Hacking SandMark

Christian Collberg

January 10, 2002

Contents

I Working with SandMark 5

1 Introducing SandMark 7
- 1.1 Introduction 7
  - Trying out the Watermarker 8
- 1.2 Installing SandMark 9
  - Building SandMark 10
- 1.3 Scripting SandMark 11

2 The SandMark Code-base 13
- 2.1 SandMark Packages 13
  - sandmark 13
  - sandmark.gui 13
  - sandmark.html 14
  - sandmark.obfuscate 14
  - sandmark.obfuscate.setfieldspublic 14
  - sandmark.obfuscate.splitclass 15
  - sandmark.obfuscate.util 15
  - sandmark.optimise 15
  - sandmark.util 15
  - sandmark.util.exec 16
  - sandmark.util.graph 17
  - sandmark.util.graph.codec 17
  - sandmark.util.javagen 17
  - sandmark.watermark 19
  - sandmark.watermark.CT 19
  - sandmark.watermark.CT.embed 19
  - sandmark.watermark.CT.encode 20
  - sandmark.watermark.CT.encode.ir 20
  - sandmark.watermark.CT.encode.ir2ir 21
  - sandmark.watermark.CT.encode.storage 21
  - sandmark.watermark.CT.recognize 21
  - sandmark.watermark.CT.trace 22
  - sandmark.watermark.constantstring 22
  - sandmark.statistics 22

3 Extending SandMark 23
- 3.1 Adding an Obfuscator 23
- 3.2 Adding a Watermarker 24
- 3.3 Adding a Graph Codec 25

3.4 Documentation 26

II SandMark Algorithms 29

4 The Collberg-Thomborson Watermarking Algorithm 31
4.1 Introduction 31
4.2 Annotation 31
4.3 Tracing 34
An Example 35
Choosing `mark()`-Locations 36
4.4 Embedding 36
Building the Graph 36
Splitting the Graph 37
Generating Intermediate Code 38
Generating Java Code 40
Inserting the Java Code 41
4.5 Recognition 43
4.6 Passing Roots in Formal Parameters 44

5 The ConstantString Static Watermarking Algorithm 45
5.1 Introduction 45
5.2 Embedding 45
5.3 Recognition 45

III Appendices 47

A Useful Tools 49
A.1 Examining Java Classfiles 49
javap 49
Jasmin 49
BCEL's Class Construction Kit 50
BCEL's listclass 50
BCEL's JustIce 51
A.2 Classfile Editors 51
BCEL 51
BLOAT 51

B Working with the SandMark Code-base 53
B.1 Coding standard 53
B.2 Using CVS 54
Installing CVS 54
Getting Started 54
Checking out code 55
Updating 56
Deleting files 56
Summary 57

Part I

Working with SandMark

Chapter 1

Introducing SandMark

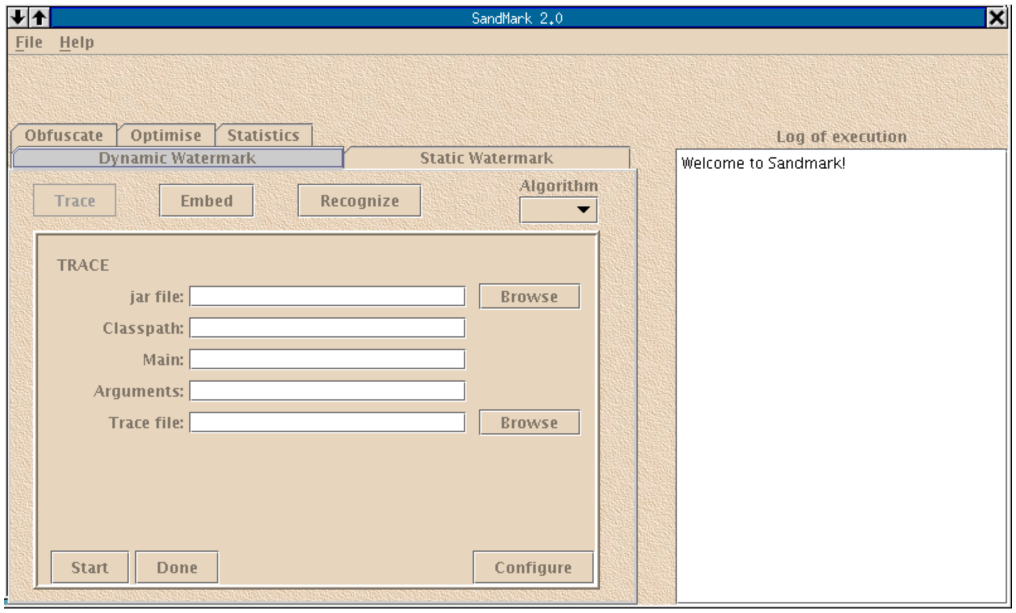
1.1 Introduction

SandMark is designed to be the Swiss-Army-Chainsaw of software protection research. In other words, we hope to build an infrastructure that makes it easy to implement algorithms for

1. code obfuscation,
2. softeware watermarking, and
3. tamper-proofing.

In fact, we hope to implement *every* software protection algorithm know to man, so that we can compare and evaluate them.

You normally interact with sandmark through its graphical interface. Start it by typing `smark` in the `smbin` directory:



Welcome to Sandmark!

Trying out the Watermarker

To get started try watermarking the `TTT` (tic-tac-toe) application. It can be found in the `smapps2` directory. Start SandMark, then do the following:

