An Evaluation of the Resilience of Static Java Bytecode Watermarks Against Distortive Attacks

James Hamilton and Sebastian Danicic *

Abstract—**Software watermarking is a software protection technique based on the insertion of copyright notices or unique identifiers into a program to prove ownership. Software watermarks can be broadly divided into two categories: static and dynamic. The former embeds the watermark in the data and/or code of the program, while the latter embeds the watermark in a data structure built at runtime. We describe and evaluate the existing Java static watermarking systems and algorithms by using them to watermark bytecode files and then applying distortive attacks to each watermarked program by obfuscating and optimising. Our study revealed that a high proportion of watermarks were removed as a result of these transformations both in the commercial and academic watermarking systems that we tested. These findings confirm the results of previous studies and this is further evidence that static watermarking techniques on their own do not give sufficient protection against software piracy.**

Keywords: java, bytecode, watermarking, obfuscation, program transformation

1 Introduction

The global revenue loss due to software piracy – the act of copying a legitimate application and illegally distributing that software, either free or for profit – was estimated to be more than $50 billion in 2008 [8].

Technical measures have been introduced to protect digital media and software due to the ease of copying computer files. Some software protection techniques, of varying degrees of success, can be used to protect intellectual property contained within Java class-files. These techniques include: using native code, encryption, obfuscation, watermarking and fingerprinting. Encryption and obfuscation aim to either decrease program understand or prevent decompilation, while watermarking and fingerprinting uniquely identify applications to prove ownership in a court of law.

Software watermarking involves embedding a unique identifier within a piece of software, to discourage software thieves by providing a means of identifying the owner of a piece of software and/or the origin of the stolen software [41]. The hidden watermark can be extracted, at a later date, by the use of a *extractor* or recognised by the use of a *recogniser*. It is possible to embed a unique customer identifier in each copy of the software distributed which allows the software company to identify the individual that pirated the software. It is necessary that the watermark is hidden so that it cannot be detected and removed and that the watermark is *robust* - that is, resilient to semantics preserving transformations (such as optimisations or obfuscations).

The Java virtual machine is a popular platform for executable programs from languages including, but not limited to, Java. The Java virtual machine provides a platform for which programs can be written once and run on any physical machine for which there is a Java virtual machine. Java bytecode is higher level than machine code and is relatively easy to decompile with only a few problems to overcome [29].

In this paper, present an evaluation of existing Java bytecode watermarking software and evaluate their effectiveness.

2 Background

Watermarking techniques are used extensively in the entertainment industry to identify multimedia files such as audio and video files, and the concept has extended into the software industry. Software watermarking is harder because media, such as video or audio, can tolerate certain distortions whereas software must remain semantically correct after watermarking. Watermarking does not aim to make a program hard to steal or indecipherable like obfuscation but it discourages theft as thieves know that they could be identified [63].

Most literature discusses techniques and problems of automatically watermarking software and there is only a small amount of literature which compares automatic watermarking with manual watermarking [45]. A manual watermark is inserted by the programmer of the application, rather than a using a third-party automatic tool. Some semi-automatic watermarking systems also exist (e.g. The Collberg-Thomborson algorithm implemented

*Department of Computing, Goldsmiths, University of London, United Kingdom, james.hamilton@gold.ac.uk, s.danicic@gold.ac.uk. This is an extended version of a previously published paper [30].

in Sandmark [13]) - where a programmer inserts markers into a program during development and the finished software is then augmented by a software watermarking tool. We are interested in automatic watermarking systems which enable a user to simply watermark a piece of software with little effort.

Watermarks can be classified as either visible, where the recogniser is public knowledge or invisible where the recogniser, or some component (such as an encryption key), is not public knowledge. Visible watermarks can act as a deterrent but also show an adversary the location of a watermark making the task of removing the watermark easier.

2.1 Difficulties of Software Watermarking

Software watermarks present several implementation problems and many of the current watermarking algorithms are vulnerable to attack. Watermarked software must meet the following conditions:

1. program size must not be increased significantly.
2. program efficiency must not be decreased significantly.
3. robust watermarks must be resilient to semantics preserving transformations (fragile watermarks, by definition, should not be).
4. watermarks must be sufficiently well hidden, to avoid removal.
5. watermarks must be easy for the software owner to extract.

Perhaps the most difficult problem to solve is keeping the watermark hidden from attackers while, at the same time, allowing the software owner to efficiently extract the watermark when needed. If the watermark is too easy to extract then an attacker would be able to extract the watermark too. If a watermark is too well hidden then the software owner may not be able to find the watermark, in order to extract it. Some watermark tools (such as Sandmark [13]) use markers to designate the position of the stored watermark - this is problematic as it poses a risk of exposing the watermark to an adversary.

Watermarks should be resilient to semantics preserving transformations and ideally it should be possible to recognise a watermark from a partial program. Semantics preserving transformations, by definition, result in programs which are syntactically different from the original, but whose behaviour is the same. The attacker can attempt, by performing such transformations, to produce a semantically equivalent program with the watermark removed. Redundancy and recognition with a probability threshold may help with these problems [36]. Ideally, software watermarks should be resilient to decompilation-recompilation attacks, as decompilation of Java is possible (though not perfect [29]).

The watermark code must be locally indistinguishable from the rest of the program so that it is hidden from adversaries [56]. For example, imagine a watermark which consists of a dummy method with 100 variables - this kind of method will probably stand out in a simple analysis of the software (such as using software metrics techniques [28]). It could be difficult to programatically generate code which is indecipherable from the human-generated program code but statistical analysis of the original program could help in generating suitable watermark code [36].

Software watermarks must be efficient in several ways: cost of embedding, cost of runtime and cost of recognition time.

The cost of embedding a software watermark can be divided into two areas: developer time and embedding cost. The former simply quantifies the time that a developer spends embedding a watermark, while the latter quantifies the execution time of a software watermarking tool. Embedding costs are not a significant problem except in certain cases such as live multimedia streaming.

Developer time is important in use of software watermarks as the developer should not have to spend a large amount of time preparing a software watermark. The complexity of a software watermark is proportional to the resilience of the watermark - that is, the greater amount of time a developer spends embedding a watermark the harder it may be for an adversary to crack. For example, a developer could spend days introducing a subtle semantic property into the program which is unique to the software and very hard to discover.

In the middle of the scale is a semi-automatic watermark which involves a developer preparing a program before a watermarking tool embeds the watermark. The preparations could include inserting markers where watermark code should be inserted, or creating dummy methods which watermarks could use. Monden *et al.* [40] describe a watermarking algorithm which requires the production of a dummy method in a program for the watermark to be stored. A programmer must create this dummy method manually and then execute watermarking software to embed the watermark.

The cost of runtime depends on the effect that the transformations applied by the watermark have had on the size and execution time. For example, Hattanda *et al.* [31] found that the size of a program, watermarked with Davidson/Myhrvold [25] algorithm, increased by up to 24% and the performance decreased by up to 14%.

