Sandmark—A Tool for Software Protection Research

Sandmark is a tool that measures the effectiveness of software-based methods for protecting software from piracy, tampering, and reverse engineering. The Sandmark team's goal is to develop techniques that will let users determine empirically which algorithms have the least performance overhead and the highest resilience to attacks.

Christian Collberg, Ginger Myles, and Andrew Huntwork
University of Arizona

The current widespread interest in protecting software from piracy, tampering, and reverse engineering has been brought to bear for several reasons. First, revenue derived from proprietary software sales is vital to many software vendors' survival. Second, more vendors distribute software in forms that attackers can easily manipulate, such as Java bytecode or Microsoft's intermediate language. (In fact, these distribution formats are essentially identical to source code.) Finally, some new types of software, such as digital rights management (DRM) systems, contain secrets that must be protected from attack. For example, users who can extract the cryptographic key stored in a DRM media player will be able to enjoy any media without having to pay for it.

Sandmark is a research tool we are developing to study the effectiveness of software-based methods for protecting software from piracy, tampering, and reverse engineering. Our ultimate goal with Sandmark is to implement and evaluate all known software-based methods of software protection. Sandmark currently contains several code obfuscation and software watermarking algorithms.[1–8] Sandmark's infrastructure makes it easy to add and combine algorithms, evaluate their performance and effectiveness, and launch automatic and manual attacks against watermarking and obfuscation algorithms. It is our hope that the tool will prove useful to software protection researchers in fairly evaluating their algorithms, to potential software protection users who wish to evaluate different protection mechanisms, and to software developers who wish to protect their software from piracy, reverse engineering, or tampering by using software protection algorithms.

Sandmark provides protection against *malicious host attacks*, which attackers launch to extract or destroy part of a program. These attacks typically target proprietary algorithms, cryptographic keys, and program registration checks. The watermarking and obfuscation algorithms that Sandmark provides can deter these types of attack. Users can use obfuscation to make it difficult for an attacker to locate sensitive information and can use watermarking to mark sensitive information so that it can be traced to the person who distributed it illegally.

Previous articles have discussed algorithms for code obfuscation, software watermarking, and tamper-proofing in more detail. An understanding of these algorithms will aid in the understanding of why Sandmark was developed.[1–4,7] In this article, we describe Sandmark's capabilities and overall design and how researchers can use it to test and evaluate these algorithms.

Software protection

There have been a variety of techniques proposed for software protection both in hardware and software. The hardware-based approaches range from the use of dongles to tamper-proof processors. The Sandmark framework is designed for the implementation and evaluation of software-based techniques such as software watermarking and code obfuscation.

Watermarking

Software watermarking deters intellectual property theft by embedding a secret message into a program. Watermarking a program is similar to adding a copyright notice to a textual document to assert copyright ownership. If the message uniquely identifies the program's seller, then

Software protection terminology

Obfuscation—A technique that prevents reverse engineering by applying 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 redistributing copies of software by embedding a message *w* into the program P. If *w* uniquely identifies the owner of P, then *w* is a copyright notice, but if *w* identifies the purchaser, then it is a fingerprint.

Tamper-proofing—A technique used to protect a secret from alteration. 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 context 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.

we call it a *watermark*. If, instead, the message uniquely identifies the program's purchaser, then we call it a *fingerprint*. Pirated software that has been fingerprinted can be traced back to its original purchaser.

Other researchers have proposed many software watermarking algorithms.[5–7,8,9] They typically embed the watermark by reordering parts of the original code or by inserting new code, which has no semantic effect when executed.

Code obfuscation

Code obfuscation protects a secret contained in a piece of software by making the software more difficult to read, understand, and reverse engineer. The secret can be the software's design, algorithms used in the software, or data (such as cryptographic keys) hidden in the software. The protection an obfuscation algorithm affords a program depends on the algorithm's sophistication, the program's size and structure, and the types of attacks to which the program will be subjected. More detailed work on code obfuscation appears elsewhere.[3,4,11]

Threat models

To evaluate a software protection technique's strength, there must be a well-defined *threat model*, which describes 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 automated attack tools are also called class attacks, an example of which is DeCSS, a C program that subverts the content scrambling system that protects DVDs from unauthorized use.

Most adversaries use a variety of tools and a hybrid attack model to execute a single attack. For example, an attacker 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 construct a fully automated attack that modifies the program without performing any analysis.