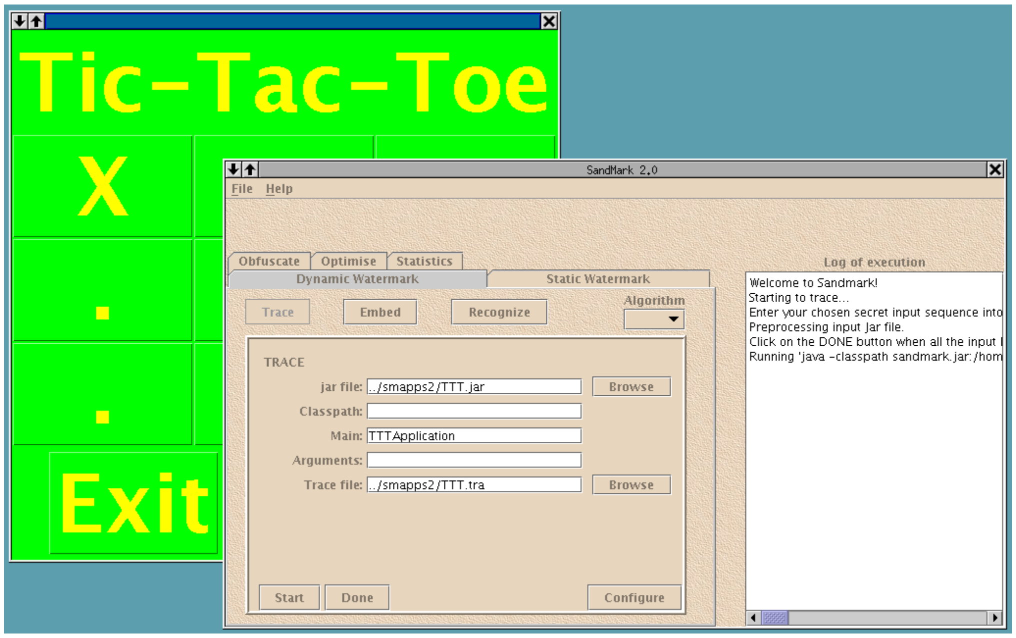
1. In the *trace* pane enter

   Jar-file to watermark: `TTT.jar`

   Main class name: `TTTApplication`

2. Hit $\boxed{\text{START}}$.

3. Click on a few X's and O's. Remember the order in which you do the clicks! It should look something like this:

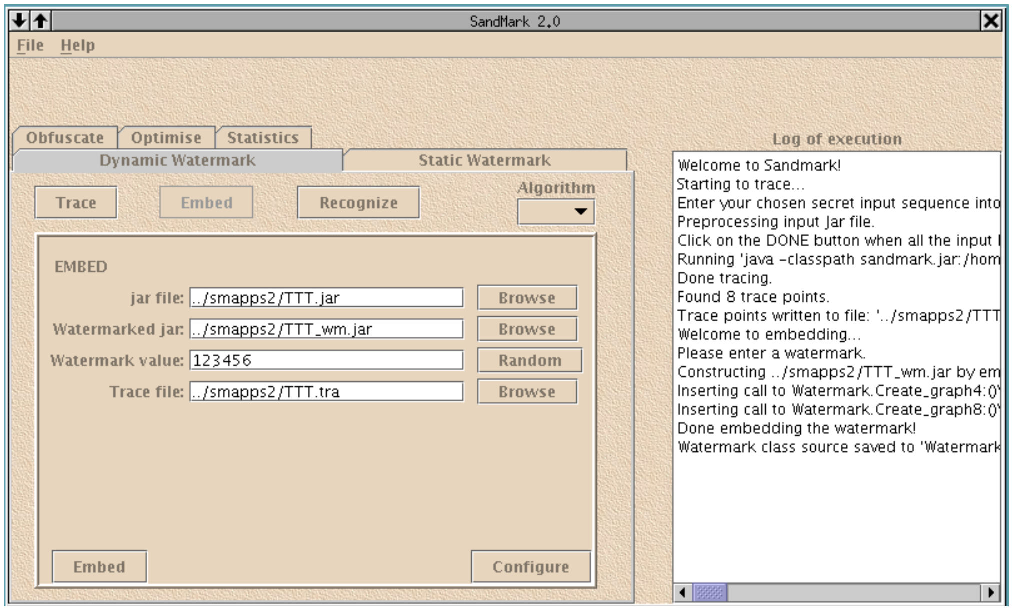


jar file: ../smapps2/TTT.jar
Classpath:
Main: TTTApplication
Arguments:
Trace file: ../smapps2/TTT.tra

```
Welcome to Sandmark!
Starting to trace...
Enter your chosen secret input sequence into
Preprocessing input Jar file.
Click on the DONE button when all the input l
Running 'java -classpath sandmark.jar:/hom
```

4. Hit $\boxed{\text{DONE}}$.

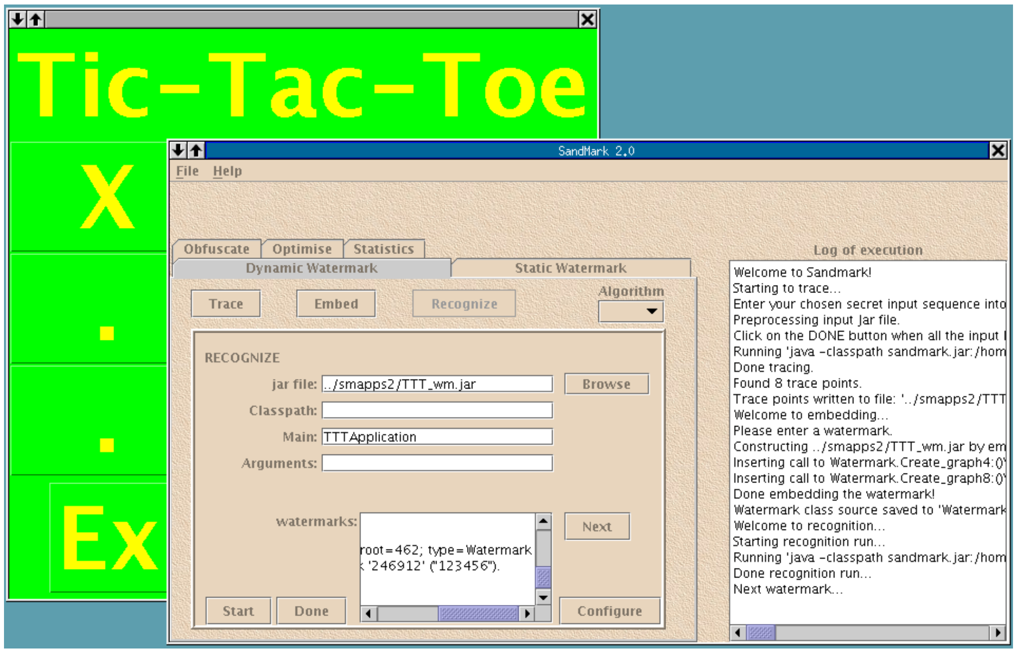
5. Go to the *embed* pane and enter the watermark value `123456`, like this:



jar file: ../smapps2/TTT.jar
Watermarked jar: ../smapps2/TTT_wm.jar
Watermark value: 123456
Trace file: ../smapps2/TTT.tra

```
Welcome to Sandmark!
Starting to trace...
Enter your chosen secret input sequence into
Preprocessing input Jar file.
Click on the DONE button when all the input l
Running 'java -classpath sandmark.jar:/hom
Done tracing.
Found 8 trace points.
Trace points written to file: '../smapps2/TTT
Welcome to embedding...
Please enter a watermark.
Constructing ../smapps2/TTT_wm.jar by em
Inserting call to Watermark.Create_graph4:()'
Inserting call to Watermark.Create_graph8:()'
Done embedding the watermark!
Watermark class source saved to 'Watermark
```

6. Hit EMBED.
7. Go to the *recognize* pane and hit the START button.
8. Click on the same X's and O's as you did in step 3), in the same order.
9. Hit DONE:



jar file: ../smapps2/TTT_wm.jar
Classpath:
Main: TTTApplication
Arguments:
watermarks:

```
root=462; type=Watermark
k '246912' ("123456").
```

```
Welcome to Sandmark!
Starting to trace...
Enter your chosen secret input sequence into
Preprocessing input Jar file.
Click on the DONE button when all the input l
Running 'java -classpath sandmark.jar:/hom
Done tracing.
Found 8 trace points.
Trace points written to file: '../smapps2/TTT
Welcome to embedding...
Please enter a watermark.
Constructing ../smapps2/TTT_wm.jar by em
Inserting call to Watermark.Create_graph4:()'
Inserting call to Watermark.Create_graph8:()'
Done embedding the watermark!
Watermark class source saved to 'Watermark
Welcome to recognition...
Starting recognition run...
Running 'java -classpath sandmark.jar:/hom
Done recognition run...
Next watermark...
```

You should see the watermark 123456 extracted from the watermarked TTT application.

