Proceedings of ACSAC'00, 16th Annual Computer Security Applications Conference, pages 308-316, 2000.

Experience with Software Watermarking

Jens Palsberg Sowmya Krishnaswamy Minseok Kwon Di Ma Qiuyun Shao Yi Zhang
CERIAS and Department of Computer Science
Purdue University
West Lafayette, IN 47907
{palsberg,madi}@cs.purdue.edu

Abstract

There are at least four U.S. patents on software watermarking, and an idea for further advancing the state of the art was presented in 1999 by Collberg and Thomborsen. The new idea is to embed a watermark in dynamic data structures, thereby protecting against many program-transformation attacks. Until now there have been no reports on practical experience with this technique.

We have implemented and experimented with a watermarking system for Java based on the ideas of Collberg and Thomborsen. Our experiments show that watermarking can be done efficiently with moderate increases in code size, execution times, and heap-space usage, while making the watermarked code resilient to a variety of program-transformation attacks. For a particular representation of watermarks, the time to retrieve a watermark is on the order of one minute per megabyte of heap space. Our implementation is not designed to resists all possible attacks; to do that it should be combined with other protection techniques such as obfuscation and tamperproofing.

1 Introduction

1.1 The Need to Prove Software Ownership

Suppose Alice has built some software and now wants to sell it for profit. She copyrights the software, but Bob still manages to make a pirate copy. Bob may be interested in the software for several reasons, including 1) private use, 2) industrial espionage, and 3) further selling for his own profit. In the first case, Alice's profits may suffer; in the second case, some of her algorithmic techniques may be found out; and in the third case, she may risk that a competitor sells her own software, perhaps somewhat modified. This leaves the question:

Question: How does Alice protect her copyright?

Answer: She must be able to prove ownership, possibly in a court of law, of a given copy of the software.

If it is known that Alice is capable of proving ownership of her software, then this capability may help deter theft.

1.2 Approaches to Anti-piracy

Various approaches to anti-piracy have been or could be tried, including:

- Keep a certified list of customers. If somebody not on the list has the software, then it must be a pirate copy.
- Link the software to the hardware of a specific machine. This makes it pointless to copy the software to some other machine. A further idea is to configure the software such that when a user is logged on to the Internet, a message with the serial number of the computer is secretly sent to the software company. Remark: in 1998 it was found out that on some Intel computers, the serial number could be read by software.
- Link the software to a movable piece of hardware that cannot easily be copied. This restricts the usage to a user who possesses the critical piece of hardware. Such a device is sometimes called a "hardware dongle"; and the technique has been called "software dog" by Tsinghua Archives (Beijing, P.R. China) [1] which since 1994 has used it for the THDA-MIS multimedia database system.
- Software watermarking: embed a secret into the software which can be retrieved on demand.

The first three of these techniques seem too inflexible for a setting where software can be downloaded from webpages, and where mobile code can roam on the Internet. Moreover, some software is distributed in a form close to source code,

for example, as Java bytecodes, and disassemblers and decompilers are getting faster and better. Such considerations have helped increase interest in software ownership protection and detection. Watermarking is a method that does not aim to stop piracy copying, but to prove ownership of the software and possibly even the data structures and algorithms used in the software. Up to now, there are at least four U.S. patents on software watermarking [10, 8, 12, 15].

1.3 Attacks on Software Watermarks

Watermarked software may be subject to attacks that have the objective of locating, distorting, or removing the watermark. The quality of a watermarking system is correlated with the degree to which the watermarked software is resistant to attacks. Until now, most approaches to software watermarking have concentrated on resisting attacks by *semantics-preserving program transformations*, including compilation, optimization, obfuscation, decompilation, and dead-code removal. Such transformations do not change the behavior of a program, but they do change the form of the program. As such, they may easily remove or distort watermarks that are embedded in the structure of the program, for example, in comments, in string constants, in the order of instructions, etc:

- a comment:

```
/* My software, version 1.0
*/
```

- a data string:

```
printf("My software, version
1.0");
```

- a particular ordering of certain instructions; for example, a particular ordering of the branches of an n-branch switch-statement can encode $\log(n!)$ bits.