Given enough time and motivation, a competent attacker can subvert any software protection scheme. Most schemes therefore focus on making the cost of a successful attack as high as possible. For example, a code obfuscation algorithm is an effective protection technique if the cost of reverse engineering the obfuscated application is the same as developing it from scratch. Similarly, a software watermarking algorithm is an effective technique if any attack that renders the mark unrecognizable also significantly slows down the de-watermarked application or disables it altogether. In other words, we want to develop software protection techniques that

- make the cost of an initial attack high enough to dissuade an adversary from trying,
- make the construction of class attack scripts difficult—for example, by making every distributed copy of a program different, and
- make a successfully attacked application unusable by being buggy, too large, or too slow.

The threat model for watermarking and fingerprinting algorithms includes additive, subtractive, distortive, and collusive attacks (see the "Attacks against watermarking systems" sidebar, p. 44). With each of these attacks, we assume that the adversary has knowledge of the watermarking technique but is unaware of the embedded message and the secret key. As noted earlier, the Sandmark framework contains tools for launching these types of attacks and evaluating the extent of their success.

Sandmark system overview

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

Software protection tools and their capabilities

Cloakware (www.cloakware.com)—Significant control and dataflow obfuscations of C source code.

DashO and Dotfuscator (www.preemptive.com)—Dead code removal and identifier renaming for Java and MSIL, respectively.

Semantic Designs' Source Code Obfuscators (www.semanticdesigns.com)—Identifier renaming and optional whitespace removal for several high-level languages.

See www.cs.auckland.ac.nz/~cthombor/Students/hlai for a list of many other obfuscation and watermarking tools.

The tool itself consists of 110,000 lines of Java source code and operates on Java bytecode. The implementation has been a team effort that has included 23 students, two professors, and one staff programmer over two years. To manipulate class files, Sandmark depends on the ByteCode Editing Library (BCEL; http://jakarta.apache.org/bcel), which provides a simple and easy-to-use interface to Java bytecode editing.

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.

We designed Sandmark's system architecture to make implementing new software protection algorithms as straightforward as possible. Each algorithm implements as a plugin to a simple interface. For example, to implement a new static watermarking algorithm, a programmer needs only to write a new Java class extending the class `StaticWatermarker`. (See the "A simple Sandmark plugin" sidebar for an example.) Any class located in the Sandmark codebase that extends one of several algorithm base classes (such as `StaticWatermarker`, `DynamicWatermarker`, `MethodObfuscator`, and so on) will load automatically and integrate into the Sandmark infrastructure, including the GUI, the regression test suite, and the obfuscation executives.

Although Sandmark strives to make extending the system with new capabilities as simple as possible, implementing new obfuscation and watermarking algorithms is not trivial. Programmers must understand basic compiler techniques (such as static analysis and code optimization) as well as in-depth knowledge of Java bytecode and the BCEL bytecode editing toolkit.

Because software protection algorithms require semantics-preserving code transformations, many of the same analyses used in code optimization are also applicable to watermarking and obfuscation. Sandmark currently implements several common static code analyses including control flow analysis, liveness analysis, inheritance hierarchy analysis, and execution stack element type analysis.

Control flow analysis is used in several Sandmark algorithms in which it is more natural to modify a method's control-flow graph than its instruction list representation. We currently use *liveness analysis* when implementing a watermarking algorithm that embeds the watermark in a program's methods register allocation, but it could also be used by obfuscations that increase obscurity by increasing the live ranges of variables. Any algorithm that changes types of method parameters or moves methods between classes uses the *inheritance hierarchy analysis*.

The stack contents often must be spilled to local variables during an algorithm's execution. Because the spill instructions depend on the types and number of stack elements, we must perform *execution stack element type analysis*.

Sandmark provides a programmer with a simple representation of an application and its component classes, fields, methods, and files, referred to as *program objects*, which ease the task of writing new plugins in several ways. Given a method object, a plugin writer can easily obtain its control flow graph, query the method for its place in the class inheritance hierarchy, or request the values of software engineering metrics applied to the method. In addition, the program object application programming interface (API) contains a cache consistency protocol that ensures the different representations are consistent.

Sandmark's tools that evaluate and attack watermarking and obfuscation algorithms, including standard benchmark suites `specjvm`, `caffeinemark`, and the `kaffe` regression tests, are well-known benchmarks for testing and performance evaluation of Java compilers and virtual machines. Sandmark supports a viewer that can display classes, methods, and bytecode sorted by various metrics. Users can take this viewer and simulate a collusive attack against fingerprinted software with the bytecode comparison tool.