You can also try

```
> smark -f ../smapps2/TTT.script
```

This will run a script that traces, embeds, and recognizes a watermark in the TTT application. You still have to enter the X's and O's and hit the $\boxed{\text{DONE}}$ buttons in the trace and recognize panes.

1.2 Installing SandMark

SandMark's source code is stored in a CVS repository at cvs.cs.arizona.edu. You can get the source anonymously (in which case you can't make any changes to it): To get the sources anonymously, do the following

```
> cvs -d :pserver:anonymous@cvs.cs.arizona.edu:/cvs/wmark login
> cvs -d :pserver:anonymous@cvs.cs.arizona.edu:/cvs/wmark checkout -P smark2
> cvs -d :pserver:anonymous@cvs.cs.arizona.edu:/cvs/wmark checkout -P smextern
> cvs -d :pserver:anonymous@cvs.cs.arizona.edu:/cvs/wmark checkout -P smapps2
> cvs -d :pserver:anonymous@cvs.cs.arizona.edu:/cvs/wmark checkout -P smbloat
> cvs -d :pserver:anonymous@cvs.cs.arizona.edu:/cvs/wmark checkout -P smbin
```

or, if you have got write access to SandMark (i.e an account on cvs.cs.arizona.edu) you should instead do

```
> cvs -d :ext:MyLogin@cvs.cs.arizona.edu:/cvs/cvs/wmark checkout -P smark2
> cvs -d :ext:MyLogin@cvs.cs.arizona.edu:/cvs/cvs/wmark checkout -P smextern
> cvs -d :ext:MyLogin@cvs.cs.arizona.edu:/cvs/cvs/wmark checkout -P smapps2
> cvs -d :ext:MyLogin@cvs.cs.arizona.edu:/cvs/cvs/wmark checkout -P smbloat
> cvs -d :ext:MyLogin@cvs.cs.arizona.edu:/cvs/cvs/wmark checkout -P smbin
```

where `MyLogin` is your account name on `cvs.cs.arizona.edu`.

You should now have four directories:

smark2: The SandMark sources.

smbin: Scripts.

smextern: External Java code i.e. jar and zip files needed to run SandMark.

smapps2: Some simple applications you can use to try out SandMark

smbloat: Some simple test cases for Bloat. Read these to get a feel for how Bloat should be used.

Once you have checked out SandMark you can get the latest version by running the `cvs update` command:

```
> cd smark2
> cvs update -dP
```

To add a new file you do `cvs add file`, to remove it `cvs rm file` and to commit your changes to the repository at `cvs.cs.arizona.edu` you say `cvs commit`.

Building SandMark

Note that you will need Java 1.4 to run SandMark properly. Get the latest version from `http://java.sun.com/j2se/1.4`.

Do the following to build SandMark:

```
> cp smark2/Makedefs.std smark2/Makedefs
# Make the 'obvious' changes to Makedefs.
# In particular, you should set these variables:
#   JDK =
#   HOME =

> cp smbin/smark.std smbin/smark
# Make the 'obvious' changes to smark.
# Again, you should set these variables:
#   JDK =
#   HOME =

> chmod a+rx smbin/smark
> make

# Build applications to watermark
> cd smapps2
> make

# Start SandMark
> smbin/smark
```

SandMark gets compiled into a jar-file `sandmark.jar`. To execute it you also need some other packages (`bloat`, `BCEL`, etc.), which can be found in the `smextern` directory. The `smark`-script takes care of setting Java's classpath correctly so that these get picked up.

You can also build the manual:

```
> cd smark2/doc
> make
```

Finally, you can generate html or latex from the JavaDoc comments in the source:

```
> make jdoc
> make jtex
```

This generates a PostScript file `classes.ps` and a directory `jdoc` of html files.

1.3 Scripting SandMark

SandMark can be scripted. Either start SandMark from the command line with the `-f` option:

```
> smark -f file.script
```

or enter the script file in SandMark's `file` pull-down manu.

- You can set a property value using the command

  ```
  set PROPERTY VALUE
  ```

  The following property values are recognized:

  AnnotatorClass:

  Encode_NodeType:

Encode_ParentClass:

Encode_ClassName:

Encode_AvailableEdges:

Encode_StoreWhat:

Encode_StoreMethods:

Encode_StoreLocation :

Encode_ProtectionMethods:

Encode_IndividualFixups:

Encode_Encoding: Should be one of `perm` or `radix`. `"*"` picks a random encoding method.

Encode_Components:

Encode_Package:

MaxTracePoints:

Encode_StoreLocation: One of `formal` or `global`.

DumpIR:

ClassPath:

- To run tracing use the command

  ```
  trace input.jar trace.tra MAINCLASS ARGUMENTS
  ```

  The classpath is set through the command

  ```
  set ClassPath ...
  ```

  Tracepoints are written to the `trace.tra` file.

- Embed watermark in input.jar using the command

  ```
  embed input.jar output.jar watermark trace.tra
  ```

  Read the tracepoints from the file `trace.tra`.

- Obfuscate input.jar, creating output.jar:

  ```
  obfuscate input.jar output.jar
  ```

- Run recognition.

  ```
  recognize input.jar watermark_count MAINCLASS ARGUMENTS
  ```

  The classpath is set through the command

  ```
  set ClassPath ...
  ```

- If the first non-blank character on a line is # the rest of the line is ignored.

- Commands are case insensitive, arguments are case sensitive.

Chapter 2

The SandMark Code-base

2.1 SandMark Packages

SandMark is a large application. Currently it is made up of some 29000 lines of java code distributed over 200 classes. The code is organized in a fairly deep package hierarchy.

sandmark

Classes at the top level:

sandmark.Console: The main entry point to SandMark. It contains code that starts up the GUI and acts as an interface between the GUI and the rest of the application.

sandmark.CLI: The command line interpreter.

sandmark.Scripting: The script interpreter.

sandmark.gui

`sandmark.gui` contains the code that build up the graphical interface.

sandmark.gui.AboutDialog:

sandmark.gui.ButtonRenderer:

sandmark.gui.CodeDialog:

sandmark.gui.EmbedConfigDialog:

sandmark.gui.ExtensionFileFilter:

sandmark.gui.GUIListener:

sandmark.gui.HelpFrame:

sandmark.gui.IntegerInput:

sandmark.gui.LayoutConstraints:

sandmark.gui.LightRenderer:

sandmark.gui.MultiHeaderRenderer:

sandmark.gui.ObTableModel:

sandmark.gui.ObfuscateConfigDialog:

sandmark.gui.RelativeLayout:

sandmark.gui.SandMarkFrame:

sandmark.gui.SandMarkGUIConstants:

sandmark.gui.SkinPanel:

sandmark.gui.StatDialog:

sandmark.gui.StatTableModel:

sandmark.gui.TablePanel:

sandmark.gui.SMarkGUIConstants:

sandmark.gui.SandMarkOSConstants:

sandmark.html

`Html` help files are kept here.

sandmark.obfuscate

Each obfuscation lives in its own subdirectory in `sandmark.obfuscate`. `sandmark.obfuscate.setfieldspublic`, for example, is an obfuscator that sets all fields to `public`.

sandmark.obfuscate.AllClassesObfuscator:

sandmark.obfuscate.AppObfuscator: The abstract base-class from which any obfuscator that works on an entire application should inherit. It inherits from `sandmark.obfuscate.GeneralObfuscator`.

