An Evaluation of Static Java Bytecode Watermarking

James Hamilton and Sebastian Danicic *

Abstract—**-The rise in the use of the Internet and bytecode languages such as Java bytecode and Microsoft's Common Intermediate Language have made copying, decompiling and disassembling software easier. The global revenue loss due to software piracy was estimated to be more than $50 billion in 2008. Watermarking is a technique which attempts to protect software by inserting copyright notices or unique identifiers into software to prove ownership. We 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. 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

Software theft, also known as software piracy, is the act of copying a legitimate application and illegally distributing that software, either free or for profit. Legal methods to protect software producers such as copyright laws, patents and license agreements [1] do not always dissuade people from stealing software, especially in emerging markets where the price of software is high and incomes are low. Ethical arguments, such as fair compensation for producers, by software manufacturers, law enforcement agencies and industry lobbyists also do little to counter software piracy. The global revenue loss due to software piracy was estimated to be more than $50 billion in 2008 [2].

Software watermarking involves embedding a unique identifier within a piece of software, to discourage software theft. Watermarking does not prevent theft but instead discourages software thieves by providing a means to identify the owner of a piece of software and/or the origin of the stolen software [3]. The hidden watermark can be extracted, at a later date, by the use of a *recogniser* to prove ownership of stolen software. It is also 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. It is also necessary (in most cases), that the watermark is *robust* - that is, resilient to semantics preserving transformations (such as optimisations or obfuscations).

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.

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 [4].

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. We present a survey 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. 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 [5].

*Submitted August 2010. Department of Computing, Goldsmiths, University of London, United Kingdom, james.hamilton@gold.ac.uk, s.danicic@gold.ac.uk

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

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 [6]) 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 [7]. Ideally, software watermarks should be resilient to decompilation-recompilation attacks, as decompilation of Java is possible (though not perfect [4]).

The watermark code must be locally indistinguishable from the rest of the program so that it is hidden from adversaries [8]. 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 [9, 10]). It could be difficult to programatically generate code which is indecipherable from the human-generated program code but statisical analysis of the original program could help in generating suitable watermarks [7].

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.* [11] 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.* [12] found that the size of a program, watermarked with Davidson/Myhrvold [13] 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' [14], 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 [14]. Such attacks rely on the repetitive execution of a recogniser thus fast recognition time helps an adversary.

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

2.2 Types of Watermark

Software watermarks can be broadly divided into two categories: static and dynamic [15]. 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. Additionally, Nagra *et al.* define four categories of watermark [14]:

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 evalaute 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 [16].

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.* [16] 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.

3 Emperical 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.

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 only available systems that we could obtain for watermarking Java programs.

Some of the algorithms have been evaluated before (for example Collberg *et al.* have compared the Davidson/Myhrvold and Monden algorithms [17]) and we re-evaluate them with our set of test programs. To the best of our knowledge, the commercial watermarking systems, Allatori and DashO, haven't been evaluated before.

3.1.1 Sandmark

SandMark [6] 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 [6]. We used version 3.4.0 released in 2004.

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

3.1.2 Allatori

Allatori [18] is a commercial Java obfuscator complete with a watermarking system created by Smardec [19]. 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' [18]. We used version 2.8 released in 2009.

3.1.3 DashO

DashO [20] is a commerical Java security solution, including obfuscator, watermarking and encrypter - similar to Allatori. DashO is made by PreEmptive Solutions [21] 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[1] static Java bytecode watermarking algorithms [22]:

