Electron Commerce Res (2006) 6: 155–171
DOI 10.1007/s10660-006-6955-z

Software watermarking via opaque predicates: Implementation, analysis, and attacks

Ginger Myles · Christian Collberg

© Springer Science + Business Media, LLC 2006

Abstract Within the software industry software piracy is a great concern. In this article we address this issue through a prevention technique called software watermarking. Depending on how a software watermark is applied it can be used to discourage piracy; as proof of authorship or purchase; or to track the source of the illegal redistribution. In particular we analyze an algorithm originally proposed by Geneviève Arboit in *A Method for Watermarking Java Programs via Opaque Predicates*. This watermarking technique embeds the watermark by adding opaque predicates to the application. We have found that the Arboit technique does withstand some forms of attack and has a respectable data-rate. However, it is susceptible to a variety of distortive attacks. One unanswered question in the area of software watermarking is whether dynamic algorithms are inherently more resilient to attacks than static algorithms. We have implemented and empirically evaluated both static and dynamic versions within the SANDMARK framework.

Keywords Software piracy · Copyright protection · Software watermarking · Opaque predicate

Software piracy and copyright infringement have been issues of concern for some time. The ease with which people can access and use the Internet has lead to the widespread dissemination of illegal software. To compound the problem, software is being legally distributed in architectural neutral formats, such as Java bytecode. These formats closely resemble source code and can be easily decompiled and manipulated. Not only do these formats make it easy for software pirates to bypass license checks but they also allow unscrupulous programmers to steal algorithmic secrets. This may allow them to decrease their own production time to get an edge on the competition.

This work is supported by the NSF under grant CCR-0073483, by the AFRL under contract F33615-02-C-1146, and the GAANN Fellowship.

G. Myles (✉) · C. Collberg
Department of Computer Science, University of Arizona, Tucson, AZ, 85721, USA
E-mail: mylesg@cs.arizona. edu; collberg@cs.arizona. edu

Of course, there are legal ramifications associated with software piracy, such as a $150,000 fine for each program copied [1]. However, these fines are often targeted at an unsuspecting end user and not at the person actually responsible for the piracy. When a person unknowingly purchases and uses an illegal piece of software it is often difficult to trace this software back to the guilty party. In addition, it is also hard to detect and prove that an unscrupulous programmer has taken advantage of a trade secret.

Various organizations do perform audits to verify that corporations are not using illegal software [1]. Unfortunately, auditing does not identify an unknown software pirate or unscrupulous programmer. *Software watermarking* is one technique currently being investigated to tackle this issue. Software watermarking embeds a unique identifier in a program. The unique identifier can be used to identify the author or the legal purchaser of the program. An authorship mark can be used against the unscrupulous programmer and a purchase mark can be used to track the source of the illegal redistribution [18].

In this paper we present an implementation and empirical evaluation of a software watermarking technique originally proposed by Arboit [5]. To the best of our knowledge this technique has never been implemented nor empirically evaluated. The general idea behind the algorithm is to embed the watermark by appending opaque predicates to branching points selected throughout the application. The implementations have been incorporated into the SANDMARK framework [2]. This allows us to evaluate the resilience of the algorithms to manual and automated attacks. In particular, we consider attacks by tools such as static statistics and code obfuscations. We also present a novel extension of this idea which uses a dynamic recognition technique.

The remainder of the paper is structured as follows. We begin with a discussion of software watermarking and previously proposed software watermarking algorithms. In Section 2 we discuss opaque predicates, one of the most important aspects of the technique proposed by Arboit. We present a general description of the technique as it was proposed in Section 3. This is followed, in Section 4, by the details of our implementation and the dynamic technique. Section 5 provides a detailed empirical evaluation of the algorithms and an evaluation of static versus dynamic versions of the algorithm. Finally, in Section 6 we summarize our findings.

1. Software watermarking

Software watermarking is just one of many techniques that is currently being studied to prevent or discourage software piracy and copyright infringement. The idea is similar to media watermarking where a unique identifier is embedded in image, audio, or video data through the introduction of errors not detectable by human perception. Due to the nature of software it is not possible to strictly apply the ideas found in media watermarking. Instead embedding an identifier in a piece of software must be done in such a way that the original functionality is maintained.

Definition 1. (Software watermarking System). Given a program $\mathcal{P}$, a watermark w, and a key k, a software watermarking system consists of two functions:

- $\texttt{embed}\,(\mathcal{P}, w, k) \rightarrow \mathcal{P}'$
- $\texttt{recognize}\,(\mathcal{P}', k) \rightarrow w$.

There are two general categories of watermarking algorithms, static and dynamic. A dynamic algorithm relies on information gathered from the execution of the application to embed and recognize the watermark. Static algorithms only examine the static code and data of the application. A variety of techniques have been proposed for software watermarking but there are few publications describing the implementation and evaluation of these algorithms.

There are far more static watermarking algorithms than dynamic due to the multitude of locations where information can be hidden in an executable. For example, in a Java classfile a static watermark can be embedded in the constant pool table, method table, etc.

Davidson and Myhrvold [12] proposed a static watermarking algorithm which embeds the watermark by reordering the basic blocks of a control flow graph. Venkatesan et al. [25] build on this idea in an algorithm which embeds the watermark by extending a method's control flow graph through the insertion of a subgraph. Monden et al. [14, 15] propose a technique which embeds the watermark in a dummy method through a specially constructed instruction sequence. Stern et al. [23] also consider instruction sequences for embedding the watermark. Their technique modifies instruction frequencies to represent the watermark. Qu and Potkonjak [20] make use of the graph coloring problem to embed a watermark in the register allocation of an application.