Dummy methods, which are not executed, will have minimal effect on runtime cost but dynamic watermarks may have a high runtime cost as the watermark is built during program execution. The *fidelity* of watermark, 'the extent to which embedding the watermark deteriorates the original content' [45], should also be taken into account for the effects caused by watermarking, for example embedding a watermark may introduce unintentional errors.

The ideal recognition time of a watermark will most likely be quick but in some cases it may be important to artificially slow watermark recognition time to prevent *oracle* attacks [45]. Such attacks rely on the repetitive execution of a recogniser thus fast recognition time helps an adversary.

2.2 Types of Watermark

Software watermarks can be broadly divided into two categories: static and dynamic [16]. The former embeds the watermark in the data and/or code of the program, while the latter embeds the watermark in a data structure built at runtime. Figure 1 shows a conceptual diagram of a simple static watermarking system.

Additionally, Nagra *et al.* define four categories of watermark [45]:

> **Authorship Mark** identifying a software author, or authors. These watermarks are generally visible and robust.
>
> **Fingerprinting Mark** identifying the channel of distribution, i.e. the person who leaked the software. The watermarks are generally invisible, robust and consist of a unique identifier such as a customer reference number.
>
> **Validation Mark** to verify that software is genuine and unchanged, for example like digitally signed Java Applets. These watermarks must be visible to the end-user to allow validation and fragile to ensure the software is not tampered with.
>
> **Licensing Mark** used to authenticate software against a license key. The key should become ineffective if the watermark is damaged therefore licensing marks should be fragile.

In this paper, we evaluate static watermarking systems which enable software authors to prove ownership of their software and/or identify the customer responsible for the copyright infringement. We are therefore interested in only the first two kinds of watermarks: authorship marks and fingerprint marks.

2.3 Program Transformation Attacks

Program transformation attacks on watermarked software can be divided into three categories:

2.3.1 Additive

An additive attack involves inserting another watermark into an already watermarked application, thus overwriting the original watermark. This attack will usually work if a watermark of the same type is embedded but not necessarily if a different type of watermark is embedded [43]. However, even if the original watermark is not over-written by the new watermark the attacker could claim ownership of the software if both watermarks are recognisable. It will be difficult to prove which party inserted their watermark first and thus result in a dispute over ownership.

2.3.2 Subtractive

A subtractive attack involves removing the section, or sections, of code where the watermark is stored while leaving behind a working program. This could be achieved by *dead code elimination*, *statistical analysis* or *program slicing*.

2.3.3 Distortive

Distortive attacks involve applying semantics preserving transformations to a program, such as obfuscations or optimisations thus removing any watermarks which rely on program syntax. For example, *renaming variables*, *loop transformations*, *function inlining*, etc.

Both static and dynamic watermarks can be susceptible to program transformation attacks. Myles *et al.* [43] conducted an evaluation of dynamic and static versions of the *Arboit* algorithm by watermarking and obfuscating test files. They found the dynamic version to be only minimally stronger than the static version, and both versions could be defeated by distortive attacks.

In this paper, we attack watermarked programs with distortive attacks.

3 Empirical Evaluation

We evaluate the existing static watermarking software by watermarking 60 *jar* files with all available watermark algorithms and then apply a distortive attack to each watermarked program, by obfuscating and optimising. After all the programs have been transformed we attempt to extract the watermarks from the programs. We expect that many watermarks will be lost during the transformations and attempt to find which transformations most affect the watermarks.

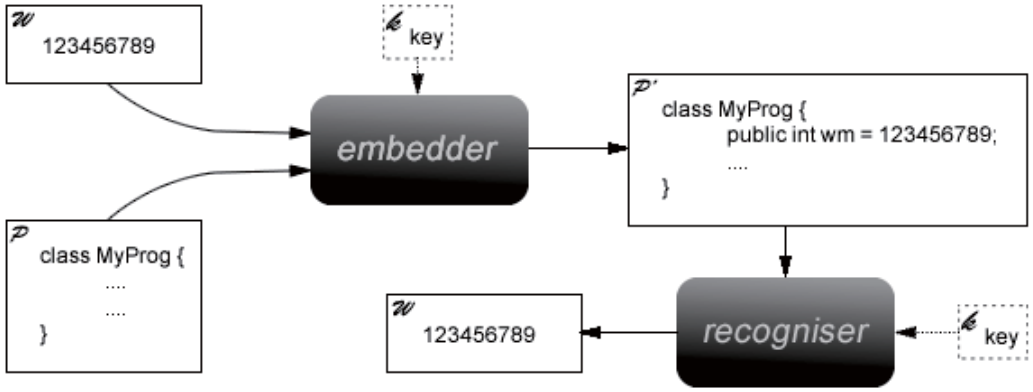

Figure 1: A simple static watermarking system

3.1 The Watermarkers

We are testing 14 different static watermarking algorithms from 3 different watermarking systems: Sandmark, Allatori and DashO. The latter two are commercial systems, while the former is an academic open-source framework. These are the available systems that we could obtain for watermarking Java programs.

name	type	version	year
Sandmark	open-source	3.4.0	2004
Allatori	commercial	2.8	2009
DashO	commercial	6.3.3	2010

Table 1: The Watermarking Systems

Some of the algorithms have been evaluated before (for example Collberg *et al.* have compared the Davidson/Myhrvold and Monden algorithms [44]) and we re-evaluate them with our set of test programs. To the best of our knowledge, the commercial watermarking systems, Allatori and DashO, have not been evaluated before. Both commercial systems use proprietary algorithms, and are therefore undocumented - we include a short description of their algorithms as far as we understand.

3.1.1 Sandmark

SandMark [13] is a tool developed by Christian Collberg *et al.* at the University of Arizona for research into software watermarking, tamper-proofing, and code obfuscation of Java bytecode. The project is open-source and both binaries and source-code can be download from the SandMark homepage [13]. We used version 3.4.0 released in 2004.

3.1.2 Allatori

Allatori [52] is a commercial Java obfuscator complete with a watermarking system created by Smardec [51]. The company claim that 'if it is necessary for you to protect your software, if you want to reduce its size and to speed up its work, Allatori obfuscator is your choice' [52]. We used version 2.8 released in 2009.

3.1.3 DashO

DashO [1] is a commerical Java security solution, including obfuscator, watermarking and encrypter - similar to Allatori. DashO is made by PreEmptive Solutions [2] who claim that 'DashO provides advanced Java obfuscation and optimization for your application'. We used version 6.3.3 released in 2010.

3.2 The Watermark Algorithms

We evaluate 14 static watermarking algorithms available to us from the 3 watermarking systems. Sandmark contains 12 static Java bytecode watermarking algorithms [18]; both Allatori and DashO have 1 watermarking algorithm. Of the 12 algorithms, in Sandmark, some are fairly trivial while others have been studied extensively in software watermark literature.

3.2.1 String Constant

The String Constant algorithm is a simple watermarking algorithm which simply embeds the watermark string in an unused constant in the constant pool of a class-file.

3.2.2 Add Expression

The Add Expression [10] algorithm simply adds a bogus addition expression to a class-file, containing 2 integers which add up to the watermark number. For example, `int wm = 451 + 123`. This is added to the constant pool of a classfile which is used to form an expression during recognition.