A program transformation which 1) removes all comments, 2) breaks up all strings into shorter strings, and 3) reorders instructions whose order is insignificant will remove all of the example watermarks above. Moreover, if part of a watermark resides in portion of the program which can be determined to be dead code, then a straightforward dead-code removal may be a successful attack.

The main approaches to defending against such attacks are to either 1) make the watermark an inherent part of the programs behavior, or 2) to make the watermark be represented by data structures that are created at run-time. In the first category, there is the notion of an *Easter Egg* watermark which is a piece of code that gets activated after an unusual input. For example, a particular input sequence to Microsoft Excel 97 starts an alien-world-exploration game. In the second category, we find the approach of Collberg and Thomborsen [3] who explained how to exploit that a number can be represented as a graph which, in turn, can be built as an object structure during the execution of a Java program. The idea is that object structures are difficult to analyze precisely at compile time because such analysis involves flow analysis and pointer analysis [13, 6]. An attacker might therefore learn little about the location of a watermark by statically analyzing a program. Note that the building of an object structure introduces the need for extra code which builds that object structure. An attacker can attempt to locate that code even without knowing the exact details of the watermark. Until now there have been no reports on practical experience with Collberg and Thomborsen's technique.

There are other forms of attack than semantics-preserving program transformations. Using terminology of Collberg and Thomborsen [3], a *subtractive* attack is one where Bob, perhaps with some tool support, tries to locate and remove a watermark. Easter Egg watermarks may be vulnerable to such attacks because the Easter Egg code may stand out in a way that a human reader can detect without much effort. An *additive* attack is one where Bob inserts his own watermark in an attempt to override Alice's watermark, or at least make it plausible that Alice's watermark was not inserted before Bob's. One possible defense against this form of attack is to let Alice show that her copy of the software contains only her watermark, not Bob's, which means Bob's watermark is added after hers. Another possible protection mechanism is that Alice engages in a protocol with a trusted authority to timestamp the hash value of her watermark. A *collusion* attack is one where two attackers have two copies of the same watermarked program—but watermarked with different watermarks—and compare the two watermarked programs in an attempt to locate the watermark.

An attacker may do more than just analyzing the program text; the attacker may also monitor the state of the heap, registers, etc. during execution. This may reveal a data structure that is suspiciously long lived, it may also reveal a condition that always evaluates to true, etc. Such observations may help locate the watermark.

1.4 Problem Statement

Our goal is to gain practical experience with Collberg and Thomborsen's idea of embedding a watermark in dynamic data structures. Our first goal is to verify the hypothesis:

> *is such a watermark resilient to program-transformation attacks?*

Peticolas, Anderson, and Kuhn [14] wrote: "the problem [with watermarking] is not so much inserting the marks as

Attack	Protection
program-transformation attack	represent watermark as an object structure
subtractive attack	tamperproofing and obfuscation
collusion attack	randomization

Table 1. Attacks and Protection Methods

recognizing them afterwards." This leads us to ask the question:

can we efficiently retrieve such a watermark?

It is *not* our goal to produce a watermarking system which is resilient to *all* of the attacks listed in Section 1.3.

1.5 Our Results

We have implemented and experimented with a watermarking system for Java which embeds a watermark in dynamic data structures. Our experiments show that watermarking can be done efficiently with moderate increases in code size, execution times, and heap-space usage, while making the watermarked code resilient to a variety of program-transformation attacks. For a particular representation of watermarks, the time to retrieve a watermark is on the order of one minute per megabyte of heap space.

Our prototype watermarking system, called JavaWiz, is publicly available from our web site at http://www.cs.purdue.edu/s3/, together with all benchmarks and test inputs. Thus, an interested reader should be able to reproduce the experimental results presented in Section 2, and try some attacks.

The version of JavaWiz available from our website is not designed to resists all possible attacks; to do that it should be combined with other protection techniques such as obfuscation and tamperproofing. In this paper we outline how to combine the techniques implemented in the publicly available version of JavaWiz with the current best practices for obfuscation, tamperproofing, and randomization, see table 1.

