Software Watermarking in the Frequency Domain: Implementation, Analysis, and Attacks

Tapas Ranjan Sahoo, Christian Collberg
Department of Computer Science,
University of Arizona,
Tucson, AZ 85721, USA,
Email: {collberg,tapas}@cs.arizona.edu

Technical Report TR04-07

March 3, 2004

Abstract

In this paper we analyze the SHKQ software watermarking algorithm, originally due to Stern, Hachez, Koeune and Quisquater. The algorithm has been implemented within the SandMark framework, a system designed to allow effective study of software protection algorithms (such as code obfuscation, software watermarking, and code tamper-proofing) targeting Java bytecode.

The SHKQ algorithm embeds a watermark in a program using a spread spectrum technique. The idea is to spread the watermark over the entire application by modifying instruction frequencies. Spreading the watermark over the code provides a high level of stealth and some manner of resilience against attack.

In this paper we describe the implementation of the SHKQ algorithm, in particular the issues that arise when targeting Java bytecodes. We then present an empirical examination of the robustness of the watermark against a wide variety of attacks. We conclude that SHKQ, while stealthy, is easily attacked by simple distortive transformations.

1 Introduction

Software watermarking is a technique used to dissuade illegal copying and resale of programs. Such *software piracy* is estimated to be a $15 billion per year business. Software watermarking does not prevent piracy, but rather allows the tracing and prosecution of the pirates after the fact. The idea is to embed a copyright notice (a *watermark*) or a customer identification number (a *fingerprint*) into the code or data segments of an application. The watermark asserts the ownership of the program and the fingerprint allows the tracking of intellectual property violators.

In this paper we analyze the SHKQ software watermarking algorithm, originally due to Stern, Hachez, Koeune and Quisquater [1]. The algorithm has been implemented within the SandMark framework, a system designed to allow effective study of software protection algorithms (such as code obfuscation, software watermarking, and code tamper-proofing) targeting Java bytecode.

The SHKQ algorithm embeds a watermark in a program using a spread spectrum technique. The idea is to spread the watermark over the entire application by modifying instruction frequencies.

In their original description of the SHKQ algorithm [1] the authors proposed techniques for applying the algorithm to x86 assembly code. The techniques primarily focussed on code transformations such as semantic preserving code substitution and code reordering for embedding the watermark. Unfortunately, there was no publicly available information regarding the detailed implementation. Also, no attack-model was proposed and no in-depth analysis of the robustness of against various attacks were offered.

This paper makes the following contributions:

1. We describe an implementation of the algorithm for Java bytecode applications.

2. We provide several improvements to the original watermarking scheme. In particular, we present two new approaches for altering the frequencies of selected groups of instruction in the code and propose efficient embedding heuristics. These improvements turn out to be necessary in order to target instruction sets (such as Java bytecode) that are more regular than the x86.

3. We propose detailed and novel evaluation criteria for software watermarking algorithms.

4. We provide an empirical evaluation of the SHKQ algorithm with respect to performance overhead, stealth, and resilience to attack by semantics-preserving transformations.

The remainder of the paper is organized as follows. In Section 2 we will discuss previous related work in this field. Section 3 explains the watermarking algorithm with a brief explanation of the spread spectrum watermarking technique. In Section 4 we describe the SandMark framework for software protection research within which the SHKQ algorithm has been implemented. In Section 5 we evaluate the algorithm empirically with respect to its resilience against attack and its effect on the performance of the watermarked program. Section 6, shows an example of watermarking in a small program code. In Section 7 we summarize our findings.

2 Related Work

Bender et al. [2] describe various techniques for hiding data in media formats such as audio, video, images, graphics, and text documents.

Software watermarking targets application code. There are two broad categories of algorithms, static and dynamic. Static watermarking techniques embed the watermark in the target application executable. In a Unix environment this is typically within the initialized data section (where static strings are stored), the text section (executable code), or the symbol information (debugging information) of the executable. In the case of Java, information could be hidden in any of the many sections of the class file format: constant pool table, method table, line number table, etc. Dynamic watermarking techniques store the watermark in the execution state of the program. The first such technique (proposed by Collberg et. al. [3]) embeds the watermark in a graph structure constructed at execution time. Several implementations exist, for example Palsberg et al.'s [4] JavaWiz, as well as an implementation within the SandMark framework.

Venkatesan et al. [5] describe a *graph-theoretic* approach to watermarking. This is a static watermarking scheme that embeds a watermark graph by extending a function's control-flow graph by a new watermarking sub-graph. Venkatesan's algorithm is an extension of the first published software watermarking algorithm, proposed by Davidson and Myhrvold [6]. Their idea is to embed the watermark by reordering the basic blocks of a control-flow graph. Qu and Potkonjak [7] propose an algorithm that embeds the watermark by altering the register allocation of a procedure.

These algorithms are currently being implemented within the SandMark framework. When implementations are completed it will for the first time be possible to compare the relative effectiveness of the different techniques.

3 The SHKQ Algorithm

This SHKQ algorithm introduced a new approach to software watermarking. Previous algorithms were based on one of two ideas:

1. Watermarks can be embedded by reordering parts of the code [6], where such reordering can be shown to be semantics preserving. For example, data-independent instructions or statements can be reordered, as can basic blocks and switch-statement case-arms.

2. Watermarks can be embedded by inserting new (non-functional) code in the program, such that this code encodes a watermark number. For example, algorithms have been devised that embed a watermark in a method that is never called [8], in control-flow graphs that are never executed [3], or code that builds new graph structures at runtime [5].

Instead, the SHKQ algorithm extracts a vector of instruction-group frequencies from the original program and then modifies this vector to embed the watermark. To modify the vector instruction groups are replaced with different groups which are semantically equivalent but have different statistical properties.

The SHKQ algorithm watermarks code in the *frequency domain*. The main issue is how to place the watermark in the *significant* frequency components of the spectrum. Placing the watermark in the perceptually insignificant regions of the spectrum is more susceptible to detection and tampering. This is true whether the object to be watermarked is media or executable code. Code patterns and program structures which largely digress from normal can be highly susceptible to detection.

Similar reasoning can be applied to software watermarking. Code transformations that embed the watermark are more susceptible to attacks when they are localized to a portion of the target code.

3.1 Watermark Embedding and Extraction Algorithms

Figure 1 shows an overview of our implementation of the SHKQ algorithm. In step Ⓐ we profile a set of benchmark applications to extract a table of frequently occurring code patterns. In step Ⓑ we construct a code book from these patterns. This step is done manually. The code book consists of two kinds of patterns. *Insertion*-patterns give two code sequences (*Embed* and *Nullify*) which increase the occurrence of a particular vector group. They must be inserted such that *Nullify* is on every execution path from *Embed*, since it undoes any semantic change that *Embed* introduces. *Substitution*-patterns map one sequence of instructions to a semantically equivalent sequence, again so that a particular vector group becomes more frequent. The code book is used during watermark insertion and extraction.

In step Ⓒ we extract a vector of code pattern frequencies from the application to be watermarked. In our implementation the application (`app.jar`) is a collection of Java class files. We construct a new vector $\overline{c}$ by adding a watermark vector w to c. In step Ⓓ we embed $\overline{c}$ into `app.jar` by manipulating instruction group frequencies using the transformation patterns in the code book. The result is a watermarked application `app'.jar`. Step Ⓔ, finally, tests for the presence of the watermark in `app'.jar`. We extract a vector d from `app'.jar` and compare $d - c$ to w. If they are similar enough (according to some similarity measure) we output `marked` else `unmarked`.

The three algorithms Vector Extraction, Watermark Embedding, and Watermark Extraction below formalize these steps.

Vector Extraction is the central part of both the embedding and watermark extraction algorithms. The idea is to compute a vector c of frequencies $(c_1, \ldots, c_n)$ where c_i is the frequency by which instruction group i occurs in the code:

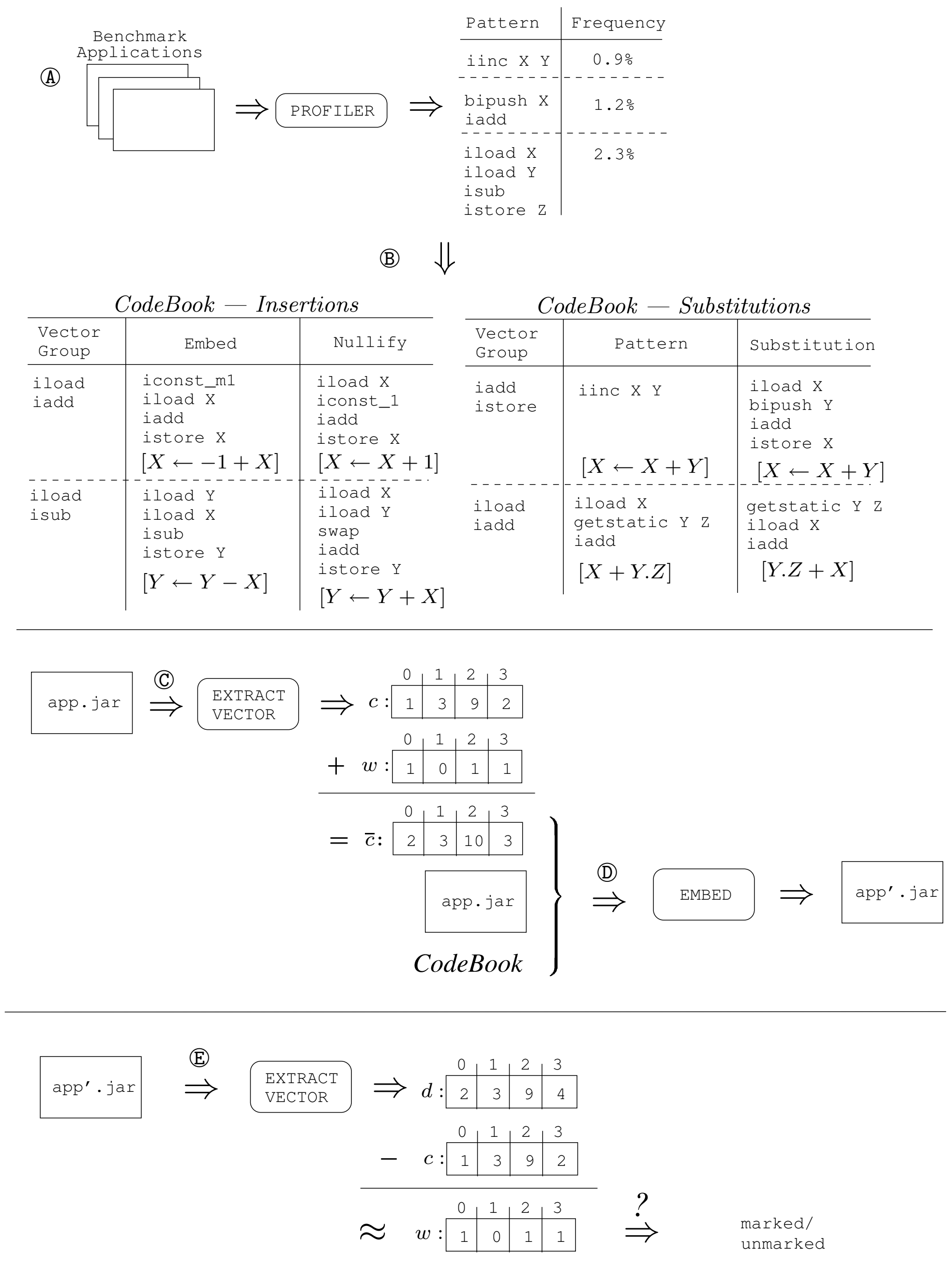
Algorithm 1 (Vector Extraction)

