SandMark Algorithms

Christian Collberg

January 28, 2003

Contents

Part I

Obfuscation Algorithms

Chapter 1

The AddBogusFields Obfuscation Algorithm

Miriam Miklofsky and Ginger Myles

1.1 Introduction

AddBogusFields is an Obfuscation algorithm that will work on a class file. It inserts assignments to a bogus field in specific places throughout the code. These locations are identified by choosing a “sister” field. The code of this algorithm resides in `sandmark.obfuscate.addbogusfields`.

1.2 Apply

To obfuscate the code a new field in added to the class and then assignments are made to this field. The way the obfuscation works is that a “sister” field is identified which is used to insert an assignment to the new field. The sister field is chosen randomly. Everytime there is an assignment to the sister field an assignment to the new field is inserted into the code.

```
newFieldName = ‘‘sm$’’ + i
sisterFieldIndex = randomNumberGen() % fields.length
insert newFieldName in the class
for(each method in class){
   search instruction list for assignment to sisterField
   insert assignment to newField
}
```

Chapter 2

Bogus Predicates Obfuscating Algorithm

Ashok Purushotham , RathnaPrabhu Rajendran

2.1 Introduction

This is an algorithm that implements simple boolean identities and add them to the user's code . The code of this algorithm resides in `sandmark.obfuscate.BogusPredicates`.

2.2 Apply

Our aim is to embed opaquely true constructs which must be stealthy . So we have selected some constructs based on algebraic properties and known facts in mathematics .For example ,we know for all x,y in I, (7(y X y) - 1) is not equal to (x X x) . A list of all the available constructs is maintained .At run time ,whenever we encounter a conditional expression ,we randomly select one among these to append to the current expression .Since the expression that we add is opaquely true,but the reverse engineer has to try out many inputs to find that this added expression is indeed opaquely true,if he isn't aware of mathematical properties . For example ,the original method and its transformation on applying bogusPredicates Algorithm:

```
main(){
 int a=10;
 int b=20;
 if(a<30)
   b=a+99;

}

main(){
 int a=10;
 int b=20;
 int c;
 if(a<30 && c(c+1)%2 ==0)
   b=a+99;
}

```

The original byte code in a conditional expression was

```
  iload_1
  bipush 7
  if_icmpne 29
```

The new added byte code as a result of our algorithm is

```
  iload_1
  bipush 7
  if_icmpne 29
  iload_3
  dup
  iconst_1
  iadd
```

```
imul
iconst_2
irem
iconst_0
if_icmpne 29
```

Chapter 3

Boolean Variable Splitting

A. Huntwork and X. Zhang

3.1 Introduction

This algorithm detects boolean variables and arrays and modifies all uses and definitions of these variables. In particular, every variable or array element is split into 2 variables or array elements, and the state of the original variable is reflected in the combined state of these 2 variables or array elements. This algorithm consists of two components. One is boolean variable identification. We implement a dataflow analysis algorithm over the control flow graph which can identify the range of possible values of any variable. From the range the variable we can tell if it is a boolean variable (range between 0 and 1). The other is splitting the variables that are recognized. We implement 3 splitting algorithms, described below.

3.2 Boolean variable recognition

Observations

- Boolean variables in method fields have Z type
- Boolean variables in method parameters have Z type
- Boolean functions have Z return type
- Boolean local variables have implicit I type (which means iload is used to load the variable, istore is used to store)
- An assignment to a boolean variable in java source code generates the following java bytecode sequence:

```
ifeq l1
    iconst_0
    goto l2
    l1:
    iconst_1
    l2:
    istore boolean_var
```

- An boolean operation is implemented similarly. For example, b=a&&c generates the following code sequence:

```
iload a
    ifeq l1
    iload c
    ifeq l1
    iconst_1
    goto l2
    l1:
    iconst_0       l2:
    istore_1
```

- An boolean array variable has an explicit *newarray* with type *Boolean*

- The operations over boolean array elements are byte operations (baload, bastore)

From these observations, we can tell that it is easy to recognize a boolean field, method parameter, or array variable. The difficulty resides on the identification of boolean local variables.

Code pattern matching method

Based on our observations, we have developed a method based on code pattern matching which recognizes boolean variables by looking at the code sequences preceding stores to local variables. We compare these sequences with this pattern:

ifeq l1
iconst_0
goto l2
l1:
iconst_1
l2:
istore VAR

If all stores to a variable are preceded by such a pattern, VAR will be considered boolean variable. Both the Sun and IBM java compilers generate the same code pattern for boolean assignment. This pattern accounts for almost all assignments to boolean variables

But it does not work for the following cases:

b=a, whose code sequence is:

iload a
istore b b=f(), (f is a boolean function), whose code sequence is:
invokestatic ¡Method f¿
istore b

Even though this method discovers many boolean variables, it also misses many important cases. Therefore, we developed a better method.

Variable range based method

In this method, we do data flow analysis to identify the possbile value range of any variable, if it is [0,1] , we consider it as boolean variable.

BLOAT CFG code analysis

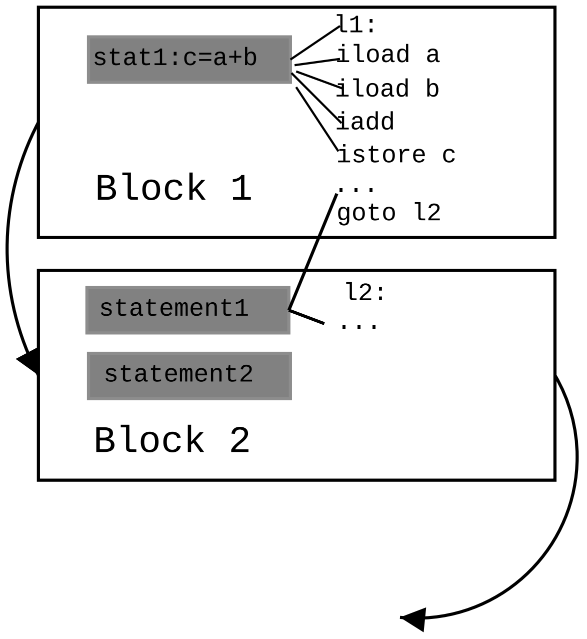
BLOAT does not have a dataflow analysis framework. However, it does have a control flow graph package which constructs the control flow graph and expression tree for a method. This CFG is used in its transformation, optimization and SSA packages.

The basic procedures of BLOAT CFG construction are:

1. Divide the code list into basic blocks (class Block extends Gnode). The division is simple. Every label starts a block, a block consists of the code sequence from one label to the next label. (This is in the buildBlocks method of class FlowGraph)

2. Starting from the first block (first label in the instruction list), BLOAT adds edges between blocks and build expression trees for block. The addition of edges and building of expression trees are in an depth first order. The depth first order means that if it finds a branch instruction, it calls itself with the target block as the parameter. Finally, BLOAT constructs a very complex graph. Blocks and the edges between them compose a conventional CFG. The CFG also consists of a forest of expression trees, the roots being statements, the edges representing the data dependence. A tree may include nodes from multiple basic blocks. (This is in the buildTreeForBlock method of class FlowGraph).

3. Add expression handler edges. Understanding this very complicated procedure is unnecessary for our purposes.



Expression tree construction is also a tedious procedure. BLOAT has a function to deal with the expression tree corresponding to **every instruction**. BLOAT also simulates the stack. For one instuction, it pops the operands from the stack, then pushes the expression of this instruction onto the stack and corresponding edges are added between this expression and the operand expressions. If this instruction does not represent an expression, nothing is pushed onto the stack. Instead, an statement is generated which is actually the root for the expression tree. (The code is in Tree class)

The CFG traversal is a DFS (anti-DFS) traversal over blocks. Inside a block, statements are traversed one by one.

Dataflow analysis

Given the complexity to build a pratical DF analyzer (the exception handler code in particular frightens the authors), we don't want to build a DF analyzer from scratch. Most of the components of a standard DF analyzer are already there. Blocks and edges between them compose a conventional CFG which can be used to do dataflow analysis. The functions which deal with expression construction for every instruction can be modified to do unit data flow procedure. The simulated stack can be used to flow the data. Our solution follows from these facts.

1. Divide the code list into blocks using the same method as CFG block division.

2. Add edges between blocks. This is also part of the CFG building function. At this point, we have a conventional CFG *dcfg*(without the confusing expression tree).

3. Generate a topological order for *dcfg*. We have to get the dominance to break a loop.

4. Traverse the blocks in a topological order. Store the range information in the stack. Store 0 for iconst_0, Store 1 for iconst_1, Store the current range of *var* for *iload var,* Store 0 for boolean function. etc. The meet operation of the data flow is simply the union of all the preceding stacks.

5. Finally, if a variable has [0,1] as its range, it is marked a boolean variable. We also mark boolean array, Boolean, Boolean array.

It turns out to be a nice solution to boolean local variable identification. It considers those integer variable with range [0,1] as boolean. But it doesn't matter.

3.3 Splitting Techniques

```

public class Test {
    public static void main(String argv[]) {
        int i;
        boolean b,a[] = new boolean[16];
        b = false;
        for(i = 0 ; i < 16 ; i++) {
            a[i] = b;
            b = !b;
        }
        for(i = 0 ; i < 16 ; i++) {
            System.out.println(a[i]);
        }
    }
}

```

The boolean variable splitting techniques presented below will be demonstrated on the sample code above.

XOR Splitting

The basic idea of the XOR splitting technique is as follows:

```
        boolean a = true;
        boolean b = false;
```

is converted to:

```
        boolean a1 = true,a2 = false;
        boolean b1 = false,b2 = false;
```

or

```
        boolean a1 = false,a2 = true;
        boolean b1 = true,b2 = true;
```

In general, a single boolean variable will be replaced with a pair of boolean variables such that the state of the original variable at any program point is equal to the XOR of the 2 new variables. Likewise, a boolean array 'a' will be replaced by an array 'b' with twice as many elements, in which the state of array element a[i] in the original array at any program point is equal to the XOR of b[2*i] and b[2*i + 1].

Parity Splitting

The basic idea of parity splitting is to replace a single boolean variable with a pair of integer variables that sum to an even number at every program point where the original variable is false, and sum to an odd number otherwise.

Equality Splitting

The basic idea of equality splitting is to replace a boolean variable with a pair of integer variables that have equal values at every program point where the original variable is true, and unequal values otherwise.

3.4 Code Transformation Techniques

Simple Variable Transformations

Our implementation only obfuscates boolean variables where all assignments to the variables are in one of the following forms:

```
iconst_0
istore_1

<or>

iconst_1
istore_1
```

or

```
<compute integer condition>
ifeq L1
iconst_1
goto L2
L1:
iconst_0
L2:
istore_1
```

These patterns are easily modified to work as split variables:

```
ifeq L1
iconst_0
iconst_1
goto L2
L1:
iconst_1
iconst_1
L2:
istore_1
istore_2
```

Boolean Array Splitting Techniques

Our implementation only obfuscates arrays that are allocated in a function and used in specific ways exclusively in that function. Allocation must look like this (all examples use the XOR obfuscation):

```
newarray boolean
astore_1
```

This pattern is converted to this pattern in the obfuscated code, which produces an array that is twice as long:

```
iconst_1
ishl
newarray boolean
astore_1
```

All stores must look approximately like this:

```
aload_1
iload_2
iload_2
ifeq L1
iconst_1
goto L2
L1:
iconst_0
L2:
bastore
```

This pattern is obfuscated as follows:

```
aload_1
iload_2
iconst_1
ishl
dup2
iload_3
ifeq L1
iconst_1
dup_x2
pop
iconst_0
goto L2
L1:
iconst_0
dup_x2
pop
iconst_0
L2:
bastore
bastore
```

All uses must look approximately like this:

```
aload_1
iload_2
baload
```

This pattern is obfuscated as follows:

```
aload_1
iload_2
iconst_1
ishl
dup
iconst_1
iadd
aload_1
swap
baload
dup_x2
pop
baload
ixor
```

Chapter 4

Class-Splitting and False Refactoring Obfuscation Algorithms

Ashok and Rathna Prabhu

4.1 Introduction

According to the Chidamber Metric [1], the complexity of a class grows with its depth in the inheritance hierarchy and the number of its direct descendents . This is an implementation of the class-splitting/False Refactoring obfuscation proposed in [2]. In the class-splitting obfuscation, we analyze the dependences between the fields and methods of a given class. Using this dependence information, the class is split into different classes forming an inheritance hierarchy, thus obfuscating the given class. In the second algorithm, we do refactoring of two unrelated classes i.e we identify some common fields in the two classes and move them into a bogus super class. Now these two classes appear to inherit from a common class. But in reality, the two classes are unrelated. The algorithms are described in the following sections with examples.

4.2 Class Splitting Obfuscation

A class C is broken into two classes C1 and C2 , such that C2 inherits from C1 . C1 has fields and methods that only refer to themselves, whereas C2 has fields and methods that can refer to themselves as well as fields and methods in C1 . Bytecode references to C will have to be replaced with references to C2. Consider the following simple code sample C.java, to which we'll apply the obfuscation technique:

```
 class C {
  float f;
  int v;
  public C()  {  }
  public void P()
  {
   f=1.2;
  }
  public void Q()
  {
     v = 1;
     P();
  }
}
```

Figure **??** shows the different phases involved in the algorithm.

Dependency Graph Generation

We create a dependency graph G for class C .The nodes of G are the members of C. There is an edge from A to B in G, if the declaration of A must be in scope of B.If there is a path from C to Y in G , then Y must be declared in the child class during splitting. If there is a path from X to Y in G , then either X and Y are both declared in the same class or X is declared in the parent class C1. Figure **??** shows the dependency graph for the given code sample.