sandmark.obfuscate.ClassObfuscator: The abstract base-class from which any obfuscator that works on a single class should inherit. It inherits from `sandmark.obfuscate.GeneralObfuscator`.

sandmark.obfuscate.GeneralObfuscator: The top-level abstract class for an obfuscator.

sandmark.obfuscate.MethodObfuscator: The abstract base-class from which any obfuscator that works on a single method should inherit. It inherits from `sandmark.obfuscate.GeneralObfuscator`.

sandmark.obfuscate.NameOverloadObfuscator: An advanced name obfuscator based on Paul Tyma's patent.

sandmark.obfuscate.Obfuscate: This class handles the incoming obfuscation requests from the GUI. It is called by `sandmark.Console`.

sandmark.obfuscate.Obfuscator: The top-level obfuscator that decides on overall strategy – which algorithms should be applied to which pieces of code.

sandmark.obfuscate.setfieldspublic

An obfuscator that sets all fields and methods to public.

sandmark.obfuscate.setfieldspublic.SetFieldsPublic:

sandmark.obfuscate.splitclass

An obfuscator that splits a class in two pieces. Non-functional.

sandmark.obfuscate.splitclass.DummyClass:

sandmark.obfuscate.splitclass.Obf:

sandmark.obfuscate.splitclass.SplitClass:

sandmark.obfuscate.util

`sandmark.obfuscate.util` contains classes of interest to all obfuscators.

sandmark.optimise

Routines to optimize Java programs. Currently based solely on the BLOAT package.

sandmark.optimise.Optimise:

sandmark.optimise.Main:

sandmark.optimise.Optimiser:

sandmark.util

`sandmark.util` contains classes of interest to all SandMark tools.

sandmark.util.Compile:

sandmark.util.BCEL: Utility routines to work with the BCEL (formerly `JavaClass`) package.

sandmark.util.ByteCodeLocation: Represents a location within a Java application. Essentially a tuple

$$\langle \texttt{sandmark.util.MethodID}, \texttt{lineNumber}, \texttt{byteCodeIndex} \rangle.$$

sandmark.util.CallGraphNode: The node in a call graph.

sandmark.util.CircularBuffer: A bounded buffer of arbitrary objects.

sandmark.util.ClassFileCollection: Routines for loading the classes of the program to be watermarked, obfuscated, etc. A class can be parsed into a `BCEL` or a `BLOAT` object.

sandmark.util.ClassHierarchy: A package for building the complete class hierarchy from a jarfile. There are many routines for querying the hierarchy, such as which methods override a particular method, which classes extend a particular class, etc.

sandmark.util.DependencyGraph: Used by `sandmark.obfuscate.splitclass.SplitClass`.

sandmark.util.Editor: Utility class for use with `BLOAT`.

sandmark.util.FileClassLoader: Used to load classes on the fly.

sandmark.util.FindClassFiles: Used to load classes on the fly.

sandmark.util.GraphViewer: Graphical display of `sandmark.util.graph.Graph`.

sandmark.util.InstructionTree:

sandmark.util.LabeledGrid:

sandmark.util.Log: Routines for printing out messages (both to a file and to the screen) to the user.

sandmark.util.MethodID: Represents a methodwithin a Java application. Essentially a tuple

$$\langle \texttt{name}, \texttt{signature}, \texttt{className} \rangle.$$

sandmark.util.MethodNode:

sandmark.util.Misc:

sandmark.util.Options: Routines for cracking command line parameters.

sandmark.util.PriorityQueue:

sandmark.util.SparseVector: A specialized version on `java.util.Vector`.

sandmark.util.StackFrame: Used to build call graphs.

sandmark.util.StatisticsRecord:

sandmark.util.Stats:

sandmark.util.StringInt: A routine to package up an int inside a string.

sandmark.util.TempDir: Routines to manage temporary directories.

sandmark.util.Time: Routines for timing Java programs.

sandmark.util.exec

`sandmark.util.exec` contains classes that allow SandMark to run another Java application under debugging. This uses Java's JDI package. JDI allows you to start up a Java program, set breakpoints, trace method calls, examine variables, etc. We use this in the implementation of the trace and recognition steps of the dynamic watermarking algorithms.

sandmark.util.exec.Breakpoint: A class for creating breakpoints.

sandmark.util.exec.DumpAll:

sandmark.util.exec.EventHandler:

sandmark.util.exec.EventThread:

sandmark.util.exec.Heap: A class for iterating through all the objects on the heap.

sandmark.util.exec.HeapData: Objects returned by `sandmark.util.exec.Heap`.

sandmark.util.exec.MethodCallData:

sandmark.util.exec.Output: A class that handles input and output from the application being executed.

sandmark.util.exec.Overseer: Extend this class to run a program under JDI.

sandmark.util.exec.TracingException:

sandmark.util.graph

Many watermarking algorithms make use of graphs. This package contains a multigraph package `sandmark.util.graph.Graph` as well as operations on such graphs, such as all-pairs-shortest-path, depth-first-search, etc.

sandmark.util.graph.GraphOp: Additional algorithms on graphs.

sandmark.util.graph.Dfs: Classify the edges of a graph as `tree`, `cross`, and `back`.

sandmark.util.graph.Edge: Edge objects of a `sandmark.util.graph.Graph`.

sandmark.util.graph.Graph: A class for constructing multi-graphs.

sandmark.util.graph.Matrix: A class where graphs are represented as an adjacency matrix.

sandmark.util.graph.Node: Node objects of a `sandmark.util.graph.Graph`.

sandmark.util.graph.Path: A class for representing a path in a `sandmark.util.graph.Graph`.

sandmark.util.graph.codec

This package implements various algorithms for embedding a number into the topology of a graph.

sandmark.util.graph.codec.DecodeFailure:

sandmark.util.graph.codec.GraphCodec: Base-class for graph codecs.

sandmark.util.graph.codec.PPCT: Non-functional.

sandmark.util.graph.codec.PermutationGraph: A permutation graph encoding.

sandmark.util.graph.codec.RadixGraph: A radix graph encoding.

sandmark.util.javagen

`sandmark.util.javagen` contains classes for building up Java methods and classes. It is essentially a layer on top of `BCEL`. `sandmark.util.javagen` allows you to build up a Java AST and then turn that into bytecode.

sandmark.util.javagen.AssignField:

sandmark.util.javagen.AssignIndex:

sandmark.util.javagen.AssignStatic:

sandmark.util.javagen.Block:

sandmark.util.javagen.Cast:

sandmark.util.javagen.Class:

sandmark.util.javagen.Comment:

sandmark.util.javagen.CondNotNullExpr:

sandmark.util.javagen.Discard:

sandmark.util.javagen.EmptyStatement:

sandmark.util.javagen.Expression:

sandmark.util.javagen.Field:

sandmark.util.javagen.FieldRef:

sandmark.util.javagen.Formal:

sandmark.util.javagen.IfNotNull:

sandmark.util.javagen.Java:

sandmark.util.javagen.List:

sandmark.util.javagen.LiteralInt:

sandmark.util.javagen.LiteralString:

sandmark.util.javagen.LoadIndex:

sandmark.util.javagen.Local:

sandmark.util.javagen.LocalRef:

sandmark.util.javagen.Method:

sandmark.util.javagen.MyClass:

sandmark.util.javagen.New:

sandmark.util.javagen.NewArray:

sandmark.util.javagen.Null:

sandmark.util.javagen.Return:

sandmark.util.javagen.Statement:

sandmark.util.javagen.StaticCall:

sandmark.util.javagen.StaticFunCall:

sandmark.util.javagen.StaticRef:

sandmark.util.javagen.Test:

sandmark.util.javagen.Try:

sandmark.util.javagen.VirtualCall:

sandmark.util.javagen.VirtualFunCall:

sandmark.watermark

This package contain all the watermarking algorithms. Each algorithm resides in its own directory.