1. Define n as a security parameter.

2. Define a vector $S = (s_1, \ldots, s_n)$ of n ordered groups of machine language instructions.

3. For each group i in S, count the frequency c_i of the group in the code, and form the vector $c = (c_1, \ldots, c_n)$. Return c. □

The Watermark Embedding algorithm modifies the instruction frequencies of the program in order to encode the watermark:



Pattern	Frequency
iinc X Y	0.9%
bipush X iadd	1.2%
iload X iload Y isub istore Z	2.3%

CodeBook — Insertions

Vector Group	Embed	Nullify
iload iadd	iconst_m1 iload X iadd istore X $[X \leftarrow -1 + X]$	iload X iconst_1 iadd istore X $[X \leftarrow X + 1]$
iload isub	iload Y iload X isub istore Y $[Y \leftarrow Y - X]$	iload X iload Y swap iadd istore Y $[Y \leftarrow Y + X]$

CodeBook — Substitutions

Vector Group	Pattern	Substitution
iadd istore	iinc X Y $[X \leftarrow X + Y]$	iload X bipush Y iadd istore X $[X \leftarrow X + Y]$
iload iadd	iload X getstatic Y Z iadd $[X + Y.Z]$	getstatic Y Z iload X iadd $[Y.Z + X]$

Figure 1: An overview of the implementation. Steps Ⓐ and Ⓑ build a code book. This step is one only once, at implementation time. The code book is used during watermark insertion (Ⓓ) and extraction (Ⓔ).

Algorithm 2 (Watermark Embedding)

1. Apply the vector extraction step to obtain a vector c of length n.
2. Choose an n-coordinate vector $w = (w_1, \ldots, w_n)$ whose coefficients are randomly distributed following a normal law with standard deviation α.

3. Modify the code in such a way that the new extracted vector $\overline{c}$ is $c + w$. □

Modifying the frequencies of the instruction groups is done in an iterative manner. A random modification is done to the code while preserving its correctness and the new frequency vector is computed. If the new vector is closer to $\overline{c}$, the process is continued, else the modification is refused.

The WATERMARK EXTRACTION algorithm compares the frequency vectors extracted from the original and watermarked programs. If they differ by approximately w (the watermark vector) we conclude that the watermark is, in fact, present:

ALGORITHM 3 (WATERMARK EXTRACTION)

1. Set a detection threshold σ, $(0 < \sigma < 1)$.
2. Apply the vector extraction step to obtain a vector c from the original (unwatermarked) application.
3. Apply the extraction step to obtain a vector d from the watermarked application.
4. Compute a similarity measure Q between $d - c$ and w.
5. If Q is higher than σ then the algorithm outputs `marked`, else it outputs `unmarked`. □

4 Implementation

Our implementation of the SHKQ watermarking algorithm was done entirely within the SANDMARK environment. SANDMARK is a tool developed for research into software watermarking, tamper-proofing, and code obfuscation of Java bytecode. The ultimate goal of the project is to implement and study the effectiveness of all known software protection algorithms. Currently, the tool incorporates several dynamic and static watermarking algorithms, a large collection of obfuscation algorithms, a code optimizer, and tools for viewing and analyzing Java bytecode. SANDMARK is designed to be simple to use. A graphical user interface allows novices to easily try out watermarking and obfuscation algorithms. Algorithms can be combined, the resulting watermarked and/or obfuscated code can be examined, and attacks can be easily launched. SANDMARK has been designed using a plugin-style architecture which makes it easy to extend with additional algorithms.

As shown in Figure 1 the SHKQ algorithms for watermark embedding and extraction make use of a *code book*. The code book is assumed to be kept secret from an attacker. It contains information on which code insertions and substitutions are legal and how they affect the instruction group frequencies.

The code transformations in the code book are constructed based on what are frequently occurring code sequences in real programs. To collect this information a *profiler* was built. The profiler collects statistics from different applications and identifies specific groups of instructions which have more likelihood of occurrence in real code. These groups of instructions form the components of the watermark vector which is to be embedded. We will refer to these instruction groups as *vector instruction* groups.

Once a code book has been constructed one or more vector instruction embedding schemes have to be designed. Having a variety of schemes available increases the likelihood of a particular embedding being successful. Stern [1] introduces the idea of *code substitution* where the frequency of a vector group is increased by replacing one group of instructions with a semantically equivalent one. We have identified and implemented two additional approaches. The first inserts two sequences of instructions where the second one nullifies the effects of the first. The second one uses method overloading to insert code into new methods which will never be called.

4.1 Profiler

The profiler extracts instruction groups which are more likely to occur in general Java code. Since the frequency patterns of Java bytecode instructions can be closely related to the type of application we ran our profiler over different applications. In our case, we selected a few large applications such as specJVM benchmark, JFIG and SandMark . We were only interested in small groups of instructions, specifically instruction groups of size less than 4. Hence we collected the code statistics of instruction groups limiting the group size to between 1 to 4. Since we are only interested in the opcodes of the instructions we abstracted away the operands while calculating the frequencies of the groups. The instruction-groups obtained form potential candidates for watermark vector components.

4.2 Building the code book

The code book defines a set of vector instruction groups as well as a set of mappings between these groups necessary for the various embedding schemes. The vector instruction groups are obtained by profiling a range of different applications. From the instruction profiles we select frequently occurring instruction groups which have "scope for substitution." In other words, they can be transformed to manipulate the frequency of instruction groups.

To illustrate these points we will next consider a simple example of a substitution transformation. Assume the existence of the following vector instruction group:

```
bipush
iadd
```

`bipush` X pushes literal integer X onto the evaluation stack. `iadd` pops two integers from the top of the stack, adds them together, and pushes the result. If an instruction `iinc` X Y (which increments local variable X by literal integer Y) is found in the code it can be replaced with discrete instructions that compute $X \leftarrow X + Y$. The code book could contain the following substitution pattern:

```
iinc X Y          iload X
                  bipush Y
           ==>    iadd
                  istore X
```

$[X \leftarrow X + Y]$ $\Longrightarrow$ $[X \leftarrow X + Y]$

After a substitution has taken place we have increased the frequency of the vector instruction group `[bipush, add]` by one.

After profiling a large number of applications we discovered that there are many frequently occurring instruction sequences for which we cannot construct substitution patterns. For example, the `new` bytecode (which allocates new heap objects) is very common in all Java programs. However, there are no other bytecode sequences that perform the same actions as `new`. Thus `new` cannot be part of any substitution pattern.

For this reason our implementation of the SHKQ algorithm includes *insertion patterns* in addition to the substitution patterns first discussed by Stern [1]. Insertion patterns are used when substitution alone is not enough to sufficiently manipulate instruction group frequencies.

For example, to increase the frequency of the instruction group `[bipush, add]` we could insert the following code:

```
iload X
bipush -1
iadd
istore X
```

$[X \leftarrow X + (-1)]$

However, this is not a semantics-preserving transformation since the value of X is altered. We can nullify this effect by embedding a nullifying instruction sequence such as:

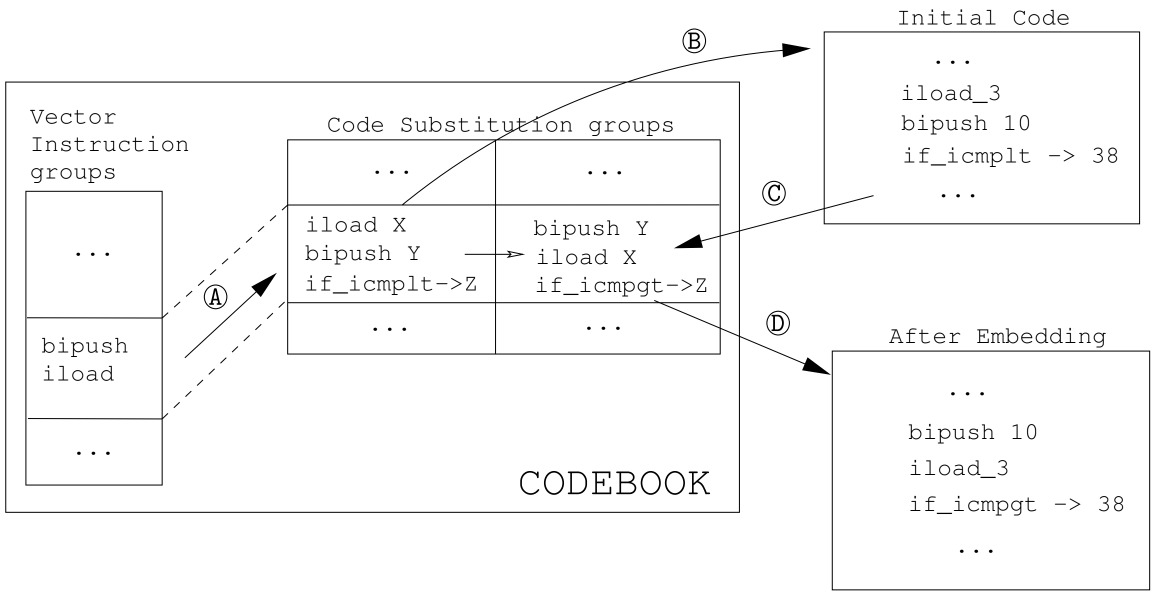


Figure 2: The code substitution procedure.

```
iload X
iconst_1
iadd
istore X
```

$[X \leftarrow X + 1]$

We will next discuss these methods in detail.

