**Software Security Project Assignment**

**Topic: SandMark Tool**

**Prepared by :**

**Shafiya Heena(301199792)**

**Najmun Nahar(301160081)**

**Gagandeep Kaur(301144458)**

**Table of Content**

* Introduction
* About the Tool
* Software protection terminology
* Threat models
* Sandmark system overview
* A simple Sandmark plugin
* Watermarking in Sandmark
* Sandmark manual attack tools
* Sandmark evaluation techniques
* References

**Introduction:**SandMark is a tool developed at the University of Arizona for software watermarking, tamper-proofing, and code obfuscation of Java bytecode. The ultimate goal of the project is to implement and study the effectiveness of all known software protection algorithms. Currently, the tool incorporates several dynamic and static watermarking algorithms (such as those proposed by Venkatesan, Collberg, Stern, and others), a large collection of obfuscation algorithms, a code optimizer, and tools for viewing and analysing Java bytecode.

**About the Tool:**

Software watermarking algorithms can be used to embed a customer identification number (a fingerprint) into a Java program in order to trace software pirates. A SandMark software watermarking algorithm consists of two programs:

* + The embedder takes a Java jar-file and a string (the watermark) as input and produces the a new jar-file that embeds the string.
  + The recognizer takes the watermarked jar-file as input and produces the watermark string as output.

Typically, the watermark is a copyright notice or a customer identification number.

* The code obfuscation algorithms in SandMark take a Java jar-file as input and produce an obfuscated jar-file as output. They have many applications:
  + The obfuscations can be used to protect the intellectual property of Java programs (by rendering the code difficult to understand).
  + Obfuscations can protect fingerprinted programs from collusive attacks (by making differently fingerprinted program differ everywhere, not just in the parts where the mark is embedded).

Obfuscations can also be used to attack software watermarks (by reorganizing the code such that the mark can no longer be recognized).  
  
SandMark is designed to be simple to use. A graphical user interface allows novices to easily try out watermarking and obfuscation algorithms. Algorithms can be combined, the resulting watermarked and/or obfuscated code can be examined, and attacks can be easily launched. SandMark has been designed using a plugin-style architecture which makes it easy to extend with additional algorithms.

We are currently using SandMark to study which software watermarking algorithms are vulnerable to which code optimizations and code obfuscations. We are also interested in evaluating the effectiveness and performance overhead of obfuscation algorithms.

# **Software protection terminology**

**Obfuscation**—A technique that prevents reverse engineering by ap- plying semantics-preserving code transformations in an attempt to make the code as complex and confusing as possible.

**Watermarking**—A technique used to dissuade a user from illegally re- distributing copies of software by embedding a message w into the program P. If w uniquely identifies the owner of P, then w is a copy- right notice, but if w identifies the purchaser, then it is a **fingerprint. Tamper-proofing**—A technique used to protect a secret from alter- action. Tamper-proofing code must be able to detect that an alteration has occurred and then cause the software to fail in a stealthy manner. In this way, the attacker cannot detect the code that caused the failure.

Malicious host—A computing system that compromises a program’s integrity (the “client”) that it executes. A compromise in this con- text includes reverse engineering, tampering, or piracy.

Malicious client—A program that compromises the integrity of the computing system (the host) on which it executes. A compromise in this context includes leaking or destroying the host’s data.

### **Threat models**

To evaluate a software protection technique’s strength, there must be a well-defined threat model, which de- scribes the tools and techniques an adversary is likely to employ. Manual attack models assume that a programmer skilled in reverse-engineering techniques inspects and modifies the software “by hand.” Automated attack models assume that software protection schemes are attacked with tools that do not require user interaction. The auto- mated attack tools are also called class attacks, an example of which is DeCSS, a C program that subverts the con- tent scrambling system that protects DVDs from unauthorized use.

