A Method-based Whole-Program Watermarking Scheme For Java Class Files

Anshuman Mishra, Rajeev Kumar, and P. P. Chakrabarti
Department of Computer Science & Engineering
Indian Institute of Technology Khargapur
Kharagpur, WB 721 302, India

Abstract

We outline a scheme to perform static watermarking on an entire Java class file, method by method. We show that our scheme offers a structural approach towards increased resistance against tampering attacks. We demonstrate how the watermark can be used to determine in what way the class file has been modified, if tampering has happened. We compare against other previously known watermarking schemes and show that our scheme provides watermarking with increased robustness and decreased visibility.

Keywords: Software watermark, software piracy, Java class file, control-flow diagram

1 Introduction

Software piracy has reached rampant proportions in today's world. When coupled with the ease of decompiling program code written to work on virtual execution-based systems such as Microsoft's .NET or Sun's Java platform, the protection of intellectual property rights over software becomes a paramount issue which needs to be addressed. Watermarking schemes which had been developed to assert authorship or ownership control over digital artifacts such as images, video or audio, are being extended to cover software objects. However, most software watermarking schemes fail due to the ease by which software can be tampered with, keeping the overall semantics of the program constant. In the following subsection, we give a brief overview of previous work done in the field of software watermarking.

1.1 Previous Work in Software Watermarking

In general, watermarking implies the embedding of some form of marker into a digital artifact such as an image, an audio file, a video file, or even a software package, so as to assert authorship, or to perform validation and authentication, or to enforce licensing restrictions, or a combination of any of the above. Software watermarking is a specific form of watermarking in which the digital object which is to be modified to carry the watermark is a software package. A major constraint which is encountered while watermarking software is that of keeping the semantics of the program unchanged while embedding the watermark. This is a difficult task to perform, and various specialized schemes have been proposed keeping the above constraint in mind.

Though a general survey of the field of software watermarking has been carried out before, for example by Zhu et al. [9], we would like to outline a few major advances in this area. The first formal software watermarking approach was described by the patent of Davidson and Myhrvold [4], in which they described a scheme to embed a watermark inside a program by rearranging the order in which basic blocks of the program were arranged. Curran et al. [3] described a scheme in which the call graph depths of the method are changed to reflect the watermark encoding. A scheme of encoding the watermark inside a dummy method in the form of opcodes is described by Monden et al. [5]; since the dummy method is never executed, any type of watermark may be stored by proper encoding.

Venkatesan et al. [8] described a scheme of embedding watermarks by modifying the control-flow graph with the addition of specially marked nodes which denote the value of the watermark.

As opposed to the static watermarking schemes described above (so called because the watermark is embedded in the static *text* of the program), Collberg & Thomborson [2] described a novel method of watermarking, in which the watermark is constructed dynamically during the program execution, in the form of a graph. Inspection of the graph at runtime alone can verify the value of the watermark, and is not susceptible to static-time analysis. Opaque predicates, the values of which are previously known to the programmer but cannot be determined by static analysis of the program, can be constructed using the graph.

Some other experimental watermarking methods (e.g., [7]) include using threads to encode watermarks in the choice of execution of basic blocks depending on

the input pattern of the program. The input is kept secret and the watermark is recognized by tracing the program execution with the secret input. This is a low-capacity scheme, since with every encoded watermark bit, the size of the software increases by nearly 1 KB.

1.2 Problems with the Above Approaches

One problem with most of the above approaches is that there is no close coupling between the program and the watermark, i.e. the watermark can be easily damaged or removed without changing the program by simply applying semantics-preserving transforms. This is especially true for the schemes described in [4, 5].

Another problem is that in the case of schemes such as that of [2], it is not very difficult for humans to recognize which parts of the software belong to the original code and which belong to the dynamic watermark generating code. The scheme of [8] comes closest to our approach. One major difference is that the former marks the pieces of code which correspond to the watermark in a special manner – no such marking is done in our scheme, which ensures that, once the watermark is inserted, there cannot be any distinction between the original code and the watermark-based code.

2 Problem Outline

Let us envisage the following scenario - Alice, producer of the software **S**, wants to distribute the latter over some insecure channel, where there is a possibility of code tampering. Alice wants to embed a watermark onto the software which should fulfill the following criteria:

1. The watermark should be **robust**, i.e., it should be resistant to software transforms which preserve the meaning of the program.
2. The watermark should be **invisible**, i.e., it should only be detectable by using special forensic software.
3. The watermark should be resistant to transforms which try to overwrite it with another watermark or otherwise modify it in some manner.
4. The watermark should contain information which validates the claim that Alice could be the original creator of the software **S**.
5. The software consumer should have no way of removing the watermark, even if provided with the forensic software.