4.3 Embedding by Code Substitution

Substitution of semantically equivalent groups of instructions increments the frequency of a vector instruction group, while increasing the code size minimally. The details are given in Algorithm 4 below.

Algorithm 4 (Code Substitution)

1. Given a vector group V whose frequency needs to be increased, search the code book for a substitution group $S : P \to R$ mapped to V. P is the pattern and R its replacement.
2. Search the application for a code sequence T that matches P.
3. Replace variables in R with actual instruction operands extracted from T.
4. Replace T by R. □

Figure 2 illustrates the group substitution operation. At Ⓐ, a substitution group corresponding to the vector group `[bipush; iload]` is selected. At Ⓑ, we search the target code for a matching opcode pattern. At Ⓒ, the actual parameters $\langle X = 3, Y = 10, Z = 38 \rangle$ are extracted from the target code. At Ⓓ, finally, we replace the target code with a new instruction group formed by filling in the actual parameters in the mapped substitution group.

Code substitution is our default embedding scheme. Only when there is no further scope for substitution do we revert to other embedding techniques. These will be described next.

4.4 Embedding by Instruction Insertion

Each vector group has an embed instruction group associated with it in the code book which encapsulates the vector group. This instruction group is inserted into a randomly selected method. The instructions in these "embed" groups make maximum use of the existing local variables in the method to keep the code size to a minimum by eliminating the need for variable initialization. Additional nullifying instructions have to be inserted to cancel out the effect of the inserted code, assuming that the affected variables have a next use in the method. At a first glance it might appear that these nullifying codes create additional increase to code size, but in practice, the code book has been designed such that the nullifying codes create further scope of substitution in the same or other vector groups. For instance, assume a vector group

```
iadd
istore
```

together with instruction embedding and nullifying groups

```
iload X                    iload X
bipush -1                  iconst_1
iadd           and         iadd
istore X                   istore X
```

$[X \leftarrow X + (-1)]$ and $[X \leftarrow X + 1]$

The nullifying group can assist in further substitution of a vector group, such as

```
iconst
isub
```

which has a substitution group

```
iload X              iload X
iconst_1    ==>      iconst_m1
iadd                 isub
```

$[X + 1]$ $\Longrightarrow$ $[X - (-1)]$

It should be clear that the embed and nullify instruction groups cannot be inserted in random places in the code. Java bytecode has very strict structural and semantic requirements that our inserted code must not violate. We must, for example, ensure that regardless of the execution path taken to reach a particular point p in the bytecode the elements on the execution stack have the same types. We use a conservative estimate of the points in the bytecode that are safe for insertions, namely the points in the code that have stack height 0. Java also requires a local variable to be initialized before use. We must therefore ensure that code that uses a variable x is only inserted at points where x is defined, i.e. points dominated by definitions of x.

An interesting case arises when the embedding instruction group contains a branch instruction. In such a case, the branch target is set to the point where the nullifying instruction group is embedded. The corresponding nullify group uses the same branch condition to return back to the point of the embed instruction. Hence the correct flow of control of the program is maintained. Consider, for example, the following vector group

```
iconst
if*
```

(where the `if*` pattern matches any branch instruction) with the embed and nullify patterns below:

```
A: iload X                        goto --> Z
   iconst_0                    B: iload X
   if_icmplt --> B      and       iflt --> C
C:                             Z:
```

[**if** $X < 0$ **goto** B] and [**goto** Z ; **if** $X < 0$ **goto** A]

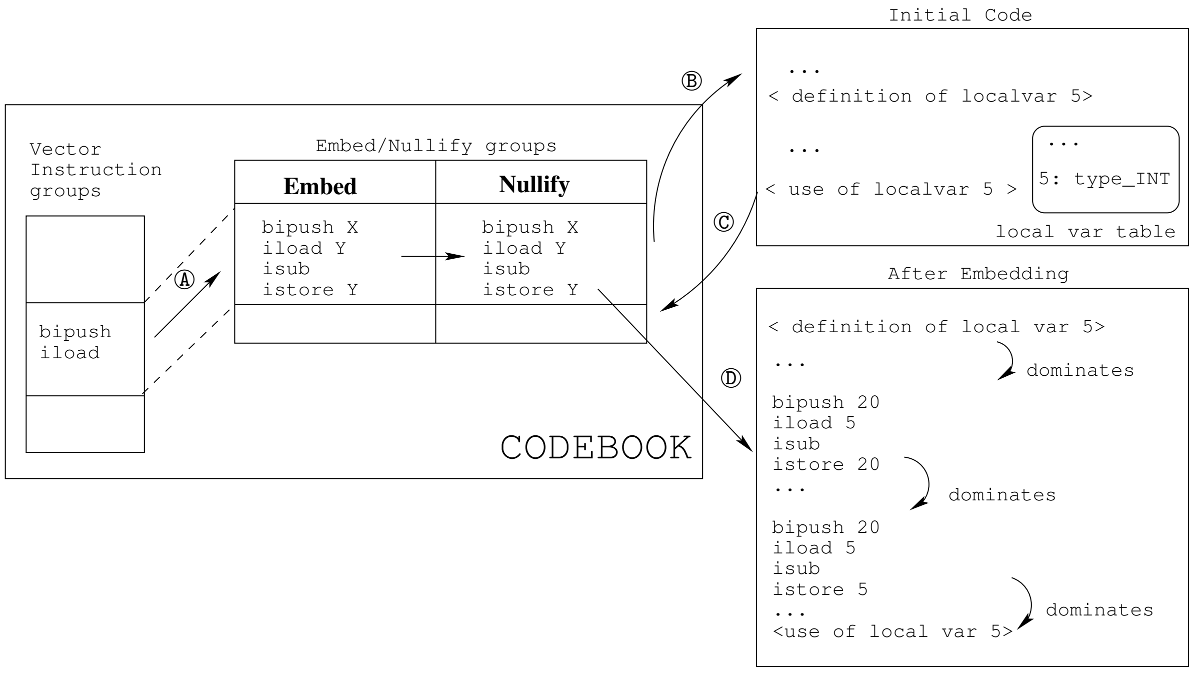


Figure 3: The instruction insertion procedure.

If $X < 0$ when control reaches the beginning of the embed group at point `A`, we will branch to point `B` in the nullify group. We will then test $X < 0$ again and flow back to the end of the embed group at point `C`. Thus, inserting these embed and nullify groups will have no effect on the semantics of the program.

Algorithm 5 summarizes the insertion technique.

Algorithm 5 (Instruction Insertion)

1. Given a vector group V whose frequency needs to be increased, search the code book for an insertion group $S : (E, N)$. E is the embed group and N the nullify group.
2. Identify a method M containing V.
3. For every free variable I in E and N, locate local variables of the appropriate type in M. If no such variables exist, create new ones, inserting the appropriate initialization code.
4. Select an embed group insertion point such that all initializations of the used variables dominate this point.
5. Select a nullify group insertion point such that all initializations of the used variables as well as the embed group insertion point dominate this point.
6. If no safe insertion points are found repeat the process on a different method. Otherwise, replace free variables in E and N with actual operands and insert them at the chosen insert points. $\square$

Figure 3 illustrates the instruction insertion operation. At Ⓐ, we select an embed group E and a nullify group N corresponding to the vector group `[bipush; iload]`. E and N have two free operands, an integer literal X and an integer local variable Y. At Ⓑ, the application is searched for a suitable location for insertion. At Ⓒ, the literal 20 is selected for X and local variable 5 is selected for Y. Finally, at point Ⓓ, two points are selected for insertion of E and N based on dominator analysis of variable 5.

4.5 Embedding by Method Overloading

In Java, two methods are overloaded if they are declared in the same class and have the same name but different parameter lists. This is a frequently occurring phenomenon in real programs. In cases when neither embedding by code substitution nor instruction insertion is sufficient, we add new overloaded methods to the application to increase the frequency of a particular vector group. Algorithm 6 describes how this embedding takes place.

Algorithm 6 (Method Overload Embedding)

1. Identify a method M containing the vector group V.
2. Make a copy M' of M and add it to the application.
3. Modify the parameter list of M' to be different from M and from any other method in the class. This can be done by either adding or deleting a parameter:
   - If a formal parameter F is *deleted*, update the body of M' either by deleting all instructions referencing F or by adding a new local variable of the appropriate type and with appropriate initialization code to substitute for F.
   - If a formal parameter F is *added* to M' update the body of M' by inserting instructions that use F. In particular, we can insert additional vector groups using F further increasing the vector group frequency.
4. Add bogus calls to M' to ensure that ambitious optimizers will not remove it:
   (a) Locate a method N in any class that can access M'.
   (b) Insert an invocation of M' in N. The call is protected by an *opaque predicate* to prevent M' from actually being called.
   (c) Add appropriate actual parameters to the call. Any available local variables or literal values can be used as actuals.

   The call may thus look like this:

```
void N() {
   int a = ···;
   float b = ···;
         ...
   if (P^F)
      M'(a, b, 10);
         ...
}
```

   P^F is an opaquely false predicate — a boolean expression which always evaluates to false but which is constructed in such a way that evaluation is difficult for an adversary. Collberg [9] describes the design of such predicates. SandMark contains a library of opaque predicate construction routines. □

Care has to be taken when choosing the method to overload. Ideally, we will find a method that contains several vector groups such that when a new copy of the method is made we increase the frequency of all of them. At the same time, we must ensure that the new copy will not increase the frequency of any one vector group beyond what is required. We must also be careful to set a reasonable upper bound on the size of the chosen method to avoid excessive code bloat.