3.2.3 Add Initialization

The Add Initialisation [18] watermarking algorithm inserts several bogus integer assignments in randomly chosen class files. The variables store 2 digits of a watermark which are concatenated together during recognition. The variable names include numbers to ensure the watermark is reconstructed in the correct order. The length of the watermark is also stored in a bogus variable. Variable names are prefixed with `sm$`.

3.2.4 Add Method and Field

The Add Method and Field [18] algorithm splits a watermark in two: one half is stored in the name of a bogus field, the other half store in the name of a bogus method. The new method accesses the field, while a randomly chosen method calls the new method to make it seem like they are part of the program.

3.2.5 Add Switch

The Add Switch [18] algorithm embeds the watermark in the case values of a switch statement, inserted at the beginning of a randomly chosen method.

3.2.6 Register Types (HatTrick)

Register Types [18] embeds a watermark by introducing local variables of certain Java standard library types, based on the encoding in table 2, into the first non-abstract method of the first class of a Jar file. The variables are prefixed with a secret name to enable recognition.

3.2.7 Davidson/Myhrvold

Davidson and Myhrvold [25] proposed one of the first software watermarking algorithms which encodes the watermark by basic block re-ordering. The embedding algorithm was described in a patent issued to Microsoft but the extraction algorithm was not discussed. Collberg *et al.* [44] proposed a method of watermark extraction and implemented the DM algorithm in Sandmark [13].

Table 2: Register Types Encoding

Number	Type
0	`java.util.GregorianCalendar`
1	`java.lang.Thread`
2	`java.util.Vector`
3	`java.util.Stack`
4	`java.util.Date`
5	`java.io.InputStream`
6	`java.io.ObjectStream`
7	`java.lang.Math`
8	`java.io.OutputStream`
9	`java.lang.String`

Collberg *et al.*'s extraction algorithm is an *informed* extraction algorithm; that is, it requires the original P and the watermarked program P^w to extract the watermark w. The embedding algorithm re-orders only unique basic blocks in all methods as there is no way of knowing which method(s) the watermark is stored in. This would result in the extraction of many watermarks; Collberg *et al.* overcome this in their implementation by prefixing and suffixing magic numbers to the watermark to guarantee recognition. Non-unique basic blocks can be made unique by inserting bogus code (such as *no op* instructions) until all the basic blocks are unique [14].

The basic idea is to convert the watermark into a number w; then the w^{th} permutation of a set of basic blocks B is generated. The permutated basic blocks B' are relinked to retain the original program semantics and B is replaced by B' to produce the watermarked program P'. To extract a watermark, the ordering of the original basic blocks is compared against the new ordering, to obtain the permutation number; this number is then converted back into the watermark number.

A program method containing n unique basic blocks can embed $[log_2 n!]$ watermark bits. The method should not contain exception handling code as this can impose an ordering of basic blocks which is difficult or impossible to alter [44].

Hattanda and Ichikawa [31] evaluated the DM watermarking algorithm by watermarking several C programs and analysing metrics such as program size and program performance. In their implementation they found that the size increase of a watermarked program was between 9% and 24% while the performance was 86% to 102% of the original program. [5] implemented and evaluated a version of the DM watermarking algorithm for machine code where groups of chains of basic blocks are re-ordered. They concluded that their watermarking algorithm is stealthier as it has a minimal affect on code

locality.

Hattanda *et al.* reported a data-rate of approximately 0.2% of program size (in bytes) based on their implementation that used a partial permutation scheme, which only used 6 basic blocks. Collberg *et al.*'s implementation was not constrained in this way and the data rate is dependent on the number of basic blocks in program methods.

The DM algorithm is highly unstealthy due to the fact that a normal compiler would not linearise the control flow graph as in DM watermarked programs [14]. A simple way to discover a DM watermark is to examine the ratio of *goto* statements to the total number of instructions in a method - methods with the DM watermark show a high ratio compared to an unwatermarked method [44].

3.2.8 Graph Theoretic Watermark

Collberg et al. [19] describe several techniques for encoding watermark integers in graph structures. Graph watermarking algorithms rely on the fact that graph-generating code is difficult to analyse due to aliasing effects [26] which, in general, is known to be un-decidable [49].

An ideal class of watermarking graph should have the following properties [22]:

- ability to efficiently encode a watermark integer; and be efficiently decodable to a watermark integer
- a root node from which all other nodes are reachable
- a high data-rate
- a low outdegree to resemble common data structures such as lists and trees
- error correcting properties to allow detection after transformation attacks
- tamper-proofing abilities
- have some computationally feasible algorithms for graph isomorphism, for use during recognition

Venkatesan et al. [56] proposed the first static graph watermarking scheme, Graph Theoretic Watermarking (GTW), which encodes a value in the topology of a program's control-flow graph [3]. The idea was later patented by Venkatesan and Vazirani [55] for Microsoft. The basic concept is to encode a watermark value in a reducible permutation graph and convert it into a control flow graph; it is then merged with the program control flow graph by adding control flow edges between the two (see figure 2).

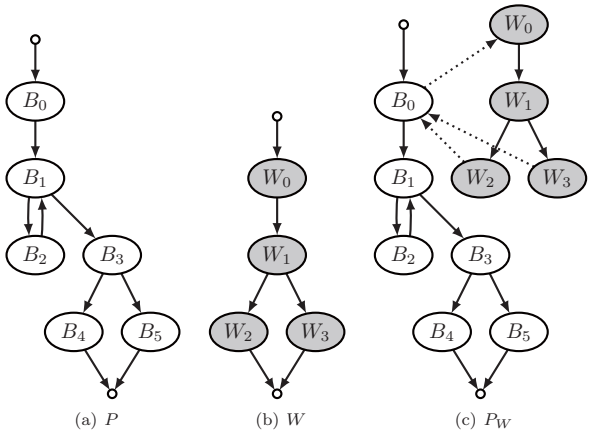

Figure 2: Graph theoretic watermarking

In a permutation graph encoding scheme encoding scheme a permutation $P = \{p_1, p_2, \ldots, p_n\}$ is derived from the watermark integer n; the permutation is then encoded in the graph by adding edges between vertices i and p_i.

Reducible permutation graphs (RPG) [55, 56] are very similar to permutation graphs but they closely resemble control-flow graphs as they are reducible-flow graphs [32]. They resemble control-flow graphs constructed from programming constructs such as *if*, *while* etc. [14]. This family of graphs is resistant to edge-flip attacks, where an attacker inverts the condition of conditional jumps in a program.

RPGs, like CFGs, contain a unique entry node and a unique exit node, a preamble which contains zero or more nodes from which all other nodes can be reached and a body which encodes a watermarking using a self-inverting permutation [11].

The algorithm adds bogus control flow edges between random pairs of vertices in the program CFG and watermark CFG in order to protect against static analysis attacks looking for sparse-cuts [7] in the control-flow graph. A sparse-cut would indicate a possible joining point of the original program CFG and the watermark CFG where the attacker could split the program with as few edges broken as possible.