Topological sort of the Dependency Graph

We perform a topological sort of the dependency graph to know what methods and variables are to be put in which classes . For the dependency graph created, topological sort gives the following results:

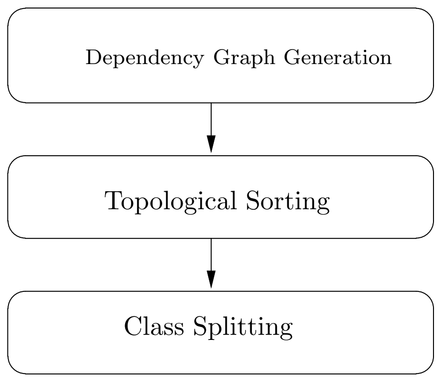


Figure 4.1: Phases in the Class-Splitting algorithm

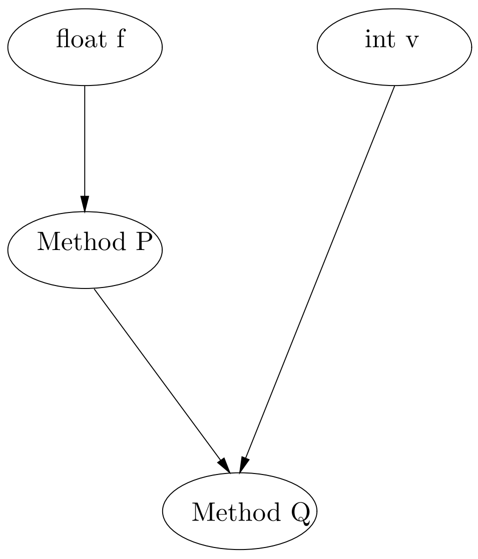


Figure 4.2: Dependency Graph

```
Level 0 : Field f
Level 1 : Method P
Level 2 : Method Q
```

From the above results we can decide on the placement of the methods and fields in the inheritance tree. For instance, the method P must be in the scope of the class in which Q is defined. So Q can be put in the same class as P or in some class inheriting from the class defining P. This ensures that the obfuscation is semantics preserving and gives the same results as before.

Results

The result of applying the algorithm to C.java is as follows :

```
class C0 {
        float f;
        int v;
        public C0() { }
 }
class C1 extends C0 {
        public C1() { }
        public void P() {
         f=1.2
         }
 }
class C extends C1 {
        public C() { }
        public void Q() {
          v=1;
          P();
        }
 }
```

Note that the sub class has been named as C. This makes sure that the references to the original class C, in other java programs are still valid.

4.3 False Re-factoring of Classes

False refactoring is performed on two classes C1 and C2 that have no common behavior. If both classes have instance variables of the same type, these can be moved into the new parent class C3 . C3 's methods can be buggy versions of some of the methods from C1 and C2 . Consider the following sample code

```
C1.java
class C1 {
        String customerName="";
        public void process() {
         //Process customer
         customerName=...
        }
 }

C2.java
class C2 {
        String furnitureName="";
        public int polish(){
         furnitureName=...
         //Polish the furniture
        }
 }
```

After false refactoring the classes:

```
class SMBaseC1 {
        String smstring=""
        public void process() {
          //Buggy code
        }
}

class C1 extends SMBaseC1 {
        public void process() {
          smstring=...
        }
}

class C1 extends SMBaseC1 {
        public int  polish() {
         smstring = ...

        }
}
```

If one of the classes inherit from some other class other than java.lang.Object, then SMBaseC1 has to be an interface and the two classes have to implement SMBaseC1 instead of extending it.

4.4 Reference

1. Chidamber, S. and C. Kemerer, Towards a Metrics Suite for Object Oriented Design, Proceedings of OOPSLA, July 1991.

2. Breaking Abstractions and Unstructuring Data Structures - Christian Collberg, Clark Thomborson, Douglas Low

Chapter 5

The Degrading Obfuscation Algorithm

D. Mandel, A. Segurson

5.1 Introduction

The Degrading algorithm for obfuscation is designed to slow down java applications. It consists of two main parts, the promotion obfuscator, and the thread contention obfuscator. There is a motivation behind the desire to make java applications run slower: Software company X wants to release a new program. But, company X wants the users to be able to try the new software before they buy it, as a motivation to own it. If the customer already has a working version of the software, why would they still buy it? Because company X only released a slightly crippled version of the software and the customer will enjoy the optimal speed of the full version. Thus, company X will degrade their product and release it as a free demo, and sell the fastest working version.

What makes java programs run slower? Memory managment, and thread contention are two things in java code that we focus on that can slow things down. Our goal is to increse the amount of these two things in the java application with some control.

The following is the discussion of the two algorithms used in the Degrading Obfuscation, followed by some analysis of programs in which the degradation was applied.

5.2 Promotion Obfuscation

The main goal of the promotion obfuscator is to increase the amount of memory management in the java application. Almost everything in java is already an object, so the target was clear: primitive local variables. Primitive local variables are used quite frequently in programming, which in java includes the types: " `boolean`, `byte`, `char`, `int`, `double`, `float`, `long`, and `short`." Luckily, java already has provided us with objects that can hold these values.

The details on how to promotote the primitive local variables to an object local variables vary slightly from primitive to primitive. The basic idea follows these steps:

- cast all primitive variable parameters passed into the method to local objects
- when a primitive variable is loaded, replace the load with a call to the local object's value function
- when a primitive variable is stored, replace the store with a constructor for a new object and store the new object

Special `int`'s

The primitive `int` is probably the most common used primitive for a local variable. Also, on the java byte code level the primitives `boolean`, `byte`, `char`, and `short` are treated exactly the same as `int` variables. See the figure for an example. So, there is no support for `boolean`, `byte`, `char`, and `short`'s on the byte code level, they are all `int`'s.

As a result of java's treatment of the small primitives (`boolean`, `byte`, `char`, and `short`), the small primitives can all be stored in the object, `Integer`. There are objects for all of the small primitives, but detailed analysis of the program would be required to use the same object as small primitive, and there would be no beneficial results. For example, one could do data analysis on the local variable 2, and if it is an integer which is always 0 or 1 then it could be stored in a `Boolean` object, but saving it in the `Boolean` object offers no advantage over saving it in a `Integer` object. For the rest of the section `int`'s will include all small primitives.

```
  public static char myMethod(short b, int c, boolean bb) {
     if (bb) {
        int x = b + c;
        return 'a';
     }
     return 'b';
  }
```

```
.method public static myMethod(SIZ)C
.limit stack 2
.limit locals 4

Label1:
.line 7
        iload_2
        ifeq Label0
.line 8
        iload_0
        iload_1
        iadd
        istore_3
.line 9
        bipush 97
        ireturn
Label0:
.line 11
        bipush 98
Label2:
        ireturn

.end method
```

Figure 5.1: The listing of the java, and the java byte code for the method, `myMethod`. `myMethod`'s signature in the byte code accepts a `short`, `int` and a `boolean` as paramaters, but when the method uses these paramaters it always uses an `iload`. Also, when the method returns the `char` it uses an `ireturn`.

Promoting Primitives

There are only four main types of primitives: `int`, `float`, `long` and `double`. To implement the promotion obfuscator four method obfuscator algorithms were added to Sandmark:

- DPromote - `double` promote
- FPromote - `float` promote
- IPromote - `int` promote
- LPromote - `long` promote

All four method obfuscator algorithms begin by loading all of the primitive parameters into an object. Figure **??** is the new byte code created for each primitive paramter type at the beginning of the method.

Now, there are only local variables that are objects. But, we still have to change the rest of the method to correct the stores and loads to reference the objects, instead of the primitives. Figures **??** - **??** itemize all the substitutions that need to be made.

Long	**Float**

`new java/lang/Long`
`dup`
`lload 4`
`invokespecial java/lang/Long/<init>(J)V`
`astore 4`
	`new java/lang/Float`
	`dup`
	`fload_2`
	`invokespecial java/lang/Float/<init>(F)V`
	`astore_ 2`

Int	**Double**

`new java/lang/Integer`
`dup`
`iload_3`
`invokespecial java/lang/Integer/<init>(I)V`
`astore_3`
	`new java/lang/Double`
	`dup`
	`dload_0`
	`invokespecial java/lang/Double/<init>(D)V`
	`astore_0`

Figure 5.2: The code inserted at the beginning of the method for each of the four primitive types. In this example the signature of the static method was `myMethod(DFIJ)V`.

Old Byte Code	Replacement Byte Code

`istore <index>`	`new Ljava/lang/Integer;`
	`dup_x1`
	`dup_x1`
	`pop`
	`invokespecial java/lang/Integer/<init>(I)V`
	`astore <index>`

`iload <index>`	`aload <index>`
	`invokevirtual java/lang/Integer/intValue(V)I`

`iinc <index><num >`	`new Ljava/lang/Integer;`
	`dup`
	`ldc <num>`
	`aload <index>`
	`invokevirtual java/lang/Integer/intValue(V)I`
	`iadd`
	`invokespecial java/lang/Integer/<init>(I)V`
	`astore <index>`

Figure 5.3: Substitution for integers.

Controling Promotion

The number of local variables that are promoted can be controled to a limited extent. The following parameter can be set in the Degrade configuration:

- `OBF_DE_PERCENT` - The approximate percent of the primitive local variables that you want to promote.

The algormithm will then apply the promotion obfuscation to methods in the jar file until it has reached or exceeded the desired percent. It uses the greedy algorithm to achieve this.

Old Byte Code	Replacement Byte Code

`lstore <index>`	`new Ljava/lang/Long;`
	`dup_x2`
	`dup_x2`
	`pop`
	`invokespecial java/lang/Long/<init>(J)V`
	`astore <index>`

`lload <index>`	`aload <index>`
	`invokevirtual java/lang/Long/longValue(V)J`

Old Byte Code	Replacement Byte Code

`fstore <index>`	`new Ljava/lang/Float;`
	`dup_x1`
	`dup_x1`
	`pop`
	`invokespecial java/lang/Float/<init>(F)V`
	`astore <index>`

`fload <index>`	`aload <index>`
	`invokevirtual java/lang/Float/floatValue(V)F`

Old Byte Code	Replacement Byte Code

`dstore <index>`	`new Ljava/lang/Double;`
	`dup_x2`
	`dup_x2`
	`pop`
	`invokespecial java/lang/Double/<init>(D)V`
	`astore <index>`

`dload <index>`	`aload <index>`
	`invokevirtual java/lang/Double/doubleValue(V)D`

Figure 5.4: Substitution for longs, floats and doubles.

5.3 Thread Contention Obfuscation

This algorithm attempts to automagically slow down some method in a program by inserting threads that perform busy-waiting while the original code in the method runs. While the idea of slowing down a program may seem odd, commercial software developers may find this idea interesting in order to give away a slower version of their product while charging for the complete version. This algorithm may also be used as a benchmarking scheme for java's multithreaded features. In addition, the obfuscation value of this algorithm (when it is completely implemented with sufficient stealthiness) should not be underestimated. Multithreaded code has been known to confuse even advanced programmers, and obfuscated multithreaded code should further help in this quest for confusion.

Apply

For the algorithm to work, the user must have configured the following parameters beforehand:

- `OBF_TC_ClassName` - The name of the class where the method resides
- `OBF_TC_MethodName` - The name of the method to obfuscate
- `OBF_TC_NumThreads` - The number of threads to insert into that method

If a method with the given class and name isn't found, then the obfuscation does nothing. Once these values have been set, the user may run the algorithm. A summary of the steps taken during the obfuscation are as follows:

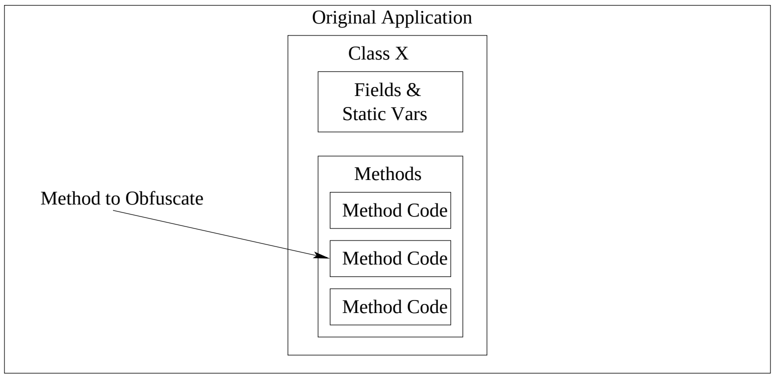


Figure 5.5: The original application to obfuscate. A method is selected for obfscation

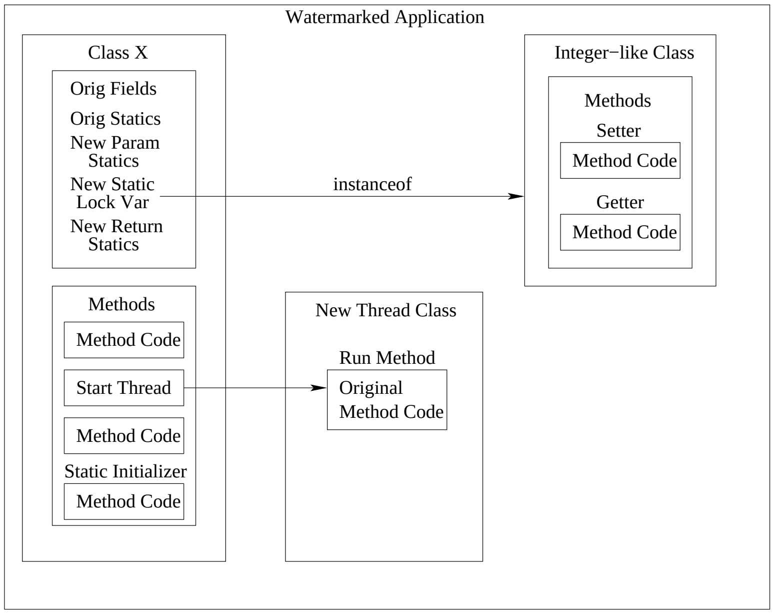


Figure 5.6: The application after obfuscation. The original method's code has been moved to a thread which the original method now starts. An Integer-like class has been added for locking purposes.

- A new `java.lang.Integer`-like class is added to the jar file
- A static initializer is either modified (if it already exists) or created (if it doesn't already exist) and a static lock variable is added to the original class
- A new inner-thread class that contains the original method code is added to the original class
- Some number of other inner-thread classes are added to the original class
- The original method is modified so it starts the new inner thread class and waits for that thread to complete

New java.lang.Integer Class

In order to support various locking methods, the first step in the obfuscation is to add a class similar to `java.lang.Integer`. The main difference between the new class and java's version is that the new class can change the value stored in the Integer object. This difference is significant because it allows a different way of coordinating threads. Java's built-in synchronization mechanism relies on monitors which function through object locks. For our purposes, we want to use this in addition to value-based synchronization mechanisms. The standard `java.lang.Integer` class cannot support value-based locking due to the value being immutable once the object has been created. If one wishes to change the value of an Integer, one must create a new Integer, which throws away the reference to the old Integer, which renders the java's built-in object locking mechanism useless.

Static Initializer and Lock Variable Modified or Added

For locking to take place, there needs to be some sort of locking variable. While typical locking in java uses the `synchronized` keyword in method headers, this synchronizes on the `this` object associated with that method, which defeats our purpose of having more than one thread attempting to run while our original method is running. To get around this, we introduce a new static class variable of our Integer-like class that the methods can synchronize on. A static initializer is added to the class in order to construct our new Integer variable.