4.6 Embedding Heuristic

Algorithm 7 outlines the heuristic used by the embedder. The main complication is that a code substitution or insertion can affect more than one vector group. For example, assume that the frequencies of vector groups A and B both need to be increased. If we choose a code substitution that increases the frequency of A but, as a side effect, *decreases* the frequency of B then we are not moving towards a successful embedding. Therefore, before we commit to use a particular transformation we must first verify that the transformation actually converges towards the final vector value.

To simplify the heuristic, we shifted the complexity onto the code book. Each substitution group in the code book is mapped to exactly one vector instruction group. Similar restrictions are applied on the embed/nullify groups. This implies that a substitution or an embedding by instruction insertion will not have any side-effects of changing multiple vector group frequencies.

In the case of method overloading, we always verify that the frequency updates to the vector groups in the newly created method do not exceed the required updates within a certain threshold before committing to overload the method.

Since both these approaches are directed hill-climbing approaches i.e. at each step we move closer towards the final result, they guarantee that the algorithm converges. Both code substitution and instruction embedding approaches precisely increment the instruction group frequencies. For method overloading approach, by selecting a desired threshold, we can guarantee that the final increment of the frequencies of the instruction groups done is within a preselected threshold distance from the actual frequency increment of the instruction groups targeted.

Hachez [10] followed a random hill-climbing approach. This approach is good in scenarios where embedding time is not a constraint. One reason that we can attribute to the different embedding approaches is the nature of our respective code book . Our scheme takes up at most one vector component for each code modification(substitution or new instruction insertion). So their are no side-effects of embedding in these two cases. Our only side-effects come from the method overloading scheme, which we restrict within a limit by having a threshold factor. In contrast, their scheme intends to change multiple vector components at each step of modification. So the side-effects always exists. So it seems reasonable to follow the random hill-climbing approach, whereas we revert to the directed approach.

However, following a random update heuristic creates a non-deterministic scenario where the vector frequency updates might not converge to the targeted updates, and even if they do, the convergence may be time-consuming. Consequently, both the performance and reliability of the embedding process degrades.

Algorithm 7 (Embedding Heuristic)

1. If all vector component frequencies have reached their desired level then return *success*.
2. If too many embedding attempts have failed, return *failure*.
3. Select the next vector component V whose frequency needs to be increased.
4.
   - Attempt a code substitution for V, according to Algorithm 4.
   - If substitution fails, attempt an instruction embedding for V, according to Algorithm 5.
   - If instruction embedding fails, attempt a method overload embedding for V, according to Algorithm 6. Add false invocations of the new method to the application.
5. Repeat from 1. □

4.7 Recognition Procedure

Software watermarking protocols can be classified along three 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, however, 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 true/false depending on whether the mark is present or not. In a fingerprinting algorithm the recognizer returns the mark itself (a number) which can be different for every 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 the the unwatermarked program and/or the watermark itself.

As described in Stern [1], the SHKQ algorithm is a static, informed, watermarking algorithm. Thus, the recognizer is provided with the watermark, the watermarked code, as well as the original code of the application and answers with a true/false whether the given watermark is present in the target code or not. A drawback of the similarity measure they proposed was that it failed to relate the recognition process with the actual mechanism of testing a presence of a transmitted signal within a received signal. In such scenarios, some sort of correlation measure is required to carry out the recognition process.

Hachez [10] followed a slightly different approach for recognition. During embedding, he marks the methods in which the watermark is present by a key. This key is basically formed by information such as the method signature, number of exceptions, local variable table size, maximum stack size, etc. The recognizer looks at these methods to retrieve the watermark. This makes it possible to embed more than one watermark in the same code by identifying disjoint set of methods to embed them. This is not possible in our scheme, since we run our recognizer over the entire code. Another advantage of his implementation scheme is that additive attacks such as adding a new method will most likely not disrupt the watermark, since the likelihood of the method being inspected is less. One major disadvantage of his approach is that the key components used to identify the watermarked methods can get altered when subjected to different method level obfuscations. So some watermarked methods might go unchecked. The additive attack mentioned above can also disrupt this approach sometimes. The new method may have identical features to an existing watermarked method, in case of which, this method will also become a part of the recognition. He mentioned another approach, in which the method can be partitioned into x regions and different watermarks can be embedding in different regions. This approach also seem to be susceptible to similar method obfuscations. The authors however did not mention much about the false positive/negative ratios of this scheme.

In our implementation the recognizer uses the code book to run the vector extraction procedure to retrieve the vector frequencies from the original code and the target code. It then calculates the normalized linear correlation vector to get the difference vector and the given watermark.

The linear correlation between two vectors, v and w_r, is the average product of their elements:

$$z_{lc} = \frac{1}{N} \sum v_i w_r[i].$$

where, v_i and $w_r[i]$ denote the ith vector components in their respective vectors, and N denotes the number of vector components in each vector.

Cox [11] describes a method to test for the presence of a transmitted signal, w_r in a received signal, v, by computing $z_{lc}(v, w_r)$ and comparing it to a threshold. This practice is referred to as match filtering. One problem with this approach is that the detection values are highly dependent on the magnitude of the vectors extracted from the target application. This means that the watermark will not be robust against modifications. The problem is solved by normalizing the extracted vector and the reference vector to unit magnitude before computing the inner product between them. That is,

$$\tilde{v} = \frac{v}{|v|}$$

$$\begin{aligned}\tilde{w_r} &= \frac{w_r}{|w_r|} \\ z_{nc} &= \frac{1}{N}\sum \tilde{v_i}\tilde{w_r}[i].\end{aligned}$$

We make use of this normalized correlation to detect the watermark. Under ideal condition, the normalized correlation will give a value of 1. We make the assumption that a correlation ratio greater than 0.9 signifies watermark detected. Any ratio less than 0.6 implies watermark is not present, or has been destroyed. The range between 0.6-0.9 is a probabilistic area where we are not completely sure whether the watermark is present or absent.

5 Experimental Evaluation

To date most software watermarking research has focused on the discovery of novel embedding schemes. Unfortunately, very little work has been done on the evaluation of these techniques. One of the goals of the SandMark project is to implement every proposed software watermarking algorithm and to develop procedures for how these algorithms should be evaluated.

It is our belief that a software watermarking algorithm should be evaluated according to the following criteria:

data rate: What is the ratio of the number of bits of watermark that can be embedded in the target code to the size of the code?

embedding overhead: How much slower/larger is the watermarked application compared to the original?

resilience against manual attacks (stealth): Does the watermarked program have statistical properties that are different from typical programs? Can an adversary use these differences to locate and attack the watermark?

resilience against semantics-preserving transformations: Will the watermark survive transformations such as code optimization and code obfuscation? If not, what is the overhead of these transformations? In other words, how much slower/larger is the application after enough transformations have been applied that the watermark no longer can be recognized?

resilience against collusive attacks: Given two or more differently fingerprinted copies of the same application can the location of the fingerprints be determined?

false positive rate: Given a random value to the watermark recognizer what is the probability that it is recognized as a valid watermark?

Our evaluation of the SHKQ algorithm shows the following:

- For typed and architecture neutral instruction sets such as Java bytecode the data rate of the SHKQ algorithm is low, on the order of 8 bits (Section 5.1).
- The cost of embedding is typically low. Moderately sized Java programs increase on the order of 5% in size. For larger programs the increase can be as low as 1%. We typically see less than a 10% increase in execution time (Section 5.2).
- The algorithm is stealthy. Our statistical measures show very minor changes between watermarked and un-watermarked programs (Section 5.3).
- The watermarks are very easily destroyed by semantics-preserving transformations. Very simple obfuscating transformations will alter the program enough that the watermark can no longer be recognized. The overhead of these transformations is also low, typically less than 10% increase in size and less than 4% increase in speed (Section 5.4).

- SandMark has a tool to carry out collusive attacks. This tool compares Java bytecode applications for similarities. Such manual attack can sometimes be successful in detecting the code which forms the watermark. We discuss this further in Section 5.5.
- The false positive rate depends on the watermark vector length as well as the range of values taken by the actual watermark vector components. With random vector values (but with length equal to the length of the watermark vector currently implemented in our code book), the correlation ratio lies mostly in the 0.6-0.9 range. In few cases the correlation did exceed the 0.9 threshold, that is, gave false detection. (Section 5.6).

5.1 Data Rate

The data rate of this watermark essentially depends on the number of vector instruction groups defined in the code book. Each vector instruction group encodes certain number of bits of watermark. Hence, the total number of bits that can be encoded depends on the number of vector instruction groups.

Small applications may not provide enough scope to embed all the bits due to the various constraints during the embedding procedure (for instance, method size, availability of local variables, etc). As the application size increases, we may be able to embed more bits. So the data rate increases gradually. But once we are able to embed all the bits in an application, further increasing the application size does not increase the bit rate any further, since the number of bits embedded remains constant after that.

5.2 Embedding Overhead

Code substitutions have a minimal impact on the size and performance of the watermarked application. The reason is that for most substitutions the original and the substituted code segments are of the same or similar lengths. For this reason, it is important to try to construct as many code substitution patterns as possible.

However, it is not always possible to create substitution patterns from the higher frequency instruction groups returned by the profiler. The reason is that for a vector group V we need to find two groups of semantically equivalent instructions A and B such that B encloses V while A does not, and, moreover, A or B do not enclose any other vector group. On complex instruction set architectures (such as the X86 targeted in the first implementation of the SHKQ algorithm [1]) this may be fairly easy since there are many possible code sequences with the same semantics. On a RISC machine or a virtual machine such as the JVM this is typically much harder since there are fewer instructions and addressing modes available.

For this reason it becomes important to construct additional methods to increase the frequencies of vector groups, such as the new instruction embedding and method overloading schemes introduced in this paper.

Both instruction embedding and method overloading increase code size but our studies have shown that the effect on execution time is minimal. The effect of the embedding on execution time is also minimal. Table 2 summarizes the effect of embedding a watermark formed by 8 vector instruction groups in a few Java applications. The column *VecGroups* shows the actual number of vector instruction groups embedded out of the maximum 32. Note that running the substitution process alone does not embed the entire watermark in the application because of absence of enough scope in the code for substitution. As expected, the percentage of code size increase is minimal for code substitution when compared to the new instruction embedding scheme. Especially for smaller applications, this difference between percentage of code size increase for the two embedding schemes is large.