sandmark.watermark.DynamicEmbed: Handles the embedding phase of a dynamic watermarker.

sandmark.watermark.DynamicRecognize: Handles the recognition phase of a dynamic watermarker.

sandmark.watermark.DynamicTrace: Handles the tracing phase of a dynamic watermarker.

sandmark.watermark.DynamicWatermarker: The base-class for all dynamic watermarkers.

sandmark.watermark.GeneralWatermarker: The base-class of all watermarkers

sandmark.watermark.StaticWatermarker: The base-class for all static watermarkers.

sandmark.watermark.StaticEmbed: Handles the embedding phase of a static watermarker.

sandmark.watermark.StaticRecognize: Handles the recognition phase of a static watermarker.

sandmark.watermark.WatermarkingException:

sandmark.watermark.Watermarking:

sandmark.watermark.CT

This package implements the Collberg-Thomborson dynamic watermarking algorithm.

sandmark.watermark.CT.CT: The main class for the Collberg-Thomborson algorithm. Extends `sandmark.watermark.DynamicWatermarker`.

sandmark.watermark.CT.embed

This package implements the embedding of a graph into a Java program. This is part of the Collberg-Thomborson watermarking algorithm.

sandmark.watermark.CT.embed.AddParameters:

sandmark.watermark.CT.embed.CallGraphPath:

sandmark.watermark.CT.embed.DeleteMarkCalls:

sandmark.watermark.CT.embed.EditedClass:

sandmark.watermark.CT.embed.EmbedData:

sandmark.watermark.CT.embed.Embedder:

sandmark.watermark.CT.embed.InsertStorageCreators:

sandmark.watermark.CT.embed.PrepareTrace:

sandmark.watermark.CT.embed.ReplaceMarkCalls:

sandmark.watermark.CT.encode

This package implements the encoding of a graph into a Java program. This is part of the Collberg-Thomborson watermarking algorithm.

sandmark.watermark.CT.encode.Encoder:

sandmark.watermark.CT.encode.Graph2IR:

sandmark.watermark.CT.encode.Split:

sandmark.watermark.CT.encode.ir

This package implements the encoding of a graph into a simple intermediate code. This is part of the Collberg-Thomborson watermarking algorithm.

sandmark.watermark.CT.encode.ir.AddEdge:

sandmark.watermark.CT.encode.ir.Build:

sandmark.watermark.CT.encode.ir.Construct:

sandmark.watermark.CT.encode.ir.Create:

sandmark.watermark.CT.encode.ir.CreateNode:

sandmark.watermark.CT.encode.ir.CreateStorage:

sandmark.watermark.CT.encode.ir.Debug:

sandmark.watermark.CT.encode.ir.Destroy:

sandmark.watermark.CT.encode.ir.Destruct:

sandmark.watermark.CT.encode.ir.Field:

sandmark.watermark.CT.encode.ir.Fixup:

sandmark.watermark.CT.encode.ir.FollowLink:

sandmark.watermark.CT.encode.ir.Formal:

sandmark.watermark.CT.encode.ir.IR:

sandmark.watermark.CT.encode.ir.Init:

sandmark.watermark.CT.encode.ir.List:

sandmark.watermark.CT.encode.ir.LoadNode:

sandmark.watermark.CT.encode.ir.Method:

sandmark.watermark.CT.encode.ir.NodeStorage:

sandmark.watermark.CT.encode.ir.PrintGraph:

sandmark.watermark.CT.encode.ir.ProtectRegion:

sandmark.watermark.CT.encode.ir.SaveNode:

sandmark.watermark.CT.encode.ir.StaticCall:

sandmark.watermark.CT.encode.ir2ir

This package implements transformations of the intermediate code used in the encoding of graphs in the Collberg-Thomborson watermarking algorithm.

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

sandmark.watermark.CT.encode.ir2ir.AddFormals:

sandmark.watermark.CT.encode.ir2ir.Builder:

sandmark.watermark.CT.encode.ir2ir.CleanUp:

sandmark.watermark.CT.encode.ir2ir.Debug:

sandmark.watermark.CT.encode.ir2ir.Destructors:

sandmark.watermark.CT.encode.ir2ir.InlineFixups:

sandmark.watermark.CT.encode.ir2ir.Protect:

sandmark.watermark.CT.encode.ir2ir.SaveNodes:

sandmark.watermark.CT.encode.ir2ir.Transformer:

sandmark.watermark.CT.encode.storage

This package implements operations on the storage of graph nodes in Collberg-Thomborson watermarking algorithm.

sandmark.watermark.CT.encode.storage.NodeStorage:

sandmark.watermark.CT.encode.storage.Array:

sandmark.watermark.CT.encode.storage.Hash:

sandmark.watermark.CT.encode.storage.GlobalStorage:

sandmark.watermark.CT.encode.storage.Pointer:

sandmark.watermark.CT.encode.storage.StorageClass:

sandmark.watermark.CT.encode.storage.Vector:

sandmark.watermark.CT.recognize

This package implements the recognition part of Collberg-Thomborson's software watermarking algorithm. It should be generalized to handle recognition in *any* dynamic software watermarking algorithm.

sandmark.watermark.CT.recognize.Heap2Graph:

sandmark.watermark.CT.recognize.RecognizeData:

sandmark.watermark.CT.recognize.Recognizer:

sandmark.watermark.CT.trace

This package currently implements the trace part of Collberg-Thomborson's software watermarking algorithm. It should be generalized to handle tracing in *any* dynamic software watermarking algorithm.

sandmark.watermark.CT.trace.Annotator:

sandmark.watermark.CT.trace.CallForest:

sandmark.watermark.CT.trace.Preprocessor:

sandmark.watermark.CT.trace.TracePoint:

sandmark.watermark.CT.trace.Tracer:

sandmark.watermark.constantstring

This package implements a trivial static watermarking algorithm.

sandmark.watermark.constantstring.ConstantString:

sandmark.watermark.constantstring.Test:

sandmark.watermark.constantstring.Program:

sandmark.statistics

This package counts all types of elements of a Java program.

sandmark.statistics.Statistics:

Chapter 3

Extending SandMark

3.1 Adding an Obfuscator

SandMark is designed to make it easy to add a new obfuscation algorithm. Assume that we want to add a new obfuscation `ReorderMethods`. The process would be the following:

1. Create a new directory `sandmark/obfuscate/reorderMethods`.

2. Create a new class `sandmark/obfuscate/reorderMethods/ReorderMethods.java`.

3. The `ReorderMethods` class should extend one of the base classes `AppObfuscator` (if the algorithm works on the entire user program), `MethodObfuscator` (if the algorithm works one method at a time), or `ClassObfuscator` (if the algorithm works on one individual class at a time). Let's assume that our algorithm reorders methods within one class. `ReorderMethods` should therefore extend `ClassObfuscator`, which looks like this:

```
package sandmark.obfuscate;
public abstract class ClassObfuscator extends GeneralObfuscator {
protected ClassObfuscator(String label) {
    super(label);
}
abstract public void apply(
    sandmark.util.ClassFileCollection cfc, String classname)
    throws Exception;
}
public String toString() {
    return "ClassObfuscator(" + getLabel() + ")";
}
}
```

4. The `ReorderMethods` class should look something like this:

```
public class ReorderMethods extends sandmark.obfuscate.ClassObfuscator {
   public ReorderMethods(String label) {
      super(label);
   }
   public void apply(
       sandmark.util.ClassFileCollection cfc, String classname) throws Exception {
        // Your code goes here!
     }
  }
```

5. Use BCEL or BLOAT to implement your obfuscation. The `cfc` parameter represents the set of classes to be obfuscated. Use routines in `sandmark.util.ClassFileCollection` to open a class to be edited by BCEL or BLOAT.