Rest of the Paper. In section 2 we summarize Collberg and Thomborsen's idea of embedding a watermark in dynamic data structures, we discuss the JavaWiz implementation, and we show our experimental results. In section 3 we give a summary and a critique of various protection techniques, and we outline how to integrate them with our other techniques.

2 Implementation and Experimental Results

We now describe our prototype watermarking system, called JavaWiz. It can watermark Java 1.2 source programs, and it is written in Java. If presented with a Java bytecode program, then we first decompile it to source code using one of the available bytecode decompilers. JavaWiz does need access to all parts of the input program, and it does not touch the standard library.

2.1 Watermark Representation

For simplicity, we assume from now on that a watermark is a natural number. A number can be represented as a graph in many ways, e.g., as a Planted Plane Cubic Tree (PPCT). For use in a watermarking system, it is best to have a choice between as many kinds of representations as possible. The current version of JavaWiz uses PPCTs. Here follow a few details of PPCTs.

A *PPCT*, see Figure 1, is essentially a binary tree with one extra node called the Origin, The Origin has a pointer to the root of the binary tree. Moreover, all leaves of the binary tree are linked into a circular linked list which includes the Origin. Finally, each leaf has a self-pointer. Notice that every node in a PPCT has two outgoing pointers.

A particularly important property of PPCTs is that if we have a pointer to any node in the PPCT, then we can recover the Origin. This can help locate a PPCT in an arbitrary heap.

Let us now define mappings from integers to PPCTs and back. Let $c(n)$ be the number of PPCTs with n leaves (this is called a Catalan number [7].) We have

$$c(n) = \frac{1}{n} * \left(\begin{array}{c} 2n-2 \\ n-1 \end{array} \right)$$

We can enumerate the PPCTs with n leaves as shown in Figure 2.

We denote by $T(L,R)$ the set of binary trees with L leaves in the left subtree, and R leaves in right subtree. We can recursively define the minimum integer represented by the trees in $T(L,R)$ as follows:

$$\begin{aligned} min\_int(L,R) &= min\_int(L-1,R+1) \\ &\quad +c(L-1)\times R \quad (L \neq 1) \\ min\_int(1,R) &= 0 \end{aligned}$$

Now let us define the integer that a PPCT represents. Given a PPCT in which the binary tree is T, we denote the

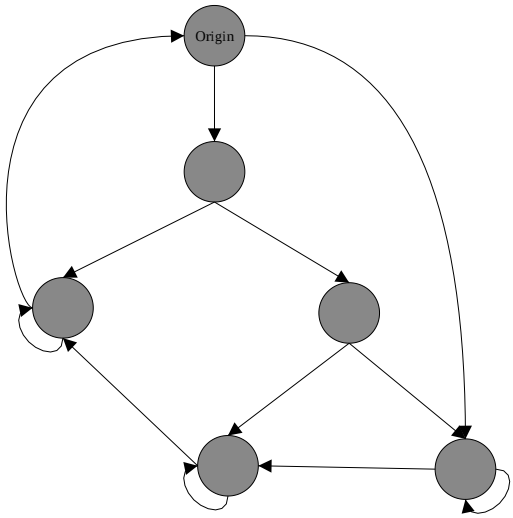

Figure 1. PPCT

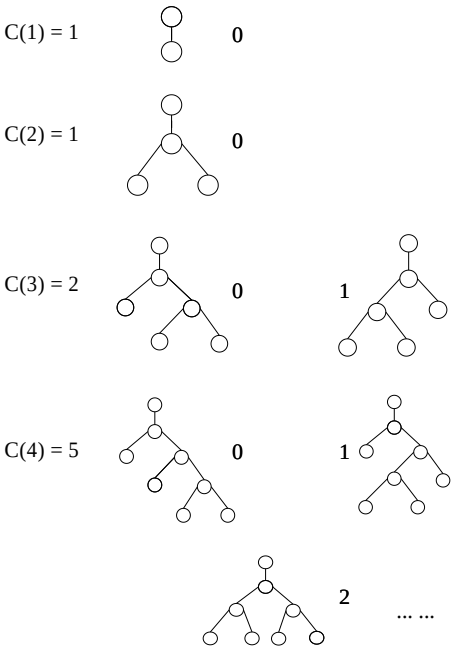

Figure 2. Enumeration of PPCT

represented integer by $int(T)$. Then

$$\begin{aligned} int(T) &= int(T.left) \times c(LeafNum(T.right)) \\ &\quad + int(T.right) \\ &\quad + min\_int(LeafNum(T.left), \\ &\qquad LeafNum(T.right)) \\ int(leaf) &= 0 \end{aligned}$$

where function $LeafNum(node)$ gives the number of leaves of the tree rooted by $node$.