The first dynamic watermarking algorithm, CT, was proposed by Collberg et al. [8]. In this technique the watermark is embedded through a graph structure which is built on the heap at runtime. A second technique by Cousot and Cousot [11] makes use of abstract interpretation to embed a watermark in values assigned to integer local variables during execution. Collberg et al. [10] proposed a dynamic path-based technique which embeds the watermark in the dynamic branching behavior of the application by modifying the sequence of branches taken and not taken on the secret input sequence. A final dynamic technique by Nagra and Thomborson [17] relies on multi-threading to embed the watermark.

Of the early algorithms very little has been published on their implementation and evaluation. There are a few existing implementations of the CT algorithm, such as the one within the SANDMARK framework and that by Palsberg et al. [19]. A recent dissertation by Hachez [13] provides an analysis of the Stern algorithm, as does Sahoo [21]. The Qu and Potkonjak technique was evaluated by Myles [16] and Collberg et al. [6] provide an evaluation of the Venkatesan technique.

SANDMARK [7] is a research tool for studying software protection techniques and in particular software watermarking, code obfuscation, and tamper-proofing of Java bytecode. One of the goals of the SANDMARK project is to implement and evaluate all known software watermarking algorithms. The system includes a variety of tools that permit the study of watermarking algorithms with respect to such properties as resiliency and stealth. Through the implementation and evaluation of known software watermarking algorithms we will be able to gain an understanding of what makes a software watermarking technique strong.

2. Opaque predicates

Opaque predicates were first presented by Collberg et al. [9] as a technique to aid in code obfuscation and later incorporated in a software watermarking technique proposed by Monden et al. [14, 15]. Informally, opaque predicates are inserted to make it difficult for an adversary to analyze the control-flow of the application. This makes it more difficult to identify that certain portions of the application are superfluous. For example, the Monden algorithm uses opaque predicates to disguise the fact that a dummy method is never invoked.

Table 1 Number theoretically true opaque predicates used in the implementation of the Arboit Algorithms

$\forall x, y \in \mathbb{Z}$	$7y^2 - 1 \neq x^2$
$\forall x \in \mathbb{Z}$	$2 \mid \lfloor \frac{x^2}{2} \rfloor$
$\forall x \in \mathbb{Z}$	$2 \mid x(x+1)$
$\forall x \in \mathbb{Z}$	$x^2 \geq 0$
$\forall x \in \mathbb{Z}$	$3 \mid x(x+1)(x+2)$
$\forall x \in \mathbb{Z}$	$7 \nmid x^2 + 1$
$\forall x \in \mathbb{Z}$	$81 \nmid x^2 + x + 7$
$\forall x \in \mathbb{Z}$	$19 \nmid 4x^2 + 4$
$\forall x \in \mathbb{Z}$	$4 \mid x^2(x+1)(x+1)$

Definition 2. (Opaque predicate). A predicate P is opaque at a program point p, if at point p the outcome of P is known at embedding time. If P always evaluates to `True` we write P_p^T, for `False` we write P_p^F, and if P sometimes evaluates to `True` and sometimes to `False` we write $P_p^?$ [9].

Definition 3. (Opaque method). A boolean method M is opaque at an invocation point p, if at point p the return value of M is known at embedding time. If M always returns the value of `True` we write M_p^T, for `False` we write M_p^F, and if M sometimes returns `True` and sometimes `False` we write $M_p^?$.

The key challenge to using opaque predicates or opaque methods is to design them in such a way that they are resilient to various forms of analysis. If an adversary can easily decipher the value of an opaque predicate it provides very little protection for the software. A variety of techniques such as using number theoretic results, pointer aliases, and concurrency have been suggested for the construction of opaque predicates [9]. In addition to the number theoretic results, Arboit also suggests a technique for constructing a family of opaque predicates through the use of quadratic residues. Our current implementation of the Arboit algorithms uses number theoretically true opaque predicates and opaque methods. The nine we have implemented thus far can be seen in Table 1. An important aspect of the Arboit algorithms is that the opaque predicate library must remain secret. If an adversary knows even a few of the opaque predicates used in the embedding he may be able to identify them in the application and then remove them.

None of the nine opaque predicates used in the current implementation are considered cryptographically secure or even resilient to analysis. While this does weaken the implementation it does not invalidate the analysis in Section 5. The disadvantage of using these opaque predicates is that the algorithm is not as stealthy and is susceptible to manual attacks that will be elaborated on. As more sophisticated opaque predicates become available within the SANDMARK framework they will be used to embed the watermark in place of the simple ones in Table 1.

3. Arboit algorithm

Arboit proposed two watermarking techniques both based on opaque predicates [5]. The first algorithm (henceforth GA1) is the basic insertion algorithm which directly uses the opaque predicates. To embed a watermark, w is split into k pieces, $w_0, \ldots, w_{k-1}$, and k branching

points, $b_0, \ldots b_{k-1}$, are randomly selected throughout the application. At each branching point b_i, either $\wedge P^T_{b_i}$, $\vee\neg P^T_{b_i}$, or $\vee P^F_{b_i}$ is appended to the predicate at that location. The bits of the watermark are embedded through the opaque predicate that has been chosen. Within the opaque predicate the bits can be encoded either as constants or by assigning a rank to each of the opaque predicates. To recognize the watermark the application is scanned, extracting all identifiable opaque predicates. The bits of the watermark are then decoded from the opaque predicate. As an example, suppose our watermark is encoded in the opaque predicate $x_2 \geq 0$. A watermark could be embedded as follows:

```
class c{
   void ml(int a, int b){
      ...
      if (a <= b) {...}
      else{...}
      ...
   }
}
```

$\overset{\mathcal{W}}{\Longrightarrow}$

```
class c{
   void ml(int a, int b){
      ...
      int x = 1;
      if (a <= b)  &&
          (x*x>= 0)){...}
      else{...}
      ...
   }
}
```