Most adversaries use a variety of tools and a hybrid at- tack model to execute a single attack. For example, an at- tacker might wish to disable a license check that requires the user to enter a valid registration number. The attacker might begin by locating the registration number input code using a debugger, and setting breakpoints in code

that open a new window in the user interface. He or she might then perform a static data dependency analysis to find where the registration number is used, disassemble the code that uses it, and manually edit this code to disable the license check. Thereafter, the attacker could con- struct a fully automated attack that modifies the program without performing any analysis.

**Sandmark system overview**

Sandmark’s current implementation includes a variety of software watermarking and code-obfuscation algorithms.

The Sandmark architecture is composed of eight main parts: a set of watermarking and obfuscation algorithm plugins, a set of static code analyses, a manager of program objects (classes, methods, and fields that the algorithms can manipulate), a set of software engineering metrics and other static code statistics, a set of manual attack tools, several “obfuscation executives” (which use different heuristics to select an optimal sequence of obfuscations), a graphical user interface (GUI), and a test suite for measuring correctness and evaluating software protection algorithms.

**Text

Description automatically generated**

# **A simple Sandmark plugin**

package sandmark.watermark.constantstring;

public class ConstantString extends StaticWatermarker { public String getShortName() { return “ConstantString”; } public String getAuthor() { return “Christian Collberg”; }

public String getAuthorEmail() { return [“collberg@cs.arizona.edu”;](mailto:collberg@cs.arizona.edu) } public String getDescription() {

return “Embed a watermark in a string in the constant pool”;

}

public String getAlgURL() { return “path to html documentation”; } public void embed(Application app, Properties props)

throws WatermarkingException {

String watermark = props.getProperty(“WM\_Encode\_Watermark”); Iterator classes = app.classes();

if (!classes.hasNext()) throw new WatermarkingException (“There must be at least one class to watermark.”);

sandmark.program.Class aclass = (sandmark.program.Class)classes.next();

aclass.getConstantPool().addString(“sm$watermark=” + watermark);

}

class Recognizer implements Iterator Vector result = new Vector();

int current = 0;

public Recognizer(Application app)

{

throws Exception {

for(Iterator classes = app.classes() ; classes.hasNext() ;) { sandmark.program.Class aclass =

(sandmark.program.Class)classes.next(); ConstantPoolGen cpg = aclass.getConstantPool(); for (int i=0; i < cpg.getSize() ; i++) {

if (cpg.getConstant(i) instanceof ConstantString) { ConstantString s = (ConstantString)cpg.getConstant(i); String v = (String)s.getConstantValue(cp);

if (v.startsWith(“sm$watermark”)) result.add(v.substring(“sm$watermark”.length()));

}

}

}

}

public boolean hasNext() { return current < result.size(); } public Object next() { return result.get(current++); }

}

public Iterator recognize (Application app, Properties props) throws WatermarkingException {

return new Recognizer(app);

}

}

## **Watermarking in Sandmark**

Sandmark’s watermarking module includes both static and dynamic algorithms of varying complexity. Our goal is to develop techniques that will let us determine empir- ically which embedding and recognition algorithms have the smallest performance overhead and the highest re- silience to attacks.

A watermarking/fingerprinting system consists of two functions: embed and recognize. Depending on the algorithm type, these functions can take different argu- ments. In the following, let P be a program, w a water- mark, key a secret key, Pw a watermarked program, and P a probability. The following signatures are possible:

embed (P, w, key)  Pw

recognize (Pw, key)  w

recognize (Pw, P, key)  w informed fingerprint recognizer recognize (Pw, w)  P

blind watermark recognizer recognize (Pw, P, w, key)  P informed watermark recognizer

## **Sandmark manual attack tools**

A common form of attack is to attempt to find code sequences with unique features that could indicate the presence of software protection code.

For example, xor instructions are not commonly found in real code but are commonly used in software protection techniques to encrypt a program’s instructions. To allow attackers to browse and search bytecode for suspicious code, we incorporated a view pane into Sandmark (see Figure 1).

The view pane lets users view an application’s Java bytecode. The application is displayed in a tree structure that illustrates the relationships between packages, classes, and methods. Users can view a method’s bytecode by selecting the desired method in the tree.