2.2 Watermark Embedding

To watermark a Java program with a given number, we first determine its PPCT representation. Next we choose a base class from the original program and convert it into a *node class* by adding some fields. Each additional field will hold an outgoing edge to another node object. The node class is now a building block for constructing the PPCT, and off-line we generate straight-line code that constructs the PPCT. The graph construction code is then merged with the original program. We do this in a particularly simple way to illustrate the idea without getting into implementing full-fledged randomization, obfuscation, and tamperproofing. We view these other protection techniques as building blocks that can be added on top of our approach. In our implementation only the "new" expressions in the original code will be changed. Intuitively if there is an expression "new A()", then we may change it to "new A1()", and add a class A1 which extends A and place some watermarking code in its constructor.

For example, suppose we have the following code for class A:

```
class A{
   A(){
      a1 = 0;
   }
   int a1;
}
```

We can change the code into:

```
class A1{
   A1(){
      a1 = 0;
      <code for building
      watermark>// produced offline
   }
   int a1;
}
```

While this may seem simple, bordering on trivial, it is sufficient to protect against a variety of program-transformation attacks, as we will show later.

To resist attacks based on dead-code removal, we create data dependencies from the original program to the watermark. We do this by replacing a statement S in the original program with a statement of the form

```
if (x != y) S
```

where `x,y` are distinct nodes in the watermark graph. This transformation makes the watermark graph part of the computation of the original program, and it requires a powerful pointer analysis to determine that `x != y` is always true. If a given dead-code removal tool does not determine that `x != y` is always true, then the watermark will not be eliminated as dead code.

2.3 Watermark Retrieval

When retrieving a watermark, we cannot rely on the class names of the watermark because class names can be changed by obfuscators. We retrieve a watermark by accessing the heap image at some point during execution of the watermarked program. Getting access to a heap image is supported by the Java 2 SDK [11] (also known as the Java Development Kit, Version 1.2.), and we take a snapshot of the heap image by using the `-Xhprof` option to the `java` command.

The processing of the heap image proceeds in four steps.

1. **Extract potential node classes.** We do that by extracting the set of classes used in the heap image, *except* those in the standard Java runtime packages. (Our watermarking system does not use standard classes as node classes.)

2. **Extract potential node objects.** We do that by extracting the set of objects in the heap image that are of a potential node class, as determined in step 1.

3. **Determine potential edges.** For each field in a potential node object, determine whether it can potentially hold an outgoing edge in a watermark graph. We do that by comparing the field's type declaration with the potential node classes, as determined in step 1. After this step, we have completed building a graph representation of the part of the heap image that has a chance of containing a watermark.

4. **Graph search.** We are now ready to search for the watermark graph. In general, this is the problem of subgraph isomorphism which is known to be NP-complete.

The rightful owner of the watermarked software knows what the watermark is. We can use that information to do a more efficient graph search that takes advantage of knowing special properties of the watermark graph. Our prototype watermarking system contains an implementation of such a specialized graph search for the case where the watermark is a *PPCT*. Our search strategy proceeds by, repeatedly, first finding a potential leaf of a *PPCT*, then locating a potential origin node, and finally checking that the potential origin node is indeed an origin node for the watermark *PPCT*. Each of these steps involves the traversal of paths in the graph, and since we know the size of the watermark *PPCT*, we have an upper bound on how far we need to pursue a given path. Notice that when searching for a *PPCT*, we can prune away all potential node classes with less than two potential edge fields.

With our specialized search for a *PPCT*, retrieving a watermark is tractable, although time consuming, as we show next.