The second Arboit algorithm (henceforth GA2) is similar to GA1 except opaque methods are used to embed the watermark. Again k branching points $b_0, \ldots, b_{k-1}$ are randomly selected throughout the application. For each b_i, M^T_{bi} or M^F_{bi} is created and a method call is appended. The bits of the watermark are encoded in the opaque method through the opaque predicate that it evaluates. To recognize the watermark the application is scanned, extracting all opaque methods which are first identified through their signatures. Once a possible candidate has been identified the method body is examined to find the opaque predicate. To illustrate, suppose we use the same opaque predicate as above. Using GA2 the application would be transformed in the following way:

```
class c {
   void ml(int a, int b){
      ...
      if (a <= b) {...}
      else{...}
      ...
   }
}
```

$\overset{\mathcal{W}}{\Longrightarrow}$

```
class c {
   boolean m2(){
      int x = 1;
      return (x*x >=0);
   }
   void ml(int a, int b){
      ...
      if ((a <= b) &&
          m2()){...}
      else{...}
      ...
   }
}
```

Arboit claims that GA2 is more secure. The main argument is that changing the signature of a method is difficult. However, this claim is untrue and SANDMARK includes code obfuscations which can do just that. In Section 5 we will show that GA1 is in fact a stronger algorithm than GA2. This claim demonstrates the importance of implementation and evaluation in the proposal of a software watermarking algorithm.

4. Implementation details

Our implementations of GA1 and GA2 follow from the algorithms presented by Arboit [5]. A few modifications described below were made in an attempt to make the algorithms more resilient to attack. In addition, we developed and implemented dynamic versions of the algorithms.

4.1. Watermark encoding

Arboit proposed an encoding technique in which each piece of the watermark also includes an index value. By including the index value the watermark pieces can be recovered in any order. Our implementation also splits the watermark so that it can be recovered in any order, but the index value is not required. Prior to embedding the watermark w it is encoded as an integer and split into k pieces $\{w_1, w_2, \ldots, w_k\}$ such that $0 \leq w_i \leq n$. The technique used to split the watermark relies on a 1-1 correspondence between a multiset S of size m (where $S = \{s_i : 0 \leq s_i \leq n\}$) and combinations of size n chosen from $m + n$ elements. Given this correspondence, the splitter enumerates combinations of n chosen from the $m + n$ elements for some fixed n. By using this particular splitting technique the order of the pieces is unimportant.

The k pieces of the watermark are encoded in the opaque predicates in one of two ways: through the use of constants in the predicate or by assigning a rank to each of the opaque predicates in the library. If the opaque predicate is a number-theoretic result, w_i can be encoded:

1. in the constants contained in the predicate, or
2. by inserting new constants in the predicate.

For example, consider encoding the value 42 using the opaquely true predicate $4|x^2(x+1)(x+1)$. This predicate has a constant value of 6 because it contains the constants 4, 1, and 1. Thus the value 36 still needs to be encoded. This is accomplished by multiplying both sides by 18 which produces the opaque predicate $[(18)(4)]|[(18)x^2(x+1)(x+1)]$. This technique does not change the value of the opaque predicate and it permits the encoding of any $n \in \mathbb{N}$. To encode an odd valued watermark select an opaque predicate that already has an odd constant value such as $2|x(x+1)$.

Either technique for encoding the watermark using constants is valid, but using only the constants that are contained in the predicate is restrictive. For example, using the 9 opaque predicates in Table 1, only the values $\{0, 3, 4, 6, 8, 27, 88\}$ can be encoded. The disadvantage of inserting new constants is that it makes the opaque predicate more obvious.

To encode w_i using rank, each of the opaque predicates are assigned a value starting at 0. Using SANDMARK's library the values $\{0, \ldots, 8\}$ can be encoded. While this technique is simple, it does require that the opaque predicate library be a fair size in order to be useful.

4.2. Watermark embedding

The embedding process is dependent on identifying a set B of possible branching points. This set is identified through preprocessing each method in the application. For each $w_i \in w$ an opaque predicate $P^T_{b_j}$ or a call to an opaque method $M^T_{b_j}$ is appended to a selected $b_j \in B$. In an attempt to increase the strength of the algorithm we identify local variables in the method which can be used in the opaque predicate. These variables are identified through the use of a forward slice [24] centered around b_j.

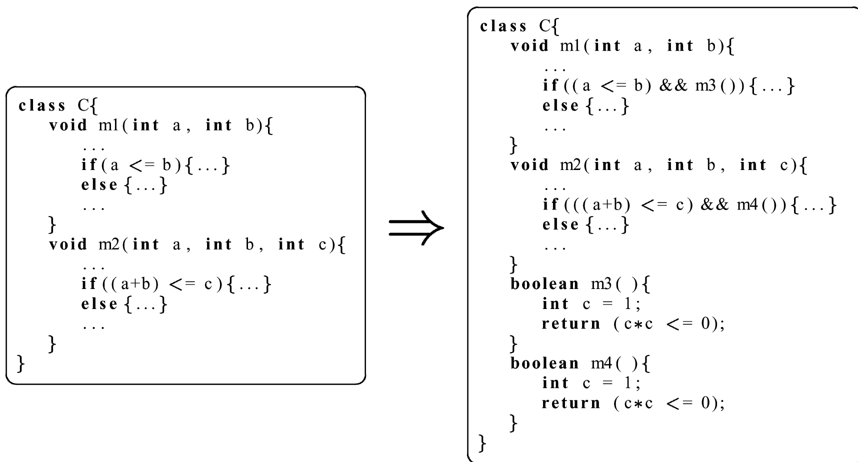


Fig. 1 Transformation without method reuse

The most significant advantage to using live variables in the opaque predicate (as opposed to inserting new variables) is that it aids in disguising the superfluous nature of the predicate. The current disadvantage to this technique is that it is not always possible to identify local variables containing integers around a selected b_j. Thus, some branching points are unusable. This disadvantage will be alleviated as other types of opaque predicates become available.