Collberg et al. [12] implemented a version GTW_{SM} of GTW in Sandmark [13]. They measured the size and time overhead of watermarking and evaluated the algorithm against a variety of attacks. They also introduce two methods (Partial Sum splitting and Generalised Chinese Remainder Theorem splitting) for splitting a watermark integer into redundant pieces so that a large integer can be stored in several smaller CFGs. They found that stealth is a big problem; for example, the basic blocks of the generated watermark method consisted of 20% arithmetic instructions compared to just 1% for standard Java

methods [20]. Watermarks of up to 150 bits increased program size by between 40% and 75%, while performance decreased by between 0% and 36% [12].

3.2.9 Monden

Some of the first patented software watermarking algorithms [33, 50] were based around the idea of code replacement; that is, they replaced a pre-determined portion of code and/or data in a program with the watermark value.

Monden et al. [38, 39, 40] describe a code replacement technique, MON, for watermarking Java programs by swapping bytecode instructions within dummy methods (implemented as jmark [37]). The dummy methods used by MON are created either manually or automatically, and method calls are protected by opaque predicates [17] to ensure they are not executed.

The basic idea is to assign bit values to certain Java bytecode instructions and replace the existing instructions with the encoding bits which correspond to the watermark value. As the dummy method is not executed there are no semantic restrictions on the replacements but the watermarked method must be semantically correct, in order to pass the Java bytecode verifier [57].

Myles et al. [44] implemented a version, MON_{SM}, in Sandmark [13] and compared it to the Davidson/Myhrvold watermarking scheme [25]. MON_{SM} differs from the jmark implementation as it automatically generates a dummy method, so is completely automatic. However, it is difficult to generate code which is similar to the original program and it may be discoverable by a statistical analysis of the bytecode. In this evaluation we use the MON_{SM} version.

3.2.10 Qu/Potkonjak (Register Allocation)

The QP algorithm [46] is a constraint-based watermarking (and fingerprinting [48]) algorithm based on the concept of graph colouring. In the QP algorithm edges are added to a graph based on the value of the watermark. The graph used for watermarking programs is the interference graph [9], which is used to model the relationship between the variables in a program method. Each vertex in the graph represents a variable and an edge between two variables indicates that their live ranges overlap. We colour the graph in order to minimise the number of registers required and ensure that two live variables do not share a register. Zhu and Thomborson [61] described a clarified version of the originally published algorithm.

A major flaw in the QP algorithm is that it is *not extractable* as it is possible to insert two different messages into an interference graph and obtain the same watermark graph [60–62]. It has also been shown that the QP graph solution can be modified in such a way that any message could be extracted [35]. Qu and Potkonjak dismiss this problem, claiming that it will be hard to build a meaningful message particularily if the original message is encrypted by a one-way function [47].

Myles and Collberg [42] implemented a new algorithm, QPS, in Sandmark [13]. In the QPS algorithm triples of vertices are selected such that they are isolated units that will not effect other vertices in the graph. Experimental results [42] showed that the QPS algorithm has a very low data-rate and is susceptible to a variety of simple attacks, such as obfuscations. However, the QPS algorithm was found to be quite stealthy and is extremely credible. In other words, the watermarks are hard to detect by an attacker whilst readily detectable by the watermark author.

In this paper, we use the QPS version of the algorithm, which is implemented in Sandmark.

3.2.11 Static Arboit

An opaque predicate is a predicate whose outcome is known *a priori*. It is difficult for automated software analysis to find the value of the predicate; therefore it is not known whether the enclosed code (which may or may not be a watermark) could be removed [17].

Arboit [6] proposed a watermarking method where pieces of a watermark are encoded as constants within opaque predicates. The watermark is extracted by searching a program for opaque predicates and decoding them back into the watermark value.

Myles and Collberg [43] implemented the algorithm in Sandmark [13] and found that the algorithm could, fairly easily, be defeated by semantics-preserving transformation attacks.

3.2.12 Stern (Robust Object Watermarking)

Stern et al. [54] introduce robust object watermarking ROW, based on a spread-spectrum technique previously used for multimedia watermarking [23]. This technique differs from many other techniques because it views the code as a whole statistical object, rather than a sequence of instructions. The technique is more resilient against collusion attacks because the watermark is spread out over the program, rather than being in one location.

The approach modifies the frequencies of groups of instructions in order to watermark the code (though other statistical properties of the program could be used).

Stern et al. [54] implemented their technique for x86 assembly language and later Hachez [27], and separately Collberg and Sahoo [15], implemented the technique for Java bytecode. Curran et al. [24] describe a spread-spectrum technique using a vector derived from the call graph depth of a program; Ai et al. [4] attempt to improve on the original algorithm by introducing a collusion-attack resistant variation.

Finding the perfect transformation is difficult ; a previous study found that the watermark can survive many high-level obfuscations that effect classes, fields, and method signatures [15] but some transformations are easily undoable by trivial obfuscations [14]. In this paper, we use the Sandmark implementation.

3.2.13 Allatori

Allatori embeds watermarks in a sequence of *push* and *pop* operations inserted into multiple methods. The sequences consists of 4 instructions pushing integers, followed by 4 pop instructions, repeated many times in a method. We have not determined the encoding used but the watermark is unstealthy as code like this is unusual. It is also not resilient to attack - we believe the optimiser will be effective at removing this watermark because a dead-code optimisation will remove the watermark code.

3.2.14 Dash-O Pro

Dash-O Pro embeds a watermark by obfuscating and inserting extra code - each of the class files are renamed and some code is added to each class file. The extra code is added to the class initialisation method and is therefore invoked implicitly by the Java Virtual Machine. The extra code includes the ability to 'expire' the software if this option is activated in the Dash-O Pro configuration.

3.3 The Transformation Attacks

Sandmark contains a variety of semantics preserving obfuscations which we will use to evaluate the watermarking systems. We also use Proguard [34] to optimise the test programs, as another form of obfuscation. In total, there are 37 different transformations to be applied.

3.4 The Jar files

All the jar files that we use in the tests are plugins for the open-source text editor jEdit [59]. These files are fairly small (average 30KB) but represent a collection of real-world Java software[1]. The range of plugins represent a variety of code, and were all written by different

[1]we found that larger files cause problems with Sandmark's obfuscator resulting in crashes and/or extremely long embed times

Table 3: Test file statistics.

