A Method for Watermarking Java Programs via Opaque Predicates

Extended Abstract

Geneviève Arboit

McGill University
School of Computer Science, Crypto and Quantum Info Lab
Montreal, Québec, Canada
garboit@cs.mcgill.ca

Abstract

In this paper, we present a method for watermarking Java programs that uses opaque predicates, improving upon those presented in two previous papers [13, 9]. We present two algorithms: the first is simpler to implement and to analyze, but certain distortive attacks can make watermark extraction difficult; the second is more complex, but under realistic assumptions yields good resistance to all usual types of attacks.

Keywords Cryptography, steganography, information hiding, compiler security, watermarking, fingerprinting, software copyrights.

1 Introduction

Software copyright and patent provide legal protection against intellectual property theft. However, the classification of software with respect to these laws is vague and whether the reverse engineering and decompiling of software is legal is a complex issue [4], and can be argued to be "fair" in some cases, while "malicious" in others [11]. Furthermore, even when possible, the application of these laws is laborious.

Watermarking is a steganographic technique that aims at providing cryptographically secure tools to aid in the application of copyright laws. This paper is concerned with code watermarking, that is, "Authorship Marks" (AM) and "Fingerprint Marks" (FM) [15], as we will discuss in Section 3.1. Ideally, the removal of an AM / FM should be as difficult as breaking a reputedly hard cryptographic problem.

In this paper, we will describe and analyze a method for watermarking programs, in particular Java programs, that uses opaque predicates, improving upon those presented in two previous papers [9, 13]. For an efficiency tradeoff, we achieve an arguably cryptographically secure insertion of an AM, which can be easily extended to the one of an FM. The security of our method will be discussed in Section 5.

2 Background

In 1999, Collberg and Thomborson [7], wrote that, "...apart from Grover [10] and a few recent US patents, very little (publicly available) information seems to exist on software watermarking in which a copyright notice (AM) or customer identification number (FM) is embedded into a program". In 2002, Nagra, Thomborson and Collberg [15], added a number

of publications to the problem [13], [18], [17], [16], and [20]. We have found that two more papers needed to be added to this list [9] and [2].

Our results improve on those presented in Monden et al. [13], which describes a technique that uses a never-executed dummy method (of a class), appended to a target Java source program, to encode an AM. This principle is improved upon by Collberg, Thomborson and Low [9], by guarding the never-executed call to the method by an "opaque predicate" (this notion will be discussed in Section 3.2), in order to avoid the method being easily eliminated as dead-code.

In our scheme, this improved principle is further revised by having these opaque predicates even further exploited, as to encode the whole AM / FM within them, with or without associated never-executed calls to methods.

3 Problem Settings

In general, the role of an AM / FM insertion algorithm is to insert a copyright notice within a program, such that it is hard to retrieve except for the programmer, and that its presence can be proven to be deliberate under reasonable assumptions.

3.1 Definitions and Notation

Following the terminology set by Nagra, Thomborson and Collberg [15, Section 4], an AM is a watermark that embeds in the software, information identifying its author. In other words, an AM aims at preventing an adversary from unrightfully claiming authorship of software. Single or Multiple AM's may be embedded. On the other hand, an FM "is a watermark that embeds information in the software identifying the serial number or purchaser of that software". Put differently, an FM aims at tracking the channel of distribution of a particular copy of the software.

Using the notation set in Section 5 of the paper just cited, we let O be a computer program that is available for manipulation in the current state, $S = [O, ...]$, of a computer system. Let ω be a watermark, and $\mathcal{E}$ be a watermark embedding function, then $\mathcal{E}(O, \omega) \rightarrow O_\omega$ is an embedding of the watermark in O. As noted in the same paper, the following properties should be understood in a probabilistic sense.

Correctness of extraction. The corresponding watermark extraction function $\mathcal{X}$ has the property,

$$\forall O, \omega \ : \ \mathcal{X}(O_\omega) = \omega \ .$$

Soundness of extraction. The extraction function $\mathcal{X}$ also has the property of not allowing false-recognitions, in the larger context of the program state, $S = [O, ...]$, which may also include the OS, the hardware, etc.,

$$\forall S, \omega, \omega' \neq \omega \ : \ \mathcal{X}(S_\omega) \neq \omega' \ .$$

Robustness of embedding. Additionally, we will require our embedding process $\mathcal{E}$ to be robust. An embedding process $\mathcal{E}$ is robust to a transform T if and only if it is legible after T. And $\mathcal{E}$ is said to be legible after $T : O \rightarrow O$ if

$$\forall O, \omega \ : \ \mathcal{X}(T(\mathcal{E}(O, \omega))) = \mathcal{X}(\mathcal{E}(O, \omega)) \ .$$

This is the mathematical opposite of fragility, which is rather used for software watermarks of the kind of Validity and Licensing Marks, otherwise known as tamper-proofing.

