Securing Java through Software Watermarking

D. Curran
Information Hiding Laboratory
Computer Science
Department
University College Dublin
Dublin 4, Ireland
David.Curran@ucd.ie

N. J. Hurley
Computer Science
Department
University College Dublin
Dublin 4, Ireland
Neil.Hurley@ucd.ie

M. Ó Cinnéide
Computer Science
Department
University College Dublin
Dublin 4, Ireland
Mel.Ocinneide@ucd.ie

ABSTRACT

An important advantage of Java is its portability due to its use of bytecode. However the use of bytecode allows decompilation of Java programs to gain access to their source code. This makes it easier to pirate Java programs, infringing their copyright. This is a disadvantage of Java in comparison with programming languages that compile to native object code.

Software watermarking is a relatively new approach to the problem of copyright protection that involves embedding ownership information in an executable program. Watermarking has been extensively researched in the context of multimedia and significant progress has been made toward the development of robust and secure techniques. In this paper we investigate a new software watermarking scheme. This is derived from signal detection theory which is used in multimedia watermarking.

Keywords

Copyright infringement, java protection, software watermarking, spread spectrum watermarking

1. INTRODUCTION

Software piracy is a major obstacle to the wider use of Java [8]. The ability to prove ownership of pirated Java programs would ease copyright concerns over Java programs.

Software watermarking entails altering a program to include ownership information. In this paper we describe current state of the art in software watermarking and how signal detection theory can improve our knowledge of software watermarking. We present a watermarking scheme that is based on signal detection theory.

1.1 The Basics of Watermarking

Steganography is the art and science of communicating in a way which hides the existence of the communication. In short, a *secret* message is hidden in a *host* message. Watermarking is a special case of steganography. The purpose of watermarking is to securely embed information in host content. A watermark should be robust against deliberate attempts to remove it.

The effectiveness of a watermark can be described by three traits. When applied to software watermarks, these are described as follows:

- *Robustness* represents the ability of a watermark to withstand an attack by an adversary. In general, a software watermark must at least be able to withstand common transformations such as optimization, decompilation-recompilation, obfuscation, etc. Robustness requires that a watermark is not only present but also detectable after an attack. Areas of watermark producing code should not be easily separable from other program code so as to increase robustness [4].

- *Capacity* is the amount of information that can be embedded in the watermark.

- *Perceptibility* is a measure of how apparent a watermark is in the target program. A watermark that alters the quality of the host message is perceptible. In the case of software watermarking negative alterations to functional program behavior, memory usage and maintainability increase the perceptibility of the watermark.

There is a trade-off between capacity, robustness and perceptibly. For instance the more information that is present in a watermark the more perceptible it tends to be.

2. SOFTWARE WATERMARKING

Software watermarking differs from multimedia watermarking in that the watermark must be embedded into an executable program, rather than passive data such as an image or audio file.

The essential difficulty of copyright protection of digital data is that prevention of copying of digital media is infeasible. This was aptly described by Bruce Schneier [6] when he stated "*Digital files cannot be made uncopyable, any more than water can be made not wet.*" A pirate cannot be prevented from copying Java source code. Watermarking this code to allow ownership to be proved is one method of copyright protection.

Software watermarking schemes can be split into two types, static and dynamic.

PPPJ 2003, 16-18 June 2003, Kilkenny City, Ireland.

2.1 Static Watermarks

Static watermarks are placed in the code or data of the software being watermarked. They hide the watermark in redundant areas of the program. This is similar to how watermarks are hidden in redundant areas of multimedia files, which humans cannot detect due to imperfections in our perception.

Code watermarks are stored within the instructions of a program. Program instructions contain two types of redundant information that can be used to store such watermarks.

The first exploits the lack of data or control dependencies between statements. These statements are then reordered to encode a watermark. A simple example of this is reordering the *case* statements inside a Java *switch* statement. Another form of this watermark is reordering the control flow graph of a method. These methods store information based on which arrangement out of a number of possible arrangements is used.