Common optimizations also can effectively attack some watermarks. Consider, for example, an algorithm that embeds a watermark by reordering instructions. An optimizing transformation that reorganized the code by moving or deleting code segments would destroy this type of watermark. Sandmark, therefore, provides access to the BLOAT bytecode optimizer (www.cs.purdue.edu/homes/hosking/bloat). Optimizations are also useful for reducing the performance impact that obfuscation and

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"; }
  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 often incur.

To allow novices to easily experiment with different types of software protection algorithms, Sandmark supports a GUI that gives easy access to obfuscations, watermarking algorithms, and various configuration and support tools. Its GUI presents users with seven panels that characterize the seven main types of operations it supports: static watermarking, dynamic watermarking, obfuscation, optimization, viewing of software engineering metrics, viewing of Java bytecode, and a bytecode comparison tool.

Attacks against watermarking systems

Assume Alice develops a program P that she watermarks with her message *w* and secret key *k* prior to selling a copy to Bob. Bob must destroy Alice's watermark before he can illegally redistribute P. Bob could employ additive, subtractive, distortive, and collusive attacks in an attempt to destroy Alice's watermark.

- In an *additive attack*, Bob adds a watermark of his own to Alice's program. Alice may be unable to prove that her watermark was embedded in the program prior to Bob's.
- In a *subtractive attack*, Bob examines the program in an attempt to discover the watermark and remove it. He must perform the removal in such a way that the program's functionality is maintained.
- In a *distortive attack*, Bob applies one or more semantics-preserving transformations to the watermarked application in an attempt to destroy the watermark. Bob aims to render Alice's watermark unrecognizable but preserve the functionality of P.
- In a *collusive attack*, Bob obtains two differently fingerprinted programs P1 and P2. He attempts to isolate the fingerprints' location by comparing them, and then he removes them.

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 empirically which embedding and recognition algorithms have the smallest performance overhead and the highest resilience to attacks.

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

embed $(P, w, \text{key}) \rightarrow P_w$
recognize $(P_w, \text{key}) \rightarrow w$

recognize $(P_w, P, \text{key}) \rightarrow w$
  informed fingerprint recognizer
recognize $(P_w, w) \rightarrow$ P
  blind watermark recognizer
recognize $(P_w, P, w, \text{key}) \rightarrow$ P
  informed watermark recognizer

Typically, embed takes P, w, and a secret *key* as its arguments and produces a new program P_w in which w has been embedded. Ideally, without access to the key, an attacker should be unable to extract the watermark.

An *informed recognizer* has access to the original unwatermarked program, whereas a *blind recognizer* does not. A watermark recognizer returns the probability of w existing in P_w. A *fingerprint recognizer*, on the other hand, returns the mark w (typically an integer value) stored in P_w.

Informed recognizers (such as Julien Stern's algorithm[5]) have the disadvantage that the unwatermarked program P must be disclosed to the public to prove an intellectual property violation.

If all generated watermarks $w_1, w_2, \ldots w_n$ are known, a watermark recognizer can be turned into a fingerprint recognizer simply by testing whether each w_i occurs in P_w. Pure fingerprint recognizers (such as the CT algorithm[7]) do not require this step.

Software watermarking algorithms can also be classified as static or dynamic. A static watermarking algorithm stores the watermark directly in the program executable; a dynamic watermarking algorithm embeds the watermark in the dynamic state of the program. The information from the program's execution can then be used to recognize the watermark.

Christian Collberg and Clark Thomborson proposed the first dynamic watermarking algorithm, called the CT algorithm.[7] In it, the watermark is embedded in runtime structures built by code inserted for that purpose. The secret key is a sequence of inputs to the application. When the application runs using the secret sequence, the watermark (a graph structure) is built and can be extracted by the recognizer.

Because embedding a static watermark does not require runtime information, the process of embedding and recognition is far simpler, which has led to a plethora of static watermarking algorithms. We have implemented 11 algorithms in Sandmark and others are under development. Of those, Table 1 describes seven simplistic algorithms. The remaining four algorithms are implementations of algorithms described elsewhere.[5,6,8-10, 12]

Obfuscation algorithms

In other literature, Collberg, Thomborson, and Low presented a taxonomy of obfuscating transformations.[2–4] *Layout* obfuscations modify a program's formatting, naming, or meta-information (such as comments). Most commercial obfuscators fall in this category. *Control* obfuscations modify a program's control flow, for example, by inserting bogus branches using *opaque predicates*. *Data* obfuscations modify the storage and encoding of data structures, such as splitting one variable into multiple parts.