Filename	Size (KB)	Classes	Methods	Fields	Locals
Accents	18.4	6	33	16	96
Activator	21.1	17	86	47	212
Ancestor	4.9	5	16	9	48
AxisHelper	12.9	7	39	33	90
Background	11.0	6	35	26	100
BufferLocal	13.1	5	31	20	105
BufferSelector	15.2	9	44	29	110
CheckStylePlugin	4.7	3	19	9	47
CodeLint	12.4	3	19	11	82
CommentFolder	3.3	2	4	3	15
CommonControls	271.6	62	436	218	1189
ConfigurableFoldHandler	24.0	15	83	53	223
ContextHelp	16.7	2	21	48	87
ContextMenu	20.3	11	64	32	136
DBTerminal	23.6	22	101	56	203
Dict	10.9	6	40	31	89
GroovyScriptEnginePlugin	3.6	1	5	1	5
HelperLauncher	7.3	4	23	10	63
HexEdit	22.7	27	137	35	321
Hyperlinks	17.0	18	74	34	194
IncludesParser	22.3	15	51	43	123
InformSideKick	24.2	10	78	82	252
JFuguePlugin	24.7	12	51	12	84
JNAPlugin	1.9	1	3	0	3
JVMStats	4.8	4	11	16	35
JalopyPlugin	22.2	22	70	13	117
JavaFold	5.2	3	10	9	50
JavaInsight	26.6	10	52	20	237
JavaScriptShell	27.6	6	38	7	103
JavascriptScriptEnginePlugin	3.6	1	5	1	5
JcrontabPlugin	19.1	11	52	35	133
JinniConsole	7.9	5	37	13	86
LineGuides	15.3	8	57	24	156
LispPaste	8.4	7	25	20	68
MacOSX	8.9	4	29	8	94
MetalColor	9.0	4	36	40	68
MibSideKick	8.5	6	23	9	60
MouseSnap	4.6	1	8	3	19
MyDoggyPlugin	24.1	17	94	46	237
Nested	15.9	12	47	23	132
NetComponents	17.8	21	137	49	336
Optional	15.4	11	68	29	226
Outline	4.6	4	13	7	33
PerlSideKick	7.2	2	4	10	15
ProjectViewer	712.1	169	1103	523	3004
PrologConsole	17.6	3	22	7	60
RETest	16.9	6	54	31	121
RecentBufferSwitcher	10.6	6	31	9	87
RecursiveOpen	8.3	4	17	9	48
Rename	5.0	6	18	14	49
SaxonAdapter	7.6	3	20	13	67
SaxonPlugin	26.3	1	1	0	1
ScriptEnginePlugin	21.2	7	54	21	166
SendBuffer	5.5	2	7	11	34
ShortcutDisplay	10.9	9	39	18	85
Sudoku	16.8	17	62	50	221
SuperScript	27.6	14	70	39	201
SwitchBuffer	22.3	17	66	43	171
TableLayout-20050920	10.1	5	70	47	296
TomcatSwitch	17.9	7	60	45	159
Average	30.0	11	66	35	180

programmers but as they are plugins they share some characteristics. For example, some classes may subclass jEdit's abstract plugin classes to use jEdit's plugin API. All the test files were obtained by installing jEdit and then using the built-in plugin manager to download the plugin jar files. The average number of classes per jar is 11, while the average number of methods per jar is 66. The average number of fields 35 and the average number of local variables is 180.

The biggest program jar was 712.1KB while the smallest was 1.9KB. The largest program jar had 169 classes and the smallest had only 1. Two programs had no fields while the largest program contained 523. The largest program contained 3004 locals variables. Further program statistics can be found in table 3.

4 Results

4.1 Watermarking

After embedding watermarks we obtained 671 out of an expected 840 watermarked jars. Some watermark algorithms failed to embed the specified watermark, due to error or incompatible program jar. For example, Qu/Potkonjak could only embed watermarks in 1 of the programs because the class files were too small for the watermark. Allatori, String Constant and Add Expression managed to correctly embed watermarks in all 60 test programs - they were embedded and recognised correctly. Only 79.9% of the expected watermarked jar files were actually produced (see figure 3).

Out of the 671 watermarked jar files only 588 contained watermarks which were successfully recognised before the transformation attacks were applied. This means only 87.6% of the watermarks in the watermarked jar files produced were actually recognised (see figure 3).

4.2 Obfuscation

We obfuscated the 671 jar files with 36 obfuscations, 1 optimisation and 2 obfuscation combinations which should have resulted in 26,169 attacked watermarked jars. Some algorithms failed to output some jars so we actually obtained 23,626 attacked watermarked jars using 39 semantics preserving transformations. We believe this is due to bugs in the implementation rather than a fundamental problem with the algorithms. This means only 90.3% of the expected attacked watermarked jar files were actually produced (see figure 3).

4.3 Recognition

The result of recognising the watermarks in the obfuscated jar files are shown in table 5. The number of successful recognitions before transformations is shown in the first column, while the remaining columns show the number of successful recognitions after transformations.

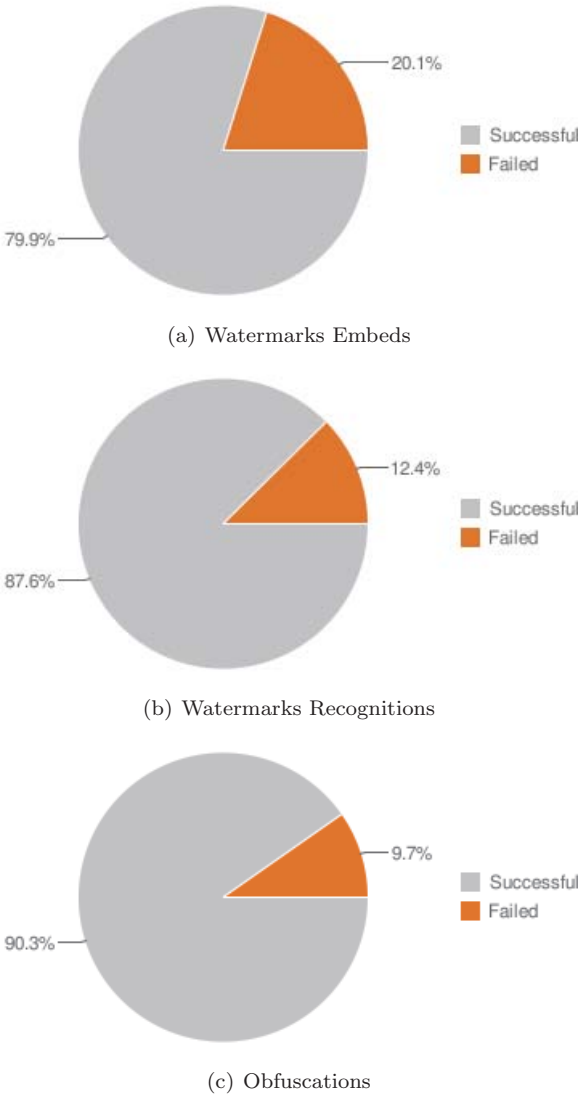

Figure 3: Watermark and Obfuscation success. Out of the 840 expected watermarked jars, only 671 were produced by the watermarkers (a), while only 588 of these were correctly recognised (b). Out of the 26,169 expected attacked watermarked jars only 23,626 were produced (c).

A number of zeros can be seen throughout the table indicating that no watermarks was recognised with that combination of the watermark and transformation. These are the combinations of watermark and transformation that we are interested.

4.4 Analysis

By examining the table we can see that Proguard Optimizer produces the best results overall - with a low number of recognitions for all watermarkers, except String Constant.

Some obfuscation algorithms remove certain watermarks better than others, for example 'Rename Registers' removes all of the 'Add Expression' watermarks – this is because this attack renames the variables and the watermarking system can no longer identify which are the bogus variables. Other obfuscations which affect variable names also remove these watermarks.