The second redundancy in program code exploits alternate instructions or sets of instructions that have the same resulting programming behavior. A form of static code watermark is to find *mutable instructions* [5]. Mutable instructions are instructions that have two or more equivalent variants. This scheme has obvious connections to Java bytecode which possesses many instructions that are equivalent in certain circumstances.

In the case of data watermarks, a copyright string is placed in the areas of a program that do not contain instructions. In Java classes one such area is as the constant pool that can be altered to encode a watermark.

Static watermarks are not robust to distortive attacks such as optimization and obfuscation.

2.2 Dynamic Watermarks

Dynamic watermarks create a watermark in the structures created by the running program. They hide the watermark in redundant computations of the program. Dynamic watermarks are generated by the program during its execution, usually on a particular input sequence. The four main types of dynamic watermarks are:

2.2.1 Easter Egg Watermarks

An *Easter Egg* is a piece of code that gets executed for a highly unusual input to the program and produces an obvious and easily recognisable output. See [1] for an example of such an Easter Egg in Java. The obvious output of an Easter Egg means once an attacker enters a correct input the watermark is discovered, and so easily removable. A more effective system is to create a watermark that is hard to separate from normal program behavior, thus making it harder to discover.

2.2.2 Dynamic Data Structure and Execution Trace Watermarks

These involve a watermark that is inserted into the state or trace (instructions, addresses or both) of the program during execution. In contrast to Easter Eggs these watermarks are not easily perceivable to the user. The watermark is then extracted by examining the state after the input using a watermark detector such as a debugger.

An elementary example of a data structure watermark is to store the watermark in the variables of a Java program, e.g. **char water[];**

water[0]='C';
water[1]='o';
water[2]='p';
water[3]='y';
water[4]= ...

An elementary execution trace watermark can store the watermark in the operand stack of a method. By examining what is popped from the operand stack the watermark can be detected.

These types of watermarks are susceptible to distortive attacks. For example arrays in a program can be split, values in an array/stack can be altered using obfuscation and methods can be split into submethods. The weakness of these methods is that dependencies do not exist within the watermark, so the watermark is easy to disrupt.

2.2.3 Dynamic Graph Watermarks

These involve a watermark that is added to a program by having it create a graph structure. This graph structure enumerates a number that is the watermark. It is difficult to find program behavior preserving transformations that will alter this graph. This makes this form of dynamic watermarking more robust than general data structure watermarks [2].

This form of watermark is believed to be the most promising, type to withstand distortive attacks. Other graph watermark attacking schemes require the analysis of the behavior of a program that can be difficult and impractical. This is because detecting code that does not affect functional program behavior is challenging. See [2] for a more in-depth review of software watermarking.

In the next section we present a model that can be used to argue about the effectiveness of a software watermark.

3. THE SIGNAL DETECTION APPROACH

The signal detection model of watermarking has been a topic of much research in multimedia watermarking [3]. Using this approach significant progress has been made toward the development of robust schemes. If software watermarking can be modeled as a signal detection problem then the knowledge gained from this research can be applied to software watermarking.

Multimedia content can be represented as signals in space and time. In general such content is watermarked by modulating a weak signal (the watermark signal) onto the original host signal. Watermark detection is then the problem of detecting this weak signal, even after signal transformations. A popular technique for modulating the watermark signal on the host signal is *Spread-Spectrum* watermarking. Stern *et al* [7] suggest a scheme using mutable instructions that uses Spread-Spectrum watermarking. This shows that multimedia watermarking models can be applied to software watermarking.

Spread-Spectrum watermarking requires a vector **r** to be extracted. Our approach is to extract this vector from the properties of a running program. One simple way to do this is to measure the call graph depth at distinct points during the execution of the program on some particular input. There are many other possible signals present in java programs that could be watermarked, for example memory usage.