New Inner-Thread Class

A new inner-thread class is created in order to run the old method code. The new class contains one method, `run()`, which is required in order to implement the `java.lang.Runnable` interface. The method body of the `run()` method contains the instructions that were formerly in the original method. However, there are two slight details that make this process difficult, and both of them stem from the signature of the `run()` method required by `java.lang.Runnable`. Since the signature of this method is set, it is impossible to pass parameters or return a value in the conventional manner. The workaround for this is to add static class variables to the original class so that both the original class and the new class can access the values. It should be noted that all generated code (with the exception of the first thread class which always has the name `<className>`$Inner) has random identifier strings associated with them. As well, all generated code is declared static so as to be accessible from both a static and dynamic context. For parameter passing, if the code to be edited originally looked like this:

```
public class ParamPassingExample {
   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++) {
         System.out.println(``Arg `` + i + `` = `` + args[i]);
      }
   }
}
```

Then we would add the new class and a static variable so the resulting code and modified `main(String[])` method would look like this:

```
public class ParamPassingExample {
   // New static variable added to mimic parameter-passing
   private static String[] d3345;

   public static void main(String[] args) {
      Thread newThread;
      ParamPassingExample$Inner inner;

      // Assign the parameters to the new static class variable
      d3345 = args;

      inner = new ParamPassingExample$Inner();
      newThread = new Thread(inner);
      newThread.start();
   }

   private static class ParamPassingExample$Inner implements java.lang.Runnable {
      public void run() {
         // Assign the static class variable to a local for manipulation like
         // how a parameter would normally be manipulated
         String[] local = d3345;

         for (int i = 0; i < local.length; i++) {
            System.out.println(‘‘Arg ‘‘ + i + ‘‘ = ‘‘ + args[i]);
         }
      }
   }
```

Return types are treated in a similar manner, in that a new static variable is added to the class. However, for return types to work correctly, the modified original method must spin on the lock until the thread has completed. To illustrate this, consider the following example:

```
public class ReturnTypeExample {

   public static void main(String[] args) {
      System.out.println(‘‘Square root of 25 is: ‘‘ + squareRoot(25));
   }

   public static double squareRoot(int value) {
      return Math.sqrt((double) value);
   }
}
```

After adding the lock variable and modifying the `squareRoot(int)` method to construct the thread and spin until the thread has completed, we have the following code:

```
public class ReturnTypeExample {
   // New static variable for param-passing
   private static int d546;
   // New static variable for return-type
   private static double g6789;
   // New static lock variable
   private static ReturnTypeExample$n7632 k3452;

```

```
    // static initializer added to construct new lock variable
    static {
       k3452 = new ReturnTypeExample$n7632(0);
    }

    public static void main(String[] args) {
       System.out.println(``Square root of 25 is: `` + squareRoot(25));
    }

    public static double squareRoot(int value) {
       ReturnTypeExample$Inner inner;
       Thread thread;

       // Store the parameter in a static class variable
       d546 = value;
       // Then start the thread
       inner = new ReturnTypeExample$Inner();
       thread = new Thread(inner);
       thread.start();
       // Spin on the lock until the thread completes its work
       while (k3452.j213() != 1) {
       }
       // Then return the newly added static variable
       return g6789;
    }

    private class ReturnTypeExample$Inner implements java.lang.Runnable {
       public void run() {
          // Re-assign the value of the lock variable to 0 to delay execution
          // of other threads
          k3452.n4795(0);
          // Assign the parameters to a local variable
          int local = d546;
          // Then assign the original return value to our static class member
          g6789 = Math.sqrt(d546);
          // Then assign the lock value to 1 so other threads can complete
          k3452.n4795(1);
       }
    }

    // Integer-like class
    private class ReturnTypeExample$n7632 {
       private int myValue;

       public ReturnTypeExample$n7632(int value) {
          super();
          myValue = value;
       }

       // Getter method
       public int j213() {
          return myValue;
       }
```

```

      // Setter method
      public void n4795(int value) {
         myValue = value;
      }
   }
}
```

Other New Thread Classes

Any other new thread classes created during the process will have a `run()` method with an empty spin loop waiting until the original thread has completed. In future revisions of the algorithm, these additional threads may have domain-specific code inside of the spin loops, but for now they are relatively unstealthy. As an example, here is a possible inner-thread class:

```
private class SomeClass$s78 implements java.lang.Runnable {
   public void run() {
      // Just spin on some lock variable
      while (b321.m987() != 1) {
      }
   }
}
```

Of course, the original method would also be modified to start this thread as well.

5.4 Sample Results

These experiments were performed on an Intel Pentium III 1.2 ghz machine running Red Hat Linux 7.2. The application that was obfuscated was the famous Sieve of Eratosthenes, an algorithm used to compute all prime numbers up to some **n**. In the test case, the **n** used was 99999999.

Number of threads used	*Percent Promotion*	*Seconds Needed*
1 (original)	0.0	12 seconds
1	1.0	52 seconds
2	0.0	25 seconds
2	1.0	73 seconds
3	0.0	38 seconds
3	1.0	122 seconds
4	0.0	64 seconds
4	1.0	411 seconds

Table 5.1: Sample results obtained using Degradation on the Sieve of Eratosthenes

Chapter 6

The Method 2R Madness Obfuscation Algorithm

Martin Stepp and Kelly Heffner

6.1 Introduction

The Method 2R Madness Obfuscation (M2M) algorithm is based on the idea that useful information about program organization lies in the parameter list and separation of methods. In order to destroy that information, the M2M obfuscator performs a six pass obfuscation:

1. In pass one, all fields and methods are made public.
2. In pass two, all dynamic method bodies are moved into a static helper method. The dynamic shell is simply a call on this static helper.
3. In pass three, all primitive types residing in a static method signature are promoted to their respective object wrapper types.
4. In pass four, the parameter lists of all static methods are rearranged.
5. In pass five, the parameter lists of all static methods are condensed into one large `Object []` array.
6. In pass six, static methods in the same class with the same signature are merged into one method.

6.2 Pass Two - Method Splitting

As mentioned above, the second phase of the M2M obfuscation splits each dynamic method into two parts. The first part is the same dynamic method, except that the body of the method is changed to a simple call on a static helper method. The first parameter to this static method is the actual `this` reference which was implicitly argument zero in the dynamic method. Take for example the following class to computer Ackermann's function:

```
public class Ackermann
{
    public static void main(String[] args)
    {
      int num = Integer.parseInt(args[0]);
      System.out.println("Ack(3," + num + "): " + new Ackermann().Ack(3, num));
      }

    public int Ack(int m, int n)
    {
      return (m == 0) ? (n + 1) : ((n == 0) ? Ack(m-1, 1) :
                  Ack(m-1, Ack(m, n - 1)));
    }
}
```

After being split into a static section by the StaticSplit obfuscator phase, and then run through the SourceAgain decompiler, the code looks as follows:

```
//
// SourceAgain (TM) v1.10i (C) 2001 Ahpah Software Inc
//

import java.io.PrintStream;

public class Ackermann {

    public static void main(String[] as)
    {
        int i = Integer.parseInt( as[0] );

        System.out.println( "Ack(3," + i + "): " + (new Ackermann().Ack( 3, i )) );
    }

    public static int Ackintint0(Ackermann this, int arg0, int arg1)
    {
        return (arg0 == 0) ? arg1 + 1 : (arg1 == 0) ? Ack( arg0 - 1, 1 ) : Ack( arg0 - 1, Ack( arg0, ar
    }

    public int Ack(int arg0, int arg1)
    {
        return Ackintint0( this, arg0, arg1 );
    }
}
```

6.3 Pass Three - Primitive Promoting

The third phase of the M2M obfuscation consists of promoting the argument list and return type of the static methods so that all primitive typed arguments or returns are now reference types. A restriction of reference only parameters and returns allows for easier parameter reordering and array parameterize, and this primitive promoting obfuscation makes it possible for more static methods to fit the “reference only” criteria for reordering. We will again use the Ackermann example from above, and tack on the primitive promoting pass.

```
//
// SourceAgain (TM) v1.10i (C) 2001 Ahpah Software Inc
//

import java.io.PrintStream;

public class Ackermann {

    public static void main(String[] arg0)
    {
        int i = Integer.parseInt( arg0[0] );

        System.out.println( "Ack(3," + i + "): " + (new Ackermann().Ack( 3, i )) );
    }

    public static Integer get0(Ackermann this, Integer arg0, Integer arg1)
    {
        return new Integer( (arg0.intValue() == 0) ? arg1.intValue() + 1 : (arg1.intValue() == 0) ? Ack
    }

    public int Ack(int arg0, int arg1)
    {
        return get0( (Ackermann) new Object[3][0], (Integer) new Object[3][1], (Integer) new Object[3][
    }
}
```

Notice that the decompiler has a problem interpreting that the arguments to method calls were promoted using a temporary array of Objects.

6.4 Pass Four - Parameter Reordering

The parameter reorderer adds an extra dose of confusion to the algorithm, so that there is no direct mapping between the order of the nonstatic stub methods, and their static helpers. This also shuffles the `this` reference out of slot zero, making our static helpers less obvious.

```
//
// SourceAgain (TM) v1.10i (C) 2001 Ahpah Software Inc
//

import java.io.PrintStream;

public class Ackermann {

    public static void main(String[] arg0)
    {
        int i = Integer.parseInt( arg0[0] );

        System.out.println( "Ack(3," + i + "): " + (new Ackermann().Ack( 3, i )) );
    }

    public static Integer get0(Integer this, Ackermann arg0, Integer arg1)
    {
        return new Integer( (arg1.intValue() == 0) ? intValue() + 1 : (intValue() == 0) ? arg0.Ack( arg
    }

    public int Ack(int arg0, int arg1)
    {
        return get0( (Integer) new Object[3][0], (Ackermann) new Object[3][1], (Integer) new Object[3][
    }
}
```

6.5 Pass Five - Signature Bludgeoning

The bludgeon takes the final step in confusing the signatures of methods, by making all of the static method signatures the *same*, “(L[java/lang/Object;)Ljava/lang/Object;”. This prepares the classes for the final pass, which is to combine any methods with the same signature (which is now all static methods).

```
//
// SourceAgain (TM) v1.10i (C) 2001 Ahpah Software Inc
//
```

```
import java.io.PrintStream;

public class Ackermann {

    public static void main(String[] arg0)
    {
        int i = Integer.parseInt( arg0[0] );

        System.out.println( "Ack(3," + i + "): " + (new Ackermann().Ack( 3, i )) );
    }

    public static Object get0(Object[] this)
    {
        return new Integer( (((Integer) this[2]).intValue() == 0) ? ((Integer) this[0]).intValue() + 1
    }

    public int Ack(int arg0, int arg1)
    {
        return ((Integer) get0( new Object[] { (Integer) new Object[3][0], (Ackermann) new Object[3][1]
    }
}
```

6.6 Pass Six - Method Merging

The method merger combines static methods with the same signature into one method, that is multiplexed on an extra `int` parameter. Consider the code for the original Ackermann, with an extra static method added in:

```
public class Ackermann
{
    public static void main(String[] args)
    {
      int num = Integer.parseInt(args[0]);
      System.out.println("Ack(3," + num + "): " + new Ackermann().Ack(3, num));
      }

public static void hi(long p)
{
System.out.println(p);
}

    public int Ack(int m, int n)
    {
      return (m == 0) ? (n + 1) : ((n == 0) ? Ack(m-1, 1) :
                Ack(m-1, Ack(m, n - 1)));
    }
}
```

After a pass through all five other phases, the two static methods (which now share an Object[] to Object signature) are merged together as follows:

```
//
// SourceAgain (TM) v1.10i (C) 2001 Ahpah Software Inc
```

```
//

import java.io.PrintStream;

public class Ackermann {

    public static void main(String[] arg0)
    {
        int i = Integer.parseInt( arg0[0] );

        System.out.println( "Ack(3," + i + "): " + (new Ackermann().Ack( 3, i )) );
    }

    public int Ack(int arg0, int arg1)
    {
        return ((Integer) keldaANDmartio0( new Object[] { (Integer) new Object[3][0], (Integer) new Obje
    }

    public static Object keldaANDmartio0(Object[] keldaANDmartio0, int switchVar)
    {
        if( 1 == switchVar )
            return new Integer( (((Integer) keldaANDmartio0[1]).intValue() == 0) ? ((Integer) keldaANDma
        if( 0 != switchVar )
            ;
        System.out.println( ((Long) keldaANDmartio0[0]).longValue() );
        return null;
    }
}
```

MethodRearranger is a static obfuscation algorithm that moves all of the static methods that it can into their own class. Only static methods that do not rely on private information within its residing class can be moved.

The MethodRearranger works as follows:

1. A list of movable methods is collected, and those methods are moved into MasterStaticClass.
2. Constant pool references within each moved method are updated.
3. Constant pool references for the method calls for those methods are updated to reference the new class.

Chapter 7

The PromoteLocals Obfuscating Algorithm

Danny Mandel and Anna Segurson

7.1 Introduction

This is an algorithm that changes all of the local `int` variables in a method to local `java.lang.Integer` variables.

For this to work every reference to the old local `int` has to be replaced by the appropriate reference to the new local `java.lang.Integer`. We identified three java byte code instructions that needed to be modified to make this change. The following table lists the necessary modifications:

Old Byte Code	Replacement Byte Code

`istore <index>`	`new Ljava/lang/Integer;`
	`dup_x1`
	`dup_x1`
	`pop`
	`invokespecial java/lang/Integer/<init>(I)V`
	`astore <index>`

`iload <index>`	`aload <index>`
	`invokevirtual java/lang/Integer/intValue(V)I`

`iinc <index><num >`	`new Ljava/lang/Integer;`
	`dup`
	`ldc <num>`
	`aload <index>`
	`invokevirtual java/lang/Integer/intValue(V)I`
	`iadd`
	`invokespecial java/lang/Integer/<init>(I)V`
	`astore <index>`

The code of this algorithm resides in `sandmark.watermark.promotelocals`.

7.2 Apply

To apply this obfuscation we circulate through the byte code of a given method and if the instruction is one of the ones in the table above we replace that instruction with the corresponding replacement byte code.