2.4 Experimental Results

Our experiments were run under Solaris 2.5.1 on a Sun Ultra 2 Model 2200, with 256MB RAM, and two 200MHz UltraSPARC-I processors, each with 1MB external cache in addition to their on-chip instruction and data caches. The UltraSPARC-I data cache is a 16KB write-through, non-allocating, direct-mapped cache with two 16-byte subblocks per line.

We used the Sun Java 2 SDK Version 1.2.1 Solaris Production Release Virtual Machine with the default Just-In-Time compiler turned on.

We tested our JavaWiz watermarking system on seven medium-sized Java programs, see table 2. In each case, we embedded a 40-digit watermark. For each test program, the third column of the table shows an input that we have used when measuring the run time and the needed heap space before and after watermarking.

In the following table we use "before" to denote *before watermarking*, and we use "after" to denote *after watermarking*. The listed *code sizes* do not include the standard libraries. Each test was done five times. The numbers in the table are the averages. Dimensions: *code size* is in kilobytes; *wm time* is in seconds and means the time to embed a watermark; *retr time* is in minutes and means the time to retrieve watermark; *execution time* is in seconds and means the watermarked program's execution time; and *heap space usage* is in kilobytes and means that heap space used by the watermarked program.

¿From the table 3, one can see that, for our benchmarks: 1) JavaWiz adds 4–12 kilobytes of code (less than 10 kilobytes of code on average), 2) embedding a watermark is done in less than 20 seconds, often faster, 3) watermarking increases a program's execution time less than 7%, often much less, 4) the heap space requirement increases, al-

program	description	test input
javac	a compiler for Java	the JavaCup source code
javadoc	a Java API documentation generator	the JavaCup source code
JavaCup	an LALR parser generator for Java	the CORBA grammar
JTB	JTB [16] is a frontend for The Java Compiler Compiler from Sun Microsystems	the Java 1.2 grammar
JavaWiz	the watermarking system reported in this paper	the JavaCup source code
compress	a java virtual machine spec benchmark	some tar files shipped with compress
BLOAT	BLOAT [9] is a Java bytecode optimization tool	the JavaCup source code

Table 2. Programs on which we have experimented

program	code size		wm time	retr time	execution time		heap space usage	
	before	after			before	after	before	after
javac	192	201	18.8 s	7.1 min	79.4 s	82.5 s	6,415	6,453
javadoc	187	191	19.9 s	8.9 min	26.7 s	27.4 s	9,770	10,000
JavaCup	362	373	5.6 s	4.6 min	4.3 s	4.6 s	4,041	4,080
JTB	810	815	5.2 s	0.6 min	9.9 s	10.1 s	440	475
JavaWiz	582	591	4.3 s	2.2 min	4.7 s	4.9 s	2,012	2,045
compress	24	32	4.6 s	0.6 min	68.8 s	72.4 s	477	514
BLOAT	1,415	1,427	7.0 s	3.6 min	55.7 s	57.9 s	3,322	3,362

Table 3. Experimental Results

though it should be noted that the increase depends on the size of the objects of the class which is chosen as node class, and 5) watermark retrieval is done in about 1 minute per 1 megabytes of heap.

We have tried to attack the watermarked programs with the Java bytecode obfuscator WingGuard [5] and the Java packaging tool JAX [17]. We can view obfuscation and packaging as attacks because they are semantics-preserving program transformations. JAX is particularly interesting as an attack because it attempts to eliminate dead code. In all cases, we found that the watermark was intact after the attacks.

3 Integration of Protection Techniques

Our experience shows that a watermarking system can greatly benefit from the protection mechanisms of randomization, obfuscation, and tamperproofing. In the following, we give a summary and a critique of the current best practices, and we discuss how they can be integrated with our other techniques into a full-fledged watermarking system.

We will assume a worst-case scenario where the attacker has access to:

1. the watermarked code,
2. a graphical display of the heap during a run of the watermarked code, and
3. the source code for the watermarking system (but not necessarily the form of data structures used to represent watermarks.)

