be attacked by obfuscating the program such that the local variables representing the watermark cannot be located or such that the abstract interpreter becomes confused and cannot say what values are assigned to those local variables.

7. CONCLUSIONS

Software watermarking is an important tool for combating software piracy. It is important that software watermarks be resilient against semantics-preserving code transformations. Unfortunately, most existing proposals for software watermarking turn out to be vulnerable to fairly straightforward code transformations. This paper introduces a new approach to watermarking, called path-based watermarking, that embeds the watermark in the dynamic branch structure of the program, and shows how error-correcting and tamperproofing techniques can be used to make path-based watermarks resilient against a wide variety of attacks.

Experimental results, using both Java bytecode and IA-32 native code, indicate that the cost of watermarking is relatively modest, even for relatively large watermarks (ranging from 128 to 512 bits). For Java bytecode, we see that if the number of pieces that the watermark is broken into is kept small, the runtime overhead of watermarking is essentially negligible. As the number of pieces is increased, thereby making it increasingly difficult for an attacker to destroy the watermark, there is a concomitant increase in the runtime overhead. The space cost of watermarking Java bytecode is independent of the size of the application being watermarked, and is quite small: it varies roughly linearly with the size of the watermark, and required about 8 Kbytes for a 512-bit watermark. Native code watermarking on an Intel IA-32 platform incurred mean size increases of about 12–13% and mean runtime slowdowns of about 3.5%.

8. REFERENCES

[1] Business Software Alliance. Eighth annual BSA global software piracy study: Trends in software piracy 1994–2002, June 2003. `http://global.bsa.org/globalstudy/2003_GSPS.pdf`.

[2] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *In Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Jan. 1999)*, 1999.

[3] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *POPL*, 2004. To appear.

[4] R. L. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, September 1996. Assignee: Microsoft Corporation.

[5] S.K. Debray, R. Muth, S. Watterson, and K. De Bosschere. ALTO: A link-time optimizer for the Compaq Alpha. *Software — Practice and Experience*, 31:67–101, January 2001.

[6] S.K. Debray, B. Schwarz, G.R. Andrews, and M. Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*, September 2001.

[7] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[8] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, third edition, 1997.

[9] C.M. Linn and S.K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, pages 290–299, October 2003.

[10] 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, November 1998. `http://www.cs.ucla.edu/~gangqu/publication/gc.ps.gz`.

[11] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[12] J.P. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999.

[13] R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, Pittsburgh, PA, April 2001.