For example, this simple class:

```
public class VerySimp {
   public static void main(String[] args) {
      int a,b,c;
      a = 1;
      b = 22;
      c = a + b;
      for (int i = 0; i < 3; i++) {
         a+=2;
      }
   }
}
```

is obfuscated to the following class:

```
public class VerySimp {
   public static void main(String[] as)
   {
      Integer integer2 = new Integer( 1 );
      Integer integer3 = new Integer( 22 );
      Integer integer4 = new Integer(integer2.intValue() + integer3.intValue());
      Integer integer1 = null;
      for( integer1 = new Integer( 0 ); integer1.intValue() < 3;
           integer1 = new Integer( 1 + integer1.intValue() ) )
         integer2 = new Integer( 2 + integer2.intValue() );
   }
}

```

/algorithmNodeSplitting ObfuscationRathnaPrabhu Rajendran /sectionIntroduction The NodeSplitting obfuscation algorithm obfuscates a class file by splitting a node into two, i.e. some of the fields from the class are moved into a newly created class and all references to those fields in the given class are modified to reflect the changes. /sectionCode Sample

Consider the following code: /beginverbatim class A int i; private float f; public A() public void dummy() i=10; f=12.2;

After Obfuscation : class A int i; A1 next; public A() next = new A1(); public void dummy() i=10; next.f = 12.2; class A1 public float f; public A1()

Part II

Watermarking Algorithms

Chapter 8

The AddMethField Watermarking Algorithm

Miriam Miklofsky and Ginger Myles

8.1 Introduction

AddMethodField is a watermarking algorithm that embeds a watermark by splitting it into two parts. The first part is attached to a field name and the second is attached to a method name. Both the field and the method are new items inserted into the code. The code of this algorithm resides in `sandmark.watermark.addMethField`.

8.2 Embedding

To embed the watermark we split the watermark into two parts, wmPart1 and wmPart2. wmPart1 is combined with “sm$” to create the name of a new field that is added to the class. In order to embed wmPart2 we first randomly choose a method that already exists in the class. This method is chosen by seeding a random number generator with the key. Since the key is provided as a String we convert the String to a long. If a key is not specified then the seed has a default value. The randomly chosen method is used for two different purposes. The first purpose is for the naming of the new method. The name of the random method is used as the first part of the watermarked method’s name, i.e. newMethodName = randomMethodName + “$” + wmPart2. The second purpose is that the chosen method will make a call to the watermarked method, so as to disguise the watermarked method. Since we have inserted a new method into the class, this method should do something. The watermarked method takes two parameters, whose values were also inserted into the chosen method, adds them together and places the result into the watermarked field. This assignment will aid in the recovery of the watermarked field.

```

    //break the watermark into two parts
    wmPart1 = first half of watermark
    wmPart2 = second half of watermark


    //get a pseudorandom number to indicate which method will make a call
    //to the watermarked method.
    //we are going to seed the random number generator with the key
    if(key == empty string){
       seed = 42;
    }else{
       seed = key converted to a long
    }
    methodIndex = randomGenerator(seed) % number of methods in class

    //get the name of method where the watermarked method will be
    //inserted. We are using this name to tack on the second part
    //of the watermark.
    String modMethodName = methods[methodIndex].getName();
    String newMethodName = modMethodName + "$" + wmPart2;

    create a new field

    create a new method
    //This new method takes the values of the two parameters, adds them
    //together and places the result in the watermarked field.
    //the instruction list for the method: load the value of the
    //parameters, add, put the value in the watermarked field, return.
    il.append(load1)
    il.append(load2)
    il.append(putfield(watermarkedField)
    il.append(return)

    add the new method so it becomes part of the class


    //modify the randomly chosen method so that it makes a call to the new
    //method. The instruction list that is to be inserted will be the
    //following: store 5, store 10, invoke watermarked method.
    il2.append(store5)
    il2.append(store10)
    il2.append(invoke watermarked method)
    insert il2 into instruction list of chosen method

```

8.3 Recognition

During recognition the key is again used to seed a random number generator. This is used to determine which method contains the call to our watermarked method. The watermarked method can be distinguished from any other method calls because the begining of its name will be the same as the name of the method from which it is being called. Once the watermarked method is identified the instruction list of the watermarked method is searched to find a putfield instruction. The field that is being assigned to is the watermarked field. Once the two pieces of the watermark are identified they are tacked together to form the watermark that is returned.

```

    //get a pseudorandom number to indicate which method will make a call
    //to the watermarked method.
    //we are going to seed the random number generator with the key
    if(key == empty string){
       seed = 42;
    }else{
       seed = key converted to a long
    }
    methodIndex = randomNumberGenerator(seed) % number of methods in class

    String methodName = methods[methodIndex].getName()
    //extract the second half of the watermark from the added method
    //the watermarked method will have the same beginning as methodName
    newMethodIndex = 0;
    for(each method in the class) {
       if(k != methodIndex) {
          methodName2 = methods[k].getName();
          if(methodName2.startsWith(methodName + "$" )) {
             wmprt2 = methodName2.substring((methodName.length()+1), methodName2.length())
```

```
        newMethodIndex = k;
     }
   }
}

//search the watermarked method’s instruction list for putfield
if(instruction instanceof PUTFIELD){
   fieldName = get field name
   wmprt1 = fieldName - ‘‘sm$’’
}

//put pieces of watermark together
watermark = wmprt1 + wmprt2;

```

Chapter 9

Project VIII: Class-Splitting Obfuscations

Team : Ashok Purushotham and RathnaPrabhu Rajendran

9.1 Mission Document

For our final project we would like to implement a class-splitting obfuscator for Java .

9.2 Basic Idea

The basic idea behind the project is that there are two important object splitting techniques. The first splits each object into two, such that some fields belong to the first subpart, the remainder to the second subpart.

9.3 How we plan to implement

In the bytecode every ”new”-operation turns into two ”new”s and every pointer operation (getfield and setfield) turns into two pointer operations. The second splitting technique splits at the class level. A class C is broken into two classes C 1 and C 2 , such that C 2 inherits from C 1 . C 1 has fields and methods that only refer to themselves, whereas C 2 has fields and methods that can refer to themselves as well as fields and methods in C1 . Bytecode references to C will have to be replaced with references to C 2 .

The reason we want to implement this algorithm is that, to the best of our knowledge, it hasn't been implemented before. We would like to prove this by implementing and evaluating it. The implementation will be done within the SandMark framework. This will allow us to take advantage of the many excellent, well-documented libraries that are available within this tool.

9.4 Project Tasks

We believe this will be a difficult project with many potential pitfalls. First of all, the paper talks shallowly about our project and nothing else ,so we need to work out a reasonable algorithm. Second we can't simply split classes ,but splitting should be based on inheritance principles and understanding all of them and implementing them as a single task will be an uphill task .We therefore identify the following subtasks: 1. Understanding of concepts of splitting classes,going thourgh all relevant material in Sandmark and Collberg's paper. 2. Design 3. Discussing our design and implementation algorithms with Prof.Collberg and Coding 4. Testing and evaluation. 5. Documentation.

9.5 Reference

The reference for this project will be Collberg: Breaking Abstractions and Unstructuring Data Structures.

Chapter 10

Bogus Initializers Watermarking Algorithm

Ashok Purushotham , RathnaPrabhu Rajendran

10.1 Introduction

This is an algorithm that embeds a number in the constant pool of the application. The code of this algorithm resides in `sandmark.watermark.constantstring`.

10.2 Embedding

To embed the watermark we pick one of the user's classes at random and add a few local variables to a method in the class. The initial values of the variables added, store the watermark.Suppose the number to be watermarked is 129091,we create three local variables sm$1, sm$2 and sm$3 and assign 12,90 and 91 to each of them respectively. If the number of digits in the number is an odd number, we append a zero to it, to make it even and split it among the variables appropriately. We also store the number of digits in the original watermark number, which is useful while recognizing the watermark.

```
int temp_len=watermark.length();
int stringIndex = cpg.addString("sm$len" + "=" + watermark.length();
if(temp_len%2 ==0)
    bogus_ids_no=temp_len /2;
else
  {
    bogus_ids_no=temp_len /2 +1;
    watermark=watermark+"0";
    System.out.println(watermark);
    }
de.fub.bytecode.classfile.Method[] methods=cg.getMethods()   ;
de.fub.bytecode.generic.MethodGen mg=
   new de.fub.bytecode.generic.MethodGen(methods[methods.length-1],className,cpg);
int cp_index[] = new int[bogus_ids_no];
de.fub.bytecode.generic.InstructionList il = mg.getInstructionList();
for(int i=0;i<bogus_ids_no;i++){
   String index=String.valueOf(i+1);
   de.fub.bytecode.generic.LocalVariableGen lv=
      mg.addLocalVariable("sm$"+index,de.fub.bytecode.generic.Type.INT,null,null);
   cp_index[i]=lv.getIndex();
   String str_value = watermark.substring((2*i),(2*i+1)+1);
   int int_value=Integer.parseInt(str_value);
   byte b = (byte)int_value;
   il.insert( new de.fub.bytecode.generic.ISTORE(cp_index[i]));
   il.insert( new de.fub.bytecode.generic.BIPUSH(b));
   }

```

10.3 Recognition

During recognition, we go through every class in the watermarked jar-file looking for a string in the constant pool that starts with `"sm$len"` (This string holds the length the watermark added). Once we identify the string, we extract the values of the bogus variables added during embedding and combine them to reveal the watermark. We also take care of the extra '0' added to make the number of digits even, during the embedding process.

```
if (v.startsWith("sm$len")) {
   String w = v.substring("sm$len".length()+1);
   int no_of_vars=Integer.parseInt(w);
   if(no_of_vars%2==0)
       no_of_vars/=2;
   else
       no_of_vars=no_of_vars/2 +1;
   String wm="";
   for(int j=0;j<no_of_vars;j++){
       de.fub.bytecode.generic.BIPUSH bip= (de.fub.bytecode.generic.BIPUSH)instr[2*j];
       int sub_value= bip.getValue().intValue();
       String str = String.valueOf(sub_value);
       if( sub_value < 10)
          str= "0"+str;
       wm = str + wm;
       }
       if(wm.length()==Integer.parseInt(w))
          result.add(wm);
       else
          result.add(wm.substring(0,wm.length()-1));

```

Chapter 11

The Bogus Switch Watermarking Algorithm

Andrew Huntwork and Xiangyu Zhang

11.1 Introduction

This is an algorithm that embeds a watermark in the 'case' statements of a switch block. The code of this algorithm resides in `sandmark.watermark.bogus_switch`.

11.2 Embedding

To embed the watermark we insert a lookup switch at the beginning of the instruction stream for a randomly chosen method:

11.3 Recognition

During recognition we find the switch we inserted, and decode its 'case' statements

Chapter 12
The ConstantString Static Watermarking Algorithm

C. Collberg

12.1 Introduction

This is a trivial algorithm that embeds a string

`sm$watermark=`*WATERMARK*

in the constant pool of the application. *WATERMARK* is the string to be embedded. The code of this algorithm resides in `sandmark.watermark.constantstring`.

12.2 Embedding

To embed the watermark we pick one of the user's classes at random and adds the appropriate string:

```
String jarInput  = props.getProperty("WM_Embed_JarInput");
String jarOutput = props.getProperty("WM_Embed_JarOutput");
String watermark = props.getProperty("WM_Encode_Watermark");

sandmark.util.ClassFileCollection cfc =
   new sandmark.util.ClassFileCollection(jarInput);

java.util.Iterator classes = cfc.classes();
String className = (String) classes.next();
de.fub.bytecode.classfile.JavaClass origClass = cfc.getClass(className);
de.fub.bytecode.generic.ClassGen cg = new de.fub.bytecode.generic.ClassGen(origClass);

de.fub.bytecode.generic.ConstantPoolGen cp = cg.getConstantPool();
int stringIndex = cp.addString("sm$watermark" + "=" + watermark);

de.fub.bytecode.classfile.JavaClass newClass = cg.getJavaClass();
cfc.addClass(newClass);
cfc.saveJar(jarOutput);
```

12.3 Recognition

During recognition we go through every class in the watermarked jar-file looking for a string in the constant pool that starts with `"sm$watermark"`:

```
String jarInput  = props.getProperty("WM_Recognize_JarInput");
sandmark.util.ClassFileCollection cfc =
    new sandmark.util.ClassFileCollection(jarInput);
java.util.Iterator classes = cfc.classes();
while (classes.hasNext()) {
   String className = (String) classes.next();
   de.fub.bytecode.classfile.JavaClass clazz = cfc.getClass(className);
   de.fub.bytecode.classfile.ConstantPool cp = clazz.getConstantPool();
   for (int i=0; i<cp.getLength(); i++) {
      de.fub.bytecode.classfile.Constant c = cp.getConstant(i);
      if (c instanceof de.fub.bytecode.classfile.ConstantString) {
          de.fub.bytecode.classfile.ConstantString s =
             (de.fub.bytecode.classfile.ConstantString) c;
          String v = (String)s.getConstantValue(cp);
          if (v.startsWith("sm$watermark")) {
             String w = v.substring("sm$watermark".length()+1);
             // w is the watermark
          }
      }
   }
}
cfc.close();
```

Chapter 13

The Collberg-Thomborson Watermarking Algorithm

C. Collberg, J. Nagra, G. Townsend

13.1 Introduction

The Collberg-Thomborson watermarking algorithm (henceforth, CT) is a dynamic algorithm. The idea is that rather than embedding the watermark directly in the *code* of the application, code is embedded that *builds* the watermark at runtime. The algorithm assumes a secret key $\mathcal{K}$ which is necessary to extract the watermark. $\mathcal{K}$ is a sequence of inputs $I_0, I_1, \ldots$ to the application. As seen in Figure **??**, the watermark (a graph structure) is built by the application only when the user runs it with the special input $I_0, I_1, \ldots$. Figure **??** shows a simple example of a what a program may look like after having been watermarked.

In the SandMark implementation of CT, watermark embedding and extraction runs in several steps (See Figure **??**):

Annotation: Before the watermark can be embedded the user must add *annotation* (or *mark*) points into the application to be watermarked. These are calls of the form

```
      sandmark.trace.Annotate.mark();
      String S = ...;
      sandmark.trace.Annotate.mark(S);
      long L = ...;
      sandmark.trace.Annotate.mark(L);
```

The `mark()` calls perform no action. They simply indicate to the watermarker locations in the code where (part of) a watermark-building code can be inserted. The argument to the `mark()` call can be any string or integer expression that (directly or indirectly) depends on user input to the application.

Tracing: When the application has been annotated the user should do a *tracing* run of the program. The application is run with the chosen secret input sequence, $\mathcal{K}$. During the run one or more annotation points are hit. Some of these points will be the locations where watermark-building code will later be inserted.