1. **Add Expression** adds a bogus addition expression containing the watermark to a class-file.
2. **Add Initialization** adds bogus local variables to a method in a class-file.
3. **Add Method and Field** splits a watermark in two - one half 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.
4. **Add Switch** embeds the watermark in the case values of a switch statement, inserted at the beginning of a randomly chosen method.
5. **Davidson/Myhrvold** [13] embeds the watermark by re-ordering basic blocks in a suitable method. A previous study found that this algorithm is susceptible to semantics-preserving transformation attacks [17].
6. **Graph Theoretic Watermark** [8] embeds the watermark in a control-flow graph, which is added to the original program. It has previously been shown that if fewer than about half of the blocks in the watermarked application are modified this watermark survives [23].
7. **Monden** [11] embeds the watermark by replacing opcodes in a dummy method, generated by Sandmark. A previous study found that this algorithm is susceptible to semantics-preserving transformation attacks [17].
8. **Qu/Potkonjak** [24,25] embeds the watermark in local variable assignments by adding constraints to the interference graphs. Collberg *et al.* [26] implemented their version of the QP algorithm, which they call QPS, in Sandmark.
9. **Register Types** embeds a watermark by introducing local variables of certain Java standard library types.
10. **Static Arboit** [16, 27] embeds a method by encoding the watermark in an opaque predicate and then appending the predicate to a selected branch.
11. **Stern (Robust Object Watermarking)** [28] embeds the watermark as a statistical object by creating a frequency vector representation of the code. A previous study found that the watermark can survive many high-level transformations that affect classes, fields, and method signatures [29].
12. **String Constant** is a simple watermarking algorithm which simply embeds the watermark string into the constant pool of a class-file.

The two commercial watermarking systems contain one algorithm each:

13. **Allatori** embeds watermarks as a sequence of *push* and *pop* operations inserted into multiple class-file methods.
14. **Dash-O Pro** renames classes and inserts some extra static code in each of the class-files.

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 [30] 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 [31]. These files are fairly small (average 30KB) but represent a collection of real-world Java software[2]. The range of plugins represent a variety of code, and were all written by different

[1] A 13th static algorithm is included, but not counted here - Steganography. This algorithm stores a watermark within PNG files in the program jar file.

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

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

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.

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 1).

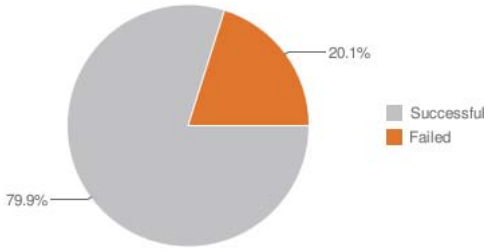
Out of the 671 watermarked jar files only 588 contained watermarks which were successfully recognised before the transformation attacks were applied. This means only 87.6% of the watermarks in the watermarked jar files produced were actually recognised (see figure 1).

4.2 Obfuscation

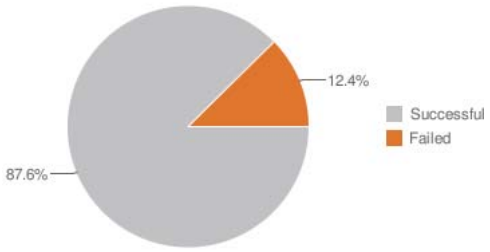
We obfuscated the 671 jar files with 36 obfuscations, 1 optimisation and 2 obfuscation combinations which should have resulted in 26,169 attacked watermarked jars. Some algorithms failed to output some jars so we actually obtained 23,626 attacked watermarked jars using 39 semantics preserving transformations. 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 1).

4.3 Recognition

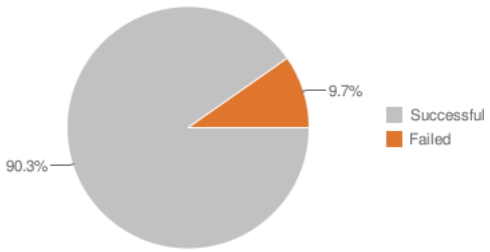
The result of recognising the watermarks in the obfuscated jar files are shown in table 2. The number of successful recognitions before transformations is shown in the first column, while the remaining columns show the number of successful recognitions after transformations.



