Experience with Software Watermarking for Java

Jens Palsberg
Purdue University

Secure Software Systems Group

CERIAS, Center for Education and Research in Information Assurance and Security

Joint work with Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, Yi Zhang.

Work supported by an NSF CAREER award and by Lilly Endowment Inc.

Overview

- ≥ 4 U.S. patents on software watermarking;
- Collberg and Thomborsen (1999): embed a watermark in dynamic data structures;
- we have new techniques for obfuscation and tamperproofing;
- we have an implementation for Java; and
- experiments show: efficient, cheap, resilient to attacks.

The Need to Prove Software Ownership

1. Suppose Alice has built some software and now wants to sell it for profit.
2. She copyrights the software, but Bob still manages to make a pirate copy.
3. Bob may be interested in the software for several reasons, including
   (a) private use,
   (b) industrial espionage, and
   (c) further selling for his own profit.

Question: How does Alice protect her copyright?
Answer: She must be able to prove ownership, possibly in a court of law, of a given copy of the software.

Approaches to Anti-piracy

- Keep a certified list of customers.
- Link the software to the hardware of a specific machine.
- Refinement: configure the software such that when a user is logged on to the internet, a message is secretly sent to the company.
- Link the software to a movable piece of hardware that cannot easily be copied.
- Software watermarking: embed a secret into the software which can be retrieved on demand.

Approaches to Software Watermarking

- *Easter Egg* watermarks: a piece of code that gets activated after an unusual input.
- Embed a representation of: 1) a company identifier, 2) a customer number, 3) a serial number, and 4) a timestamp into the software.

A static watermark resides in the text of the software. Examples:

- A comment:

```
/* My software, version 1.0 */
```

- A data string:

```
printf("My software, version 1.0");
```

- A particular ordering of certain instructions. For example, a particular ordering of the branches of an n-branch switch-statement can encode $\log(n!)$ bits.

Dynamic Watermarks

Resides in data structures built at run time.

Stack allocated vs. heap allocated.

Attacks on Watermarks

Collberg and Thomborsen 1999:

- **Subtractive attack:** Bob attempts to locate and remove the watermark.
- **Distortive attack:** Bob applies uniform semantics-preserving software transformations in an attempt to distort the watermark so much that Alice can no longer retrieve the watermark.
- **Additive attack:** Bob inserts his own watermark in an attempt to override Alice's watermark, or at least make it plausible that Alice's watermark was not inserted before Bob's.

Attacks on Watermarks

Easter Egg watermarks: subtractive attack!

Static watermark: distortive attack!

Dynamic watermarks are less vulnerable to distortive attacks.

Heap-allocated data structures are more difficult to analyze (pointer analysis!) than stack-allocated data structures.

To successfully defend against additive attacks, it seems necessary to involve a trusted third party and to use time stamps.

An attacker may monitor the state of the heap, registers, etc. during execution.

Attacks on Watermarks

Problems with use of dynamic data structures:

- Need code which builds those data structures.
- An attacker can attempt to locate that code even without knowing the exact details of the watermark.
- Attack: look for anomalies in the code, for example, unusual instructions or sequences of instructions, etc.
- Collusion attack: if two attackers have two copies of the same watermarked program, watermarked with different watermarks, then they can compare and contrast.

Our Results

- JavaWiz: an implementation of a watermarking system for Java that refines the ideas of Collberg and Thomborsen.
- We use novel techniques to obfuscate the watermarking code, and to achieve a level of tamperproofing.
- Experiments show:
  - watermarking can be done efficiently
  - code size, execution times, and heap-space usage only increase moderately, and
  - the watermarked code is resilient to various attacks.
- Open problem: it seems necessary simultaneously to do:
  - watermarking,
  - tamperproofing, and
  - obfuscation.

  How?

A Representation of a Watermark

A watermark is a natural number.

Represent a number as a Planted Plane Cubic Tree (PPCT).

PPCT

no of leaves	biggest number representable
1	$1 * 10^0$
10	$5 * 10^3$
20	$2 * 10^9$
30	$1 * 10^{15}$
40	$7 * 10^{20}$
50	$5 * 10^{26}$
60	$4 * 10^{32}$
70	$3 * 10^{38}$
80	$3 * 10^{44}$
90	$3 * 10^{50}$
100	$2 * 10^{56}$
109	$2 * 10^{61}$

Build PPCT: 6 $\times$ (no of leaves) lines of code.

generate : number $\rightarrow$ straight-line code.

Extract watermark: 60 lines of code.

The Design of our Watermarking System

$$
\begin{tikzcd}
\text{original data structures} \arrow[r, "\text{built by}"] & \text{original code} \arrow[d, "\text{obfuscated by}"] \\
& \text{a watermark PPCT} \arrow[d, "\text{built by}"] \\
& \text{code that builds the watermark PPCT} \arrow[d, "\text{obfuscated by}"] \\
& \text{a random PPCT} \arrow[d, "\text{built by}"] \\
& \text{code that builds the random PPCT}
\end{tikzcd}
$$

Watermarking

W: the watermark
P: the Java 1.2 program to be watermarked.

Watermarking proceeds in three steps:

1. produce generate(W);
2. produce the merger of P and generate(W), including obfuscation based on two PPCT's; and
3. obfuscate the outcome of (2) with some other obfuscator.

The result is a Java 1.2 program.