We must note here that it is very difficult, if not impossible, to produce a watermark that fulfills the above criteria, since the watermark will be encoded in some form in the software, and errors will creep in when a semantics-preserving transform is applied. Another reason is that since the watermark will be an externally-injected piece of code, it will be very difficult to stop an automated software tool such as a flow analyzer from tracing out code which does not belong to the main algorithm being implemented.

In this paper, we suggest the following general methodology to be followed by Alice:

- Alice embeds a watermark consisting of some encoded authorship information into each and every method of a Java class file.
- Alice employs a *web-spider* to search for suspicious pieces of code, which could match Alice's software **S** or use it in an enveloped format. Suspicious pieces of code could also be reported to Alice by neutral third parties, such as law enforcement agencies.
- Alice employs a specific software system (**forensic software**) to extract the watermark from the software. For proper extraction, the forensic software should require some additional input which should be known only to Alice. In this way, watermark removal by malicious third parties becomes very difficult.

3 Proposed Watermarking Scheme

The basis of our scheme lies in the fact that what Alice is trying to protect is her intellectual property in the form of the program's algorithm as well as implementation. Any algorithm can be implemented by using the structural paradigm of sequence, selection and iteration. Hence, a minor modification of the program structure in the sense of semantics-preserving rearrangement of code, or usage of a FOR loop instead of a DO-WHILE loop should not affect the fact that the program remains the same. However, a major modification in the sense of embedding the original program into a much larger program would ideally still allow us to extract relevant parts and check whether the original producer's intellectual property rights still hold.

The scheme we outline in this paper is hence based completely on the structure of the program. A minor modification of the program would lead to a minor tampering of the watermark - redundancy has to be built into the watermark to take care of such possibilities. We will point out a few mechanisms by which this can be done.

3.1 Basic Blocks and Value Assignments

Let us consider a Java class file consisting of many methods. Each method can be decomposed into a control-flow

diagram [1, 6], consisting of only four possible types of basic blocks:

- Simple sequential blocks, consisting of at most 8 instructions.
- Extended blocks, consisting of a sequence of one or more sub-blocks, which can be of any type.
- Extended if-then-else blocks, consisting of two extended sub-blocks, corresponding to the THEN block and the ELSE block.
- Extended iteration blocks, consisting of one extended sub-block, which undergoes iteration.

We notice that the above four types of block suffice in describing all program behavior, and that all higher-level constructs such as FOR loops or DO-WHILE loops can be decomposed into combinations of the above block types.

To each basic block and extended block we assign a value, based on the following scheme:

1. A simple sequential block is assigned the value of the number of its instructions, up to a maximum of 8.
2. An extended block is assigned the value of the sum of the values of its individual sub-blocks, taking into account the number of variables defined in the sequence - each such variable is assigned a value of 4. For example, suppose an extended block B consists of three simple sequential blocks {8,8,6}, and the number of variables defined in the sequence is 6, then the value of the block B is 22+6*4=46.
3. An extended if-then-else block is assigned the value of the sum of the values of the THEN extended block and the IF extended block, multiplied by 16. For example, suppose an if-then-else block ITE consists of the sub-blocks THEN={46} and ELSE={48}, the value of ITE will be (46+48)*16=752.
4. An extended iteration block is assigned the value of the extended sub-block which is to be iterated, multiplied by 256. For example, suppose an iteration block IB consists of the sub-block SB={46} which is to be iterated, the value of IB will be 46*256=11776.

The value of the entire method is taken to be the value of the entire control-flow diagram, taken as an extended block whose value is summed up, also taking into account the number of variables defined in the sequence, the latter being assigned a value of 4 each. For example, say a method consists of the extended block {8,8,6,ITE{46,48},IB{46},1}, and defines 6 variables, then the value of the entire method will be 8+8+6+16*(46+48)+256*46+1+6*4=12575. We notice that as the complexity of a method increases, so does its assigned value.

3.2 Watermark Insertion Algorithm

Our algorithm is divided into the following two phases:

1. The software producer Alice generates a watermark **W** which needs to be inserted into the Java class file **P**. **W** can be expressed in the form of either a number or a sequence of bits.
2. For each method **M*i*** in **P**, do the following:
   (a) Extract a control-flow diagram from **M*i***.
   (b) Assign a value **V*i*** based on the scheme described in the previous section.
   (c) Generate a control-flow structure **S'** consisting of a sequence of basic blocks of *dummy code* and *dummy variables* whose value **V*w*** is the same as that of the watermark.
   (d) Insert **S'** at appropriate places into the method **M*i***, thus changing the value of **M*i*** from **V*i*** to **V*i*+W**.