(a) Watermarks Embeds



(b) Watermarks Recognitions



(c) Obfuscations

Figure 1: 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. We can also see that some of the other transformations remove some of the other watermarks completely. We therefore used a combination of well performing watermarks to remove more watermarks overall (see table 1).

The results of running this combination of transformations are shown at the end of table 2, in the 'Combo 1' col-

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Table 1: 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	✓	✓

umn. 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. We can easily remove unused constant pool items with a simple static analysis and therefore remove the 52 String Constant watermarks.

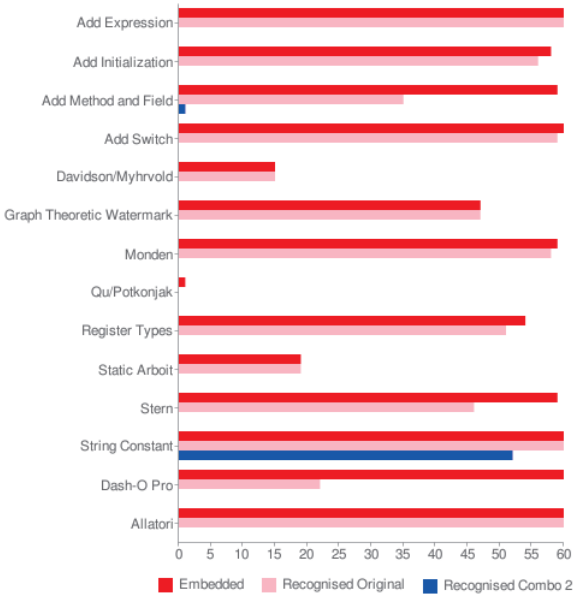


Figure 2: The number of files in which watermarks were correctly embedded and recognised.

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

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Table 2: 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	56	55	55	56	55	56	56	56	56	0	0	0	7	56	10	51	48	56	56	1	0	0
Add Method and Field	35	35	35	35	33	30	35	35	7	6	34	35	29	35	23	32	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	27	35	6	1
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	59	59	59	59	59	59	59	59	59	59	59	59	1	1	0
Davidson/Myhrvold	15	15	12	14	13	15	15	15	12	15	12	15	7	8	15	15	15	13	15	15	15	11	12	13	8	7	15	11	6	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	1	47	58	47	47	47	33	0	47	1	0	5	47	47	2	47	45	0	0	0	0
Monden	58	56	58	56	55	58	58	58	28	58	58	58	58	58	58	58	57	58	58	7	58	58	48	55	56	31	44	25	32	44	46	56	58	58	58	56	7	5	0	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	51	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	10	19	2	2	0	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	42	2	0	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	55	52	
Dash-O Pro	22	4	11	0	8	8	0	2	6	4	0	0	4	11	6	0	0	2	0	1	0	2	9	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	60	56	60	60	60	60	59	59	58	1	0	0	

forms of protection [32], such as obfuscations or tamper-proofing techniques [33], 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 [16]. 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 [34] in order to perform subtractive watermark attacks.

References

[1] G. Cronin, "A taxonomy of methods for software piracy prevention," Department of Computer Science, University of Auckland, New Zealand, Tech. Rep., 2002.

[2] B. S. Alliance, "Sixth annual BSA and IDC global software piracy study," Business Software Alliance, Tech. Rep. 6, 2008.

[3] G. Myles, "Using software watermarking to discourage piracy," *Crossroads - The ACM Student Magazine*, 2004. [Online]. Available: http://www.acm.org/crossroads/xrds10-3/watermarking.html

[4] J. Hamilton and S. Danicic, "An evaluation of current java bytecode decompilers," in *Ninth IEEE International Workshop on Source Code Analysis and Manipulation*, vol. 0. Edmonton, Alberta, Canada: IEEE Computer Society, 2009, pp. 129–136.

[5] W. F. Zhu, "Concepts and techniques in software watermarking and obfuscation," PhD Thesis, The University of Auckland, 2007.