Henceforth we will refer to such an attacker as an "expert attacker." Of course, one could adopt a business model where anybody who wants their software to be watermarked has to send it in, and get it watermarked by the owner of the watermarking software. Still, the watermarking software may be stolen or simply handed over via bribery. Moreover, one might be interested in selling the watermarking software and let the buyers do the watermarking themselves. If the watermarking software is sold, then it is difficult to prevent that an attacker gets access to it.

Throughout the this section, we use P to denote the program to be watermarked, W to denote the watermark itself, and C to denote a piece of straight-line code (generated off-line) which, when executed, will produce a watermark graph. Each of the techniques discussed in this section concern the question of producing a merger of P and C which is resilient to attacks.

3.1 Randomization

To merge P and C, one possibility is to insert C right at the beginning of the main program of P. This would ensure

that C is executed at the beginning of a run of the watermarked code. It has the drawback that an expert attacker will know where to look for the C code. An alternative is an Easter Egg approach where the code is executed after an unusual input. This has the drawback of being more difficult to fully automate. Yet another alternative is randomization, that is, weaving P and C in a random way, and changing the order, in a random way, of parts of P and C in cases where the order does not matter. The opportunities for such random weaving may be limited by the nature of P, but it has the advantage that even an expert attacker may have to examine a large part of the watermarked code to find C.

Randomization can protect against collusion attacks to some extent. Suppose two attackers have access to two copies of the same software which have been watermarked with different watermarks (this is also known as "fingerprinting".) The attackers may do a "diff" of the two copies, and thereby learn the location of the watermarking code.

3.2 Obfuscation

Obfuscation is useful for two related purposes:

1. to make C unrecognizable, and
2. to make P and C look alike.

If we can achieve (1) and (2), then it will be difficult even for an expert attacker to locate the obfuscated version of C.

There is a wide variety of obfuscation techniques: padding, opaque predicates, renaming, variable splitting/merging, method inlining/outlining, and many others, see the survey of Collberg, Thomborsen, and Low [4]. All of these techniques are general, and any general-purpose obfuscation tool that implements them can be helpful. Let us here focus on two obfuscation opportunities that can be given twists that are specific to the watermarking setting.

First, we can obfuscate C by *padding* it with statements that lead to the building of a *bigger* graph than the one for representing W. Given that the kind and the size of the graph for W are secret, the further padding effectively makes it impossible for an expert attacker to know what part of the graph is the watermark.

Second, we can obfuscate P based on *opaque predicates* [4]. The idea of obfuscation with opaque predicates is to change a statement S into a conditional statement "if c then S", where c is an opaque predicate. An opaque predicate is a condition which always evaluates to **true** (or always evaluates to **false**), but where this property is difficult to determine for an attacker. The use of opaque predicates will therefore make it more difficult to determine the control flow by just inspecting the program.

A problem with opaque predicates is that if an attacker can monitor the heap, registers, etc. during execution, then it may quickly be revealed that a given predicate always evaluates to true. To protect against such an attack, one needs to avoid that a predicate always evaluates to the same result. This can be done by obfuscating with "dynamically" opaque predicates, that is, a family of *correlated* predicates which all evaluate to the same result in any given run, but in different runs they may evaluate to different results. For example, if we have five correlated predicates, then we might observe that they evaluate as shown in the following table.

Predicate	Run 1	Run 2	Run 3	Run 4	Run 5
Pred 1	T	T	F	T	F
Pred 2	T	T	F	T	F
Pred 3	T	T	F	T	F
Pred 4	T	T	F	T	F
Pred 5	T	T	F	T	F

It seems advantageous to use both traditional opaque predicates and dynamically opaque predicates for obfuscation because it makes it difficult for an attacker to determine whether a given predicate is either

- one of those from the original program, or
- a traditional opaque predicate, or
- a dynamically opaque predicate.

The attacker may monitor the results of evaluating the predicates in a program in various runs, but this may not give a solid basis for classifying the predicates correctly. If there are many predicates overall, then the classifying task will require a large effort. In particular, determining which predicates are correlated may mean keeping track of an exponentially many sets of predicates.