Embedding: During the embedding stage the user enters a watermark, a string or an integer. A string is converted to an integer. From this number a graph is generated, such that the topology of the graph embeds the number. The graph is split into a number of subgraphs, depending on the number of locations where watermarking code should be inserted. Each subgraph is converted to Java bytecode that builds the graph. The relevant `mark()`-calls are replaced with this graph-building code.

Recognition: During recognition the application is again run with the secret input sequence as input. The same `mark()`-locations will be hit as during the tracing run. Now, however, these locations will contain code for building the watermark graph. When the last part of the input has been entered, the heap is examined for graphs that could potentially be watermark graphs. The graphs are decoded and the resulting watermark number or string is reported to the user.

13.2 Annotation

The CT watermark consists of dynamic data-structures. This means that the code inserted in the application will look like this:

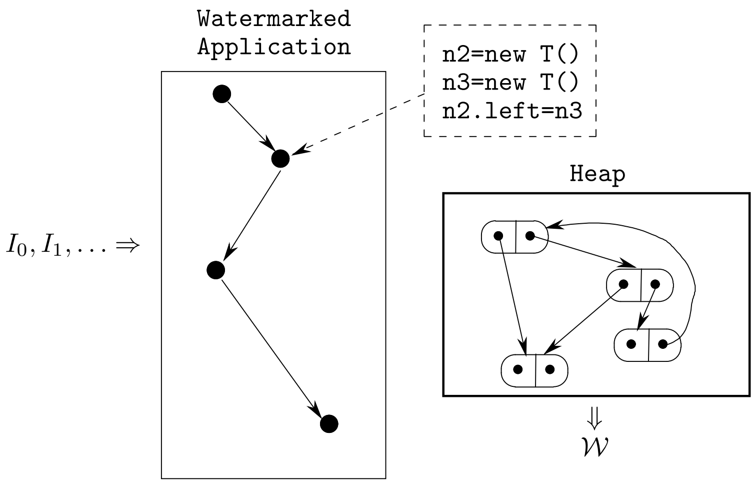


Figure 13.1: Overview of how the CT algorithm recognizes a watermark $\mathcal{W}$. At runtime the watermarked application will – given the special secret input key sequence $I_0, I_1, \ldots$ – traverse certain points in the program. At these points code has been inserted which builds a graph $G_{\mathcal{W}}$ on the heap. The topology of the graph embeds the watermark $\mathcal{W}$.

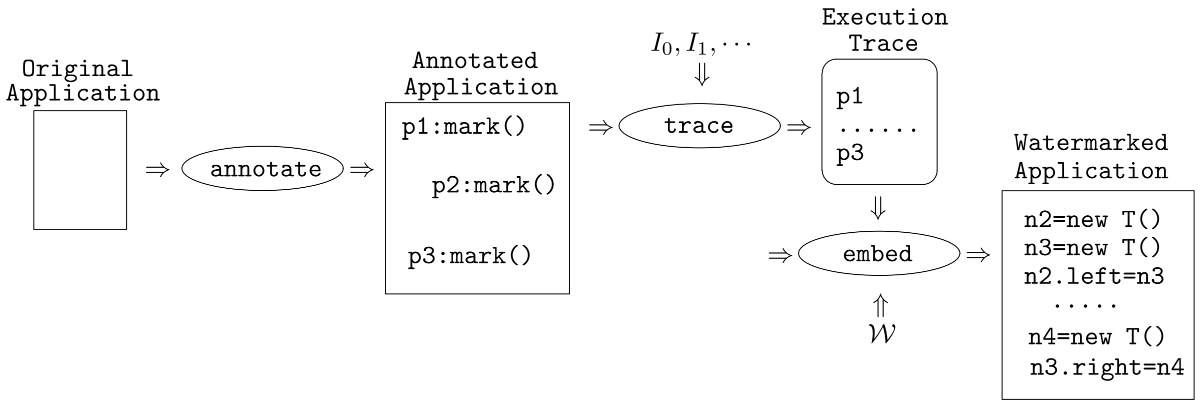


Figure 13.2: Overview of how the CT algorithm watermarks an application. First, the user adds *annotation points* (`mark()`-calls) to the application. These are locations where watermarking code may be inserted. Secondly, the application is run with a secret input sequence, $I_0, I_1, \ldots$ and the trace of `mark()`-calls hit during this run is recorded. Finally, code is embedded into the application (at certain `mark()`-call locations) that builds a graph G_W at runtime. The topology of $G_{\mathcal{W}}$ embeds the watermark $\mathcal{W}$.

```
public class Simple {
   static void P(String i) {
      System.out.println("Hello " + i);
   }
   public static void main(String args[]) {
      P(args[0]);
   }
}

public class Simple_W {
   static void P(String i) {
      if (i.equals("World")) Watermark.Create_G4();
      System.out.println("Hello " + i);
   }
   public static void main(String args[]) {
      Watermark.Create_G2();
      P(args[0]);
   }
}

public class Watermark extends java.lang.Object {
  public Watermark edge1;
  public Watermark edge2;
  public static java.util.Hashtable sm$hash = new java.util.Hashtable();
  public static Watermark[] sm$array = new Watermark[4];

  public static void Create_G2 () {
     Watermark n3 = new Watermark();
     Watermark n2 = new Watermark();
     Watermark.sm$array[1] = n2;
     n2.edge1 = n3;
     n2.edge2 = n3;
  }
  public static void Create_G4 () {
     Watermark n1 = new Watermark();
     Watermark n4 = new Watermark();
     Watermark.sm$hash.put(new java.lang.Integer(4), n4);
     n4.edge1 = n1;
     Watermark n2 = Watermark.sm$array[1];
     n1.edge1 = n2;
     Watermark n3 = (n2 != null)?n2.edge1:new Watermark();
     n3.edge1 = n1;
  }
}
```

Figure 13.3: Simple watermarking example. The class `Simple` is modified into `Simple_W` by adding calls into the generated watermark class `Watermark`. The static methods `Watermark.Create_G2()` and `Watermark.Create_G4()` are only called when the application is run with the secret input argument `"World"`. When this happens, the watermark graph is built on the heap.

```
Node n1 = new Node();
Node n2 = new Node();
n1.edge = n2;
...
```

Hence, we should prefer mark locations that

- allocate objects and manipulate pointers, and
- directly depend on user input.

We should avoid mark locations that

- are hot-spots, and
- are executed non-deterministically.

In other words, `mark()`-calls should be added to locations where the resulting watermark code will be fit in (is *stealthy*), won't affect performance, and will be executed consistently from run to run, depending only on user actions.

For example, the following code is undesirable since `Math.random()` may generate different values during different runs of the program:

```
if (Math.random() < 0.5) {
   ...
   sandmark.trace.Annotate.mark();
}
```

Similarly, if thread scheduling, network activity, processor load, etc. can affect the order in which some locations are executed, these locations are not valid annotations points and should be avoided.

13.3 Tracing

SandMark makes heavy use of Java's `JDI` (*Java Debugging Interface*) framework. During tracing and recognition SandMark starts up the user's application as a subprocess running under debugging. This allows SandMark to set breakpoints, examine variables, and step through the application – all the operations that can be done under an interactive debugger. During tracing we are interested in obtaining a trace of the `mark()`-calls that are hit while the user enters their secret input. We also want to know the argument to the `mark()`-call and the stack trace at the point of the call.

Unfortunately, `JDI` is not yet a perfect product and we have to jump through a couple of hoops to make it do what we want. First of all, examining the value of the argument to the `mark()`-call may or may not work. Examining static global variables seems to work, however, so we always start by storing the argument in a global, and then call the placeholder method `MARK()`. See Figure **??**. During tracing we only have to put a breakpoint on the `MARK()` method.

The second problem is that we need a stack trace at the point of each `mark()`-call. This trace is used during embedding to compute an accurate call-graph of the program at each `mark()` location. The call graph allows us to compute ways to pass information between `mark()`-calls in method parameters. While `JDI` allows us to examine the stack frames at any point in the program, it is not possible to tell if two stack-frames are *the same*. That is, JDI stack-frames do not have unique identities. To solve this problem we add the following statement to the beginning of every method in the program:

```
 long sm$stackID = sandmark.trace.Annotator.stackFrameNumber++;
```

This is done prior to tracing in `sandmark.trace.Preprocessor`.

When, during tracing, a `mark()`-call is hit we walk the stack, collecting the `sm$stackID`s in each frame.

At the end of tracing run we have gathered a list of `sandmark.trace.TracePoint`-objects. Each object represents a `mark()`-call that was hit during the trace and contains three pieces of information:

```
package sandmark.trace;
public class Annotator {
   static String VALUE = "";
   public static long stackFrameNumber=0;
   public static void MARK(){}
   public static void mark() {
      long sm$stackID = sandmark.trace.Annotator.stackFrameNumber++;
      VALUE = "----";
      MARK();
   }
   public static void mark(String s) {
      long sm$stackID = sandmark.trace.Annotator.stackFrameNumber++;
      VALUE = "\"" + s + "\"";
      MARK();
   }
   public static void mark(long v) {
      long sm$stackID = sandmark.trace.Annotator.stackFrameNumber++;
      VALUE = Long.toString(v);
      MARK();
   }
}
```

Figure 13.4: The class `sandmark.trace.Annotate`.

1. the location in the bytecode where the `mark()` was located (a `sandmark.util.ByteCodeLocation`);
2. the value that the user supplied as an argument to the `mark()`-call (a `String`);
3. a list of the stack-frames active when the `mark()`-call was hit (`sandmark.util.StackFrame[]`).

An Example

Consider the following example application:

```
public class SimpleA {
   static void P(int i) {
      sandmark.trace.Annotator.mark(6*i+9);
   }
   public static void main(String args[]) {
      P(3);
   }
}
```

After tracing we will have found only one trace point. It is described by a structure like this:

$$\langle \texttt{value} = 27, \texttt{location} = \langle \texttt{P}, \texttt{pc} = 8\rangle, \texttt{stack} = [\langle \texttt{P}, \texttt{pc} = 8, \texttt{frame} = 1\rangle, \langle \texttt{main}, \texttt{pc} = 8, \texttt{frame} = 0\rangle]\rangle$$

We have stored the argument to the `mark()` call (`value=27`), the bytecode location where that call was made (`pc=8`), and complete stack trace (with unique identifiers for each frame) at this location.

Choosing `mark()`-Locations

The tracing phase will have generated one or more `mark()`-locations. However, cannot be used to build the watermark graph and have to be removed. Also, we need k locations to insert code to build a k-component graph, and any extra locations should be deleted.

`sandmark.watermark.CT.embed.PrepareTrace` examines the trace to find a set of `mark()`-locations that can be used to build the watermark graph. An annotation point $\langle \textit{value}, \textit{location} \rangle$ is valid if

1. there is exactly one trace point at *location*, or
2. there are multiple trace points at *location*, but they all have unique *value*s.

For example, consider the following `mark()`-points:

$$\begin{array}{l}\langle -, L_0 \rangle \\ \langle 1, L_1 \rangle \\ \langle 1, L_1 \rangle \\ \langle 10, L_2 \rangle \\ \langle 11, L_2 \rangle \\ \langle 12, L_2 \rangle\end{array}$$

`value` $=-$ is used for `mark()`-calls that take no argument. $\langle -, L_0 \rangle$ is valid, because it is the only `mark()`-point at location L_0. $\langle 1, L_1 \rangle$ is not valid because there are two identical annotation values at this location. If we were to insert watermark-building code at this location we would not be able to tell the difference between the first and the second time we arrive. $\langle 10, L_2 \rangle, \langle 11, L_2 \rangle, \langle 12, L_2 \rangle$ are valid because the *value*s are unique. If there is one unique value at a location, this `mark()`-call is said to be `LOCATION`-based, otherwise it is `VALUE`-based.

13.4 Embedding

Once the application has been traced we can finally start embedding the watermark. The input to this phase is tracing information (as described in the previous section), a watermark $\mathcal{W}$ to be embedded, and a jar-file containing the classfiles in which to embed the mark. The embedding is divided into five phases:

1. First we generate a graph G whose topology embeds $\mathcal{W}$.
2. Next, we split G into k subgraphs $\langle G_1, \ldots, G_k \rangle$.
3. From each subgraph G_i we generate an *intermediate code* C_i that builds this graph and connects it to the subgraphs $\langle G_1, \ldots, G_{i-1} \rangle$.
4. We translate each intermediate code C_i into a Java method M_i that, when executed, will build G_i.
5. Finally, based on the tracing information, we replace some of the `mark()`-calls with calls to one of the M_i-methods. The remaining `mark()`-calls are removed.

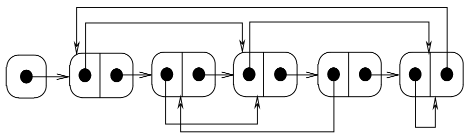
The result is a new jar-file that when executed with the special input sequence will execute the methods $\langle M_1, \ldots, M_k \rangle$ (in this order), and, consequently, build the watermark graph G on the heap.

Building the Graph

Eventually we hope to have a whole library of algorithms for building watermark graphs. Currently, only two have been implemented. The algorithms are located in `sandmark.util.graph.codec`. See Section **??** for a description of how to add a new *Graph Codec* to SandMark.

Radix Encoding is the simplest algorithm. The codec is in `sandmark.util.graph.codec.RadixGraph`. Figure **??** illustrates the idea of a Radix-k encoding using a circular linked list. An extra pointer field encodes a base-k digit in the length of the path from the node back to itself. A null-pointer encodes a 0, a self-pointer a 1, a pointer to the next node encodes a 2, etc.

The *Permutation Encoding* codec is in `sandmark.util.graph.codec.PermuationGraph`. The idea is to represent the watermark $\mathcal{W}$ by a permutation of the numbers $\langle 0, \ldots, n-1 \rangle$. For example, the number



$$61 \times 73 = 3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0$$

Figure 13.5: Radix-6 encoding. The right pointer field holds the `next` field, the left pointer encodes a base-k digit.

could be represented by the permutation

$$\langle 9, 6, 5, 2, 3, 4, 0, 1, 7, 8 \rangle$$

of the numbers

$$\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle.$$

A permutation of length n is encoded into a graph structure similar to the one in Figure **??**. The nodes of the graph form a circular linked list and a pointer from node i to node j represents the fact that i has been permuted with j.

It should be noted that the graphs we use in the `CT` algorithm are, in fact, hyper-graphs. They are implemented by the package `sandmark.util.graph`.