We were also able to add one more detail to the implementation that not only increases the stealth but decreases the overhead. To embed a watermark using GA2, k new methods are added to the application. This increase in code size could be unacceptable to size sensitive applications such as those on mobile devices. One solution is to encode w_i using rank and reuse the new methods that are added to the application. For example, without method reuse the example class C could be transformed into the class in Figure 1. With method reuse it is transformed into the class in Figure 2. This detail increases the stealth by further disguising the superfluous nature of the opaque method.

Arboit discusses a technique to inhibit the adversary's ability to destroy the watermark using method overloading. If the adversary attempts to modify the types of the overloaded

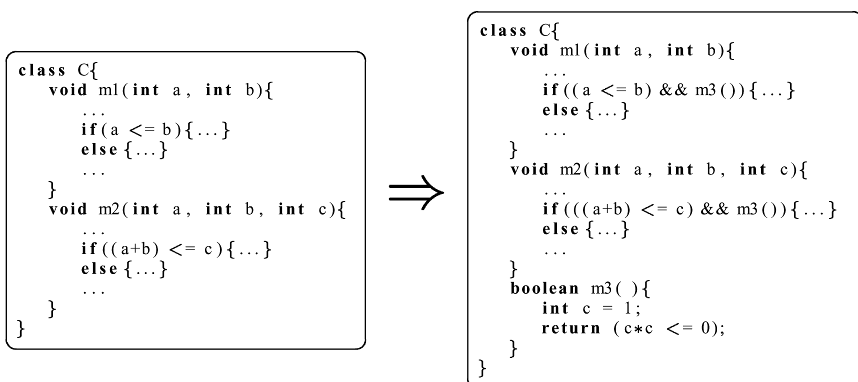


Fig. 2 Transformation with method reuse

method, overriding occurs which could lead to faulty behavior. The current implementation does not support this technique, but we will see in Section 5 that such a technique does not prevent watermark distortion in those instances where GA1 outperforms GA2.

4.3. Watermark recognition

The recognition procedure varies slightly depending on which embedding technique is used. Watermark recovery using GA1 involves an exhaustive search of each method. To identify sets of instructions that may be opaque predicates the basic blocks of the control flow graph (CFG) [4] and expression trees are constructed. Each opaque predicate will end with an `if` instruction which can be found as the last instruction of a basic block. The instructions that comprise the expression tree for that `if` instruction are compared to the entries in the opaque predicate library.

If the watermark was embedded using GA2 then each method is scanned looking for `invoke` instructions which call a method that has the same signature as one of the opaque methods. Currently all opaque methods have a return type of `boolean` and either 1 or 2 parameters of type `int`. In the case when opaque methods are not reused the recognition process could have been simplified to checking the signature of each method. Unfortunately this does not yield the correct number of pieces when methods are reused. Within each opaque method is an opaque predicate that is identified using the same technique as in GA1.

If w_i is encoded using rank, the rank of that particular opaque predicate is identified. If constants are used, the sum of the constants is extracted from the predicate. Once all possible w_i have been identified the values are combined to produce the watermark value.

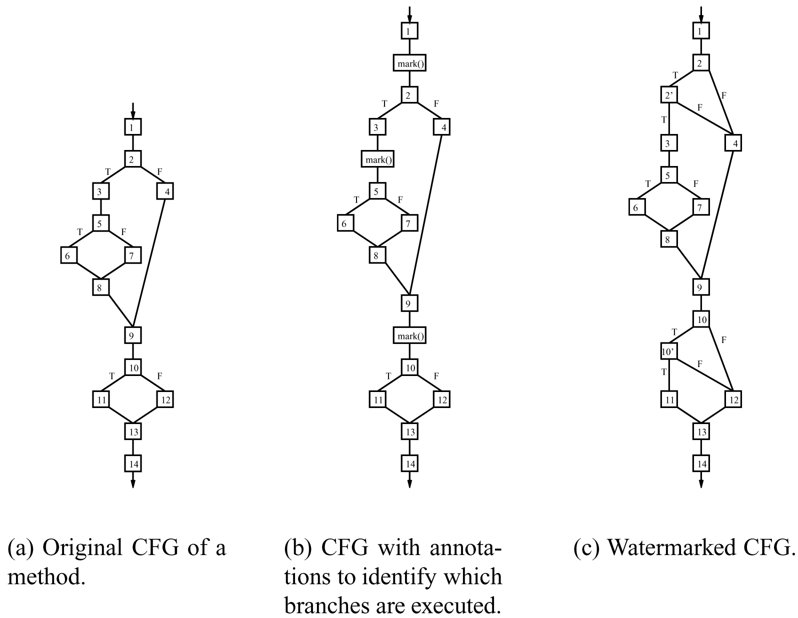
4.4. Dynamic arboit algorithms

One of the yet unanswered questions in the area of software watermarking is whether dynamic algorithms are inherently more resilient to attack than static algorithms. One technique to investigate this idea is to develop, implement, and evaluate a dynamic version of an already known static algorithm. To this end we have developed and implemented dynamic versions of GA1 and GA2 (DGA1 and DGA2 respectively).