3.2 Assumptions

Ideally, the removal of an AM / FM should be as difficult as breaking a reputedly hard cryptographic problem. For this purpose, we state an "opaque construct assumption" as being cryptographically hard to break. The definition of an opaque predicate is borrowed from Collberg, Thomborson, and Low [9, Section 4.1].

Table 1: Examples of number-theoretical true opaque predicates

$\forall x, y \in \mathbf{Z}$	:	$7y^2 - 1 \neq x^2$
$\forall x \in \mathbf{Z}$	:	$3 \mid (x^3 - x)$
$\forall x \in \mathbf{N}$	:	$14 \mid (3 \cdot 7^{4x+2} + 5 \cdot 4^{2x-1} - 5)$
$\forall x \in \mathbf{Z}$	:	$2 \mid x \vee 8 \mid (x^2 - 1)$
$\forall x \in \mathbf{Z}$	:	$\sum_{i=1,\, 2 \nmid i}^{2x-1} i = x^2$
$\forall x \in \mathbf{N}$	:	$2 \mid \lfloor \frac{x^2}{2} \rfloor$

Opaque predicate. A predicate P is opaque at p if its outcome is known at [watermarking] time. We write P_p^F (P_p^T) if P always evaluates to `False` (`True`) at a program point p, and $P_p^?$ if P may sometimes evaluate to `True` and sometimes to `False`.

Definition 1. Informally, a one-way (non trivial) opaque predicate P is one that is difficult for an adversary to resolve, i.e. to find the truth value solution of.

Formally, let s be the security parameter of P and the success probability (resolution probability) of an adversary A for P be, for $V \in \{T, F\}$,

$$\delta(s) = \Pr[A(P_p^V) = V]$$

where the probability is taken over the random bits of A. A one-way opaque predicate is such that no efficient A can compute V much better than by random guessing. Given that $s = |P|$, the average size of the available opaque predicates, it is required for all polynomials R in s, that $\delta(s) < 1/2 + 1/R(s)$. [1]

Construction. The manufacturing of opaque predicates can be done using objects and aliases, as well as concurrency [9, Section 5]. On the other hand, they can also be constructed using elementary number theory facts. Collberg [6, Lecture 13, Section H] lists many examples, such as those given in Table 1. [2]

Assumption 1. Most number-theoretical opaque predicates are complex. By complex, it is meant that the minimal number of arithmetic operations required to resolve them is large, so that the expected resolving time of an adversary is super-polynomial. In particular, the predicates of Table 1 are complex.

Assumption 2. Complex number-theoretical opaque predicates are one-way.

3.3 Construction of a family of opaque predicates

The bank of predicates $\{P_i\}_{i=1}^n$ is kept secret in the way of a secret key. However, if it is of restricted size, an adversary can test to detect some P_i's that are known to be commonly used.

To counter this, we would want to dispose of a large set of opaque predicates, and intuitively, sets of exponential size. One way to achieve this is by using parametrized predicates. Take for instance some of the rules for finding square roots of known quadratic residues [3, p. 206–207], where p is a prime number of the given form.

1. Solutions of $y^2 \equiv a \bmod p = 4x + 3$, are $y \equiv \pm a^{x+1}$ and the corresponding family of opaque predicates is
$$\left[a^{x+1}\right]^2 \equiv a \bmod p \ .$$
2. Solutions of $y^2 \equiv a \bmod p = 8x + 5$, are $y \equiv \pm[(4a)^{x+1}/2]$ and the corresponding family of opaque predicates is
$$\left[(4a)^{x+1}/2\right]^2 \equiv a \bmod p \ .$$

[1] Intuitively, if the size and hence the number of predicates increases, it is harder to resolve any particular one.
[2] Note that the values of the variables x and y should be determinable at run-time only.