The 'Add Initialization' watermark was easily defeated by the 'Merge Local Integers' because this watermarking algorithm stores parts in a bogus addition expressions. The 'Irreducibility' obfuscation inserts jumps into a method, protected by opaque predicates, which makes the control flow graph irreducible – this greatly effects the 'Graph Theorectic Watermark' which stores a watermark in a reducible control-flow graph in the program. The 'Moden' watermarking algorithm is effected by any transformation to the watermarked method – this is because the watermark is encoded in the syntactic properties of that method.

Allatori, one of the commercial systems performed well under most obfuscations but was easily defeated by the optimiser due to it's use of 'dead-code' to embed the watermark. The watermarking algorithm 'Static Arboit' uses opaque predicates to encode the watermark which are not resilient to obfuscations which introduce other opaque predicates.

We can also see that some of the transformations remove some of watermarks completely. We therefore used a combination of well performing watermarks to remove more watermarks overall (see table 4).

The results of running this combination of transformations are shown at the end of table 5, in the 'Combo 1' column. This removes many of the watermarks, leaving just 71 remaining and some watermark algorithms with no remaining watermarks. We then generated 'Combo 2' by selecting transformations which contained files in 'Combo 1' but which had the watermark removed.'Combo 2' removed some more of the remaining watermarks resulting in just 53 files containing watermarks and the Add Switch, Davidson/Myhrvold, Monden and Allatori watermark algorithms completely defeated, compared to 'Combo 1'.

There are still 52 watermarks recognisable after Combo 2 using the 'String Constant' watermark algorithm but these can easily be removed. The 'String Constant' algorithm creates a new, unused entry in a class-file's constant pool containing the watermark value. The constant is not used within the code of classfile therefore we can easily remove it with a simple static analysis and therefore remove the 52 'String Constant' watermarks.

Table 4: The combinations of transformations used for Combo 1 and Combo 2.

Transformation	**Combo 1**	**Combo 2**
Array Folder		
Array Splitter		
Block Marker		
Constant Pool Reorderer		✓
Dynamic Inliner		
FalseRefactor		
Integer Array Splitter		
Interleave Methods		✓
Overload Names	✓	✓
ParamAlias		
Rename Registers	✓	✓
Split Classes		✓
String Encoder		✓
Class Splitter		✓
Field Assignment		
Method Merger		
Objectify		
Publicize Fields		
Simple Opaque Predicates		✓
Static Method Bodies		
Bludgeon Signatures		
Boolean Splitter		✓
Branch Inverter		
Duplicate Registers		
Insert Opaque Predicates		✓
Irreducibility		✓
Merge Local Integers		✓
Opaque Branch Insertion		✓
Promote Primitive Registers	✓	✓
Promote Primitive Types		
Random Dead Code		
Reorder Instructions		
Reorder Parameters		
Transparent Branch Insertion		
Variable Reassigner		✓
Inliner		✓
Proguard Optimize	✓	✓

The last remaining watermarked file contains an 'Add Method and Field' watermark. This jar file caused the obfuscators to crash and therefore could not be obfuscated. We believe that this happened due to bugs in obfuscation implementations rather than a fundamental problem with the algorithms. We therefore suggest that this remaining watermark could be removed if the obfuscation implementations were corrected.

A watermarking system can fail in two ways: it fails to embed the watermark, or the watermark is easy to remove. A good watermarking system is one where embedding succeeds often and the watermark is not often removed. Our results show that the static watermarking systems performed badly at embedding and watermarks were easily removed.

5 Conclusion

We confirmed that none of the 14 static watermark algorithms are resilient to semantics preserving transformations. Our results compare similarly with previous evaluations of some of the static watermarking algorithms. A combination of transformations removed all but 52 'String Constant' watermarks and 1 'Add Method and Field' watermark from the test files. 52 of the remaining watermarks can be destroyed by removing (or overwriting) unused constants in a class-file's constant pool. The last watermarked file was rejected by some of the obfuscations and we assume that the watermark in this file would be removed if the bugs in the obfuscations were fixed.

Software watermarking must be supplemented with other forms of protection [53], such as obfuscations or tamperproofing techniques [21], in order to better protect a program from copyright infringement and decompilation.

Though we have not evaluated all aspects of the watermarking algorithms, we have shown that static watermarks are insufficient to prove ownership of software due to their lack of resilience to semantics preserving transformations.

5.1 Future Work

Further work will involve extending the evaluation to dynamic watermarks which, in theory, should be resilient to semantics preserving transformations. However, it has been shown that at least one dynamic algorithm is only minimally stronger than the static version [43]. We intend to investigate this claim and extend the investigation to evaluate other dynamic watermarking algorithms and their advantages over static algorithms. Furthermore, we plan to evaluate more factors such as runtime and embedding costs, and stealthiness. Additionally, we intend to look at the use of program slicing techniques [58] in order to perform subtractive watermark attacks.

Table 5: Evaluation results - along the top is the name of the transformation performed and along the left is the name of the watermark system.

	Original	Array Folder	Array Splitter	Block Marker	Constant Pool Reorderer	Dynamic Inliner	FalseRefactor	Integer Array Splitter	Interleave Methods	Overload Names	ParamAlias	Rename Registers	Split Classes	String Encoder	Class Splitter	Field Assignment	Method Merger	Objectify	Publicize Fields	Simple Opaque Predicates	Static Method Bodies	Bludgeon Signatures	Boolean Splitter	Branch Inverter	Duplicate Registers	Insert Opaque Predicates	Irreducibility	Merge Local Integers	Opaque Branch Insertion	Promote Primitive Registers	Promote Primitive Types	Random Dead Code	Reorder Instructions	Reorder Parameters	Transparent Branch Insertion	Variable Reassigner	Inliner	Proguard Optimize	Combo 1	Combo 2
Add Expression	60	60	60	60	57	60	60	60	57	60	60	0	28	60	60	60	60	60	60	60	24	60	60	60	60	60	59	56	60	0	7	47	60	60	59	1	10	2	0	0
Add Initialization	56	56	56	56	54	56	56	56	55	56	56	56	56	56	44	56	56	56	55	56	55	56	56	5	56	31	44	25	32	44	46	56	58	10	51	48	56	1	0	0
Add Method and Field	35	35	35	35	33	30	35	35	35	7	6	34	35	29	35	23	32	35	35	35	35	35	35	35	35	0	0	0	0	0	0	0	0	0	0	0	0	6	1	0
Add Switch	59	59	59	59	55	59	59	59	59	59	59	59	59	59	58	59	59	59	59	59	59	59	59	59	59	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Davidson/Myhrvold	15	15	12	14	13	15	15	15	15	12	15	15	7	8	15	15	15	13	15	15	15	11	12	13	8	7	15	11	6	4	4	14	13	13	9	2	13	8	3	0
Graph Theoretic Watermark	47	47	47	47	45	45	47	47	29	47	47	47	46	47	47	47	47	47	47	47	47	47	47	47	47	33	0	47	1	0	5	47	47	47	2	47	45	0	0	0
Monden	58	58	58	56	55	58	58	58	28	58	58	58	58	58	58	58	58	58	58	58	58	48	55	56	56	31	44	25	32	44	46	56	58	58	58	58	58	56	7	5	0
Qu/Potkonjak	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Register Types	51	51	51	51	49	49	51	51	50	9	51	0	15	51	32	51	51	6	51	51	5	51	51	51	8	51	51	51	51	51	51	51	51	51	51	1	8	1	0	0
Static Arboit	19	19	19	19	18	12	19	19	3	19	19	19	19	19	19	19	1	19	19	0	19	19	19	19	19	3	19	19	0	0	2	19	19	19	10	19	2	2	0	0
Stern	46	43	46	45	44	44	45	46	39	45	46	45	45	46	45	45	45	45	45	1	45	45	46	45	45	10	26	16	42	0	5	22	46	45	45	45	42	2	0	0
String Constant	60	60	60	60	57	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	55	52
Dash-O Pro	22	4	11	0	8	8	0	2	6	4	0	0	4	11	6	0	2	0	1	0	2	9	0	0	0	10	9	0	0	0	2	0	9	2	0	1	7	22	0	0
Allatori	60	60	60	60	57	54	60	60	59	60	60	59	60	60	60	60	60	60	60	60	60	59	60	60	60	60	60	56	60	60	60	60	59	59	59	60	58	1	1	0