The following points should be taken care of while inserting **S'** into each method **M*i***:

- **S'** contains dummy code and dummy variables. It is mandatory that the dummy code use the dummy variables and generate values which can be merged with that of the original code, e.g. if the original code is using some constant, **S'** could be used to calculate the constant and feed it to the original code.
- Depending on the user's preferences, the watermark could be inserted at a lesser time-cost and higher space-cost at the outermost layers of the method, e.g. just before the end of the method or right at the beginning. The watermark could also be inserted at a higher time-cost and lesser space-cost into some inner loop of the method. It is worthwhile to remember that each length increase of a loop by 1 increases the value of the method by 256.

Other important points to note during the watermark insertion process are:

- For redundancy purposes which will be explained later in the paper, if **N** methods need to be protected, it is best if **2N+1** methods are watermarked.
- The watermark **W** which is inserted should ideally be of the form of a digital fingerprint or author authentication mark, utilizing public-key cryptography.
- The original values **V*i*** are stored safely for use in the watermark extraction phase. These values should only be known to the software producer Alice and to nobody else.

3.3 Watermark Extraction and Inspection

Extraction. The watermark extraction algorithm proceeds as follows:

1. The original values **V*i*** of each method **M*i*** are accepted from the user.
2. The values **V*i'*** are extracted from each method in the Java class file.
3. For each method, the difference **V*i'* - V*i*** is calculated.
4. For any **2N+1** watermarked methods, if more than **N+1** methods generate an equal difference **W'**, then the forensic software deems the watermark of the entire class file to be **W'**.
5. Any method whose difference **V*i'* - V*i*** is different from **W'** by a factor of $\triangle$, say, is assumed to have been tampered with - the forensic software can generate various combinations of possible transforms which could have created the difference $\triangle$.For example, if the difference $\triangle$= -8, the tampering agent could have:
   (a) Deleted a sequential basic block of size 8, or
   (b) Deleted two variables, or even
   (c) Deleted an IF-THEN-ELSE and inserted a block of size 8 (possible on a software *cracking* exercise).

Inspection. The job of the forensic software is to extract the watermark and detect any tampering if possible. In any case, after the watermark is extracted, the job of inspecting the watermark and verifying the authenticity of the code producer Alice still remains - inspection is required to thwart any man-in-the-middle attacks. Since the watermark that has been embedded in the software is deemed invisible, one way of ensuring that inspection leads to authentication is to encrypt some form of authorship signature using a secret key known only to Alice and use the resulting information as a watermark - thus during inspection, Alice can use her own secret key to decrypt the watermark and get back her authorship signature. This could be used to verify that at least parts of the software have been created by Alice.

3.4 Implementation and Performance Tuning

There are various factors which can affect the working of the above scheme. The correlation between structural complexity of a method and the magnitude of the method value leads us to the intuitive conclusion that if we restrict the size of the watermark to that of a fraction of the method value, the more complex a method, the greater its capacity to hold additional embedded information. Also, the redundancy which is implied in adding the watermark to **N** methods allows at most **[(N-1)/2]** methods to be tampered with in the form of addition or deletion, and such tampering to be detected by the forensic software. The scheme could thus be possibly finetuned by allowing the addition of dummy methods to ensure that all addition/deletion-based tampering of legitimate methods in a class file could be detected.

Another factor that must be taken into consideration is the quality of the code representing the watermark which is inserted into the original code. The former should be of such a quality as to ensure a much greater computational load for automated tracing/analysis tools and should be difficult, if not impossible, to detect by trained humans. One possible way in which this can be done is by a statistical analysis of the original code, and generation of code which matches the statistics, but does not affect (or affects to a very small extent) the original logical flow of the method.

Robustness of the watermark could be increased by allowing the possibility of tampering errors such as insertions/deletions by, using a crude example, multiplying the watermark number **W** by 10 and inserting the resultant number into the software. Then any program changes within the range of ± 10 could be detected and the original watermark detected.

We are currently in the process of implementing a configurable watermark insertion program as well as a watermark extraction program, written in Java and to be tested on Java class files. Though the algorithms described in the preceding two sections were created keeping Java class files in mind, they can be easily ported or extended to cover other virtual execution systems such as Microsoft's Common Language Infrastructure (CLI).