Most obfuscating transformations are constructed based on a few basic principles such as increasing a programming construct's inherent complexity or manipulating the construct so that it is as far removed from the original as possible.

If we let c be a programming language construct

Table 1. Seven simplistic algorithms contained in Sandmark.

ALGORITHM	DESCRIPTION
Bogus Expression	Watermark is embedded as an expression
Bogus Initializer	Watermark is embedded by adding bogus local variables into the constant pool
Constant String	Watermark is embedded in a string in the constant pool
Add Method Field	Watermark is split into two parts, which are embedded in the names of a method and field that are added to the class
Bogus Switch	Watermark is embedded in the cases of a switch block
Method Renamer	Watermark is embedded in the methods' names
Bogus Locals	Watermark is embedded by encoding it as a series of special local variables in which the encoding is based on the local type (each type maps to a base-10 digit that encodes a numerical watermark).

(such as a class, method, statement, or expression), we get the following basic ideas for constructing obfuscating transformations:

- Increase or decrease `C`'s dimensionality, such as the number of dimensions of an array.
- Split `C` into subparts or merge several constructs `C1`, `C2`,… into `C`.
- Wrap `C` in a layer of abstraction or remove a layer of abstraction from it.
- Add or remove a level of indirection to access `C`.
- Reorder two adjacent constructs.
- Rename a labeled construct.

Consider splitting transformations. To split a class `C` we create a superclass `C'` that holds some of the fields and methods of `C`:

```
                          T    class C'{
class C {                 ⇒      int x;
  int x, y;
  method m(){ x++; }             }
}                                class C {
                                   int y;
                                   method m(){ x++;}
                                 }
```

The splitting must be performed so that the identifiers' visibility is not affected. In this case, it would have been wrong for `x` to be in `C` and `m` in `C'`.

Similarly, we can split a basic block (a straight-line code sequence) by inserting a bogus branch:

```
method m (int x, int y){   method m (int x, int y) {
                         T
   x++;                  ⇒     x++;
   y += x;                     if (P^T) y += x;
}                            }
```

P^T is an opaquely true predicate. This means that it always evaluates to `true`, but attackers find determining this invariant to be difficult. Constructing opaque predicates that are computationally hard for an attacker to break is an active area of research.[3]

We can apply the splitting principle to arrays:

```
method m(int i,int y) {      T    method m
  int[10] x;                 ⇒    (int x, int y){
  x[i] = y;                       int [5] x1,x2;
}                                 if (i%2==0)
                                  x1[i/2]=y
                                  else x2[i/2]=y;
                                  }
```

Here, the elements of `x` are distributed over `x1` and `x2`.

As our final example of the splitting principle, we can split a Boolean variable into two integer variables:

```
(1) bool A,B,C;          T    (1')short
                         ⇒       a1,a2,b1,b2,c1,c2;
(2) A = True;                (2') a1=0; a2=1;
(3) B = False;               (3') b1=0; b2=0;
(4) C = A & B;               (4') c1=(a1 ^ a2) &
(5) if (B) …;                     (b1 ^ b2); c2=0;
                             (5') if (b1 ^ b2) …;
```

In this case, we chose to represent `true` as either (0,1) or (1,0) and `false` as (0,0) or (1,1).

It is unlikely that a single application of any of these transformations will add much protection to an application. However, when several different obfuscations are applied to the same piece of code, the result will be far removed from the original. Because most obfuscations have some overhead associated with them, there will always be a trade-off between the level of obfuscation and the performance overhead induced.

Obfuscation in Sandmark

Sandmark currently contains implementations of more than 25 obfuscation algorithms and several different obfuscation executives that automatically select optimal sequences of obfuscations.