Splitting the Graph

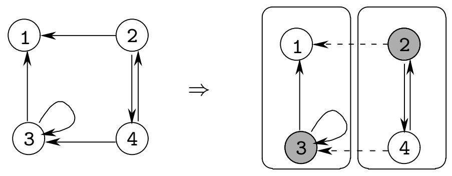
When the watermark graph has been built it needs to be split into pieces. This is done in the package `sandmark.watermark.CT.encode.Split`. There should be one graph component per `mark()`-location that we intend to use. There are three things to consider when we split the graph:

1. The subgraphs should be of roughly equal size. (It is actually not quite clear that this is a reasonable requirement. For stealth reasons it might be better if the components are of random size. The current implementation, however, splits in equal-size pieces.)

2. The splitting of G should be done in such a way that each subgraph has a root, a special node from which all other nodes in the graph can be reached. This allows us to store only pointers to root nodes to prevent the garbage collector from collecting the subgraphs. (More about this later.)

3. We should attempt to split G in such a way that the number of edges between subgraphs is minimized. The reason for this restriction is that the more edges there are between subgraphs, the more Java code we will have to generate in order to connect the subgraphs into the complete graph G.

We use a graph-splitting algorithm by Kundu and Misra. It would, for example, split the graph on the left into the two components on the right:



Root nodes have been shaded and inter-component edges have been dashed.

INSTRUCTION	DESCRIPTION
`AddEdge(`G_i`,`G_j`,`$n \xrightarrow{\texttt{edge}} m$`)`	Add an edge from node n in subgraph G_i to node m in G_j. Since the graphs are multi-graphs the out-edges are named.
`CreateNode(`G_i`,`n`)`	Create node n in subgraph G_i.
`CreateStorage(`G`,`S`)`	Create the global storage structure S.
`Debug(`*msg*`)`	Insert debugging code.
`FollowLink(`G_i`,`$n \xrightarrow{\texttt{edge}} m$`)`	Return m by following the edge `edge` from n.
`LoadNode(`G_i`,`n`,`L`)`	Load node n from global storage location L.
`PrintGraph()`	Insert code for printing the graph. Used for debugging.
`ProtectRegion(`*ops*`)`	The instructions *ops* may generate runtime errors, such as `null` dereference. Protect against such errors by, for example, putting *ops* inside a `try`-block.
`SaveNode(`G_i`,`n`,`L`)`	Save node n in global storage location L.

Table 13.1: Intermediate code instructions.

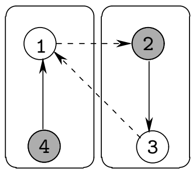


Figure 13.6: Two graph components G_2 and G_4. Components are named after their (shaded) root nodes.

Generating Intermediate Code

We could, of course, generate Java code directly from the graph components. However, it turns out to be advantageous to insert one intermediate step. From each graph component we generate a list of *intermediate code instructions*, much in the same way a compiler might generate an intermediate representation of a program, in anticipation of code generation and optimization. In a compiler, the intermediate code separates the front-end from the back-end, improving retargetability, and also providing a target-independent representation for optimizing transformations. Similarly, our intermediate representation provides

1. retargetability, in case one day we may want to generate C++ or C# code; and
2. transformability, i.e. the ability to optimize or otherwise transform the intermediate code prior to generating Java code.

In fact, we start by generating very simple intermediate code, and then run several transformations (in `sandmark.watermark.CT.encode.ir2ir`) over the code to optimize it, etc.

The intermediate code instructions are defined in the package `sandmark.watermark.CT.encode.ir`. The main operations are given in Table ??.

Consider the two graph components G_2 and G_4 in Figure ??. The following intermediate code is generated from G_2:

```
create($G_2$)
   $n_3$ = CreateNode($G_2$)
   $n_2$ = CreateNode($G_2$)
   SaveNode($n_2$, $G_2$, '$n_2$:Array/global')
   AddEdge($n_2 \xrightarrow{\texttt{edge1}} n_3$, $G_2$, $G_2$)
   AddEdge($n_2 \xrightarrow{\texttt{edge2}} n_3$, $G_2$, $G_2$)
```

Nodes are named n_1, n_2, etc. The `SaveNode` instruction is used to store the root node of a graph component in a global structure such as a hash table, vector, etc. We do this for two reasons:

1. Suppose we have two subgraphs G_1 and G_2, where G_1 is created first. After G_2 has been created, the two graphs need to be connected. At this point we need (at least) pointers to both their root nodes, so that we can access and link any two nodes in the graphs.

2. Every node in every subgraph must, at all times, be live or it may be deleted by the garbage collector.

As we will see later, we can often do away with these global pointers by passing root nodes as method arguments. This is much stealthier since most programs have few global variables but many method parameters.

Here is the intermediate code generated from the subgraph component G_4 in Figure **??**:

```
create($G_4$)
   $n_1$ = CreateNode($G_4$)
   $n_4$ = CreateNode($G_4$)
   SaveNode($G_4$, $n_4$, '$n_4$:Hash/global')
   AddEdge($G_4$, $G_4$, $n_4 \xrightarrow{\text{edge1}} n_1$)
   $n_2$ = LoadNode($G_4$, '$n_2$:Array/global')
   AddEdge($G_4$, $G_2$, $n_1 \xrightarrow{\text{edge1}} n_2$)
   $n_3$ := FollowLink($G_2$, $n_2 \xrightarrow{\text{edge1}} n_3$)
   AddEdge($G_2$, $G_4$, $n_3 \xrightarrow{\text{edge1}} n_1$)
```

Note how node n_2 from graph G_2 as been loaded from global storage in order to connect n_1 to n_2. Note also how the `FollowLink` instruction is used to traverse G_2 from n_2 to get to node n_3, which can then be connected to n_1.

To generate intermediate code from a subgraph G_i we perform a depth-first search from the root of the graph. This is done in `sandmark.watermark.CT.encode.Graph2IR`. `CreateNode(`n`)`-instructions are generated from each node, in a reverse topological order. That is, leaves are generated first and the root node last. We can issue an `AddEdge(`G_i`, `G_i`, `$n \xrightarrow{\text{edge}} m$`)`-instruction as soon as `CreateNode(`m`)` and `CreateNode(`n`)` have both been generated.

The code for G_i must also contain instructions connecting G_i to all the previous subgraphs $G_0, \ldots, G_{i-1}$. If there is an inter-subgraph edge $m \xrightarrow{\text{edge}} n$ from G_k to G_i (i.e. m is a node in G_k and n is in G_i) then we must generate

1. one or more `FollowLink()`-instructions to reach node m by traversing G_k starting at its root node,

2. and a final `AddEdge()` instruction to link m to n.

This is done by finding the shortest path from k (the root of subgraph G_k) and m and issuing a `FollowLink()`-instruction for each.

`sandmark.watermark.CT.encode.Graph2IR` generates the basic list of intermediate instructions. These "programs" are then optimized and transformed in various ways by the transformers in `sandmark.watermark.CT.encode.ir2ir.*`. Some of the more important ones are

sandmark.watermark.CT.encode.ir2ir.AddFields:

sandmark.watermark.CT.encode.ir2ir.AddFormals: Add parameters to the `Create_`G_i methods to allow graph roots to be passed in formals rather than globals. (This will be described in more detail in Section **??**).

sandmark.watermark.CT.encode.ir2ir.CleanUp: `LoadNode()`-instructions are added (in `sandmark.watermark.CT.encode.ir2ir.SaveNodes`. We do this in a greedy way, and the `CleanUp` transformer removes redundant loads.

sandmark.watermark.CT.encode.ir2ir.Debug: Add `Debug()`-instructions. These will print out trace messages as the watermark graphs are built at runtime.

sandmark.watermark.CT.encode.ir2ir.Destructors: Create bogus graph builders/destroyers that can be inserted in various places in the code. The destructors are created by modifying copies of the creator codes.

sandmark.watermark.CT.encode.ir2ir.InlineFixups: `sandmark.watermark.CT.encode.Graph2IR` generates special methods (called `Fixup_`G_i`_`G_j) which add the inter-graph links that connect too subgraphs G_i and G_j. Normally these are inlined into the code for G_j by this transformer.

sandmark.watermark.CT.encode.ir2ir.Protect: Add protection code when this is necessary to prevent `FollowLink()`-instructions from throwing unwanted exceptions at runtime.

sandmark.watermark.CT.encode.ir2ir.SaveNodes: Add code to load and store graph roots into global, static storage.

Generating Java Code

Generating Java code from the intermediate representation is relatively straight-forward. We use the `BCEL` library to generate a bytecode class `Watermark`. We can generate Java source also, but this is mostly used for debugging.

The intermediate code from the previous section is translated into the Java class in Figure **??**. Method `Create_G2` builds subgraph G_2 and `Create_G4` subgraph G_4. Note, in particular, the statements

```
Watermark n2 = Watermark.sm$array[1];
Watermark n3 = (n2 != null)?n2.edge1:new Watermark();
n3.edge1 = n1;
```

which link nodes n_3 and n_1. To get access to G_2's node n_3 we follow the edge from G_2's root node n_2 to n_3. This will work provided G_2 has been created at this point. However, if we're not doing a recognition run (i.e. the input sequence in *not* $I_0, I_1, \ldots$) then G_2 may not have been created in which case `n2` may be `null`. We can protect against this in a variety of ways:

1. if `n2` is `null` we create a new node and assign it to `n2` (as above);

2. we can enclose the entire code segment in a `try-catch`-block:

   ```
   Watermark n2 = Watermark.sm$array[1];
   try {
      Watermark n3 = n2.edge1;
      n3.edge1 = n1;
   } catch (Exception e){}
   ```

   ; or

3. we may simply not do the assignment if `n2` is `null`:

   ```
   Watermark n2 = Watermark.sm$array[1];
   if (n2 != null) {
      Watermark n3 = n2.edge1;
      n3.edge1 = n1;
   }
   ```

It is useful to have a whole library of such protection mechanisms to prevent attacks by pattern matching.

```
public class Watermark extends java.lang.Object {
  public Watermark edge1;
  public Watermark edge2;
  public static java.util.Hashtable sm$hash;
  public static Watermark[] sm$array;

  public static void Create_G2 () {
     Watermark n3 = new Watermark();       // n3 = CreateNode(G2)
     Watermark n2 = new Watermark();       // n2 = CreateNode(G2)
     Watermark.sm$array[1] = n2;           // SaveNode(n2, G2, 'n2:Array/global')
     n2.edge1 = n3;                        // AddEdge(G2, G2, n2-edge1->n3)
     n2.edge2 = n3;                        // AddEdge(G2, G2, n2-edge2->n3)
  }

  public static void Create_G4 () {
     Watermark n1 = new Watermark();       // n1 = CreateNode(G4)
     Watermark n4 = new Watermark();       // n4 = CreateNode(G4)
     Watermark.sm$hash.put(
         new java.lang.Integer(4), n4);    // SaveNode(n4, G4, 'n4:Hash/global')
     n4.edge1 = n1;                        // AddEdge(G4, G4, n4-edge1->n1)
     Watermark n2 = Watermark.sm$array[1]; // n2 = LoadNode(G4, 'n2:Array/global')
     n1.edge1 = n2;                        // AddEdge(G4, G2, n1-edge1->n2)
     Watermark n3 =                        // n3 := FollowLink(G2, n2-edge1->n3)
        (n2 != null)?n2.edge1:new Watermark();
     n3.edge1 = n1;                        // AddEdge(G2, G4, n3-edge1->n1)
  }
}
```

Figure 13.7: Java code generated from the graph components in Figure ??.

INSTRUCTION	JAVA
AddEdge(G_i,G_j,$n \xrightarrow{\text{edge}} m$)	n.edge = m
CreateNode(G_i,n)	Watermark n = new Watermark()
CreateStorage(G,S)	One of 1. static java.util.Hashtable sm$hash = new java.util.Hashtable(); 2. static Watermark sm$array = new Watermark[$m$]; 3. static java.util.Vector sm$vec = new java.util.Vector(m); sm$vec.setSize($m$); 4. static Watermark sm$n1,sm$n2,...; where m is the number of nodes in the graph and sm$n1,sm$n2,... are the root nodes of the subgraphs.
FollowLink(G_i,$n \xrightarrow{\text{edge}} m$)	Watermark m = n.edge
LoadNode(G_i,n,S)	One of 1. Watermark n = (Watermark) sm$hash.get(new java.lang.Integer(k)); 2. Watermark n = Watermark.sm$arr[$k-1$]; 3. Watermark n = (Watermark) sm$vec.get($k-1$); 4. Watermark n = Watermark.sm$n$k$ depending on how n is stored. k is n's node number.
ProtectRegion(*ops*)	try { *ops* } catch (Exception e) {}
SaveNode(G_i,n,L)	One of 1. sm$hash.put(new java.lang.Integer(k), n); 2. Watermark.sm$arr[$k-1$] = n; 3. (Watermark) sm$vec.set($k-1$, n); 4. Watermark.sm$n$k$ = n depending on how n is stored. k is n's node number.

Table 13.2: Translation from intermediate code instructions to Java.

Inserting the Java Code

`sandmark.watermark.CT.embed.Embedder` is the main class for modifying the Java program to be watermarked.

The chosen `mark()`-locations are replaced by calls to `Watermark.Create`G_i. (We're relying on the `BLOAT` optimizer to eventually inline these calls and remove the `Watermark` class.) Remaining `mark()`-calls are deleted.

As usual, this process is done in several passes over the code:

1. `sandmark.watermark.CT.embed.ReplaceMarkCalls` replaces the `mark()`-calls with calls to `Watermark.Create`G_i. There are two cases, depending on whether the `mark()`-call is `LOCATION`-based or `VALUE`-based. A `LOCATION`-based `mark()`-call is simply replaced by a call

   ```
   Watermark.Create_Gi();
   ```

   A `VALUE`-based `mark(expr)`-call is replaced by the call

   ```
   if (expr==value)
      Watermark.Create_Gi();
   ```

2. `sandmark.watermark.CT.embed.AddParameters` adds formal parameters in order to be able to pass graph root nodes in formals rather than in globals. See Section **??** for more details.

3. `sandmark.watermark.CT.embed.InsertStorageCreators` inserts code to create hashtables, arrays, vectors, etc. that are used to store subgraph root nodes.

4. `sandmark.watermark.CT.embed.DeleteMarkCalls` removes any traces of the `mark()`-calls from the application.

13.5 Recognition

`sandmark.watermark.CT.recognize.Recognizer` starts up the watermarked application as a subprocess under debugging, again using Java's JDI debugging framework. The user enters their secret input sequence $I_0, I_i, \ldots$ exactly as they did during the tracing phase. This causes the methods `Watermark.Create`G_i to be executed and the watermark graph to be constructed on the heap. When the last input has been entered it is the recognizer's task to locate the graph on the heap, decode it, and present the watermark value to the user.