A family of opaque predicates is parametrized by a prime p of the given form. This parameter can be generated by picking random values of x and then testing if the resulting p is prime, or more simply by table lookup.

The members of the resulting family of predicates are then treated as the predicates given as examples in Table 1. The value of the variable a should be determinable at run-time only. Individual predicates should be hard to resolve in the sense that they satisfy Assumption 1.

4 Solution and Implementations

The goal is to encode the bits of an AM / FM ω into one-way predicates, such as the ones listed in Table 1.

In this section, two algorithms are presented. The first, presented in Section 4.1, is simpler, but is not secure against some types of attacks as it will be made clear in Section 5. The second algorithm, presented in Section 4.2, has improved security at the cost of being less natural, though it still requires only realistic assumptions.

4.1 Algorithm 1: Basic Insertion, with Opaque Predicates Only

Value of ω. In the case that the Mark ω is an AM, the algorithm inserts it as a public piece of information, such as those from any known zero-knowledge proof, e.g. factoring [19, Chapter 13]. The proofs themselves are used when the soundness of the recognition part of the algorithm needs to be proven, as detailed in Section 5.3.

In the case that the Mark ω is an FM, the inserted public information needs to vary with each user or distribution channel. More details will be given in the recognition part of the algorithm below.

Encoding of ω. In both cases described above, the bits of ω are encoded in the information contained in the predicates, for instance, in their constants or in their rank within the predicates bank. In any case, let the number of bits encodable with predicate P_i be denoted as $N(P_i)$.

Moreover, the information contained within P_i shall not only contain a number of bits from ω, but also their index within ω. This is to avoid having to recuperate the pieces of ω in a precise order, which may or may not be preserved after an adversary has applied transformations to the Marked program O_ω. Indeed, an adversary moving predicates around in the program flow graph cannot distort ω, since each of its k pieces would remember its own order in ω.

Then, if the Mark bit length $|\omega|$ added to the number of bits in the indexing equals l, the encoding algorithm should select k predicates $\{P_{i_j}\}_{j=1}^{k}$, from the bank $\{P_i\}_{i=1}^{n}$, such that $\sum_{j=1}^{k} N(P_{i_j}) = l$. Overall, it is sufficient to select k predicates such that,

$$|\omega| \;\leq\; \sum_{j=1}^{k} N(P_{i_j}) - k\log_2 k \;\;.$$

Insertion of ω. Select k random branching program points $\{p_j\}_{j=1}^{k}$. For a predicate $P_{p_j}^{V}$, if $V = T$, then either append $\wedge P_{p_j}^{T}$ to the branching condition. Otherwise if $V = F$, then append $\vee P_{p_j}^{F}$ to the branching condition. Clearly, doing so would not change the final value of the branching condition.

Note that the results of control flow analysis are not affected, since the program O_ω executes and in particular, branches in exactly the same way as the original O. More details will be given in Section 5.1.

Recognition of ω. In the case that the Mark ω is an AM, the recognition consists in extracting back all opaque predicates from $T(O_\omega)$, for a transformation T, or more generally a composition of many T's. In other words, given $\{P_{i_j}\}_{j=1}^{k}$ and likely T's with corresponding inverses T^{-1}'s, the recognizer shows the probable presence of these P_{i_j}'s.

In the case that the Mark ω is an FM, we may need to decide on the equality of two elements from the set of possible ω. The algebraic technique of fingerprinting (this term is used in a related, but second sense to the one of the FM's) provides an efficient way of doing this [14, Chapter 7].

4.2 Algorithm 2: with Opaque Predicates and Dummy Methods

Algorithm 2 uses the same ideas as Algorithm 1, with the addition that a dummy method m_j is associated to each of the k inserted predicates P_{i_j}.

Addition for the insertion of ω. The P_{i_j}'s are inserted in the same program points as for Algorithm 1, but the evaluation of P_{i_j} itself is done within added dummy method m_j.

Addition for the recognition of ω. The extraction of ω consists in collecting the m_j's. The difficulty here is that a transformation T on O_ω may render the m_j's unrecognizable, in particular T may rename the m_j's. In fact, it is a method's signature rather than its name that is to be recognized, and this signature consist in an arrangement of the Java basic types that are to be qualifying the parameters of the method. This will be discussed in more length in Section 5.1.

5 Properties

The threat model that will be treated is only concerned with automatic adversaries, not human ones. This allows us to discuss our scheme with respect to the theory of algorithms, without being concerned about the ease that some humans may have in resolving number theoretic predicates such as the ones of Table 1.