5.3 Embedding Stealth

A typical software application can be very large, often millions of lines of code. For this reason an initial step in any attack against software is to attempt to isolate code-segments that are more likely than others to contain security-sensitive code. This classification can be based on the location of the code, the order

Watermark frequency vector: 44444444

		Substitution		**Embed/Nullify**	
Application	CodeSize	**VecGroups**	**% CodeIncr**	**VecGroups**	**% CodeIncr**
boyer	18076	12	1.5%	32	6.06%
fft	3332	15	0.65%	32	31.94%
lexgen	41469	16	1.1%	32	31.88%
lu	2938	17	1.0%	32	45.12%
nucleic	46795	16	0.84%	32	2.65%
probe	2924	12	0.80%	32	38.97%
TTT	10801	20	2.01%	32	17.24%
Sandmark	3108331	28	0.01%	32	0.04%
JFIG	1474190	25	0.08%	32	0.30%
specJVM	1686249	28	0.18%	32	0.35%

CodeSize	:	Size of target code (in bytes).
VecGroups	:	Number of groups of instructions embedded (out of maximum required 32).
CodeIncr	:	Percentage of code size increase due to transformation.

Table 1: Effect of individual embedding schemes on code size.

in which it is being executed (code executed early on is more likely to contain security checks), whether it contains unusual code sequences, etc.

It is therefore essential that the code that a watermarking algorithm inserts into an application is *stealthy*, that it is such that it does not arouse suspicion. For example, an algorithm that embeds the watermark in the number of `xor` instructions a program has would likely be un-stealthy since most real programs contain very few, if any, `xor`s.

An inserted watermark needs to be stealthy in two ways:

1. the inserted code must be similar to the code that surrounds it;

2. the inserted code must be similar to code that occurs in typical applications.

The first condition makes it difficult for the attacker to select some particular methods in which there is more chance of occurrence of the watermark. The later condition ensures that the watermarked application, in its entirety, is less likely to throw suspicion of a presence of watermark. Both the conditions makes its difficult to carry out manual as well as automated attacks in order to determine the presence and the specific location(s) of the watermark in the code.

To measure the stealth of a watermarked method we need to compare it to a *universe* of benchmark methods. We have collected a large set of Java programs including the specJVM benchmarks, SandMark itself and a large number of programs downloaded from various software repositories. Also needed is a set of similarity measures to allow us to do the actual comparisons. SandMark contains a number of *Software Complexity Metrics* [12, 13, 14, 15, 16] that allows us to measure many structural aspects of programs.

Algorithm 8 — `Normalcy`(B, X, p) — computes the stealth of a method p with respect to a universe of benchmark methods B and a set of metrics X. The basic idea is to compute a table $\mathcal{W}$ where $\mathcal{W}_{b,m}$ is the value of software complexity metric m on method b. We then compute a *clustering* $\mathcal{C}$ on the rows of $\mathcal{W}$ such that two rows are in the same cluster if their corresponding methods are "similar" with respect to the given set of metrics. `Normalcy`(B, X, p) builds the clustering table $\mathcal{C}$ and returns the size of the cluster in which p falls divided by the size of the benchmark universe. For example, given benchmark methods $B = [b_1, \cdots, b_6]$

Watermark frequency vector: 57452648

Application	Embed	Size		Execution Time		SUB	INS	OVD
	(s)	(b)	Change (%)	(s)	Change (%)			
boyer	22.7	18076	4.4	2.52	6.3	13	28	0
fft	3.4	3332	19.6	0.28	0.0	15	22	2
lexgen	28.8	41469	5.9	0.99	10.1	22	19	0
lu	3.4	2938	22.2	4.71	-0.2	17	24	0
nucleic	35.3	46795	3.6	0.61	-6.6	15	26	0
probe	3.4	2924	22.4	10.93	0.9	17	21	1
TTT	5.7	10801	7.3	-	-	24	17	0
Sandmark	161.9	3108331	0.1	-	-	37	4	0
JFIG	215.9	1474190	0.2	-	-	33	8	0
specJVM	64.7	1686249	0.2	-	-	37	4	0

SUB	:	Number of code substitution transformations.
INS	:	Number of embed/nullify groups insertion transformations.
OVD	:	Number of method overloading transformations.

Table 2: Effect of watermark embedding on code size and execution time. Sizes are in bytes, times in seconds. **Embed Time** shows the amount of time required to embed the program when running on a Linux machine operating on 1GHz Intel Pentium III processor and 256MB physical memory.

---

and metrics $X = [m_1, m_2, m_3]$ we compute a table $\mathcal{W}$ which could then be clustered into three clusters $\mathcal{C} = [C_1, C_2, C_3]$:

$$\begin{array}{c|c|c|c} \mathcal{W} & m_1 & m_2 & m_3 \\ \hline b_1 & & & \\ \hline b_2 & & & \\ \hline b_3 & & & \\ \hline b_4 & & & \\ \hline b_5 & & & \\ \hline b_6 & & & \end{array} \quad \Rightarrow \quad \begin{array}{cc|c|c|c} \mathcal{C} & & m_1 & m_2 & m_3 \\ \hline C_1 & b_3 & & & \\ \cline{2-5} & b_5 & & & \\ \cline{2-5} & b_6 & & & \\ \hline C_2 & b_4 & & & \\ \cline{2-5} & b_1 & & & \\ \hline C_3 & b_2 & & & \end{array}$$

If method p falls in cluster C_2, `Normalcy`(B, X, p) would return $|C_2|/|B| = 2/6 = \frac{1}{3}$.

Algorithm 8 (Normalcy(B, X, p))

- `Normalcy` takes three arguments, B (a universe of benchmark methods), X (a set of software complexity metrics), and p (a method). It returns a value which expresses the similarity of p to the methods in B with respect to the metrics in X.
- We compute a matrix $\mathcal{W}$ where $\mathcal{W}_{m,x}$ is the value of metric x on method m.
- From $\mathcal{W}$ we compute $\mathcal{C} = [c_1, \cdots, c_n]$, a *clustering* of $\mathcal{W}$. The i:th cluster of $\mathcal{C}$, c_i, contains the rows of $\mathcal{W}$ which are *similar* according to some similarity criterion. Thus, c_i represents the set of methods in B which are similar with respect to the given set of software complexity metrics.

```
for $b \leftarrow$ all benchmark methods in universe $B$ do
    for $x \leftarrow$ all software complexity metrics in $X$ do
        $\mathcal{W}_{b,x} \leftarrow$ compute metric $x$ on method $b$;
$\mathcal{C} \leftarrow$ a clustering of the rows of $\mathcal{W}$;
for $x \leftarrow$ all software complexity metrics in $X$ do
    $v_x \leftarrow$ compute metric $x$ on method $p$;
$d \leftarrow$ the cluster of $\mathcal{C}$ in which $v$ falls;
return $|d|/|B|$;
```

Given the `Normalcy` algorithm we can construct a family of stealth measurements. We define *Global Stealth* of a watermarked program P as the average stealth of the methods of P with respect to all known benchmark methods:

$$\frac{1}{|P|} \sum_{m \in P' smethods} \texttt{Normalcy}(\textit{all benchmark methods}, \textit{all metrics}, m)$$

An attacker would measure the global stealth of the methods of a watermarked program in order to determine if any methods have features which are unusual compared to ordinary methods. Such features might indicate that a method has been mechanically altered during the watermarking process.

We define the *Local Stealth* of P as the average stealth of P's watermarked methods as compared to all the methods of P. Letting M be the set of methods of P that contain parts of the watermark, we get:

$$\frac{1}{|M|} \sum_{m \in M} \texttt{Normalcy}(\textit{methods of } P, \textit{all metrics}, m)$$

An attacker would measure the local stealth of the methods of a watermark program to determine if any methods stand out compared to the other methods of the program.

Sandmark implements these schemes of evaluating global and local stealth. Our data set consists of the complexities of all the methods in the specJVM benchmarks. We identified a few clustering algorithms favoring data with different properties. Dynamic data, in which the data set size is not constant over time requires more efficient algorithm so that the clusters are not rebuilt completely on every data inclusion. Our scenario deals with a static numerical data set. Our implementation is based on the clustering scheme proposed by Ward [17]. This hierarchical clustering technique is based on a rule of "merging cost" and is a bottom-up approach. That is, we start with clusters of size one and iteratively merge clusters until we reach the specified number of clusters. The merging cost of two clusters takes into account their respective *centroid* and size and is given by

$$\Delta \quad = \quad \frac{n_a n_b}{n_a + n_b}(\tilde{a} - \tilde{b})^2$$

where $\tilde{a}$ and $\tilde{b}$ are the centroids and n_a and n_b are the respective cluster sizes. A cluster is merged with its neighboring cluster with minimum merging cost. Similar clustering techniques like the *K-means* approach is based on an iterative re-assignment and is essentially a hill-climbing method.

Table 3 shows the results obtained from our stealth evaluation test.

As seen from the table, the values are obtained in the range 0 to 1, with values closer to one implying higher stealth since the particular application falls in the more likely complexity range. The results show a decrease in the code stealth in almost all the cases. This can be attributed to the fact that watermark embedding inserts new local variables and additional branch instruction code. This increases the overall complexity of the data structures and control flow graph of the target code which are identified by the complexity metrics.

The absolute stealth measures obtained are somewhat related to the number of clusters preselected during test run, but nevertheless, they do show the degree of effect of the embedding scheme on the stealth of the target application.

Watermark frequency vector: 57452648

	Global Stealth			Local Stealth		
Application	**Before**	**After**	**% Decr**	**Before**	**After**	**% Decr**
boyer	0.271	0.233	14.1%	0.207	0.157	24.2%
fft	0.182	0.119	34.6%	0.300	0.127	57.7%
lexgen	0.330	0.328	0.6%	0.376	0.257	31.6%
lu	0.162	0.049	69.8%	0.214	0.049	77.1%
nucleic	0.260	0.209	19.6%	0.244	0.174	28.7%
probe	0.287	0.029	89.9%	0.200	0.019	90.5%
TTT	0.392	0.398	-1.5%	0.263	0.336	-27.7%
Sandmark	0.260	0.260	0.0%	0.354	0.123	65.3%
JFIG	0.282	0.282	0.0%	0.401	0.201	49.9%

ClusterSize for benchmark Methods(for evaluating global stealth)	:	10
ClusterSize for target application(for evaluating local stealth)	:	5

Table 3: Effect of watermark embedding on stealth of the target application. The method level code complexity was evaluated through Halstead's, McCabe's and Munson's software complexity metrics.