6. Type `make` at the top-level sandmark directory (`smark`). The new obfuscation should be loaded automagically at runtime.

3.2 Adding a Watermarker

Adding a new watermarking algorithm is similar to adding an obfuscator. Algorithms are loaded dynamically at run-time, so there is no need to explcitly link them into the system.

To create a new watermarking algorithm `wm` you

1. create a new directory `sandmark.watermark.wm`,

2. create a new class `sandmark.watermark.wm.WM` which extends `sandmark.watermark.StaticWatermarker` or `sandmark.watermark.DynamicWatermarker`. To build a new static watermarker you just have to implement two methods, one to embed the watermark into a jarfile and the other to extract it:

```
package sandmark.watermark;

public abstract class StaticWatermarker
   extends sandmark.watermark.GeneralWatermarker {

public StaticWatermarker() {}

/* Embed a watermark value into the program. The props argument
 * holds at least the following properties:
 *  <UL>
 *     <LI> Encode_Watermark: The watermark value to be embedded.
 *     <LI> Embed_JarInput: The name of the file to be watermarked.
 *     <LI> Embed_JarOutput: The name of the jar file to be constructed.
 *  </UL>
 */
public abstract void embed(
   java.util.Properties props)
      throws sandmark.watermark.WatermarkingException,
             java.io.IOException;


/* Return an iterator which generates the watermarks
 * found in the program. The props argument
 * holds at least the following properties:
 *  <UL>
 *     <LI> Recognize_JarInput: The name of the file to be watermarked.
 *  </UL>
 */
public abstract java.util.Iterator recognize(
        java.util.Properties props)
           throws sandmark.watermark.WatermarkingException,
                  java.io.IOException;

     }
```

3. Use BCEL or BLOAT to implement your watermarker. Have a look at the trivial static watermarker `sandmark.watermark.constantstring.ConstantString` for an example.

4. Type `make` at the top-level sandmark directory (`smark`). The new watermarker should be loaded automagically at runtime.

3.3 Adding a Graph Codec

Several watermarking algorithms encode the watermark as a graph. SandMark contains several methods for making this encoding, stored in the `sandmark.util.graph.codec` package.

Adding a new graph coder/decoder *codec* algorithm is similar to adding an obfuscator or watermarker: just add a new class to the `codec` directory, make sure it extends the appropriate class, type `make`, and the new algorithm will have been added to the system.

Every graph codec should extend `sandmark.util.graph.codec.GraphCodec`:

```
package sandmark.util.graph.codec;
public abstract class GraphCodec {
   public java.math.BigInteger value = null;
   public sandmark.util.graph.Graph graph = null;

/**
 * Codecs should implement this method to convert
 * the 'value' into 'graph'.
 */
   abstract void encode();

/**
 * Codecs should implement this method to convert
 * the 'graph' into 'value'. Whenever the decoding
 * failes (eg. because the graph has the wrong
 * shape) the codec should simply throw an exception.
 */
   abstract void decode() throws sandmark.util.graph.codec.DecodeFailure;

/**
 * Constructor to be used when encoding an integer into a graph.
 *   @param value   The value to be encoded.
 */
   public GraphCodec (
       java.math.BigInteger value) {
       this.value = value;
       encode();
   }

/**
```

```
 * Constructor to be used when decoding a graph to an integer.
 *   @param graph   The graph to be decoded.
 *   @param root    The root of the graph.
 *   @param kidMap  An array of ints describing which field
 *                  should represent the first child, the
 *                  second child, etc.
 */
   public GraphCodec (
       sandmark.util.graph.Graph graph,
       int kidMap[]) throws sandmark.util.graph.codec.DecodeFailure {
       this.graph = graph;
       this.kidMap = kidMap;
       decode();
   }
}
```

Note that there are two constructors. One is used when encoding an integer (`java.meth.BigInteger`) into a graph (a `sandmark.util.graph.Graph`), and another when decoding a graph into an integer.

For example, the simplest graph codec, `RadixGraph` looks like this:

```
package sandmark.util.graph.codec;

public class RadixGraph extends sandmark.util.graph.codec.GraphCodec {

    public final static String FULLNAME  = "Radix Graph";
    public final static String SHORTNAME = "radix";

// Used when encoding.
public RadixGraph (java.math.BigInteger value)  {
    super(value);
    this.fullName = FULLNAME;
    this.shortName = SHORTNAME;
}

void encode() { ... }

// Used when decoding.
   public RadixGraph (
       sandmark.util.graph.Graph graph,
       int kidMap[]) throws sandmark.util.graph.codec.DecodeFailure {
       super(graph, kidMap);
       this.fullName = FULLNAME;
       this.shortName = SHORTNAME;
   }
   void decode() throws sandmark.util.graph.codec.DecodeFailure { ... }
}
```

3.4 Documentation

To document a new obfuscator or watermarker, do the following:

1. create a new file `*name*.tex` in `smark/doc`:

```
\algorithm{The ... Algorithm}{Authors}
 ....
```

Where `Authors` is a comma-separated list of authors of this algorithm implementation.

2. add an `input`-statement to `smark/manual.tex`:

```
\part{\SM\ Algorithms}
\input{CollbergThomborson}
...
\input{name.tex}
```

3. Update `smark/makefile`:

```
SECTIONS = \
   ...
   CollbergThomborson.tex\
   ...
   name.tex \
```

4. Type `make`.

Part II

SandMark Algorithms

Chapter 4

The Collberg-Thomborson Watermarking Algorithm

C. Collberg, J. Nagra, G. Townsend

4.1 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 4.1, 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 4.3 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 4.2):

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.