The possible attacks are additive, subtractive, and distortive [8]. In an additive attack, the adversary tries to overwrite ω by adding its own AM / FM ω'. In a subtractive attack, the adversary tries to erase ω, while of course, not modifying the execution of O_ω. In a distortive attack, the adversary attempts to make ω illegible by scrambling it.

If no adversarial transformation has been applied to O, correctness and soundness are trivially obtained. Under adversarial transformations, a scheme's properties of robustness, perceptibility, and fidelity must be considered, along with the different types of attacks. Robustness measures how much computation and time an adversary is forced to spend in removing ω without damaging the original source. Perceptibility measures how difficult is it to find ω in O_ω, and fidelity, how much deterioration the embedding of ω does to the original O. Subtractive and distortive attacks also attempt at disturbing the correctness of the recognition, while additive attacks target it soundness [15, Section 6].

A note concerning fidelity: since O_ω executes in exactly the same way as the original O, the fidelity yielded by both algorithms is perfect. This was discussed in Section 4 and will be elaborated on in 5.1.

5.1 Quality

The two algorithms presented in Section 4 are arguably cryptographically secure. Terms used in this section correspond to the ones defined in Section 3.

Correctness. Robustness is first considered. In the case of Algorithm 1, we rely on Assumptions 1 and 2 to establish that the used opaque predicates could not be resolved. This means that the "essence" of these predicates must be contained within $T(O_\omega)$ for any feasible composition of transformations T. The efficiency of the problem of finding the appropriate inverse transformations in order to extract the correct ω will be elaborated upon in Section 5.2.

In the case of Algorithm 2, the robustness relies rather on the difficulty of changing the signature of dummy methods. To make this difficult for the adversary, an object-oriented programming trick is used: methods that have types which could be easily casted (or "translated" more explicitly) into other types should be *overloaded* with other methods which have just these last types. If an adversary tries to modify the types of the first methods to the ones of the second, the phenomenon of *overriding* would occur causing the program to potentially behave erroneously. Therefore, an adversary would avoid changing a signature in this way.

We next consider perceptibility. Our threat model only includes automatic adversaries whose main tool of watermark detection is control flow analysis. As mentioned in Section 4, for both algorithms, the program O_ω executes and branches in exactly the same way as the original O, since the addition of an opaque predicate does not change the truth value of the

effective branching condition. Therefore, the control flow analysis of O_ω yields exactly the same results as the one of O. Moreover, the predicates are inserted in random branching points. In this limited sense, ω is imperceptible.

A subtractive attack is difficult because it is difficult to detect where ω is, so it is difficult to remove it, as a consequence of the low perceptibility of our ω. There is a small probability that an adversary can detect and remove parts of ω, so it may be useful to introduce some redundancy in it, for instance, by using error-correcting codes as suggested by Anderson and Petitcolas [1].

Soundness. An additive attack is just as difficult as a subtractive one, since it too must find exactly where ω is, or at least parts of it, in order to overwrite it. However, another Mark ω' could be added to O_ω, which would make it difficult to establish which one was embedded first. This can be countered with a public notary registration system.

Distortive attacks are the ones to be the most concerned with. For Algorithm 1, if the predicates are so scrambled that they become infeasible to extract, the attack is successful. This is the main problem with the algorithm: the adversary can make the recognition of ω difficult. For Algorithm 2, distortive attacks are practically not an issue, in as much as overriding protects the dummy methods. However, limits to this protection were noted and recognition beyond these limits would more so resemble the one of Algorithm 1.

5.2 Efficiency

Embedding. It is easy to see from Section 4 that the embedding of an ω is cheap. However its preparation takes some effort on the part of the programmer: one must find a bank of at least k opaque predicates that are as original as possible, so that they are not known to the adversary. Also, they should be such that they are as likely as possible to satisfy Assumptions 1 and 2.

Recognition. As mentioned in Section 5.1, the recognition of ω is trickier, in particular for Algorithm 1. In order to extract very distorted predicates from $T(O_\omega)$, the recognizer has to try operations from a set of likely inverses T^{-1} on suspected distorted predicates. It seems that the results from automatic theorem proving may be illuminating in the quantization of the efficiency of recognition.