There may, of course, be an enormous number of objects on the heap and it would be impossible to examine them all. To cut down the search space we rely on the observation that the root node of the watermark graph will be one of the very last objects to be added to the heap. Hence, a good strategy would likely be to examine the heap objects in reverse allocation order. Unfortunately, `JDI` does not yet provide support for examining the heap in this way.

An elegant and efficient approach would be to modify the constructor for `java.lang.Object` to include a counter:

```
package java.lang;
public class Object {
    public static long objCount = 0;
    public long allocTime;
    public Object() {allocTime = objCount++;}
}
```

Since every constructor must call `java.lang.Object.<init>` this means that we've assigned an allocation order to the objects on the heap at the cost of only an extra add and assign per allocation.

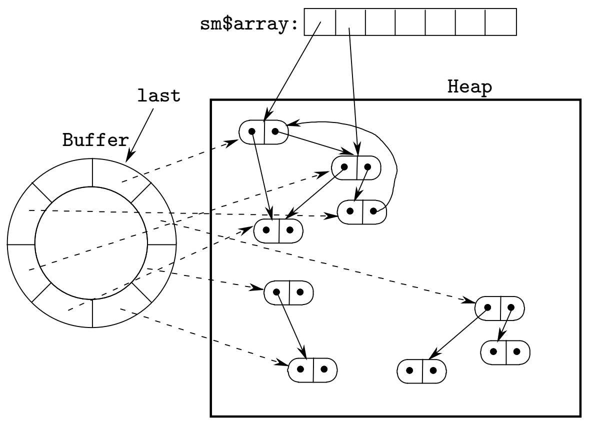


Figure 13.8: A view of memory during recognition. A circular linked buffer holds the last allocated objects. The recognizer examines the objects in reverse allocation order and extracts the subgraph reachable from each object. This is decoded into the watermark.

We've shied away from this approach, however, since it would require modifying the Java runtime system. Also, some Java compilers optimize away calls to `java.lang.Object.<init>` under the assumption that this constructor does nothing.

Instead, we rely on a more heavyweight but portable solution. Using `JDI` we add a breakpoint to every constructor in the program. Whenever an allocation occurs we add a pointer to the new object to a circular linked buffer, `sandmark.util.CircularBuffer`. This way, we always have the last 1000 (say) allocated objects available. The downside is a fairly substantial slowdown due to the overhead incurred by handling the breakpoints.

The recognition algorithm is as follows (see `sandmark.watermark.CT.recognize.Recognizer` and Figure **??**):

```
static int kidMaps[][] = {{1,2},{2,1},{1,3},{3,1},{2,3},{3,1}};
for every object O on sandmark.util.CircularBuffer, starting with the last allocated do {
    G := the graph consisting of the nodes reachable from O;
    for every graph decoder D do {
       for every kidmap K do {
          W := decode G using D, assuming K;
          print W;
       }
    }
}
```

The kidmaps are used to select 2 pointers out of each object as our outgoing pointers. The decoding is done by the codecs in `sandmark.util.graph.codec`. The class `sandmark.watermark.CT.recognize.Heap2Graph` is used to convert a heap structure to a `sandmark.util.graph.Graph` object. Note that the decoding can fail for a number of reasons. Some of the objects on the circular buffer may no longer be alive, the graph structure extracted from the heap may not be of the form expected by any of the codecs, etc. For this reason, we catch all possible exceptions (such as `null`-dereferences) and ignore any structures for which errors occur.

13.6 Passing Roots in Formal Parameters

In the descriptions above we have assumed that the roots of subgraphs are stored in static, global variables. In Figure **??**, for example, the root of each subgraph is stored in a global array `sm$array`. This is obviously un-stealthy since programs typically only contain a few globals. Instead, we would like to pass roots in the formal parameters of methods. This means we are going to have to find paths through the call-graphs from one `mark()`-call to the next.

More details about the call graph etc. here.

Chapter 14

The HatTrick Watermarking Algorithm

Danny Mandel and Anna Segurson

14.1 Introduction

This algorithm embeds a static watermark in a java *.jar file through the sandmark interface. It first converts the watermark (which is a String) to a number. This number is then encoded into the java byte code by adding bogus local variables to one method of one class in the *.jar file. The class in which the method is in is marked by a extra field variable that contains the name of the method.

Each number of the watermark is mapped to a specific type:

Number	Type
0	`java.util.GregorianCalendar`
1	`java.lang.Thread`
2	`java.util.Vector`
3	`java.util.Stack`
4	`java.util.Date`
5	`java.io.InputStream`
6	`java.io.ObjectInputStream`
7	`java.lang.Math`
8	`java.io.OutputStream`
9	`java.lang.String`

A local variable of the proper type is created for each digit of the watermark and the digit's location in the watermark is stored in the name of the bogus local variable. The rightmost digit is position 0.

The code of this algorithm resides in `sandmark.watermark.hattrick`.

14.2 Embedding

The watermark is embedded in the first non-abstract method found in the first class file of the jar file. The name of the method is recorded in the name of a field with the value of a hockey player's name. These are the local variables added to a class when the watermark "Howdy!" was embedded:

```
    Math yzerman$0 = new Math();
    String yzerman$1 = new String();
    ObjectInputStream yzerman$2 = new ObjectInputStream();
    Stack yzerman$3 = new Stack();
    InputStream yzerman$4 = new InputStream();
    Stack yzerman$5 = new Stack();
    OutputStream yzerman$6 = new OutputStream();
    InputStream yzerman$7 = new InputStream();
    InputStream yzerman$8 = new InputStream();
    OutputStream yzerman$9 = new OutputStream();
    Thread yzerman$10 = new Thread();
    Thread yzerman$11 = new Thread();
    Thread yzerman$12 = new Thread();
    ObjectInputStream yzerman$13 = new ObjectInputStream();
    Stack yzerman$14 = new Stack();
```

This is the field that is added: `public static final String hat<init>Trick = ‘‘Vrbata’’;` where `<init>` is the name of the method the watermark is in.

14.3 Recognition

The watermark is recognized when the proper field is present which stores the name of the method. Every local variable that contains the `SECRET_NAME` in that method is part of the watermark.

In the example listed above `yzerman` is the secret name. To recover the number stored by the `SECRET_NAME` variables, the algorithm initializes a number, `wm`, to zero then loops through all of the locals in the method and adds the mapped value of the variable times 10 to the power of the number after the $ to `wm`. After the number is recovered it is converted back to the watermark string.

Chapter 15

Potkonjak's Watermarking Algorithm

G. Myles

15.1 Introduction

Potkonjak's watermarking algorithm is based upon the ideas of graph coloring and register allocation. Java does not use registers to store values put instead uses local variables. So this algorithm will assign local variables based upon graph coloring instead of registers. The idea is to embed a watermark in a Java program that uniquely maps to one of the possible allocations of local variables. The reason that a watermark can be uniquely mapped to a local variable allocation is because a technique called bridge construction is used to generate multiple allocations from one specific allocation. By using this technique no additional code needs to be added so examining the code will not lead to the discovery of the watermark.

Currently this technique only works for a single method in a class, but in the future it will be extended so that an entire set of classes can be watermarked.

15.2 Embedding

The embedding of the watermark involves several different phases.

Convert watermark to binary

Converting the watermark, which is a string, to a binary is a fairly simple and straight forward task. This task is necessary because the binary is used to help choose the appropriate local variable allocation.

Generate the original coloring

The first real step of the algorithm is to determine the coloring of the method before any modifications have been made. This first step is necessary because a base is needed in order to base the binary relationship. It is not possible to used the local variable assignment that has already been allocated because BLOAT performs many optimizations in the process of completing the register allocation, which must be used in later phases of the algorithm. The majority of this step is handled by classes within BLOAT. To complete this task a control flow graph is created, SSA analysis is performed to generate the def-use information, liveness analysis is performed to generate the interference graph, and then register allocation takes place. Once these steps have been completed the instructions for the method are scaned to determine what loacal variable allocation was assigned.

Bridge construct and generate new coloring

Bridge construction is a technique used on graphs so that many colorings can be generated from the same graph. This is desirable since graph coloring is an NP problem and thus generating many colorings from the unmodified graph would be very intensive. With this algorithm it is necessary to generate as many colorings as possible so as to increase the chances of being able to embed the watermark in the method. Since the algorithm does not involve adding any code to the methods finding a suitable local variable allocation is crucial.

The bridge construction technique works by taking two unconnected nodes and connecting them and connecting the neighbors of one to the other. Each pair of nodes that can be connected generates 2^2 different colorings. If a set of three unconnected nodes can be found connecting them would generate 2^3 different colorings. It is also possible to combine these two sets to generate even more colorings. The bridge construction is performed on the interference graph which was generated from the liveness analysis. Once the bridge construction has been completed the new interference graph is passed to the register allocator. Again after the register allocation has been completed the instructions are scaned to determine the allocation.

Determine if the method can be watermarked

Determining if the method can actually be watermarked is currently folded in with generating the colorings. Once the algorithm can be performed on multiple methods it will be necessary to check if the binary will fit perfectly in the program.

Generate the pool of graphs

In order to generate all of the different colorings it is necessary to know which edges were added to the interference graph. Currently the algorithm does not generate all of the colorings at once but instead generates certain sections of the coloring and determines if any of those possibilites will work. It they don't work then the method cannot be watermarked. The sections that are generated are the pairs of nodes that were connected. If two nodes were connected then those four solutions are generated and compared with the appropriate location in the binary to see if one will work. The nodes that are not connected will maintain their same coloring and therefore result in a 0 for the binary.

Pick the correct coloring

The coloring that is chosen is based upon the binary that needs to be embedded and the relationship between the original coloring and the generated colorings. By comparing the original to the generated coloring a binary relationship can be determined which corresponds to the binary watermark.

Embed the chosen coloring

Once a coloring has been selected it is passed to the register allocator along with the corresponding node label. The node labels are necessary because BLOAT does not appear to access the nodes of the graph in any particular order. This technique of passing the colors to the modified register allocator proved to be the easiest and most reliable because it takes care of both the defs and uses (stores and loads).

15.3 Recognition

Recognizing the watermark also involves several phases.

Determine the current coloring

Because the embedded watermark is based upon the comparison of two different local variable allocations, the first thing that must be done is to determine what allocation is currently assigned to the method. This is done simply by scanning the instructions in the method and looking for places where a store is perfomed. One a store has been found we can obtain the index of the local variable. The indexes are stored in a Vector, `currentColoring`, for later reference.

Generate the original coloring for the method

In order to generate the binary that dictated which coloring was chosen we have to determine the original local variable allocation for the method. This is done following the same steps that were used in order to generate the original. First we have to generate a control flow graph for the method. We then have to perform the SSA analysis in order to generate the def-use information that is used in the liveness analysis. After the SSA analysis the liveness analysis is performed so that an interference graph is generated. The interference graph is then used by the register allocator to assign local variable. Once all of this has been completed it is necessary to scan the instructions again to determing the local variable allocation. Again we store this imformation in a Vector, `originalColoring`.

Compare the two colorings

Just as was done by in the embedding it is possible to compare the current coloring and the original coloring to get a binary. This binary will then become the watermark.

Convert binary to string

Again this is a simple task, but a necessary one. By converting the generated binary into a string we get the watermark back.

15.4 Future Work

Currently this watermarking technique has only been implemented so that a single method can be watermarked.

Chapter 16

The RenameMethodsFields Watermarking Algorithm

Martin Stepp and Kelly Heffner

16.1 Introduction

This is an algorithm that embeds a watermark into a Java program by modifying the names of different methods within the program. The code of this algorithm resides in `sandmark.watermark.RenameMethodsFields`.

16.2 Embedding

To embed the watermark we first generate a list of all of the method equivalence classes. Two methods are in the same equivalence class if one method overrides the other. This list does not contain any special methods (overriden from Java, constructors, static initializers, etc.) From there, each letter of the watermark is appended on to a method name using a random number generator (with a particular seed which remains constant between embedding and recognition) to pick which method to use next. In the event that one method is used twice in order to embed the watermark, the letters of the watermark will be added in a LIFO (Last In First Out) order. A delimeter $ is used to mark the end of the watermark.

16.3 Recognition

During recognition we generate the same list of equivalence classes as in the embedding process, and use the same seed for the random number generator in order to rescan the methods, and reconstruct the watermark.

Chapter 17

The Robust Object Watermarking Algorithm

Balamurugan Chirtsabesan, Tapas Sahoo

17.1 Introduction

The Robust Object Watermarking presents a new approach to watermarking. Instead of applying the watermarking scheme to the raw code directly, a new vector representation of the code is created. The basic idea revolving around this scheme is that instead of considering the overall structure of the code and its control flow, the code is viewed as a statistical object. The frequencies of groups of instructions in the entire code are taken into consideration in creating a new vector representation of the data. Spreading the watermark throughout the target code ensures a large measure of security against intentional and unintentional attacks.

Identifying the Vector instruction Groups: A set of instruction groups are identified that will form the watermark vector to be embedded. Ideally these instructions groups are instructions which have frequent occurence in the code and are chosen from some profiling information of the code. These instruction groups are small, each containing number of instructions in the range 1 to 4. This maximizes the probability of occurence of the instruction groups in the code while preserving the stealth.

Building the CodeBook: The CodeBook contains a mapping between semantically equivalent sets of instructions and is constructed in accordance with the vector instruction groups mentioned above. The basic motivation behind the construction of such a CodeBook is to facilitate the insertion of the watermark vector instructions through code subsitution. The CodeBook has been designed such that it can be upgraded as and when the vector instruction group is upgraded and new possible code equivalence is identified.

Vector Extraction: In this step, the vector length is chosen, the length of which is constrained by the number of the vector instruction groups. The frequency of occurence of the vector instruction groups in the code are identified and the intial vector is formed out of it.

Insertion: In this step, the watermark vector selected is to be embedded in the code. The frequencies of instructions in the vector instruction group are varied in accordance with the watermark vector by different implementation techniques such as code subsitution, new code insertion, cloning of method, etc. Care must be taken to ensure that all the operations done do not violate the semantics of the code.

Recognition: During recognition, the Vector Extraction procedure is invoked to extract the frequencies of the instructions in the vector group. The watermark is identified across a certain level of tolerance. A threshold is set prior to the extraction. Ideally, the watermark should be passed and the recognizer verifies the code and answers if the watermark vector is still present in the code within the threshold boundary.