[6] C. Collberg, "Sandmark," Department of Computer Science, Aug. 2004. [Online]. Available: http://www.cs.arizona.edu/sandmark/

[7] A. Mishra, R. Kumar, and P. P. Chakrabarti, "A method-based Whole-Program watermarking scheme for java class files," 2008.

[8] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *Proceedings of the 4th International Workshop on Information Hiding*, 2001.

[9] M. H. Halstead, *Elements of software science (Operating and programming systems series)*. Elsevier, 1977, published: Hardcover.

[10] Kearney, Sedlmeyer, Thompson, Gray, and Adler, "Software complexity measurement," *Commun. ACM*, vol. 29, no. 11, p. 10441050, 1986.

[11] A. Monden, H. Iida, K. ichi Matsumoto, K. Torii, and K. Inoue, "A practical method for watermarking java programs," in *COMPSAC '00: 24th International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2000, p. 191197.

[12] K. Hattanda and S. Ichikawa, "The evaluation of davidsons digital signature scheme," *IEICE Trans. Fundamentals*, vol. E87A, no. 1, 2004.

[13] R. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," Jun. 1996, microsoft Corporation, US Patent 5559884.

[14] J. Nagra, C. Thomborson, and C. Collberg, "A functional taxonomy for software watermarking," in *Aust. Comput. Sci. Commun.*, M. J. Oudshoorn, Ed. Melbourne, Australia: ACS, 2002, pp. 177–186.

[15] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in *Principles of Programming Languages 1999, POPL'99*, 1999.

[16] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," in *ICECR-7*, 2004.

[17] G. Myles, C. Collberg, Z. Heidepriem, and A. Navabi, "The evaluation of two software watermarking algorithms," *Softw. Pract. Exper.*, vol. 35, no. 10, p. 923938, 2005.

[18] Smardec, "Allatori java obfuscator," Sep. 2009, 2009. [Online]. Available: http://www.allatori.com/

[19] "Smardec - software development and information technology offshore outsourcing company," 2008. [Online]. Available: http://www.smardec.com/

[20] "DashO," 2010. [Online]. Available: http://www.preemptive.com/products/dasho/overview

[21] "Preemptive solutions," 2010. [Online]. Available: http://www.preemptive.com/

[22] C. Collberg, "Sandmark algorithms," University of Arizona, Department of Computer Science, Tech. Rep., Jul. 2002.

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

[24] G. Qu and M. Potkonjak, "Hiding signatures in graph coloring solutions," in *Information Hiding*, 1999, pp. 348–367.

[25] G. Qu and M. Potkonjak, "Analysis of watermarking techniques for graph coloring problem," in *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. San Jose, California, United States: ACM, 1998, pp. 190–193.

[26] G. Myles and C. Collberg, "Software watermarking through register allocation: Implementation, analysis, and attacks," in *International Conference on Information Security and Cryptology*, 2003.

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

[28] J. Stern, G. Hachez, F. Koeune, and J. Quisquater, "Robust object watermarking: Application to code," in *Information Hiding Workshop '99*, 1999, pp. 368–378.

[29] C. Collberg and T. R. Sahoo, "Software watermarking in the frequency domain: implementation, analysis, and attacks," *J. Comput. Secur.*, vol. 13, no. 5, pp. 721–755, 2005.

[30] E. Lafortune *et al.*, "ProGuard," Jul. 2009. [Online]. Available: http://proguard.sourceforge.net/

[31] world-wide developer team, "jEdit - programmer's text editor," 2010. [Online]. Available: http://www.jedit.org/

[32] J. Sogiros, "Is protection software needed watermarking versus software security," Mar. 2010. [Online]. Available: http://bb-articles.com/watermarking-versus-software-security

[33] C. S. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and obfuscation - tools for software protection," in *IEEE Transactions on Software Engineering*, vol. 28, Aug. 2002, p. 735746.

[34] M. Weiser, "Program slicing," in *ICSE '81: Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1981, p. 439449.

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)