By performing the stealth evaluation test over various watermark algorithms, we can deduce a more precise analysis over the effectiveness of each algorithm with respect to stealthiness of their respective embedding schemes.

5.4 Semantics-Preserving Attacks

The most serious threat to any watermark come from automated (or *class*) attacks. For example, there exist tools that perform optimization on binary or virtual machine code executables. For example, Debray [18, 19, 20] has developed a family of tools that optimize and/or compress X86 and Alpha binaries. Similarly, BLOAT [21] optimizes collections of Java class files. These tools are free and commonly available and it is desirable for any software watermarking scheme to survive the transformations they implement.

Moreover, SandMark implements a collection of obfuscating code transformations that can be used as class attacks against software watermarks. Typically, transformations are made up of combinations of the following operations:

fold/flatten: turn a d-dimensional construct into one with $d+1$ or $d-1$-dimensions;

split/merge: turn a compound construct X into two constructs $\{a, b\}$ or two constructs a and b into a compound construct X;

box/unbox: add or remove a layer of abstraction;

ref/deref: add or remove a level of indirection;

reorder: swap two adjacent constructs;

rename: assign a new name to a labeled construct.

Here, *construct* refers to any programming language object the obfuscator can manipulate. For example, if A is a vector then **fold(**A**)** turns A into a two-dimensional array. If P is a static method then **ref(**P**)**

turns it into a virtual one. If B is a basic block then **split**(B) breaks it into two parts by inserting a bogus branch protected by an opaque predicate. Many more such transformations have been described in the literature [22, 23, 24, 25]. Code obfuscation tools are also currently being constructed for binary code, such as the X86 [26].

We tested the robustness of our watermark on the specJVM [27] benchmark and some other applications. As attacks, we used various obfuscation algorithms currently implemented in SandMark . These obfuscators perform a range of semantic preserving modifications to the target code. Some of them perform class level modifications such as constant pool reordering or changing field accesses, whereas a bulk of the obfuscators affect method bodies. A very brief description of the various obfuscators used is given in appendix B.

Additive and subtractive attacks seem to be most effective in destroying the watermark. These attacks usually change the code size considerably, but are generally effective since they bring about random changes to the frequency of existing instruction groups. If some of these instruction groups fall within the set of the vector instruction groups then the corresponding bit is altered.

Decompiling and recompiling the watermarked application can also be a serious threat. This can be attributed to the type of instructions in the code book. Some of the substitution and embed/nullify groups are non-optimal to accommodate the required vector instruction groups. A code optimizer can exploit these groups and in the process remove the corresponding vector instruction group.

Algorithm 9 (Evaluate Resilience to Distortive Attacks)

```
for $b \leftarrow$ specJVM benchmarks and other Java applications do
    for $o \leftarrow$ all obfuscating transformations do
        $w \leftarrow$ select an 8-bit watermark;
        $b_w \leftarrow$ watermark $b$ with $w$;
        $b_o \leftarrow$ obfuscate $b_w$ with $o$;
        $r \leftarrow$ recognize($b_o$,$w$);
        if $r \geq 0.9$ then
            return "yes";
        $t \leftarrow \frac{\text{execution\_time}(b_o) - \text{execution\_time}(b)}{\text{execution\_time}(b)}$;
        if $r \geq 0.6$ then
            return $t$, "maybe";
        else
            return $t$, "no";
```

Algorithm 9 outlines the procedure for how to evaluate the SHKQ algorithm's resilience to semantics-preserving distortive attacks. We obfuscate a watermarked benchmark and then attempt to recognize the watermark. We use a 0.9 watermark detection threshold. Below 0.6, the watermark is assumed to be destroyed. When the obfuscator is successful in destroying the watermark we evaluate the performance penalty of the obfuscation.

Figure 4 shows the results of the evaluation. As expected, the watermark survived the obfuscations which do not directly affect method bodies. However, the *local variable promotion* obfuscations (which convert primitive types like `int` and `float` to their boxed equivalents, `java.lang.Integer` and `java.lang.Float`) were able to disrupt the watermark vector frequencies considerably and therefore destroy the watermark.

The *Append Bogus Code* obfuscator was also successful in destroying the watermark. This obfuscator adds large amounts of code uniformly over the entire program and was successful in disturbing some of the vector instruction groups. This particular obfuscation has significant performance overhead. We saw an increase in code size close to 10%. However, the execution time increased only by around 1.5%.

The *method merger* obfuscator reduced the correlation ratio to 0.402. The reason for this drastic change was that it changed the frequency of one vector group by a factor of 10 which was sufficient to break the watermark. The obfuscation increased the code size of by a mere 6.3% while execution time increased by 3.2%.

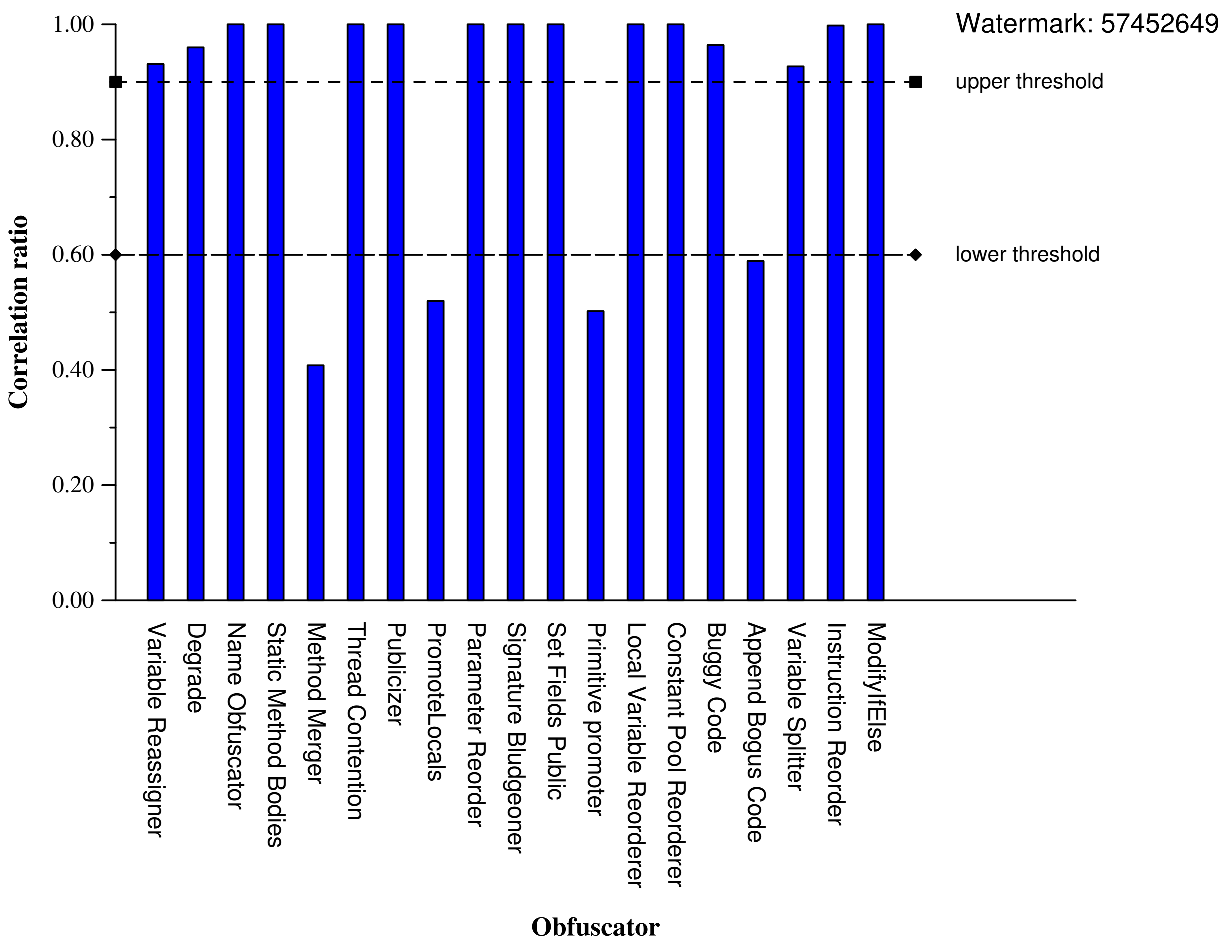


Figure 4: Results on specJVM benchmark.

There are quite a number of other attacks that can disrupt the embedded watermark vector. The recompilation attack (i.e. decompiling and recompiling the code) is a serious threat since this can completely distort the frequencies of the vector. For instance, an instruction code such as ”bipush -1” in a vector group can be converted into “iconst_m1” on recompilation. Therefore careful selection of the code book vector instruction groups are essential to minimize such transformations.

We performed the decompilation/recompilation test over a few small applications. In all the cases, the watermark could not be retrieved from the recompiled code with high confidence level. Figure 5 shows a few results of the test.[1] But such recompilations should be performed over the entire application to effect any appreciable change in the vector frequency, which can be sometimes tedious in the case of very large applications. The effect on performance was a 3.2% degradation.

Predictably, the watermark survived many high-level transformations that affect classes, fields, and method signatures. For example, obfuscations that reorder methods, classes, fields, or local variables have no effect.

[1]Decompilation was done using Ahpah’s SourceAgain decompiler.