Dynamic algorithms make use of a program's execution state to both embed and recognize a watermark. There are three different dynamic techniques: Easter Egg Watermarks, Data Structure Watermarks, and Execution Trace Watermarks [8]. DGA1 and DGA2 are execution trace watermarking algorithms because the watermark is embedded in the trace of the program as it is run with a specific input. This input represents the user's secret key. For example, suppose the application is a Tic-Tac-Toe game. The order in which the X's and O's are placed on the game board becomes the secret key.

The novel aspect of DGA1 and DGA2 is that the execution trace is used to identify the set of program branching points B instead of using randomly selected points. The motivating factor in this design is that the program will execute the original set of branching points when run with the secret key no matter how distorted an attacker makes the application. This assumption is based on the idea that most transformations that cause the execution to skip the branch will most likely alter the functionality of the application. Thus the dynamic nature will improve the algorithm's ability to withstand distortive attacks.

The set B of program branching points is required for both the embedding and recognition phases. B is compiled by annotating the application prior to execution. The annotation phase is fully automated and consists of adding a special function call immediately before each



(a) Original CFG of a method.

(b) CFG with annotations to identify which branches are executed.

(c) Watermarked CFG.

Fig. 3 The watermarking of a method using DGA1 or DGA2 requires an annotation phase which allows us to identify which branch instructions are executed in the trace.

`if` instruction. The function calls represent break points. Each time this function is called during the execution of the application it logs the location of the `if` instruction.

Figures 3(a) and 3(b) illustrate the transformation that occurs due to the annotations. To illustrate the embedding procedure, suppose the execution of the application using the secret input takes the path $\{1, 2, 4, 9, 10, 11, 13, 14\}$. Thus the set B consists of the `if` instructions in blocks 2 and 10. To watermark this method either GA1 or GA2 is used. In this example, the transformation that occurs due to watermarking is illustrated in Figure 3(c).

The recognition set B is again acquired through annotating the watermark application and collecting an execution trace. To continue with the example, the execution trace consists of the blocks $\{1, 2, 4, 9, 10, 10', 11, 13, 14\}$. What we see is that the opaque predicate inserted in block $2'$ is not executed. This is because Java uses short circuit evaluation so the second predicate does not necessarily need to be evaluated. (In the current implementation all inserted predicates are opaquely true.) Since the trace identified block 2 we can still recover the opaque predicate in $2'$. This is accomplished by examining the fall through block of every `if` instruction identified in the trace since it is a possible opaque predicate.

5. Evaluation

In order for a software watermarking technique to be effective against software piracy and copyright infringement it should be resilient against determined attempts at discovery and removal. Very little work has been done on evaluating the strength of software watermarking systems and thus a formal set of properties has yet to be established. Through our study of software watermarking algorithms using the SandMark system we have compiled the following properties which we believe aid in evaluating the strength of an algorithm [8, 13, 20]:

credibility: The recognition process should report a watermark that was embedded and should not report false watermarks.

data-rate: The algorithm should have a high data-rate to permit the embedding of a reasonably sized secret message.

overhead: Embedding a watermark should have little impact on the performance of the application and the embedding/recognition procedure should not be costly.

part protection: In order to protect the watermark it should be distributed throughout the application.

resiliency: The watermark must be resilient against determined attempts at discovery and removal. In particular it should be resilient to three important types of attacks:

- In a *subtractive attack* the attacker attempts to remove the watermark from the disassembled or de-compiled code. Through a manual or automated inspection of the code the attacker may be able to identify and remove a watermark with low transparency without damaging the application.
- In an *additive attack* the attacker adds a new watermark to the already watermarked program in an attempt to cast doubt on which watermark was embedded first.
- In a *distortive attack* a series of semantics-preserving transformations are applied to the software in an attempt to render the watermark unrecoverable but maintain the software's functionality and performance.

stealth: The embedded watermark should be difficult to detect; i.e. it should exhibit the same properties as the code or data around it.

We have evaluated both the static and dynamic versions of the Arboit algorithm within SANDMARK with respect to each of the above properties. SANDMARK includes a variety of tools that an adversary may use to discover and/or remove a watermark. These tools include:

- An obfuscation tool that permits the evaluation of resiliency of the watermark under distortive attacks.
- Additional watermarking algorithms for studying additive attacks (and in the future for comparison purposes).
- A bytecode viewer to display the watermarked bytecode and for manually examining the stealth of the watermark.
- A statistics module that provides static statistics about an application, such as the number of methods, number of conditional statements, etc., which also aids in the evaluation of stealth.

To evaluate the static GA1 and GA2 a set of 11 applications are used which vary in both size and complexity. Two of these 11 applications are also used for the dynamic algorithms: TTT (which is a Tic-Tac-Toe game) and JKeyboard (which allows a user to type using different alphabets). The evaluation of the dynamic algorithms requires applications that make use of user input. This is required so that different execution traces can be obtained. Details of 10 of the applications can be seen in Table 2. The 11th application is specjvm [3].

5.1. Credibility

The credibility of a watermarking algorithm is based on the accuracy of watermark recovery. An algorithm can have poor credibility if it recovers a watermark which was not embedded in the application (a false positive) or not recovering a watermark that was embedded (a false negative). To evaluate the algorithms with respect to this property we ran the recognition

Table 2 Benchmark applications used in the evaluation of GA1 and GA2

Application	Total classes	Total methods	Total size (bytes)
decode	4	20	5728
fft	1	10	3136
illness	16	104	13735
lu	1	7	2744
machineSim	12	110	17751
matrix	2	10	2939
probe	1	7	2699
puzzle	3	20	8627
TTT	12	51	2358
JKeyboard	30	147	32537

algorithms on non-watermarked and obfuscated versions of the benchmark applications. No false negatives or false positives were detected in any of the test cases.