Several of Sandmark's obfuscations are *reordering obfuscations* that reorder method parameters, basic block instructions, local variables, and constant pool (Java's symbol table) entries. The parameter reordering obfuscation takes care to reorder parameters so as not to interfere with the program's overloading structure. The following transformation is illegal because both methods have the same signature in the transformed code:

```
class C {                    T    class C {
   method m(int a,           ⇒       method m(int a,
     float b){ }                       float b) { }
   method m(float x,                 method m(int x,
     int y)  { }                       float y) { }
}                                  }
```

Sandmark provides an inheritance hierarchy module (which describes inheritance relationships between classes and methods) that lets obfuscations check if particular changes to method signatures are legal. It also provides a data dependency graph module that lets obfuscations verify that two instructions can be swapped without violating data dependency constraints.

Reordering obfuscations are simple to implement and have minimal performance impact, but they only add a small amount of confusion for the attacker. However, they are very useful when protecting against collusive fingerprinting attacks. By applying different reordering obfuscations to all copies of a program, a software distributor can ensure that each distributed copy differs from all others. As a result, an attacker who tries to compare two differently fingerprinted applications will find that they differ everywhere, not just in the location where the fingerprint was inserted.

Several splitting obfuscations are available in Sandmark. Classes can be split, as can basic blocks and arrays. A special node-splitting transformation is available as an effective (albeit expensive) attack against the CT watermarking algorithm. (CT embeds the watermark in the topology of a linked graph data structure.) The node-splitting obfuscation breaks every node in linked data structures (such as lists, trees, and graphs) into two parts, with an extra field in the first part linking the two together.

Sandmark supports obfuscations that merge methods, parameters, and classes. In the example below, the two classes `C1` and `C2` do not share any common behavior but can be merged using *false refactoring*. The idea is to add a bogus class `C3` as the parent of both `C1` and `C2`. If both classes have instance variables of the same type, these can be moved into `C3`. Its methods can be buggy versions of some of `C1` and `C2`'s methods:

```
                                   class C3 {
class C1 {                   T       int k;
  int x;                     ⇒       method m() { k+=2;}
  float y;                         }
  method m() { x++;  }           class C1 extends C3 {
}                                    float y;
class C2 {                           method m() { k++; }
  int a;                         }
  boolean z;                     class C2 extends C3{
  method n() { a--; }                boolean z;
}                                    method n() { k--; }
                                   }
```

Many standard code optimizations also make good obfuscations. For example, Sandmark provides a method inliner, which substitutes the method body for the method call. By inlining methods, we remove a level of abstraction that makes the program harder to understand.

Sandmark supports several obfuscation executives, the goal of which is to pick an optimal set of transformations and program parts to obfuscate. The user guides the executive by indicating what "optimal" means: for example, how much execution overhead he or she can accept, how much obfuscation to add, and which parts of the application are security- or performance-critical.

The executive is essentially a loop that repeatedly chooses a part of the application to obfuscate, chooses an appropriate obfuscating transformation from a pool of candidate algorithms, and then applies the transformation. An obfuscation executive algorithm must address several issues:

- The executive must decide when the obfuscation process should terminate.
- The user must be able to indicate the amount of protection each part of the program requires and the amount of overhead each part can absorb.
- The executive must ensure that obfuscations are applied in an acceptable order. This is essential because an obfuscating transformation destroys the application's structures. This makes the obfuscated application more difficult to analyze, and, as a result, it might not be possible to apply any further transformations.

Obfuscations should preserve program semantics. In most cases, each obfuscation algorithm can determine whether it can be safely applied by performing various types of static analysis. Sometimes, however, this is not possible. For example, it is generally not possible to determine if a class will dynamically load by name. If this is so, an otherwise safe obfuscation that changes a class's name will create incorrect behavior by causing the dynamic load to fail. The executive is designed to prevent this and similar situations by letting the user specify which parts of the program may exhibit unusual behavior, such as being loaded by name, executed asynchronously, and so on.

Sandmark currently supports three obfuscation executives. The first is a simplistic set-based model that continuously updates the list of algorithms that can run (referred to as *candidates*) using set subtraction, and then uses heuristics to choose the next algorithm to run from the set of candidates. The second is an exhaustive, deterministic, finite-state automaton (DFA)-based model that accurately captures all dependencies between obfuscation algorithms using a finite-state machine model. Unfortunately, the resulting DFAs are huge, and the executive performs poorly for large programs. The third executive is a "lazy" version of the DFA-based model that improves performance by building the finite state machine on demand.

Statistics module

One important technique used to analyze programs is computing various statistics associated with the program,

such as the number of occurrences of a particular opcode in each method. We can use these statistical values in a number of ways, in particular to evaluate a program's complexity before and after code obfuscation. Statistics also can aid the type of manual analysis that many adversaries perform to isolate a watermark.