An attacker might consider using a static analysis with the purpose of detecting branch correlations. One such analysis has been presented by Bodik, Gupta, and Soffa [2]. We see it as essential that an implementation of dynamically opaque predicates cannot be beaten by, at least, the current static analysis techniques.

One drawback of dynamically opaque predicates is that they seem to require more code to be inserted into P than traditional opaque predicates, because of the need to correlate the predicates. The extra code may be a vulnerability in itself, and it may help slow down the execution of the watermarked program.

It turns out that a *PPCT* is an excellent source for opaque predicates. For example, we can maintain two pointers from the outside to nodes in a PPCT, and from time to time move the pointers to point to other nodes in the PPCT. It is easy to maintain the invariant that, say, the two pointers point to different node. Thus, if the pointers reside in variables x and y, then the condition $(x\,!{=}y)$ is an opaque predicate which always evaluates to **true** and it is difficult for an attacker to guess the invariant.

We can use the watermark PPCT as the source of opaque predicates for obfuscation of those parts of the original program P that come after all of C has been executed. (The interleaving of P and C will determine how much of P comes after C.) A problem here is that this will leave both C and the initial portion of P unobfuscated. We can improve the situation by inserting code for building a second, random, PPCT (say) at the very beginning of the merged program, and then use that as a basis for obfuscation. We might even choose to use the random PPCT as a basis for obfuscation of all of P merged with C.

The use of a second PPCT highlights a chicken-and-egg problem because now the code that builds the random PPCT is unobfuscated and therefore a possible target for an expert attacker. It is conceivable that an expert attacker could locate the code for building the random PPCT and from there unravel the whole construction. With current technology, the only approach to protecting against such an attack seems to be to, after the other transformations are done, apply a general-purpose obfuscation tool, such as WingGuard [5]. Note though, that if the extra obfuscation technique is known to an expert attacker, then there may be ways of unobfuscating the code.

3.3 Tamperproofing

Collberg and Thomborsen [3] gave an example of tamperproofing a watermark representation using the Java reflection mechanism. The idea is to verify, at run time, that the classes used to represent the watermark are intact. However, as noted in [3], this style of tamperproofing is not stealthy.

An alternative idea is to base tamperproofing on the integrity of the watermark graph structure itself. For example, a tamperproofing mechanism may be able to detect, at run time, that the graph representing the watermark is no longer of the required form, say, a PPCT. The approach to obfuscation based on deriving opaque predicates from a watermark PPCT achieves a level of such tamperproofing because it creates data dependencies from P to the watermark graph. If an attacker manages to distort the watermark PPCT, then some opaque predicates may evaluate to **false** instead of **true**. If this is the case, then the program may afterwards begin to misbehave in a unpredictable way. Moreover, C is obfuscated with the use of the random PPCT, so if that PPCT is distorted, then C may misbehave, leading to the wrong watermark PPCT, and, in turn, a misbehaving P. By doing this, we have created a link of dependency from the correctness of the original program *P* to the integrity of the watermark *W*. The overall effect is that once the embedded watermark loses its original form, the semantics of the watermarked program will also change.

We can take this idea one step further and make it stronger by building a dependency cycle. For example, suppose we could create a dependency from the code that builds some random PPCT to the original data structures. This would create a dependency cycle (from the original program to the watermark PPCT, to the random PPCT, back to the original program) and thereby make it difficult to change something without affecting everything else. One way of achieving this would be to ask the programmer of P to supply some opaque predicates based on the original data structures. This will make the watermarking system semi-automatic rather than fully automatic.

3.4 Pool of Watermark Representations

It is beneficial to hide the *forms* of watermark representations from attackers. If an attacker knows the form of data structure used to represent a watermark, the attacker can do a heap analysis to look for specific instances of the watermark representation; and that may be easy and inexpensive to do. Thus it is an improvement to separate the watermarking system and the watermark data structures. It seems best to establish a *pool* of different kinds of watermark data structures, and not just to focus on PPCT. For example, a company can develop its own proprietary watermark representations. When watermarking a piece of program, at least one watermark data structure will be chosen from the pool.