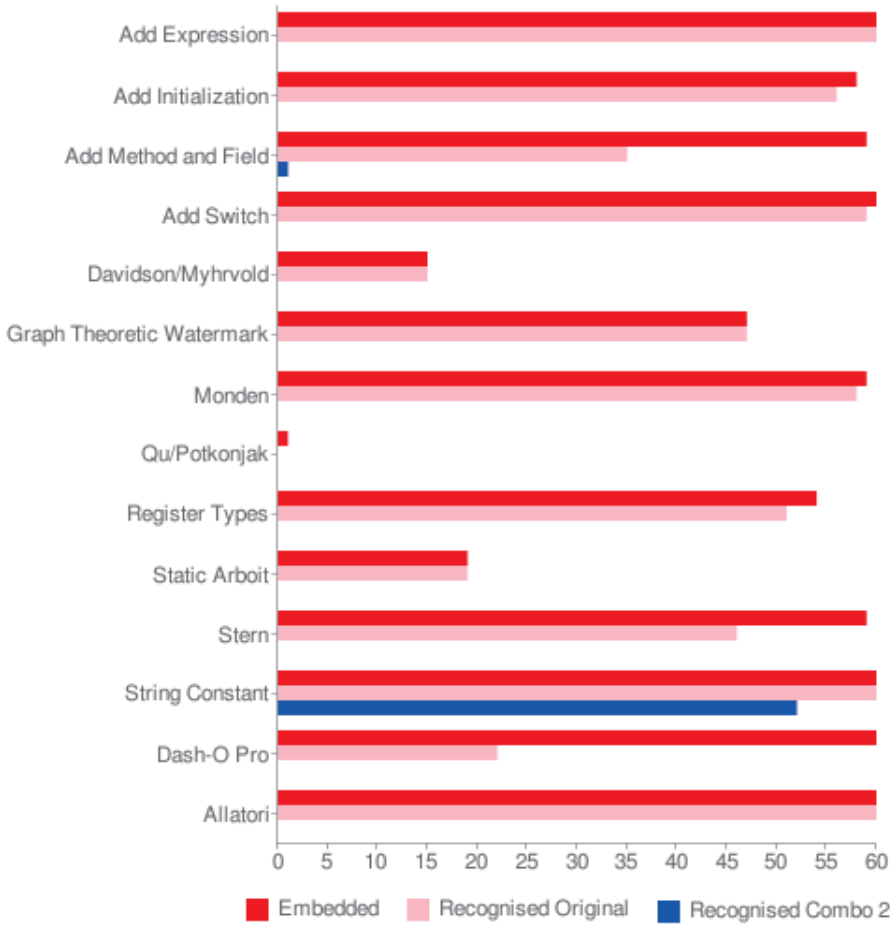

Figure 4: The number of files in which watermarks were correctly recognised.

References

[1] DashO, 2010. URL `http://www.preemptive.com/products/dasho/overview`. Accessed: 2 April, 2010.

[2] Preemptive solutions, 2010. URL `http://www.preemptive.com/`. Accessed: 26 April, 2010.

[3] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006. ISBN 0321486811.

[4] Jieqing Ai, Xingming Sun, Yunhao Liu, Ingemar J. Cox, Guang Sun, and Yi Luo. A stern-based Collusion-Secure software watermarking algorithm and its implementation. In *Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering*, pages 813–818. IEEE Computer Society, 2007. ISBN 0-7695-2777-9.

[5] Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Covert communication through executables. In *Program Acceleration through Application and Architecture Driven Code Transformations: Symposium Proceedings*, pages 83–85, 2004.

[6] Genevieve Arboit. A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[7] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 222–231, Chicago, IL, USA, 2004. ACM. ISBN 1-58113-852-0.

[8] Business Software Alliance. Sixth annual BSA and IDC global software piracy study. Technical Report 6, Business Software Alliance, 2008.

[9] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981. ISSN 0096-0551. doi: DOI:10.1016/0096-0551(81)90048-5.

[10] Balamurgan Chirtsabesan and Tapas Ranjan Sahoo. BogusExpression static watermarking algorithm. Sandmark documentation, 2004.

[11] Maria Chroni and Stavros D. Nikolopoulos. Encoding watermark integers as self-inverting permutations. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, pages 125–130, Sofia, Bulgaria, 2010. ACM. ISBN 978-1-4503-0243-2.

[12] C. Collberg, A. Huntwork, E. Carter, and G. Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. In *Workshop on Information Hiding*, 2004.

[13] Christian Collberg. Sandmark, August 2004. URL `http://www.cs.arizona.edu/sandmark/`. Accessed: 2 April, 2010.

[14] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009. ISBN 0321549252, 9780321549259.

[15] Christian Collberg and Tapas Ranjan Sahoo. Software watermarking in the frequency domain: implementation, analysis, and attacks. *J. Comput. Secur.*, 13(5):721–755, 2005.

[16] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, January 1999.

[17] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, January 1998.

[18] Christian Collberg, Miriam Miklofsky, Ginger Myles, Ashok Purushotham, RathnaPrabhu Rajendran, Andrew Huntwork, Xiangyu Zhang, Danny Mandel, Anna Segurson, Martin Stepp, Kelly Heffner, J Nagra, G Townsend, Balamurugan Chirtsabesan, and Tapas Ranjan Sahoo. Sandmark algorithms. Sandmark documentation, University of Arizona, July 2002.

[19] Christian Collberg, Stephen Kobourov, Edward Carter, and Clark Thomborson. Error-Correcting graphs for software watermarking. In *Proceedings of the 29th Workshop on Graph Theoretic Concepts in Computer Science*, pages 156–167, 2003.