17.2 Vector Instruction Group Identification

The instruction groups are identified keeping in mind the probability of occurence of the instructions in the code which is to be watermarked. Each vector group contains atmost 4 instructions. This is because a greater number of instructions in a vector group has less likelihood of occurence in the code. We have at present identified 8 vector instruction groups. The implementation has been done such that new groups can be appended to the existing vector groups as and when identified. The number of vector groups denotes the length of the watermark vector than can be embedded in the code.

```
package sandmark.watermark.objectwm;

public class ObjectWatermark extends sandmark.watermark.StaticWatermarker {

   public ObjectWatermark();

   public String getShortName();
   public String getLongName();

   public static sandmark.util.ConfigProperties getProperties();
   public static void setProperties(sandmark.util.ConfigProperties);

   public sandmark.util.ConfigProperties getConfigProperties();
   public void setConfigProperties(sandmark.util.ConfigProperties);

   public java.lang.String getAlgHTML();

   public java.lang.String getAlgURL();

   public void embed(java.util.Properties);
      - Procedure to embed the watermark. Invokes the APIs in the Insertion class.

   public java.util.Iterator recognize(java.util.Properties);
      - Procedure for watermark recognizing.

   class Recognizer implements java.util.Iterator {
      public Recognizer(String jarInput);
      public void generate();
      public boolean hasNext();
      public java.lang.Object next();
   }
}
```

Figure 17.1: The general APIs used for implementing Algorithm for Robust Object Watermarking. label

```
package sandmark.watermark.objectwm;

public class CodeBook {

   CodeBook();
      - defines the CodeBook ie. the CodeGroup instruction groups, the vector
        group instruction groups and the dependencies between them.

   int getInstructionFromCodeBook(String[], int, int, int);
      - invoked from the Insertion class to access the CodeBook.

      private String[][] getParams(String[], int);

      private String[] putParams(String[], int, String[][]);

}
```

Figure 17.2: The above figure shows the different APIs used for CodeBook access.

17.3 Building the CodeBook :

The CodeBook construction goes in parallel with the identification of the Vector Instruction Groups. Set of instructions are identified which are semantically equivalent and are placed in the same set. Moreover the equivalence should be such that it facilitates the increase of instruction vector frequencies. The following points are taken into consideration while building the CodeBook.

1. Each vector instruction group is dependent on atleast one CodeBook instruction group for its insertion.
2. Each instruction set in the CodeBook services exactly one vector instruction group. This simplifies the insertion procedure and isolates the frequency updates of different vector groups.
3. Each group in the CodeBook contains a set of instruction which has high frequency of occurence in the code. This gives more oppurtunities for code subsitution rather than new code insertion for vector frequency increment.
4. The code substitution increases the size of the code,but minimally.
5. The CodeBook has been designed such that it can be upgraded as and when new groups are identified with minimal effect on the APIs accessing the CodeBook.

A dependency array stores the dependence between each Vector Group Instruction and the set of CodeBook Instructions.

17.4 Vector Extraction

The vector extraction procedure is invoked at the initial stage prior to watermark insertion and during watermark recognition. The entire code is scanned for the occurence of the Vector Instruction Groups and the vector components are formed out of the frequencies of the corresponding groups.

17.5 The overall watermarking algorithm

Vector Extraction: Define n as a security parameter. Define a set S of n ordered groups of machine language instructions. For each group i in S, compute the frequency c_i of the group in the code, and form the vector

```
package sandmark.watermark.objectwm;

public class vectorExtraction {

   public static java.util.Vector extractVector(String);
      - implements the vector extraction procedure
}
```

Figure 17.3: VectorExtraction APIs

$c = (c_1, \ldots, c_n)$. This c is the extracted vector.

Algorithm
Initialization: Set a detection threshold σ.
Watermark Insertion:

- Apply the vector extraction step to obtain a vector c (of length n).
- Choose an n coordinates vector $w = (w_1, \ldots, w_n)$ whose coefficients are randomly distributed following a normal law with standard deviation α.
- Modify the code in such a way that the new extracted vector $\overline{c}$ is $c + w$.

Watermark Testing:

- Apply the extraction step to obtain a vector d.
- Compute a simliarity measure Q between $d - \overline{c}$ and w.
- If Q is higher than σ then the algorithm outputs *marked*, else it outputs *unmarked*.

17.6 Insertion

The insertion procedure is the most crucial step in the entire watermarking algorithm. For each Vector Instruction Group of the watermark vector, the algorithm intends to increase the frequency of that group by the vector value. It resorts to three techiniques to increase the vector group frequency.

- ***Code substitution:*** In this techinique we looks for group of instructions in the code that can be substituted by a equivalent set of instructions as enumerated in the CodeBook to increase the vector frequency. As an example, say vector group A is dependent on the CodeGroup[i,j]. We look for the occurence of CodeGroup[i,0] and substitute it by the equivalent set of instructions in the CodeGroup[i,j]. This increases the frequency of the vector by one. Some issues which we deal with while doing the substitution are enumerated below:

  1. We handle jump to a target out from this set of instructions appropriately by saving target instruction handles and later replacing it in the substituting instruction set.
  2. At present we do not handle jumps into the instruction set. So we discard the occurence of instruction sets which have branch targets as candidates for substitutions.
  3. The substitution increases the code length minimally as per the design of the CodeBook. We observed that finding an equivalence in which the vector frequency is increased as well the code size is reduced very difficult(unless the vector instruction group is restricted to almost one).

  Depending on the CodeBook implementation and the actual occurences of the instruction groups in the code there is an extent to which subsitution can be done to increment the vector frequency. Once this is exhausted, we resort to the next two techinques for vector increment.

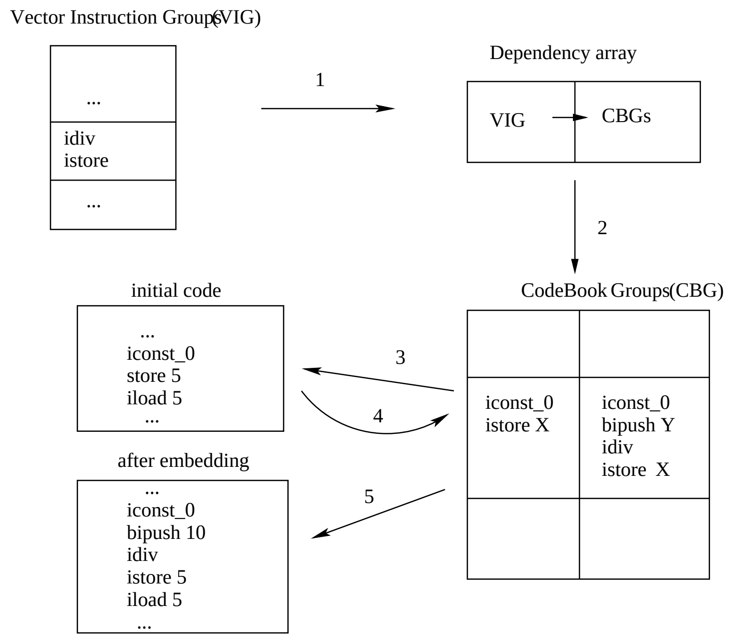


Figure 17.4:

- ***New code insertion:*** This is the most difficult part which we encountered in the entire implementation. First of all, instructions cannot be inserted at any random point in the code. After embedding, the modified code should pass the verifier. The following things were handled during the insertion:

  1. The substituting instructions corresponding to the vector are taken from the CodeBook group it is dependent on. To ensure that these set of instructions do not affect the stack size, we appended the instructions with required number of *pop* instructions.
  2. The insertion point is verified so that it does not breakup any other exisiting vector instruction group in the code.
  3. The variables used in the instructions to be inserted need to be declared and initialized in the method in which they are inserted prior to their use.
  4. To preserve the semantics of the code we inserted opaque predicates to jump over the newly inserted instructions. All the declarations for the predicate variables are done in the start of the method, though in the ideal case it could be anywhere before the point where they are used. Ideally we can have a set of such predicates in pool from which we can pick one in random and plugin into the method after inserting the vector instruction group codes.

- ***Method Cloning:*** In this techinique we identify the existing methods where we have occurence of the particular vector group instructions and create a clone of that method. This effectively increments the vector frequency. Some issues handled during this cloning are:

  1. Unlike other techniques, we have to keep track of the frequency updates of all the vectors since the replication might increase the vector frequencies of other vector groups.
  2. We maintain a threshold beyond which if the frequencies are modified, then we back track and do not replicate the method.

3. Ideally, we would make a bogus call to this clone method. But we do not implement such calls. In such a case a problem arises due to parameter passing. We need to initialize all the parameters in the clone method prior to all their use.

17.7 Comparison with the previous implementation on x86 assembly code

A previous implementation of this object watermarking was done in x86 assembly code. Each of these languages had its own constraints that restricted the implementation of the watermark embedding. In x86 assembly code we need to be careful in register selections during new instruction embedding as well as code substitution so that they dont interfere with each other. In the implementation we did on java bytecodes, we figured out that one needs to take into consideration the stack size while embedding new code. But no such problems arised during code substitutions which are semantically equivalent. The x86 code provides additional flexibility of swapping instruction sets or reordering instructions within an instruction group as long as the dependencies between the instructions are not violated, while such transformations have limited scope in the case of java bytecodes primarily due to the constraints of maintaining the proper stack height.

17.8 Analysis of attacks

The watermark faces the most serious threat from decompiling and recompiling the code. The compiler optimizations may change the sequence of instructions and even some instructions itself. The instructions in the vector group might well be the target of such modifications which results in the modification of the embedded frequency vector. But at the same time, since the watermark is spread across the entire code, the attacker has least idea about the exact location of the watermark. So to effect any substantial chenge in the vector frequency he needs to decompile and recompile the entire code rather than parts of the code which is tedious. Code compression poses the least threat since code has to be decompressed again to render it executable, so the watermark remains intact.

17.9 Brief analysis of our work and scope for future Work

We followed an extremely conservative approach during watermark embedding. One approach would have been to make an arbitrary change in the code and verify whether the new vector is closer to the target vector. If it digresses in the wrong direction then backtrack and try out another random modification. In our approach we always made sure prior to making any modification to the code that the resulting vector is always closer to the target vector. Such an approach was simple as well as preserved correctness of the code in all instances. The ratio of the code size before and after embedding increases minimally if the bulk of the code modification is done through substitution rather than new code insertion which apparently increases the code size drastically by a large factor. So its extremely important to identify the correct CodeBook and the Vector Instrution groups. The strengh of the watermark lies in the secret CodeBook which is maintained since there are always a wide range of possibilities of code equivalence and also the distribution of watermark over the entire code rather than being concentrated at a single point in the code. This increases the stealth of the watermark. Also we observed that codes with lower level of tolerance seem to withstand the tampering of this type of watermark more than code which provides more scope for modifications.

As a future work, we would like to identify some sort of profiling techniques to identify a proper set of vector and CodeBook instructions. We would also like to identify a few more approaches to embed the vector. The overall implementation needs to be modified to some extent so that we can have a framework in which we can plugin new code modification techniques.

```
package sandmark.watermark.objectwm;

public class Insertion {

   public void modifyCode(java.util.Vector );
      - Entry procedure for this 'Insertion' class


   de.fub.bytecode.generic.InstructionHandle getCodeSubstPoint(String[], int, String[]);
      - Searches and marks the instruction group occurence point in the code and returns the
        instructions to be substituted.

   void substituteCode(de.fub.bytecode.generic.InstructionHandle, String[], int);
      - Subsitutes the new instructions  obtained from the codeBook into the code at the
        marked instruction handle point.

   void newsubstituteInstruction_Embed(String[], int);
      - Implements the insertiong of previously non-existing code to increase the
        vector frequency. This sets a random class-method-instruction insertion point for
        insertion.


      void substituteNewCode( de.fub.bytecode.generic.InstructionHandle, String[], int);
         - Invoked by newsubstituteInstruction_Embed(); performs the actual low level insertion
           operation.

         String transformCode(String);
            - invoked by substituteNewCode();  it populates the new instruction set to be
              embedded with valid data and returns the final instruction to be inserted.

            int getLocalVarIndex_CreateIndex();
               - invoked by transformCode(); returns a vaild local variable index. if none
                 exist then it creates a new one and returns its index.


         de.fub.bytecode.generic.Instruction extractInstrType(String);
            - This API forms the instruction object to be inserted after parsing the input
              instruction stream and returns the object.

```

Figure 17.5: The following APIs are implemented for the watermark Insertion procedure

```
         void insertOpaque( de.fub.bytecode.generic.InstructionList,
                            de.fub.bytecode.generic.InstructionHandle,
                            de.fub.bytecode.generic.InstructionHandle,
                            de.fub.bytecode.generic.InstructionHandle, int);
            - Generates the opaque predicate and inserts it in appropriate position.


         int getOpaqueDeclPoint(int);
            - checks for and returns a $safe$ point for new instruction insertion.

      boolean checkSplitVectorGrp(int, de.fub.bytecode.generic.InstructionHandle[]);
         - Ensures that the insertion point for the new vector group does not split
           another existing vector instruction group in the code and consequently
           reduce its vector freqeuency.


   int newcopyMethod_Embed(String[], int, int, int);
      - Implements the method cloning procedure.


   int remVecfreqUpdatesInThreshold(int, int);
      - Ensures that the method cloning does not update the vecotr frequencies beyond
        the {\em threshold}.


   /* the various support APIs for the above APIs */

   int getRandomValue(int, int);

   void getInsertLocation(Integer[]);

   int varTypeIsInt(int);

}
```

Figure 17.6: The following APIs are implemented for the watermark Insertion procedure

17.10 Conclusion:

The vector extraction paradigm produces a robust watermarking technique. It mainly takes advantage of the distribution of watermark over the code to maintain its stealth. But at the same time the effectiveness in the java bytecodes lies entirely on the exact implementation technique.

17.11 Bibliography

1. Julien P. Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquate r. Robust Object Watermarking: Application to Code. In A. Pfitzmann, editor , Information Hiding '99, volume 1768 of Lectures Notes in Computer Science (LNCS), pages 368–378, Dresden, Germany, 2000. Springer-Verlag. http://ci teseer.nj.nec.com/stern00robust.html

2. I. Cox, J. Kilian, T. Leighton and T. Shamoon, "A secure, robust watermark for multimedia", in Proc. Workshop on Information Hiding, Univ. of Cambridg e, U.K., May 30 - June 1, 1996, pp. 175-190 http://citeseer.nj.nec.com/a rticle/cox96secure.html