Included in Sandmark is a statistics module that computes static code statistics. The statistics collected include the number of classes, methods, and fields; the depth of the inheritance hierarchy; the number of local variables; the number of conditional statements; the number of API calls; the level of loop nesting in a method; and the frequency of opcode instructions. We can use these statistics to manually evaluate the program or to compute various software complexity metrics. Sandmark's statistics module includes six standard complexity metrics.[13–18]

The statistics module makes it possible to evaluate aspects of a software protection scheme's overall effectiveness from both developers' and adversaries' viewpoints. For example, if a software watermarking scheme modifies a program to such an extent that its statistical properties are far removed from that of other "normal" programs, then this could indicate to an attacker that the program is, in fact, watermarked. He or she may then examine the program closer by concentrating efforts on those methods with highly unusual statistical properties.

Obfuscation executives also use software complexity metrics to evaluate how much obfuscation has been applied to a program. After each round of obfuscation, the executive evaluates the change in complexity of each part of the program. Complexity changes guide the executive to decide which part of the program most needs obfuscation.

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.

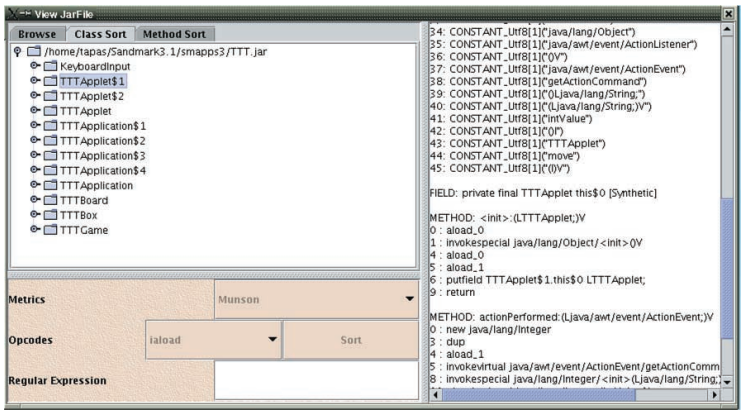

Figure 1. The Sandmark bytecode view pane, which lets users view the bytecode of any method in an application.

Software fingerprinting's most serious problem is its susceptibility to collusive attacks. An adversary who obtains several differently fingerprinted copies of the same software can compare them to find the fingerprint's location. This type of attack can be simulated in Sandmark by using the Java bytecode comparison tool. This tool accepts two Java programs as input and produces an interactive view of the code where users can view pairs of similar methods. The methods' bodies are color-coded to make locating the code segments that exist in only one copy easier, as Figure 2 illustrates.

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 reasonable 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 Sandmark'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 watermark recognizer, what is the probability that it is recognized as a valid watermark? Similarly, obfuscation algo-

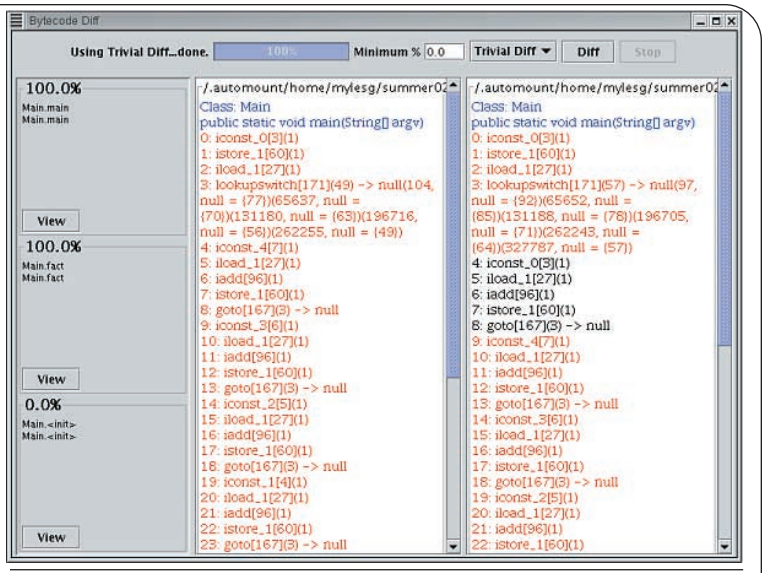

Figure 2. The Sandmark bytecode comparison tool lets users compare the methods' bytecodes. Users can easily identify differences in the bytecode with this tool.

rithms 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 attacks 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?