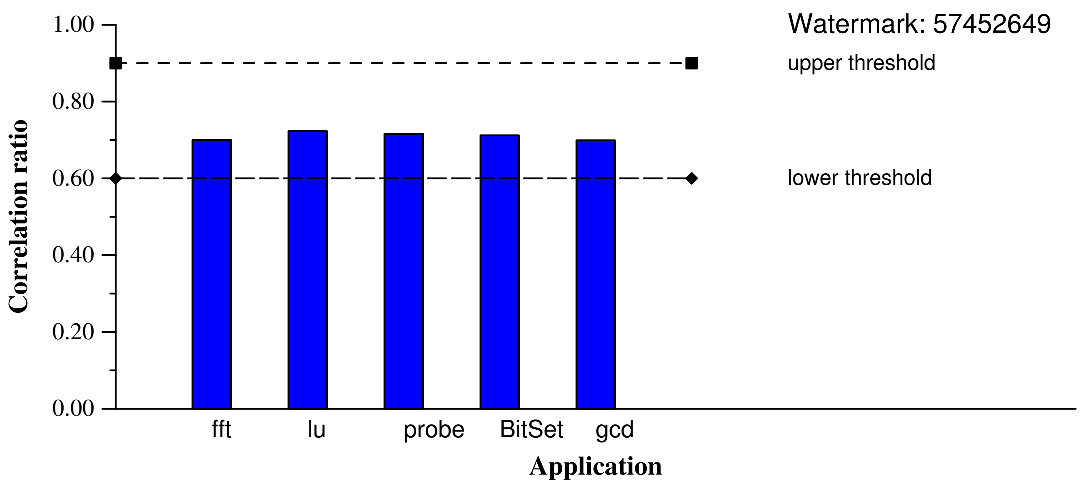


Figure 5: Effects of decompiling/recompiling on the watermark.

5.5 Collusive Attacks

A collusive attack attempts to find the location of a watermark by comparing two or more differently watermarked applications. Once the watermark locations have been found a manual inspection of the code may reveal useful details about the watermark scheme. Such information may allow the marks to be deleted directly (a so called *subtractive* attack) or a class attack might be constructed that can attack any application watermarked with the same scheme.

In the case of the SHKQ algorithm a collusive attack may be able to reveal the patterns of the code book. For example, assume that we have an application A that has been watermarked with two marks v and w, yielding A_v and A_w. At location L in A is an `iinc` instruction. In A_v this was replaced by a semantically equivalent code sequence while in A_w the `iinc` instruction remained unchanged:

```
        A                      A_v                 A_w
       ...                     ...                 ...
 L:  iinc X Y            L:  iload X         L:  iinc X Y
                             bipush Y
                             iadd
                             istore X
       ...                     ...                 ...
```

As a result, when A_v and A_w are compared it is easy to deduce that the two equivalent code sequences `iinc` and `iload, bipush, iadd, istore` form a substitution pattern from the code book. This is enough information to construct an effective class attack.

SandMark is equipped to perform collusive attacks. It has a Java bytecode comparison tool is developed which can allow an attacker to compare two Java applications for similarities. This tool is built on Baker's [28] algorithm for determining similarities between Java bytecodes.

Figure 6 shows an example of such attack on this watermark. The two columns show the results of two different watermark bits embedded in a small application. The *diff* was obtained using tool implemented in SandMark based on Baker's [28] algorithm. The difference in the code has been highlighted in the figure. By performing such collusive attacks, the attacker can sometimes be able to find out the specific code that builds the watermark.

It is our current belief that collusive attacks against software watermarks are a much less serious threat

Watermark vector: 57452649. Recognition Threshold: 0.9

Obfuscator	boyer	fft	lexgen	lu	matrix	nucleic	probe
Variable Reassigner	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Degrade	$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\ominus$	$\oplus$	$\ominus$
Bogus Arguments	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Name Obfuscator	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Method Merger	$\ominus$	$\oplus$	$\ominus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Static Method Bodies	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Thread Contention	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Publicizer	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Parameter Reorder	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Signature Bludgeoner	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
PromoteLocals	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$	$\ominus$
Set Fields Public	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Primitive Promoter	$\oplus$	$\ominus$	$\ominus$	$\ominus$	$\oplus$	$\oplus$	$\ominus$
Constant Pool Reorderer	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Local Variable Reorderer	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Buggy Code	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Append Bogus Code	$\oplus$	$\oplus$	$\ominus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Variable Splitter	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$

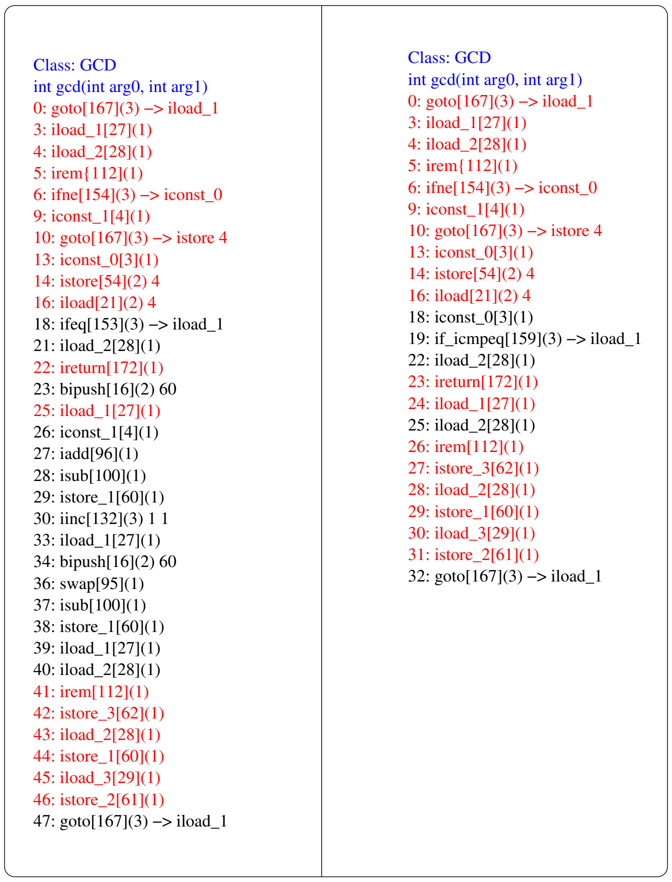
$\oplus$	:	*watermark found*
$\ominus$	:	*watermark destroyed*

Table 4: Effect of obfuscators on a set of watermarked benchmark applications.

than collusive attacks against media watermarks. The reason is that prior to watermarking an application can be preprocessed by subjecting it to a large number of randomly selected semantics-preserving transformations. As a result two differently fingerprinted versions A_v and A_w of an application will differ *everywhere*, not just at the locations of the watermarks.

5.6 False Positive Ratio

A watermarking scheme needs to be adjusted such that the probability of false detections and missed detections is below a certain limit. In the SHKQ algorithm the probability of a false detection depends on the watermark vector length as well as the range of values that the watermark vector components take. We tried out some false detections with some random watermark values (but with length equal to the length of the watermark vector currently implemented in our code book). For these values, the correlation ratio lies mostly in the probabilistic range 0.6–0.9. In a few cases, the correlation did exceed the 0.9 threshold i.e. gave a false detection. Table 5 shows a few results obtained from detection of *false* watermarks.



```
Class: GCD
int gcd(int arg0, int arg1)
0: goto[167](3) -> iload_1
3: iload_1[27](1)
4: iload_2[28](1)
5: irem{112}(1)
6: ifne[154](3) -> iconst_0
9: iconst_1[4](1)
10: goto[167](3) -> istore 4
13: iconst_0[3](1)
14: istore[54](2) 4
16: iload[21](2) 4
18: ifeq[153](3) -> iload_1
21: iload_2[28](1)
22: ireturn[172](1)
23: bipush[16](2) 60
25: iload_1[27](1)
26: iconst_1[4](1)
27: iadd[96](1)
28: isub[100](1)
29: istore_1[60](1)
30: iinc[132](3) 1 1
33: iload_1[27](1)
34: bipush[16](2) 60
36: swap[95](1)
37: isub[100](1)
38: istore_1[60](1)
39: iload_1[27](1)
40: iload_2[28](1)
41: irem[112](1)
42: istore_3[62](1)
43: iload_2[28](1)
44: istore_1[60](1)
45: iload_3[29](1)
46: istore_2[61](1)
47: goto[167](3) -> iload_1
```

```
Class: GCD
int gcd(int arg0, int arg1)
0: goto[167](3) -> iload_1
3: iload_1[27](1)
4: iload_2[28](1)
5: irem{112}(1)
6: ifne[154](3) -> iconst_0
9: iconst_1[4](1)
10: goto[167](3) -> istore 4
13: iconst_0[3](1)
14: istore[54](2) 4
16: iload[21](2) 4
18: iconst_0[3](1)
19: if_icmpeq[159](3) -> iload_1
22: iload_2[28](1)
23: ireturn[172](1)
24: iload_1[27](1)
25: iload_2[28](1)
26: irem[112](1)
27: istore_3[62](1)
28: iload_2[28](1)
29: istore_1[60](1)
30: iload_3[29](1)
31: istore_2[61](1)
32: goto[167](3) -> iload_1
```

Figure 6: Example of a collusive attack.

6 An Example

We will conclude by going through a example showing the effects of watermarking on a small code sample shown in Figure 7.[2]

The example shows a 8-bit watermark embedded in the target code. As observed, dummy variables are introduced in the watermarked code to aid insertion of additional instructions that build the watermark vector. Also included are instructions that nullify the effect of the earlier code modifications. The vector embedding due to code substitution may not be visible in the decompiled code since most of them do not produce any additional code.

7 Conclusion

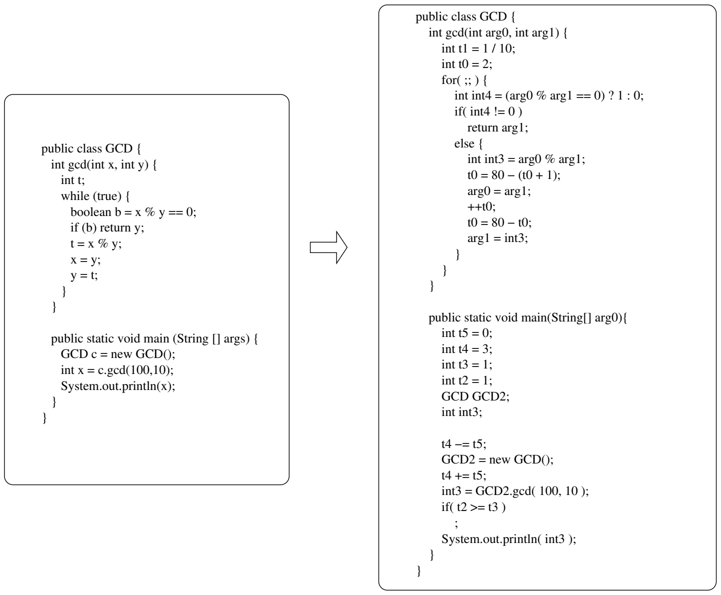
In this paper we have shown an implementation of the SHKQ algorithm for Java applications in the SandMark framework. We proposed various additions and improvements to the original scheme. We also proposed various evaluation criteria for software watermarks. We have done an empirical analysis of this watermarking scheme and shown various results based on our evaluation criteria. We showed that this

[2]The transformation was performed by the SandMark tool and decompiled using Ahpah's SourceAgain decompiler. The decompiled examples have been subjected to minor hand-editing to make them suitable for publication.