5.2. Data-rate

The data-rate for GA1, GA2, DGA1, and DGA2 will all be roughly the same. This is because the embedding process is based on identifying usable `if` instructions. The only embedding detail which can alter the data-rate is whether the watermark is encoded using constants or rank. When rank is used the watermark must be split into more pieces since the value of each piece is currently restricted to the values 0 through 8. Table 3 shows that by using constants roughly 7 times as many characters can be embedded. The table also shows the total number of `if` instructions found in the benchmark applications. From this it can be seen that there are still many locations for embedding additional characters when the opaque predicate library is expanded.

5.3. Overhead

There are various ways that the overhead property can be applied to evaluate a watermarking algorithm:

1. What effect does the watermark have on the size of the application?
2. What effect does the watermark have on the performance of the application?
3. How costly are the embedding and recognition procedures?

Table 3 Maximum characters embedded when encoding the watermark using constants and rank

Application	Max characters using constants	Max characters using rank	Total `if` instructions
decode	27	3	36
fft	14	2	16
illness	61	5	104
lu	15	2	17
machineSim	64+	5	162
matrix	27	3	36
probe	15	2	17
puzzle	64+	5	154
TTT	25	3	54
JKeyboard	46	4	147

Table 4 CaffeineMark scores before and after embedding a watermark

Category	Original	Watermarked	Slowdown
Sieve	7847	8089	−3.1%
Loop	54292	54248	0.1%
Logic	43656	43831	−0.4%
String	26173	26105	0.3%
Float	24076	24046	0.1%
Method	17077	15013	12.1%
Overall	24178	23788	1.6%

The increase in size depends on the number of watermark pieces and therefore on the size of the watermark. In addition, the encoding technique also has an impact on the overhead. When using constants the value of each w_i can be larger which means the watermark does not need to be split into as many pieces. For each w_i roughly 80 bytes are added to the application. The overhead can be reduced by reusing the methods when w_i is encoded using rank.

The CaffeineMark [22] benchmark shows the effect embedding a watermark has on the execution time of the application. Table 4 shows that embedding a watermark has very little negative impact on execution time.

The embedding and recognition procedures themselves are very efficient. Even the larger applications could be watermarked in seconds. The aspect of the algorithm that is the most costly is the preprocessing of the methods in the case of GA1 or GA2 and the annotation in DGA1 and DGA2. Once the set of branching points is gathered the time required to embed k opaque predicates is negligible.

5.4. Part protection

The idea behind the part protection property is to split the watermark into pieces and spread it across the application. The split watermark has a better chance at survival since it requires that the attack target multiple locations in the application. Both the static and dynamic algorithms incorporate part protection by splitting w into k pieces and randomly distributing those pieces. It was previously mentioned that reusing the opaque methods provided an advantage by decreasing the overhead and increasing the stealth. Unfortunately this technique also decreases the part protection. If the opaque method was used to encode three of the 10 pieces of w removing the method has a higher impact than if only one piece was destroyed.

5.5. Resilience

There are three types of attacks that an adversary could launch in an attempt to destroy a watermark: subtractive, additive, and distortive.

5.5.1. Subtractive attacks

One of the first things that an adversary may do in an attempt to eliminate a watermark is decompile the application. Once the code has been decompiled the attacker can search for aspects of the code that look suspicious such as dummy methods. If the attacker is familiar with simple number theory properties he may realize that the watermark application contains opaque predicates. If they are removed the application will still function normally and the attacker has subverted the protection. This watermarking technique will always be

Table 5 Results from applying other watermarking algorithms to the applications watermarked using GA1, GA2, DGA1, and DGA2. We found that for all test cases the results were the same. A '+' indicates that the original watermark was recovered. A '−' indicates that the original watermark was destroyed

Watermarker	Embedded using GA1	Embedded using GA2
AddMethodField	+	+
GA1	−	−
GA2	−	−
QP	+	+
BogusExpressions	+	+
BogusSwitch	+	+
BogusInitializer	+	+
ConstantString	+	+
HatTrick	+	+
MethodRenamer	+	+
MondenWmark	+	+

susceptible to subtractive attacks but using stronger opaque predicates, such as ones that are not commonly known, will make it harder for the attacker to detect the watermarked sections. In addition, maintaining the secrecy of the opaque predicate library will also improve the resiliency against subtractive attacks.

5.5.2. *Additive attacks*

Additive attacks are used by an adversary when he is either unable to locate the watermarked code or unable to remove the watermarked code. This type of attack is used to cast doubt on the validity of the original watermark or to destroy the original all together. Table 5 shows the results from applying other watermarking algorithms in the SANDMARK system to the test cases that had been watermarked using GA1, GA2, DGA1, DGA2. We found that the original watermark is quite resistant to the application of an additional watermark. However, embedding a watermark using the same algorithm or one of the other GA's destroyed the original watermark. This occurred because the recognition procedure detected additional opaque predicates. In addition we discovered that both watermarks are unrecoverable if we apply GA1 then GA1, GA2 then GA2, or GA2 then GA1. Even though the original was destroyed, the attacker will not be able to embed his own watermark using one of these techniques. The same results occur with DGA1 and DGA2 except that applying DGA2 then DGA1 does not destroy both watermarks.