4.2 Annotation

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

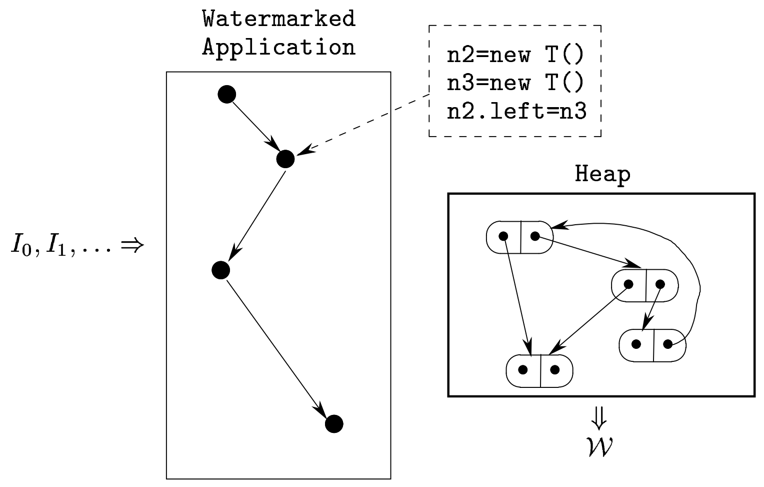


Figure 4.1: Overview of how the CT algorithm recognizes a watermark $\mathcal{W}$. At runtime the watermarked application will – given the special secret input key sequence $I_0, I_1, \ldots$ – traverse certain points in the program. At these points code has been inserted which builds a graph $G_{\mathcal{W}}$ on the heap. The topology of the graph embeds the watermark $\mathcal{W}$.

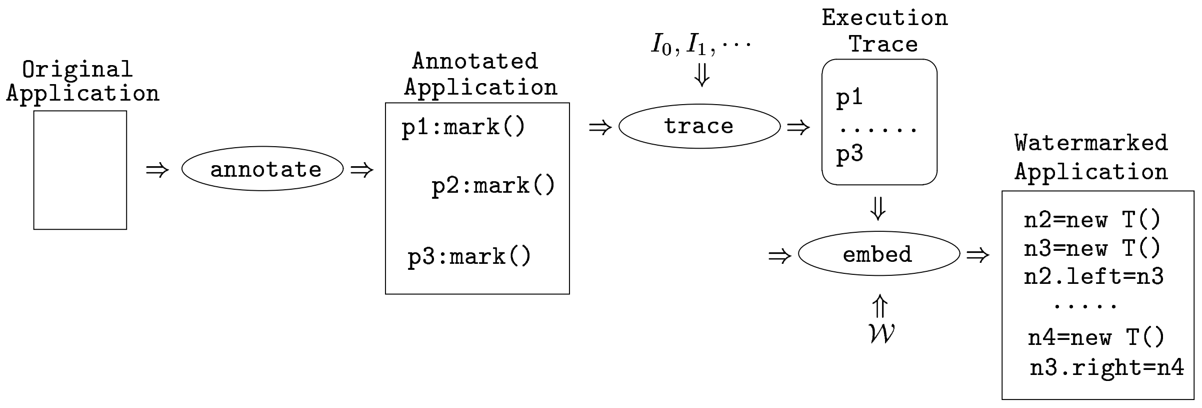


Figure 4.2: Overview of how the CT algorithm watermarks an application. First, the user adds *annotation points* (`mark()`-calls) to the application. These are locations where watermarking code may be inserted. Secondly, the application is run with a secret input sequence, $I_0, I_1, \ldots$ and the 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 4.3: 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.

4.3 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 4.4. 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++;
```

```
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 4.4: The class `sandmark.trace.Annotate`.

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:

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:

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

`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.

4.4 Embedding

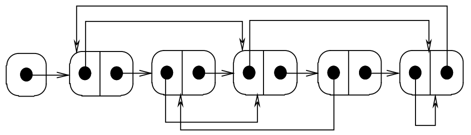
Once the application has been traced we can finally start embedding the watermark. The input to this phase is tracing information (as described in the previous section), a watermark $\mathcal{W}$ to be embedded, and a jar-file containing the classfiles in which to embed the mark. The embedding is divided 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 3.3 for a description of how to add a new *Graph Codec* to SandMark.



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

Figure 4.5: Radix-6 encoding. The right pointer field holds the `next` field, the left pointer encodes a base-k digit.

Radix Encoding is the simplest algorithm. The codec is in `sandmark.util.graph.codec.RadixGraph`. Figure 4.5 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

$$1024$$

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 4.5. 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

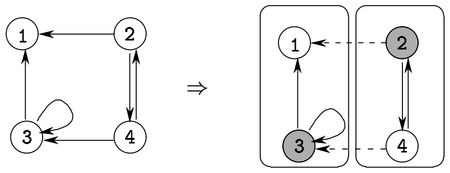
When the watermark graph has been built it needs to be split into pieces. This is done in the package `sandmark.watermark.CT.encode.Split`. There should be one graph component per `mark()`-location that we intend to use. There are three things to consider when we split the graph:

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:

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 4.1: Intermediate code instructions.



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

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 4.1.

Consider the two graph components G_2 and G_4 in Figure 4.6. 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($n_2 \xrightarrow{\texttt{edge2}} n_3$, $G_2$, $G_2$)
```

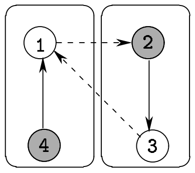


Figure 4.6: Two graph components G_2 and G_4. Components are named after their (shaded) root nodes.

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 4.6:

```
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 4.6).

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 4.7. 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

```
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 4.7: Java code generated from the graph components in Figure 4.6.

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.

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

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 4.2: Translation from intermediate code instructions to Java.

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 4.6 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.

4.5 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.

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 4.8):

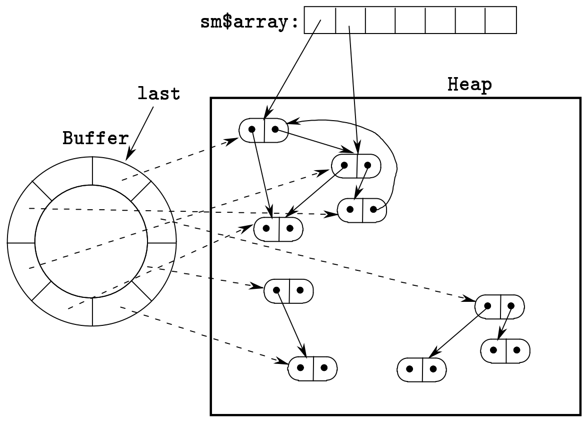


Figure 4.8: A view of memory during recognition. A circular linked buffer holds the last allocated objects. The recognizer examines the objects in reverse allocation order and extracts the subgraph reachable from each object. This is decoded into the watermark.

```
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.

4.6 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 4.8, 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 5
The ConstantString Static Watermarking Algorithm

C. Collberg

5.1 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`.