Actual Watermark frequency vector: 74893657
Embedded on specJVM bench.jar

Random watermark	Correlation ratio
44558837	0.88
15675469	0.91
28448200	0.63
81803841	0.77
69378854	0.86
94472096	0.88
50901154	0.76
10191937	0.81

Table 5: Results of detection of *false* watermarks by the recognizer



```
public class GCD {
  int gcd(int x, int y) {
    int t;
    while (true) {
      boolean b = x % y == 0;
      if (b) return y;
      t = x % y;
      x = y;
      y = t;
    }
  }

  public static void main (String [] args) {
    GCD c = new GCD();
    int x = c.gcd(100,10);
    System.out.println(x);
  }
}
```

```
public class GCD {
  int gcd(int arg0, int arg1) {
    int t1 = 1 / 10;
    int t0 = 2;
    for( ;; ) {
      int int4 = (arg0 % arg1 == 0) ? 1 : 0;
      if( int4 != 0 )
        return arg1;
      else {
        int int3 = arg0 % arg1;
        t0 = 80 − (t0 + 1);
        arg0 = arg1;
        ++t0;
        t0 = 80 − t0;
        arg1 = int3;
      }
    }
  }

  public static void main(String[] arg0){
    int t5 = 0;
    int t4 = 3;
    int t3 = 1;
    int t2 = 1;
    GCD GCD2;
    int int3;

    t4 −= t5;
    GCD2 = new GCD();
    t4 += t5;
    int3 = GCD2.gcd( 100, 10 );
    if( t2 >= t3 )
      ;
    System.out.println( int3 );
  }
}
```

Figure 7: An example.

watermarking scheme, though robust against a few code obfuscators, is susceptible to various attacks such as decompilation/recompilation and additive attacks. **Acknowledgments:** This work is supported by the

NSF under grant CCR-0073483 and by the AFRL under contract F33615-02-C-1146.

References

[1] Julien P. Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999. `http://citeseer.nj.nec.com/stern00robust.html`.

[2] W. Bender, D. Gruhl, and N. Morimoto. Techniques for data hiding. In *In Proc. of the SPIE 2420 (Storage and Retrieval for Image and Video Databases III),*, pages 164–173, 1995. `http://citeseer.nj.nec.com/context/56590/0`.

[3] Christian Collberg and Clark 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. `http://citeseer.nj.nec.com/collberg99software.html`.

[4] Jens Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of ACSAC'00, 16th Annual Computer Security Applications Conference*, pages 308–316, 2000. `http://citeseer.nj.nec.com/323325.html`.

[5] Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, Pittsburgh, PA, April 2001. `http://link.springer.de/link/service/series/0558/bibs/2137/21370157.htm%`.

[6] 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 Corporation, 1996. `http://www.delphion.com/details?pn=US05559884__`.

[7] Gang Qu and Miodrag Potkonjak. Hiding signatures in graph coloring solutions. In *Information Hiding*, pages 348–367, 1999.

[8] A. Monden, H. Iida, K. Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *compsac2000, 24th Computer Software and Applications Conference*, 2000.

[9] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, 1998. `http://www.cs.auckland.ac.nz/collberg/Research/Publications/CollbergTho%mborsonLow97a/index.html`.

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

[11] I. Cox, Matthew L. Miller, and Jeffrey A. Bloom. *Digital Watermarking.*

[12] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[13] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[14] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland, 1977.

[15] E. I. Oviedo. Control flow, data flow, and program complexity. In *Proceedings of IEEE COMPSAC*, pages 146–152, November 1980.

[16] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[17] J.H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58:236–244, 1963.

[18] Saumya Debray, Benjamin Schwarz, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*, September 2001.

[19] Saumya Debray, Robert Muth, Scott Watterson, and Koen De Bosschere. ALTO: A link-time optimizer for the Compaq Alpha. *Software — Practice and Experience*, 31:67–101, January 2001.

[20] Saumya Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[21] Nathaniel Nystrom. Bloat – the bytecode-level optimizer and analysis tool. `http://www.cs.purdue.edu/homes/whitlock/bloat`, 1999.

[22] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997. `www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborso%nLow97a`.

[23] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages, ICCL'98*, Chicago, IL, May 1998. `www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborso%nLow98b/`.

[24] Frederick B. Cohen. Operating system protection through program evolution. `all.net/books/IP/evolve.html`, 1992.

[25] Paul Tyma. Method of reducing the number of instructions in a program code sequence. US patent 5,903,761, 1999.

[26] Grzegorz Wroblewski. *A General Method of Program Code Obfuscation.* PhD thesis, Wroclaw University, 2002.

[27] Specjvm98. `www.specbench.org/osg/jvm98`, 1998.

[28] Brenda S. Baker and Udi Manber. Deducing similarities in Java sources from bytecodes. In *Usenix Annual Technical Conference*, pages 179–190, June 1998.

A The code book

VECTOR INDEX	VECTOR GROUPS	SUBSTITUTION GROUPS		NEW INSTRUCTION GROUPS	
		$\longrightarrow$		EMBED GROUPS	NULLIFY GROUPS
0	bipush iload	iload X bipush Y if_icmpne/ if_icmpeq −−> Z	bipush Y iload X if_icmpne/ if_icmpeq −−> Z	bipush X iload Y iconst_1 iadd isub istore Y	iinc Y 1 iload Y bipush X swap isub istore Y
		iload X bipush Y if_icmpgt/ if_icmpge −−> Z	bipush Y iload X if_icmplt/ if_icmple −−> Z		
		iload X bipush Y if_icmplt/ if_icmple −−> Z	bipush Y iload X if_icmpgt/ if_icmpge −−>Z		
1	iconst if	ifeq/ ifne −−> Z	iconst_0 if_icmpeq/ if_icmpne −−> Z	A: iload X iconst_0 if_icmplt −−> B	goto −−> Z B: iload X iflt −−> A Z:
		ifgt/ ifge −−> Z	iconst_0 if_icmpgt/ if_icmpge −−> Z		
		iflt/ ifle −−> Z	iconst_0 if_icmplt/ if_icmple −−> Z		
2	iload iadd	iload X getstatic Y Z iadd	getstatic Y Z iload X iadd	iconst_m1 iload X iadd istore X	iload X iconst_1 iadd istore X
3	iload iload if_icmpgt	iload X iload Y if_icmplt −−> Z	B: iload Y iload X if_icmpgt −−> A	iload X iload Y if_icmpgt −−> Z	goto −−> Z A: iload Y iload X if_icmplt -> B Z:
4	iconst_m1 isub	iload X iconst_1 iadd	iload X iconst_m1 isub	iload X iconst_m1 isub istore X	iload X iconst_1 isub istore X
5	bipush iadd	iinc X Y	iload X bipush Y iadd istore X	iload X bipush −1 iadd istore X	iload X iconst_1 iadd istore X
6	idiv istore	iconst_0 istore X	iconst_0 bipush Y idiv istore X	iload X iconst_1 idiv istore X	
		iconst_1 istore X	iconst_1 bipush 1 idiv istore X		
7	iload isub	iload X iload Y if_icmpne/ if_icmpeq −−> Z	iload X iload Y isub ifeq/ ifne −−> Z	iload Y iload X isub istore Y	iload X iload Y swap iadd istore Y

B Code Obfuscators in SandMark

- **Variable Reassigner** Reallocates local variables in a method in order to minimize the number of local variable slots used.
- **Degrade** Slows down the program execution through thread contention and local variable promotion.
- **Bogus Arguments** Obfuscates the method parameters by adding extra bogus arguments.
- **Name Obfuscator** This algorithm renames the fields and methods of an application to unique identifiers, while keeping inheritance relationships intact.
- **Method Merger** Merges all the static methods together.
- **Static Method Bodies** Changes the method bodies of dynamic methods into static methods.
- **Thread Contention** Degrades the program performance through thread contention.
- **Publicizer** Makes all fields and methods public.
- **Promote Locals** Promotes all primitive locals in the methods into objects.
- **Parameter Reorderer** Confuses the parameter lists of static methods.
- **Add Bogus Fields** Obfuscation is done by inserting a bogus field into a class and then making assignments to that field in specific locations throughout the code.
- **Signature Bludgeoner** Bludgeons the signatures of methods so that they take Object[] and return Object in order to hide the information implicitly encapsulated in methods' argument lists.
- **Set Fields Public** Changes the field access modifiers of all of the fields in a class.
- **Primitive Promoter** Promotes the local primitive variables into objects. (eg. *int* to *Integer*)
- **Constant Pool Reorderer** Reorders the constants in the constantpool and assigns random indices to them.
- **Local Variable Reorderer** Reorders only local variables and no parameter reordering is done. The local variables are randomly assigned some unique index within the range of maximum number of locals, and changes should also be made to reflect with respect to read and write references
- **Buggy Code** Creates copies of existing random basic blocks and introduces bug code within the new copy of the block. The execution of the block is bypassed through some opaque predicates.
- **Append Bogus Code** Adds bogus statements onto the end of a Java method. The appended code may include a variety of other instructions including return instructions. Multiple running of this algorithm add additional statements.
- **Variable Splitter** Creates an additional variable that has it's value changed in coordination with an original local variable. Each reference to that variable value may have been changed to reference the new variable instead.
- **Instruction Reorderer** Reorders instructions within a basic block while preserving its semantics.