5.5.3. *Distortive attacks*

Distortive attacks are any semantics preserving code transformation, such as code obfuscation or optimization algorithms. This type of attack is used to distort a watermark such that it is unrecoverable. The advantage of this attack over subtractive attacks is that the adversary need not know the exact location of the watermark. Rather, he can apply the transformation indiscriminately over the application. Through the application of the code obfuscations found in SANDMARK we discovered that GA1 is more resilient than GA2. This discovery contradicts the claim made in [5]. The author claims that GA2 is stronger since it is difficult to alter the signature of a method. The obfuscations `Method 2R Madness`, `Primitive Promoter`, and `PromoteLocals` all modify the signatures of the methods in the application. It is possible that implementing the overloading technique described in [5] would improve the resiliency against `Primitive Promoter` and `PromoteLocals`. In

Table 6 Number of watermarked applications the watermark was recovered from after the stated obfuscation was applied. The evaluation was performed on 11 applications

Obfuscation	Embedded using GA1	Embedded using GA2
AddBogusFields	11	11
AppendBogusCode	8	0
BlockMarker	11	8
BogusPredicates	11	0
BoolSplitter	7	11
Buggy Code	11	9
Class Splitter	11	11
ConstantPool Reorderer	3	3
FalseRefactor	11	11
Inliner	11	11
InstructionOrdering	11	11
IntArraySplitter	11	11
InterleaveMethods	11	0
LocalVariable Reorderer	11	11
Method 2R Madness	0	0
Method Merger	11	1
Name Obfuscator	11	11
NodeSplitter	11	11
OpaqueBranch	11	10
ParamReorder	11	11
Primitive Promoter	3	0
PromoteLocals	0	0
Publicizer	11	11
Rename Locals	11	11
SetFieldsPublic	11	11
Signature Bludgeoner	11	11
Static Method Bodies	11	11
Thread Contention	11	11
VarSplitter	11	11
Variable Reassigner	11	11

addition GA2 is susceptible to attacks which merge methods or alter the body of the methods. The results of applying all obfuscations to the 11 applications are shown in Table 6.

An important assumption made in the study of software watermarking is that the attacker knows the algorithm used to embed the watermark. Based on this assumption the GA algorithms can be easily attacked by simply applying a transformation that inserts an opaque predicate in every boolean expression throughout the application. This attack will thwart recognition and does not require knowledge of the secret key or the opaque predicate library used for embedding. Applying such an attack to the caffeine benchmark yields an overall slowdown of 97.5%. It is possible to decrease the slowdown by inserting the opaque predicate in every other or every few boolean expressions, however this does not guarantee the watermark will be destroyed.

Through application of the obfuscations we discovered that both the static and dynamic algorithms demonstrate basically the same resiliency. The results from testing all four algorithms on TTT and JKeyboard are in Table 7. Based on these results it is not clear that converting a static watermarking algorithm which is already quite resistant to distortive attacks will improve the strength of the algorithm. This however does not indicate that truly

Table 7 Results from applying obfuscations to GA1, GA2, DGA1, DGA2. A '+' indicates the watermark was recovered. A '−' indicates the watermark was destroyed

	TTT				JKeyboard			
Obfuscation	GA1	DGA1	GA2	DGA2	GA1	DGA1	GA2	DGA2
Add Bogus Fields	+	+	+	+	+	+	+	+
Append Bogus Code	+	+	−	−	+	−	−	−
Block Marker	+	+	+	+	+	+	+	+
Bogus Predicates	+	+	−	+	+	+	−	−
Boolean Splitter	+	+	+	+	+	+	+	+
Buggy Code	+	+	++	+	+	+	+	+
Class Splitter	+	+	+	+	+	+	+	+
Constant Pool Reorder	−	+	−	+	+	+	+	+
Degrade	−	−	−	−	−	−	−	−
FalseRefactor	+	+	+	+	+	+	+	+
Inliner	+	+	+	+	+	+	+	+
Instruction Ordering	+	+	+	+	+	+	+	+
Int Array Splitter	+	+	+	+	+	+	+	+
Interleave Methods	+	+	+	−	+	+	−	−
Local Variable Reorder	+	+	+	+	+	+	+	+
Method 2R Madness	−	−	−	−	−	−	−	−
Method Merger	+	+	−	−	+	+	+	+
Name Obfuscator	+	+	+	+	+	+	+	+
NodeSplitter	+	+	+	+	+	+	+	+
OpaqueBranch	+	+	+	+	+	+	+	+
ParamReorder	+	+	+	+	+	+	+	+
Primitive Promoter	−	−	−	−	−	−	−	−
Promote Locals	−	−	−	−	−	−	−	−
Publicizer	+	+	+	+	+	+	+	+
Rename Locals	+	+	+	+	+	+	+	+
SetFieldsPublic	+	+	+	+	+	+	+	+
Signature Bludgeoner	+	+	+	+	+	+	+	+
Static Method Bodies	+	+	+	+	+	+	+	+
Thread Contention	+	+	+	+	+	+	+	+
Var Splitter	+	+	+	+	+	+	+	+
Variable Reassigner	+	+	+	+	+	+	+	+

dynamic algorithms (those which can only exist in dynamic form) are not inherently stronger than static algorithms.

5.6. Stealth

Of the evaluation properties stealth is the most subjective. Currently no technique exists to quantify the meaning of stealth. What we do know is that when considering the stealth of an application it is best to look at the stealth of the watermarked code within the application (i.e. how does the watermarked code compare to non-watermarked code within the same application) and the stealth of the watermarked code with respect to other applications. These measures of stealth are called *local* and *global* stealth respectively.

One technique that can be used to evaluate stealth is to examine how the static statistics of the application change between non-watermarked and watermarked versions of the

Table 8 Static statistics of watermarked and non-watermarked version of TTT and JKeyboard. The watermark value is "wildcat"

Application	Methods	Conditional statements	Vectors	API calls	Methods in scope	Inherited methods
TTT	51	54	120	86	418	0
TTT GA1	51	61	120	86	418	0
TTT GA2	58	68	130	93	418	0
JKeyboard	147	147	554	204	2683	15
JKeyboard GA1	147	154	554	204	2683	15
JKeyboard GA2	154	161	561	211	2685	17

application. While this technique does not provide a quantitative measure it does highlight areas of the watermark application which might be suspicious to an attacker. Table 8 contains some static statistics of watermarked and non-watermarked versions of TTT and JKeyboard. What we can see from these statistics is that applications watermarked using GA1 more closely resemble the original.