We are actively developing Sandmark; you can download Version 3.1 from the Sandmark Web site at http://sandmark.cs.arizona.edu. In the future, we hope to provide more comprehensive coverage of algorithms, reverse-engineering tools, and software visualization tools. We want to add tools for dynamic analysis of programs in addition to the static statistics that the current implementation supports. We also plan to support other virtual machine architectures as well as watermarking and obfuscation of native machine code. Future versions will include a collection of tamper-proofing algorithms.[11,19,20] □

Acknowledgments

The US National Science Foundation supported this work under grant CCR-0073483 and the US Air Force Research Laboratory under contract F33615-02-C-1146. Christian Collberg's research is supported by grants and contracts from the National Science Foundation, the US Air Force Research Lab, and Microsoft.

We thank the many University of Arizona faculty, students, and staff who worked on Sandmark, in particular Edward Carter, Kelly Heffner, Zachary Heidepriem, Kamlesh Kantilal, Stephen Kobourov, Jasvir Nagra, Andrzej Pawlowski, Tapas Sahoo, Mike Stepp, Gregg Townsend, and Ashok Venkatraj.

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 Computer Science, University of Auckland, New Zealand, 1997; 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/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.
6. 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.
7. C. Collberg and C. Thomborson, "Software Watermarking: Models and Dynamic Embeddings," *Proc. 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 99), ACM Press, 1999; www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborson99a/index.html.
8. G. Qu and M. Potkonjak, "Hiding Signatures in Graph Coloring Solutions," *Information Hiding*, LNCS 1768, A. Pfitzma, ed., Springer-Verlag, 2000, pp. 348–367.
9. 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.

10. G. Qu and M. Potkonjak, "Analysis of Watermarking Techniques for Graph Coloring Problem," *IEEE/ACM Int'l Conf. Computer Aided Design*, 1998, pp. 190–193; www.cs.ucla.edu/~gangqu/publication/gc.ps.gz.
11. D. Aucsmith and G. Graunke, *Tamper Resistant Methods and Apparatus*, US patent 5,892,899, to Intel Corp., 1999.
12. G. Myles and C. Collberg, "Software Watermarking through Register Allocation: Implementation, Analysis and Attacks," submitted to 4th Int'l Workshop on Information Security Applications (WISA 03).
13. T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, 1976, pp. 308–320.
14. M.H. Halstead, *Elements of Software Science*, Elsevier, New York, 1977.
15. J.C. Munson and T.M. Kohshgoftaar, "Measurement of Data Structure Complexity," *J. Systems Software*, vol. 20, no. 217, 1993, p. 225.
16. S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, 1994, pp. 476–493.
17. S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Software Eng.*, vol. 7, no. 5, 1981, pp. 510–518.
18. W.A. Harrison and K.I. Magel, "A Complexity Measure Based on Nesting Level," *SIGPLAN Notices*, vol. 16, no. 3, 1981, pp. 63–74.
19. H. Chang and M. Atallah, "Protecting Software Code by Guards," *Workshop on Security and Privacy in Digital Rights Management 2002*. LNCS 2320, T. Sander, ed., Security and Privacy in Digital Rights Management, 2002, pp. 160-175.
20. B. Horne et al., "Dynamic Self-Checking Techniques for Improved Tamper Resistance," *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001*, LNCS 2320, Springer Verlag.

Christian Collberg *is an assistant professor at the University of Arizona, in Tucson, Arizona. He received his PhD from the University of Lund, Sweden. Prior to joining the University of Arizona, he was on the faculty at the department of computer science at the University of Auckland, New Zealand. He is an expert in the study of software protection, particularly code obfuscation and software watermarking, and is currently leading the Sandmark project. Contact him at collberg@cs.arizona.edu.*

Ginger Myles *is a PhD student at the University of Arizona. She received her BA with honors in mathematics from Beloit College in Beloit, Wisconsin. Her research focuses on software protection and she is particularly interested in watermarking algorithms. Currently, she is working on the Sandmark project. Contact her at mylesg@cs.arizona.edu.*

Andrew Huntwork *holds an MS in computer science from the University of Arizona and a BS in computer science from the University of Chicago. He is currently working on improving the Sandmark framework and integrating software protection techniques into Java Virtual Machine implementations. Contact him at ash@cs.arizona.edu.*