[20] Christian Collberg, Andrew Huntwork, Edward Carter, Gregg Townsend, and Michael Stepp. More on graph theoretic software watermarks: Implementation, analysis, and attacks. *Inf. Softw. Technol.*, 51(1):56–67, 2009.

[21] Christian S. Collberg and Clark Thomborson. Watermarking, Tamper-Proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, page 735746, August 2002.

[22] Christian S. Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29 (6):35, 2007.

[23] Ingemar J. Cox, Joe Kilian, Frank Thomson Leighton, and Talal Shamoon. A secure, robust watermark for multimedia. In *Proceedings of the First International Workshop on Information Hiding*, pages 185–206. Springer-Verlag, 1996. ISBN 3-540-61996-8.

[24] D. Curran, N.J. Hurley, and M. O. Cinneide. Securing java through software watermarking. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, page 311324, 2003.

[25] Robert Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program, June 1996. Microsoft Corporation, US Patent 5559884.

[26] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, St. Petersburg Beach, Florida, United States, 1996. ACM. ISBN 0-89791-769-3.

[27] 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.

[28] Maurice H Halstead. *Elements of software science (Operating and programming systems series)*. Elsevier, 1977. ISBN 0444002057. Published: Hardcover.

[29] James Hamilton and Sebastian Danicic. An evaluation of current java bytecode decompilers. In *Ninth IEEE International Workshop on Source Code Analysis and Manipulation*, volume 0, pages 129–136, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.

[30] James Hamilton and Sebastian Danicic. An evaluation of static java bytecode watermarking. In *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2010*, volume 1, pages 1 – 8, San Francisco, USA, October 2010. ISBN 978-988-17012-0-6. Winner of the Best Student Paper Award.

[31] Kazuhiro Hattanda and Shuichi Ichikawa. The evaluation of davidsons digital signature scheme. *IEICE Trans. Fundamentals*, E87A(1), January 2004.

[32] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, Denver, Colorado, United States, 1972. ACM.

[33] Keith Holmes. Computer software protection, February 1994. International Business Machines Corporation, US Patent: 5287407.

[34] Eric Lafortune et al. ProGuard, July 2009. URL `http://proguard.sourceforge.net/`. Accessed: 10 April, 2010.

[35] Tri Van Le and Yvo Desmedt. Cryptanalysis of UCLA watermarking schemes for intellectual property protection. In *Revised Papers from the 5th International Workshop on Information Hiding*, pages 213–225. Springer-Verlag, 2003. ISBN 3-540-00421-1.

[36] Anshuman Mishra, Rajeev Kumar, and P. P. Chakrabarti. A method-based Whole-Program watermarking scheme for java class files. 2008.

[37] Akito Monden. jmark, 2003. URL `http://se.aist-nara.ac.jp/jmark/`. Accessed: 14 July, 2010.

[38] Akito Monden, Hajimu Iida, et al. A watermarking method for computer programs. In *Proceedings of the 1998 Symposium on Cryptography and Information Security, SCIS'98*. Institute of Electronics, Information and Communication Engineers, January 1998. in Japanese.

[39] Akito Monden, Hajimu Iida, Ken ichi Matsumoto, Katsuro Inoue, and Koiji Torii. Watermarking java programs. In *International Symposium on Future Software Technology '99*, pages 119–124, October 1999.

[40] Akito Monden, Hajimu Iida, Ken ichi Matsumoto, Koji Torii, and Katsuro Inoue. A practical method for watermarking java programs. In *COMPSAC '00: 24th International Computer Software and Applications Conference*, pages 191–197, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0792-1.

[41] Ginger Myles. Using software watermarking to discourage piracy. *Crossroads - The ACM Student Magazine*, 2004. URL `http://www.acm.org/crossroads/xrds10-3/watermarking.html`. Accessed: 21 March, 2009.

[42] Ginger Myles and Christian Collberg. Software watermarking through register allocation: Implementation, analysis, and attacks. In *International Conference on Information Security and Cryptology*, volume 2971/2004 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-21376-5.

[43] Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. In *ICECR-7*, 2004.

[44] Ginger Myles, Christian Collberg, Zachary Heidepriem, and Armand Navabi. The evaluation of two software watermarking algorithms. *Softw. Pract. Exper.*, 35(10):923938, 2005. ISSN 0038-0644.

[45] Jasvir Nagra, Clark Thomborson, and Christian Collberg. A functional taxonomy for software watermarking. In Michael J. Oudshoorn, editor, *Aust. Comput. Sci. Commun.*, pages 177–186, Melbourne, Australia, 2002. ACS.

[46] Gang Qu and Miodrag Potkonjak. Analysis of watermarking techniques for graph coloring problem. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 190–193, San Jose, California, United States, 1998. ACM. ISBN 1-58113-008-2.

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

[48] Gang Qu and Miodrag Potkonjak. Fingerprinting intellectual property using constraint-addition. In *Design Automation Conference*, pages 587–592, 2000.

[49] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[50] Peter R. Samson. Apparatus and method for serializing and validating copies of computer software, February 1994.

[51] Smardec. Software development and information technology offshore outsourcing company, 2008. URL `http://www.smardec.com/`.

[52] Smardec. Allatori java obfuscator, September 2009. URL `http://www.allatori.com/`. Accessed: 2 April, 2010.

[53] Jose Sogiros. Is protection software needed watermarking versus software security, March 2010. URL `http://bb-articles.com/watermarking-versus-software-security`. Accessed: 13 April, 2010.

[54] Julien Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. Robust object watermarking: Application to code. In *Information Hiding Workshop '99*, pages 368–378, 1999.

[55] Ramarathnam Venkatesan and Vijay Vazirani. Technique for producing through watermarking highly tamper-resistant executable code and resulting watermarked code so formed, May 2006. Microsoft Corporation, US Patent: 7051208.

[56] Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *Proceedings of the 4th International Workshop on Information Hiding*, 2001.

[57] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996. ISBN 0079132480.

[58] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, page 439449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6.

[59] world-wide developer team. jEdit - programmer's text editor, 2010. URL `http://www.jedit.org/`. Accessed: 10 April, 2010.

[60] William Zhu and Clark Thomborson. Algorithms to watermark software through register allocation. In *Digital Rights Management. Technologies, Issues, Challenges and Systems*, volume 3919 of *Lecture notes in computer science*, pages 180–191, Berlin, Allemange, 2006. Springer. ISBN 978-3-540-35998-2.

[61] William Zhu and Clark Thomborson. Extraction in software watermarking. In Sviatoslav Voloshynovskiy, Jana Dittmann, and Jessica J. Fridrich, editors, *MM&Sec*, pages 175–181. ACM, 2006. ISBN 1-59593-493-6.

[62] William Zhu and Clark Thomborson. Recognition in software watermarking. In *Proceedings of the 4th ACM international workshop on Contents protection and security*, pages 29–36, Santa Barbara, California, USA, 2006. ACM. ISBN 1-59593-499-5.

[63] William Feng Zhu. *Concepts and Techniques in Software Watermarking and Obfuscation*. PhD thesis, The University of Auckland, 2007.