6. Summary

Software piracy is an ongoing problem in the software industry. While there are some legal means to handle the problem they do not always target the guilty party. Software watermarking is an additional technique that can be used in the battle. The technique makes proof of authorship or purchase possible and in some cases the source of the illegal distribution can be identified.

In this paper we provided an implementation and evaluation of two techniques proposed in [5]. In addition, we presented a novel extension of the technique to study static versus dynamic watermarking algorithms. Through our analysis we showed that both GA algorithms can be defeated. We also showed that GA1 is a stronger algorithm than GA2. We based these conclusion on six properties. Of these GA1 had a lower overhead, was more resilient to attack, and demonstrated a higher degree of stealth. With respect to the remaining three properties the algorithms were equal. We also showed that the dynamic algorithms are only minimally stronger than the static versions. From this we conclude that it is not clear that converting a known static algorithm will improve the strength. However, this does not indicate that the class of dynamic algorithms is not inherently stronger.

Acknowledgments This work is supported by the NSF under grant CCR-0073483, by the AFRL under contract F33615-02-C-1146 and the GAANN Fellowship.

References

[1] Business software alliance, http://www.bsa.org.
[2] Sandmark. http://www.cs.arizona.edu/sandmark/.
[3] Specjvm98 v1.04. http://www.specbench.org/osg/jvm98/.
[4] Aho, A. V., Sethi, R., & Ullman, J. D. (1988). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.
[5] Arboit, G. (2002). A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research* (ICECR-5).

[6] Collberg, C., Huntwork, A., Carter, E., & Townsend, G. (2004). Graph theoretic software watermarks: Implementation, analysis, and attacks. *In 6th International Information Hiding Workshop*.
[7] Collberg, C., Myles, G., & Huntwork, A. (2003). Sandmark — a tool for software protection research. *IEEE Security and Privacy*, *1*(4), 40–49.
[8] Collberg, C., & Thomborson, C. (1999). Software watermarking: Models and dynamic embeddings. In *Conference Record of POPL '99: The 26th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (Jan. 1999)*.
[9] Collberg, C., Thomborson, C., & Low, D. (1998). Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA.
[10] Christian Collberg, Edward Carter, Saumya Debray, Andrew Huntwork, Cullen Linn, & Mike Stepp. (2004). Dynamic path–based software watermarking. In *SIG–PLAN '04 Conference on Programming Language Design and Implementation*.
[11] Patrick Cousot, & Radhia Cousot (2003). An abstract interpretation–based framework for software watermarking. In *Principles of Programming Languages 2003, POPL '03*, pp. 311–324.
[12] Davidson. R. L., & Myhrvold, N. (1996). Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, Assignee: Microsoft Corporation.
[13] Hachez, G. (2003). *A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards.* PhD thesis, Universite Catholique de Louvain.
[14] Monden, A., Hajimu, I., Matsumoto, K., Katsuro, I., & Torii, K. (1999). Watermarking java programs. In *Proceedings of International Symposium on Future Software Technology*.
[15] Monden, A., Iida, H., Matsumoto, K., Inoue, K., & Torii, K. (2000). A practical method for watermarking java programs. In *compsac2000, 24th Computer Software and Applications Conference*.
[16] Myles, G., & Collberg, C. (2003). Software watermarking through register allocation: Implementation, analysis, and attacks. In *ICISC*'2003 (*International Conference on Information Security and Cryptology*).
[17] Nagra, J., & Thomborson, C. (2004). Threading software watermarks. In *6th International Information Hiding Workshop*.
[18] Nagra, J., Thomborson, C., & Collberg, C. (2002). A functional taxonomy for software watermarking. In Michael J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference* (*ACSC2002*), Conferences in Research and Practice in Information Technology, Melbourne, Australia, ACS.
[19] Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., & Zhang, Y. (2000). Experience with software watermarking. In *Proceedings of ACSAC'00, 16th Annual Computer Security Applications Conference*, pp. 308–316.
[20] Gang Qu, & Miodrag Potkonjak (1999). Hiding signatures in graph coloring solutions. In *International Information Hiding Workshop*, pp. 348–367.
[21] Sahoo, T. R., & Collberg, C. (2004). Software watermarking in the frequency domain: Implementation, analysis, and attacks. Technical Report TR04–07, Department of Computer Science, University of Arizona.
[22] Pendragon Software. Caffeinemark 3.0. (1998). `http://www.pendragon-software.com/pendragon/cm3/`.
[23] Stern, J. P., Hachez, G., Koeune, F., & Quisquater, J. (1999). Robust object watermarking: Application to code. In *International Information Hiding Workshop*, pp. 368–378.
[24] Tip, F. (1995). A survey of program slicing techniques. *Journal of programming languages, 3*, 121–189.
[25] Venkatesan, R., Vazirani, V., & Sinha, S. (2001). A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, Pittsburgh, PA.

Ginger Myles is currently a research scientist at IBM's Almaden Research Center and is finishing her Ph.D. degree in computer science at the University of Arizona. She received a B.A. in mathematics from Beloit College in Beloit, Wisconsin and an M.S. in computer science from the University of Arizona. Her research focuses on all aspects of content protection.

Christian Collberg received his PhD from the Department of Computer Science at the University of Lund, Sweden, after which he was on the faculty at the University of Auckland, New Zealand. He is currently an Associate Professor at the University of Arizona. His primary research area is the protection of software from reverse engineering, tampering, and piracy. In particular, the SandMark tool (sandmark.cs.arizona.edu) developed at the University of Arizona is the premier tool for the study of software protection algorithms.