Nevertheless, from Assumption 2, it is infeasible to resolve our predicates. This implies that the essence of a predicate cannot be lost, so that by expecting to find one of k predicates, it is possible to resolve a distorted predicate to its original form.

Furthermore, a watermarking scheme's priority should be to embed a high quality watermark, and in the event that code piracy is suspected, larger than usual efforts can be deployed in order to demonstrate it.

Run-time. The increase in running time of O into the one of O_ω is linear in the size k of the watermark, since it is the number of predicates, of a bounded length, that is inserted in the program.

5.3 Zero-knowledge proof of ownership

When piracy is detected, the owner of O_ω needs to prove that ω was deliberately inserted in O, and by no one else. After the proof, the owner also needs the watermarking process to remain unknown to other parties.

This is naturally and directly achieved by zero-knowledge proofs, where a secret is shown to be known by the prover to a verifier, while never revealing the secret itself. Many examples are detailed in a book by Stinson [19, Chapter 13]. In particular, Camenisch and Michels [5] describes how to use a ω that is the product of two safe primes.

6 Conclusion and Future Directions

We have presented two AM/FM insertion and recognition algorithms, as well as their strengths and weaknesses. Algorithm 1 is simple, but under some attacks yields a less efficient recognition, while Algorithm 2 is more robust, but requires

slightly more complex insertion, and requires the somewhat less elegant object-oriented programming trick of overloading to maintain its robustness. The analysis of the efficiency of Algorithm 1 could be improved by considering known results from automatic theorem proving.

Moreover, the key generation as explained in Section 3.3 must be improved in order not to necessitate human intervention. In other words, a more general method than one generating only a few families of opaque predicates should be developed.

However, it has been proven that watermarking is in general impossible to achieve, except for restricted complexity classes [2, Section 8, Section 9]. It would be interesting to determine how to use this impossibility result to break our scheme for a program belonging to a complexity class that was shown to be problematic.

Acknowledgements

Finally, many thanks are due to Claude Crépeau and Laurie Hendren for fruitful discussions.

The author was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Fonds québécois de la recherche sur la nature et les technologies (FCAR).

References

[1] R.J. Anderson and F.A.P. Petitcolas. On the limits of steganography. *IEEE Journal of Selected Areas in Communications*, 16(4):474–481, May 1998. Special issue on copyright & privacy protection.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Electronic Colloquium on Computational Complexity (ECCC)*, 8(057), 2001.

[3] A.H. Beiler. *Recreations in the Theory of Numbers - The Queen of Mathematics Entertains*. Dover Publications Inc., 1964.

[4] D. Burk. Copyrightable functions and patentable speech. *Communications of the ACM*, 44(2):69–75, February 2001.

[5] J. Camenisch and M. Michels. Proving in zero-knowledge that a number is the product of two safe primes. *Lecture Notes in Computer Science*, 1592:107–122, 1999.

[6] C. Collberg. CS 620 Security through obscurity, 2002. Course notes from http://www.cs.arizona.edu/~collberg/Teaching/SoftwareSecurity.html.

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

[8] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report 170, Department of Computer Science, University of Auckland, 2000.

[9] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Symposium on Principles of Programming Languages*, pages 184–196, 1998.

[10] D. Grover. Program identification. *The protection of computer software: its technology and applications, The British Computer Society monographs in informatics*, 1992.

[11] Ahpah Software Inc. Sourceagain and java decompilation. Technical report, 2000.

[12] M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, Princeton, N.J., 1996.

[13] A. Monden, H. Iida, K. Matsumoto, K. Inoue, and K. Torii. A practical method for watermarking java programs. In *compsac2000, 24th Computer Software and Applications Conference*, 2000. Also published in SCIS'98 (Japanese).

[14] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[15] J. Nagra, C. Thomborson, and C. Collberg. Software watermarking: Protective terminology. In *Australasian Computer Science Conference*, pages 177–186, 2002.

[16] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of ACSAC'00, 16th Annual Computer Security Applications Conference*, pages 308–316, 2000.

[17] J. Pieprzyk. Fingerprints for copyright software protection. In *Information Security, Second International Workshop, ISW'99*, pages 178–190, 1999.

[18] J. P. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999.

[19] D.R. Stinson. *Cryptography – Theory and Practice*. CRC Press, 1995.

[20] R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *Information Hiding, 4th International Workshop, IHW 2001*, pages 51–65, 2001.