The signal we add alters the call graph depth to encode watermarking information. This is achieved by altering a

method so that for the detection input it calls itself at the correct point. Normal method execution occurs after this fake method call.

As an example of how a method is changed to create the watermark while it is executing. The method *func1* below has been altered to encode a watermark bit on its first execution.

```
 func1(int a, int b){
  if(depth==0){
    depth++;
    func1(a,b);
  }
  else{
  /* original method */
  }
}
/*variable declaration*/
private static depth =0;
```

We now describe the Spread-Spectrum approach with reference to the method-depth scheme, first considering the embedding and then the detection of the watermark. The stages used in this scheme could be applied to watermark other signals in Java programs.

3.1 Watermarking Embedder

Extract a central vector $\mathbf{r}$ of dimension N from the target program s using the extraction function $X(s) = \mathbf{r}$. In our scheme, this can be achieved by storing distinct points of the call depth of a Java program run on a particular input, then subtracting their mean. We determine the calls to methods during a program run using a debugger. For increased security $\mathbf{r}$ can be permuted using a permutation key known by the embedder and the detector.

Modulate a central, pseudorandom watermark signal $\mathbf{w}$ on $\mathbf{r}$ with a mixing function F, producing $\mathbf{r_w} = F(\mathbf{r}, \mathbf{w})$. In our scheme $F(\mathbf{r}, \mathbf{w}) = \mathbf{r} + \mathbf{w}$. The watermark $\mathbf{w}$ is a vector that can be used to prove copyright ownership, for example $\mathbf{w} = 0, 0, 1, 1, 0, 1, 0, ..., 0$.

The additional depth of the method calls is given by

$$\sum_{i=1}^{n} w_i = c \tag{1}$$

Hence we can define a quality constraint such that the watermark should not add more than M calls to the call depth. So c must be chosen so that $c \leq M$.

Finally the vector $\mathbf{r_w}$ is re-embedded into the program s by applying the inverse of the extraction process.

In our scheme this stage involves altering the Java program so that methods call themselves to alter the call graph at the detection points.

3.2 Watermarking Detector

A program s' may be a watermarked program which has undergone transformations such as watermark attacks. We extract a feature vector $\mathbf{r}'$ which can be written as

$$\mathbf{r}' = \mathbf{r} + \mathbf{w} + \mathbf{n} \tag{2}$$

where $\mathbf{n}$ represents random noise that has been added due to an attack on s

Detect presence of watermark by computing the correlation of $\mathbf{r}'$ with the central vector $\mathbf{w}'$. Where $\mathbf{w}' = \mathbf{w} - \mu_{\mathbf{w}}$.

$$\mathbf{d}(\mathbf{r}') = (\mathbf{r} + \mathbf{w} + \mathbf{n}).\frac{\mathbf{w}'}{N} \approx \frac{|w'|^2}{N} \tag{3}$$

here $\frac{|\mathbf{w}'|^2}{\mathbf{N}}$ is a positive number that is the result of correlating on a watermarked program. Assuming that $\mathbf{r}$ and $\mathbf{n}$ are uncorrelated with the watermark vector $\mathbf{w}$ then

$$(\mathbf{r} + \mathbf{n}).\frac{\mathbf{w}'}{N} \approx 0 \tag{4}$$

Hence it is desirable that $\mathbf{r}$ and $\mathbf{w}$ be as independent as possible. If the program s' has not been watermarked the result of correlation should be near zero.

If this value $\mathbf{d}(\mathbf{r}') > T$, a predetermined threshold of the correlation, the watermark is said to be present. The value of T is usually chosen, so that the probability of declaring the unmarked content to be watermarked (i.e. probability of false alarm (pfa)) is small. That is we require.

$$Prob(\frac{\mathbf{r}.\mathbf{w}'}{N} > T) < pfa \tag{5}$$

where pfa is some small fixed constant (e.g. 10^{-4}). This pfa provides an empirical measure of the quality of a watermarking scheme.