## The view pane also facilitates analysis by making use of the statistics module. Users can sort the methods and classes based on size, the number of times a specific instruction is used, or by one of the software complexity metrics the system supports. This type of analysis aids in evaluating the level of obfuscation or in detecting a watermark. **Sandmark evaluation techniques**

There are currently no accepted metrics by which software protection algorithms should be evaluated. Most literature on software watermarking does not empirically or theoretically evaluate these algorithms against attacks, nor do they specify what might consist of a rea- sonable level of attack. It is an important aspect of the Sandmark project to develop evaluation procedures for software watermarking and code obfuscation algorithms and to implement these procedures in Sand- mark’s framework.

We believe a software watermarking algorithm should be evaluated according to the following criteria, which are analogous to the criteria used to evaluate media watermarks:

Data rate. What is the ratio of size of the watermark that can be embedded to the size of the program?

Embedding overhead. How much slower or larger is the watermarked application compared to the original?

False positive rate. Given a random value to the water- mark recognizer, what is the probability that it is recognized as a valid watermark? Similarly, obfuscation algorithms must be evaluated according to the amount of confusion they add to the program, the amount of computational overhead they incur, and their resilience to at- tacks from automatic deobfuscation tools.

Resilience against manual attacks (stealth). Does the watermarked program have statistical properties that differ from typical programs? Can an adversary use these differences to locate and attack the watermark?

Resilience against semantics-preserving transformations. Will the watermark survive transformations such as code optimization and obfuscation? If not, what is the overhead of these transformations? In other words, how much slower or larger is the application after enough transformations have been applied that the watermark no longer can be recognized?

Resilience against collusive attacks. Given two or more differently fingerprinted copies of the same application, can the fingerprints’ location be determined?

**References**

1. C.S. Collberg and C. Thomborson, “Watermarking, Tamper-proofing, and Obfuscation—Tools for Software Protection,” IEEE Trans. Software Eng., vol. 28, no. 8, 2002, pp. 735–746.
2. C. Collberg, C. Thomborson, and D. Low, A Taxonomy of Obfuscating Transformations, tech. report 148, Dept. of Com- puter Science, University of Auckland, New Zealand, 1997; [www.cs.arizona.edu/~collberg/Research/Publications/](http://www.cs.arizona.edu/~collberg/Research/Publications/) ColbergThomborsonLow97a/.
3. C. Collberg, C. Thomborson, and D. Low, “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs, Proc. 25th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 98), ACM Press, 1998; [www.cs.arizona.edu/~collberg/Research/Publications/](http://www.cs.arizona.edu/~collberg/Research/Publications/) Coll bergThomborsonLow98a/.
4. C. Collberg, C. Thomborson, and D. Low, “Breaking Abstractions and Unstructuring Data Structures,” IEEE Int’l Conf. Computer Languages (ICCL 98) 1998; www.cs. arizona.edu/~collberg/Research/Publications/Collberg ThomborsonLow98b/.
5. J.P. Stern et al., “Robust Object Watermarking: Application to Code,” Information Hiding, 1999, LNCS 1768,

A. Pfitzman, ed., Springer-Verlag, 2000, pp. 368–378.

1. R. Venkatesan, V. Vazirani, and S. Sinha, “A Graph Theoretic Approach to Software Watermarking,” Proc. 4th Int’l Info. Hiding Workshop, LNCS 2137, J.S. Moskowitz, ed., Springer Verlag, 2001. pp. 157–168.
2. C. Collberg and C. Thomborson, “Software Water- marking: Models and Dynamic Embeddings,” Proc. 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 99), ACM Press, 1999; [www.](http://www/) cs.auckland.ac.nz/~collberg/Research/Publications/ CollbergThomborson99a/index.html.
3. G. Qu and M. Potkonjak, “Hiding Signatures in Graph Coloring Solutions,” Information Hiding, LNCS 1768, A. Pfitzma, ed., Springer-Verlag, 2000, pp. 348–367.
4. R.L. Davidson and N. Myhrvold, Method and System for Generating and Auditing a Signature for a Computer Program, US Patent 5,559,884, to Microsoft, 1996.