It might seem plausible that we make all the potential candidates of watermark data structures public once we have a large number of them. This is because an attacker has to try, in the worst case, all of them to find the hidden watermark. However, this idea has a flaw: the workload for attackers to try all the candidates may not be prohibitively heavy. On the contrary, the cost is just the sum of trying each one; and one can carry out the work in parallel, which can be done in an efficient way.

We can now summarize the properties of PPCTs that make them attractive as watermark representations. When choosing other forms of representation, it seems best to choose some with similar properties. Specifically, a representation:

- should have a stealthy heap images,
- should represent a number via some enumeration method (this is generally difficult, although it may not be necessary if one can publish the hash of a graph),
- should be a source of opaque predicates (for creating dependencies), and
- should have some special properties that are easy to check (for tamperproofing) and that do not stand out (for stealthiness).

3.5 Summary

With the above discussion in mind, the merging of P and C may proceed in three steps:

1. choose from the watermark representation pool an appropriate W, generate C from W offline;
2. pad C and merge it with P using randomization, obfuscation, and tamperproofing based on one or more graphs (e.g., PPCTs); and
3. obfuscate the outcome of (1) with a general-purpose obfuscator.

An attacker may attempt to carry out the following steps:

1. Locate the code that builds the graphs;
2. remove from rest of the program all dependencies on the graphs, that is, remove all opaque and dynamically opaque predicates; and
3. remove or distort the code that builds the two graphs.

A goal of a good watermarking system should be to make it useless for an attacker to do just (1), and to make it difficult to do (2), hence also difficult to do (3).

4 Conclusion

Our current watermarking system is a promising step in the direction of a practically useful tool. The idea of embedding a watermark in dynamic data structures is viable, it leads to watermarked programs that are resilient to a variety of program-transformation attacks, and such watermarks can be retrieved efficiently. To protect against other attacks, one needs to integrate other protection techniques such as obfuscation and tamperproofing. Each of these protection methods gives protection against specific attacks. While there may be simple ways of putting these methods together in a watermarking system, it is better if we can utilize them in a correlated way so that each of them supports the others and thereby makes attacks more difficult.

Acknowledgments. We thank the reviewers for helpful comments. We also thank the other members of the Secure Software Systems group for many discussions about watermarking.

References

[1] T. Archives. http://www.tsinghua.edu.cn/docsn/dag/djse.htm, 1999.

[2] R. Bodik, R. Gupta, and M. L. Soffa. Refi ning data fbw information using infeasible paths. In *Proceedings of the 5th ACM SIGSOFT Symposium on Software Engineering*, pages 361–377, Sept. 1997.

[3] C. Collberg and C. Thomborsen. Software watermarking: Models and dynamic embeddings. In *Proceedings of POPL'99, 26th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 311–324, 1999.

[4] C. Collberg, C. Thomborsen, and D. Low. Manufactoring cheap, resilient, and stealthy opaque constructs. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 184–196, 1998.

[5] W. Corporation. http://www.wingsoft.com/.

[6] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 222–236, San Diego, California, January 1998.

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

[8] K. Holmes. Computer software protection. US Patent 5,287,407, Assignee: International Business Machines, Feb. 1994.

[9] A. Hosking and N. Nystrom. BLOAT: Bytecode-level optimizer and analysis tool. Purdue University, www.cs.purdue.edu/homes/hosking/research.html, 1999.

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

[11] Sun Microsystems. Java 2 SDK, standard edition documentation. http://www.javasoft.com/products/jdk/1.2/docs/index.html, 2000.

[12] S. A. Moskowitz and M. Cooperman. Method for stegacipher protection of computer code. US Patent 5,745,569, Assignee: The Dice Company, Jan. 1996.

[13] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

[14] F. A. P. Peticolas, R. J. Anderson, and M. G. Kuhn. Attacks on copyright marking systems. In *Proceedings of the Second Workhop on Information Hiding*, Apr. 1998.

[15] P. R. Samson. Apparatus and method for serializing and validating copies of computer software. US Patent 5,287,408, Assignee: Autodesk, Inc, Feb. 1994.

[16] K. Tao and J. Palsberg. The Java tree builder. Purdue University, www.cs.purdue.edu/jtb, 1997.

[17] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. *IBM Systems Journal*, 1999.