Watermarking schemes are classified on the basis of what information they require to be available to the detector. The less information required by the detector the wider the possible usage of a scheme. The depth count scheme requires the detector to know the secret watermark $\mathbf{w}$ though not the original host data $\mathbf{r}$. $\mathbf{w}$ and $\mathbf{r}$ must be kept secret from attackers as it will allow them to remove the watermark. This makes it a blind symmetric private scheme.

Some shortcomings of the method-depth scheme and other insights gained from experience with Spread-Spectrum watermarking are described in the next section.

4. LIMITATIONS OF THE SCHEME

Experience from multimedia watermarking provides insight into methods that can be used to attack and defend Spread-Spectrum watermarks.

An attacker can add to the method-depth count by adding his own method calls to the call graph. This attack is similar to multimedia attacks that involve adding white noise to the signal.

This attack is based on the redundant nature of the information in which the watermark is placed. The scheme is vulnerable because method-depth counts of a program can be significantly altered without affecting how the program operates. This is also what allows us to easily embed the watermark.

As with other software watermarking schemes watermarked programs should be altered using obfuscation and tamperproofing to make them more difficult to attack. A possible defense against this form of attack is to tamperproof the Java program by using reflection [2].

Our attempt to detect the watermark is hindered if a new method is called during the watermark detection run. By causing the program to call a new method the standard correlation detection scheme can be defeated. Correlation is sensitive to any new elements being added to the vector. Such desynchronisation attacks are defended against in multimedia watermarking by correlation on a number of vectors

or by embedding a synchronisation pattern in the host signal.

These defenses illustrate how work done in other multimedia watermarking can be applied to software watermarking schemes based on the signal detection model.

Because the method depth system relies on dynamic program execution it tends to be robust to optimization attacks. For example the javac -O command doesn't inline watermarked methods as they are too large. Also sophisticated dead code removal techniques are required to remove the code added to the watermarked methods.

Future work involves investigating other watermarking schemes that use the signal detection approach. This requires new signals to be extracted from programs and the more advanced Spread-Spectrum techniques that are used in multimedia watermarking to be examined.

5. CONCLUSION

Software watermarking for Java is an important technology with the possibility to combat one of the major criticisms of the use of bytecode. This paper provided an introduction to software watermarking and described why it is of particular of interest to Java programmers. We defined a signal detection model of software watermarking, providing a framework for a watermark embedding and detecting. This model also allows us to argue about the quality of a software watermarking system. We presented the method-depth scheme that uses the signal detection approach and showed how knowledge gained from other watermarking domains that use this approach can be applied to software watermarking.

6. ACKNOWLEDGMENTS

This project is supported by Enterprise Ireland Basic Research Grant, SC/2001/178.

7. ADDITIONAL AUTHORS

B. O'Donovan, (email: `Barry.Odonovan@ucd.ie`) and G. C. M. Silvestre (email: `Guenole.Silvestre@ucd.ie`).

8. REFERENCES

[1] The easter egg archive. In *www.eeggs.com/items/568.html*.

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

[3] I. J. Cox, M. L. Miller, and A. L. McKellips. Watermarking as communications with side information. *Proceedings of the IEEE (USA)*, 87(7):1127–1141, 1999.

[4] J. Nagra, C. Thomborson, and C. Collberg. A functional taxonomy for software watermarking. In M. J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.

[5] J. Pieprzyk. Fingerprints for copyright software protection. In *Proceedings of the 2nd Information Security Workshop (ISW'99)*, volume 1729 of LNCS, pages 178–190, November 1999.

[6] B. Schneier. The futility of digital copy prevention. In *CRYPTO-GRAM, cryptome.org/futile-cp.htm*, 2001.

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

[8] H. van Vliet. Mocha - the java decompiler in. In *www.brouhaha.com/∼eric/computers/mocha.html*, 1996.