If we were asked to watermark Java bytecode, then first we would attempt to decompile the bytecodes to Java source.

Obfuscation

We use obfuscation for two somewhat related purposes:

1. to make generate(W) unrecognizable, and
2. to make P and generate(W) look alike.

Opaque Predicates

Idea: change a statement S into

$$\text{if } c \text{ then } S$$

where c is an opaque predicate.

An opaque predicate always evaluates to $\mathbf{true}$, but this property is hard to recognize.

PPCT: excellent source for opaque predicates.

Open Problem

Chicken-and-egg problem.

The code that builds the random PPCT is un-obfuscated.

Currently, we wrap up by obfuscating with some other obfuscator:

- transform names to meaningless names like "A", "B", "C", and
- use overloading as much as possible.

Tamperproofing

$$
\begin{tikzcd}
\text{original data structures} \arrow[r, "\text{built by}"] & \text{original code} \arrow[d, "\text{obfuscated by}"] \\
& \text{a watermark PPCT} \arrow[d, "\text{built by}"] \\
& \begin{array}{l}\text{code that builds} \\ \text{the watermark PPCT}\end{array} \arrow[d, "\text{obfuscated by}"] \\
& \text{a random PPCT} \arrow[d, "\text{built by}"] \\
& \begin{array}{l}\text{code that builds} \\ \text{the random PPCT}\end{array}
\end{tikzcd}
$$

Trade-off: obfuscation+tamperproofing *versus* how much effort will an expert attacker have to spend to distort or remove a watermark.

Dynamically Opaque Predicates

Problem with opaque predicates: if an attacker can monitor the heap, registers, etc. during execution, then it may quickly be revealed that a given predicate always evaluates to true.

To defend against such an attack, JavaWiz uses a novel technique called *dynamically opaque predicates*.

Idea: branch correlation!

A group of predicates will in the same run all give the same result, but in different runs they may give different results.

Predicate	Run 1	Run 2	Run 3	Run 4	Run 5
Pred 1	T	T	F	T	F
Pred 2	T	T	F	T	F
Pred 3	T	T	F	T	F
Pred 4	T	T	F	T	F
Pred 5	T	T	F	T	F
Pred 6	T	T	F	T	F

Dynamically Opaque Predicates

Known static analyses of branch correlation, for example the one by Bodik, Gupta, Soffa (1997), seems to have difficulty attacking dynamically opaque predicates.

Experimental Results

Embed a 64-digit watermark.

program	description
javac	a compiler for Java
javadoc	a Java API documentation generator
JavaCup	an LALR parser generator for Java
JTB	JTB is a frontend for The Java Compiler Compiler from Sun Microsystems
JavaWiz	our watermarking system
JAX	JAX is a Java program packaging tool from IBM that reduces the program's size
BLOAT	BLOAT is a Java bytecode optimization tool

program	test input
javac	the JavaCup source code
javadoc	the JavaCup source code
JavaCup	the CORBA grammar
JTB	the Java 1.2 grammar
JavaWiz	the JTB source code
JAX	the Hanoi demo shipped with JAX
BLOAT	the JavaWiz source code

Experimental Results

All tests were run on a Dell 450MHz Pentium II with 128MB RAM and 7.5GB disk.

We used the Java virtual machine that comes with the JDK from Sun Microsystems.

program	lines of code		time	running time		needed heap space	
	before	after	to wm	before	after	before	after
javac	6,053	6,946	3.4 s	2.8 s	3.2 s	1.6 MB	1.9 MB
javadoc	7,812	8,407	3.7 s	11.8 s	13.3 s	4.6 MB	6.0 MB
JavaCup	11,029	11,799	2.8 s	5.3 s	7.4 s	2.3 MB	3.5 MB
JTB	13,125	14,059	3.0 s	5.6 s	7.9 s	1.6 MB	2.5 MB
JavaWiz	22,397	23,152	3.4 s	3.0 s	5.3 s	2.0 MB	2.6 MB
JAX	22,705	23,537	2.7 s	12.3 s	13.6 s	7.2 MB	7.8 MB
BLOAT	55,518	56,052	4.1 s	3.6 s	3.9 s	0.5 MB	1.0 MB

Simulated Attacks

- Compile to bytecodes and then decompile.
- Do bytecode optimization along the way.
- The Java bytecode obfuscator OuyaSafe.
- The Java packaging tool JAX.

The watermark was intact after the attacks.

Simulated Collusion Attack

We watermarked a program with twice with 64 digit watermarks that differed only on the last digit.

The two watermarked programs were then submitted to MOSS (UC Berkeley) which computes a measure of software similarity.

The MOSS result was that the two programs are 55% similar, and then MOSS output seems to give little help with locating the code that builds the watermark PPCT.

We ran a "diff" on the two files, and we found that although many lines differ, they differ only in some numerical constants.

Next we handed all of the above to two graduate students who know a bit about our watermarking approach and asked them to locate the watermarking code. After an hour, they gave up.

Conclusion

1. Our watermarking system is efficient;
2. the watermarked programs use only a little more time and space; and
3. the watermarked programs are resilient to attacks.

Open problem: avoid the "weak spot" from where the whole construction can be unraveled.

Other things to consider:

- More choices for data structures that PPCT.
- Encrypt the watermark before building the graph.
- Move the watermark around at run time.