3.5 Salient Features of the Proposed Scheme

Some points to be noted about our proposed scheme, in comparison to others, are that, firstly, the scheme offers an **infinite-capacity channel for storage of watermarks with minimal increase in code size**, albeit at the cost of decreased performance of the software. This can be achieved by appropriate insertion of nested loops inside the program, each loop having a multiplying effect of 256 times on the value of the sub-block inside. So, two nested loops with 1 dummy instruction inside the innermost loop would give an added program value of 256*256*1 = 65536, a range of 16 bits, and could be encoded with less than 10 instructions.

Secondly, once the watermark is inserted into the program in the form of code, there is theoretically **no distinction between the inserted code and the original code**. When code insertion is performed taking into account the statistical properties of the original program, and if appropriate logical connections are made between the original

program and the inserted code, seamless integration is possible and would lead to greater difficulties for malicious third parties. Of course, the watermark extracting forensic software would have no problems since it is only concerned with the overall structure of each method. In comparison to other watermarking schemes, there is no special *marking* of the objects which are carrying the watermarking information.

Thirdly, our scheme offers **various ways to achieve the same task** of watermark insertion. When Alice decides to insert her watermark, she can decide to insert it keeping in mind that she wants minimal time cost, in which case she can add cheaper blocks such as sequential blocks or IF-THEN-ELSE blocks. If Alice wants minimal size increase, she can add more blocks with more time-cost such as iteration blocks, which can be implemented with much less code size increase. Thus we see that our scheme can be configured to insert the watermark in a variety of ways, depending upon the requirements of the software producer, without making a slightest amount of difference to the watermark extracting forensic software. This is in contrast to other schemes, where once the insertion mechanism is known to malicious attackers, the latter can undertake targeted attacks probing the weaknesses of the mechanism.

4 Results and Ongoing Work

Let us now look back at the original scenario and try to determine how many of the criteria that were required by Alice were fulfilled, and to what degree:

1. The scheme allows the watermark to be resistant to sequence modifications - insertion/deletion of the code will result in a change in the extracted watermark. However, such changes could be trapped as explained previously, the exact methodology being used being dependent on the particular implementation of our scheme.

2. The scheme allows watermarks to be nearly invisible, provided the inserted code is of similar quality to that of the original code.

3. The insertion of an encrypted authorship mark ensures that, even if a malicious third party attempts to insert their own code, Alice's original watermark would still be embedded inside the code, provided the above two conditions have been fulfilled properly.

4. If the watermark contains Alice's authorship information encrypted using Alice's secret key, then only Alice could have had access to it and thus only Alice could have inserted the watermark. Thus we ensure that Alice's authorship of the software remains validated.

5. The forensic software requires the original set of method values as input in order to be used to extract watermarks. Hence, a malicious user of Alice's software, even with access to the forensic software, would not be able to tamper with Alice's watermark.

The scheme outlined in this paper is a work in progress. As mentioned previously, we are in the process of building configurable watermark insertion and extraction programs. We need to evaluate exactly what kind of code to insert, and should allow users configuration options such as the level of the method at which to insert the code, e.g. inner loops, outer loops, etc. as also the strength of the watermark, by modifying the fraction of the method value up to which watermark code can be inserted, e.g. whether a code increase of 10%/20%/30% is allowed. Of course, any mechanized algorithm would fail in the face of human ingenuity, and so we are constantly trying to find flaws in the scheme which could be exploited.

References

[1] Aho, Ullman & Sethi, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] C. Collberg, C. Thomborson, *Software Watermarking: Models and Dynamic Embeddings*, POPL-99, 1999

[3] D. Curran, N. Hurley, M. Cinneide, *Securing Java through Software Watermarking*, PPPJ 2003, Kilkenny, Ireland, 2003.

[4] R. L. Davidson, N. Myhrvold, *Method and System for Generating and Auditing a Signature for a Computer Program*, US Patent No. 5,559,884; 1996.

[5] A. Monden, H. Iida, K. Ichi Matsumoto, *A Practical Method for Watermarking Java Programs*, 24th Computer Software and Applications Conference, 2000.

[6] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[7] J. Nagra, C. Thomborson, *Threading Software Watermarks*, 6th Intl Information Hiding Workshop 2004 (IH 2004), LNCS 3200, 2004.

[8] R. Venkatesan, V. Vazirani, S. Sinha, *A Graph-Theoretic Approach to Software Watermarking*, 4th Int. Information Hiding Workshop, Pittsburgh, PA, 2001.

[9] W. Zhu, C. Thomborson, F-Y Wang, *A Survey of Software Watermarking*, LNCS vol 3495, Springer-Verlag, pp. 454 - 458, April 2005.