5.2 Embedding

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

```
String jarInput  = props.getProperty("Embed_JarInput");
String jarOutput = props.getProperty("Embed_JarOutput");
String watermark = props.getProperty("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);
```

5.3 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("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();
```

Part III

Appendices

Appendix A

Useful Tools

A.1 Examining Java Classfiles

There are a number of tools that are helpful for viewing Java classfiles.

javap

`javap` lists the contents of a Java classfile. It's particularly bad at displaying corrupted classfiles.

Normally, we call `javap` like this:

```
javap -c -s -verbose -l TTTApplication
```

These are the available options:

```
Usage: javap <options> <classes>...

where options include:
   -b                        Backward compatibility with javap in JDK 1.1
   -c                        Disassemble the code
   -classpath <pathlist>     Specify where to find user class files
   -extdirs <dirs>           Override location of installed extensions
   -help                     Print this usage message
   -J<flag>                  Pass <flag> directly to the runtime system
   -l                        Print line number and local variable tables
   -public                   Show only public classes and members
   -protected                Show protected/public classes and members
   -package                  Show package/protected/public classes
                             and members (default)
   -private                  Show all classes and members
   -s                        Print internal type signatures
   -bootclasspath <pathlist> Override location of class files loaded
                             by the bootstrap class loader
   -verbose                  Print stack size, number of locals and args for methods
                             If verifying, print reasons for failure
```

Jasmin

`Jasmin` is a Java bytecode assembler. It reads a text file containing Java bytecode instructions and generates a classfile. It can be found here: `http://mrl.nyu.edu/~meyer/jasmin/about.html`.

Here's is a simple class `hello.j` in the `Jasmin` assembler syntax:

```
.class public HelloWorld
.super java/lang/Object

;
; standard initializer (calls java.lang.Object's initializer)
;
.method public <init>()V
   aload_0
   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

;
; main() - prints out Hello World
;
.method public static main([Ljava/lang/String;)V
   .limit stack 2   ; up to two items can be pushed

   ; push System.out onto the stack
   getstatic java/lang/System/out Ljava/io/PrintStream;

   ; push a string onto the stack
   ldc "Hello World!"

   ; call the PrintStream.println() method.
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

   ; done
   return
.end method
```

Call `Jasmin` like this:

```
java -classpath smextern/jasmin.jar jasmin.Main hello.j
```

which will produce a class file `HelloWorld.class`.

BCEL's Class Construction Kit

`cck` is an interactive viewer and editor of Java classfiles. It is built on `BCEL`. Call it like this:

```
java -classpath .:smextern/BCEL.jar -jar smextern/cck.jar
```

and open a classfile from the `file`-menu.

BCEL's listclass

`listclass` comes with the BCEL package. It's a replacement for `javap`, and very useful in cases when `javap` crashes. Call it like this:

```
java -classpath smextern/BCEL.jar listclass -code TTTApplication
```

The following options are available:

```
java listclass [-constants] [-code] [-brief] [-dependencies] \
     [-nocontents] [-recurse] class... [-exclude <list>]
```

`-code`: List byte code of methods.

`-brief`: List byte codes briefly

`-constants`: Print constants table (constant pool)

`-recurse`: Usually intended to be used along with

`-dependencies`: When this flag is set, listclass will also print i

BCEL's JustIce

`JustIce` verifies classfiles. Call it like this:

```
java -classpath .:smextern/BCEL.jar:smextern/JustIce.jar \
  de.fub.bytecode.verifier.GraphicalVerifier
```

or like this:

```
java -classpath .:smextern/BCEL.jar:smextern/JustIce.jar \
   de.fub.bytecode.verifier.Verifier TTTApplication.class
```

A.2 Classfile Editors

SandMark reads Java classfiles, modifies them, and writes out the modified classfiles. There are a number of free packages available to parse, modify, and unparse classfiles. We are currently using two such packages, namely `BCEL` and `BLOAT`. Generally, `BLOAT` is comprehensive and hard to use, `BCEL` is simpler to use but not as complete.

BCEL

Add a BCEL manual here

BLOAT

Add a BLOAT manual here

Appendix B

Working with the SandMark Code-base

B.1 Coding standard

- Don't use tab characters.
- Indent by typing three (3) blanks.
- Put the left brace ({) on same line as preceding statement. I.e. format an if-statement like this:

  ```
  if () {
     ...
  }
  ```

  not like this:

  ```
  if ()
  {
    ...
  }
  ```

  or like this:

  ```
  if ()
     {
       ...
     }
  ```

- Don't use `import`-statements. Instead, always use fully qualified names. For example, say

  ```
  java.util.Properties p = new java.util.Properties();
  ```

  instead of

  ```
  import java.util.*;
  ...
  Properties p = new Properties();
  ```

  In a large system like SandMark it is difficult to read the code when you don't know where the a particular name is declared. In SandMark we use deep package hierarchies to organize the code and always refer to every object using its fully qualified name. This also prevents name clashes.

- We make one exception: you are allowed to say `String` rather than `java.lang.String`!
- We use the following naming strategy:
  - Package names are short and in lower case. We favor deep package hierarchies over packages with many classes.
  - Class names are typically long and descriptive. They start with an uppercase letter.
  - Method names start with a lowercase letter.

B.2 Using CVS

CVS is a *source code control system.* It is used extensively in industry and in the open source community to organize source code that many people are working on simultaneously.

The basic idea is to have one master copy of the source code, residing in a special source code repository. Programmers check out the latest version of the code to their local account and work on it until they're ready to share their new code with their co-workers. They then check in the new code to the repository from which the rest of the team can download the new changes.

A programmer can have several versions of the code checked out at the same time: one at work, one at home, one on the laptop, etc. Furthermore, every change ever made to a file is stored in the repository allowing a programmer to check out "the version of the program from last Monday."

Installing CVS

If you are running Linux on your home machine (and why wouldn't you be?) you should have CVS installed already. Otherwise you can download CVS from here: `http://www.cvshome.org/downloads.html`. The manual is here: `http://www.cvshome.org/docs/manual`. ⌜`man cvs`⌝ gives you the basic information you need.

You also need to have `ssh` installed on your machine in order to communicate with our CVS server, `cvs.cs.arizona.edu`.

If you just intend to do your assignments on `lectura`, no installation is necessary.

Getting Started

Let's assume that your team consists of Alice and Bob whose `lectura` logins are `alice` and `bob` with the passwords `alice-pw` and `bob-pw`, respectively. The team name is `cs453bd`.

One team member (in our case Alice) should do the following to get the team's source repository set up:

```
> whoami
alice
> mkdir ass1            # Create a temorary directory.
> cd ass1
> setenv CVS_RSH ssh    # Or export CVS_RSH=ssh. Must always be set.
> cvs -d :ext:alice@cvs.cs.arizona.edu:/cvs/cvs/cs453/cs453bd import -m "New!" ass1 aaa bbb ass1
   alice@cvs.cs.arizona.edu's password: alice-pw
```

Now, Alice deletes the temporary directory:

```
> rmdir ass1
```

Checking out code

Everything should now be set up properly on the CVS server. Alice can check out the code (which so-far only consists of a single directory):

```
> cvs -d :ext:alice@cvs.cs.arizona.edu:/cvs/cvs/cs453/cs453bd checkout ass1
   alice@cvs.cs.arizona.edu's password: alice-pw
cvs server: Updating ass1
> ls ass1
CVS
```

Alice now wants to start programming. She creates a new C module in her CVS directory:

```
> cd ass1
/home/alice/ass1
> cat > interpreter.c
main() {
}
> cvs add -m "Started the project" interpreter.c
   alice@cvs.cs.arizona.edu's password: alice-pw
   cvs server: scheduling file `interpreter.c' for addition
   cvs server: use 'cvs commit' to add this file permanently

> cvs commit -m "Finished first part of interpreter."
   cvs commit: Examining .
   alice@cvs.cs.arizona.edu's password:
   RCS file: /cvs/cvs/cs453/cs453bd/ass1/interpreter.c,v
   done
   Checking in interpreter.c;
   /cvs/cvs/cs453/cs453bd/ass1/interpreter.c,v  <--  interpreter.c
   initial revision: 1.1
   done
```

The `add` command told the CVS system that a new file is being created. The `commit` command actually uploaded the new file to the repository.

Now Alice realizes that she needs to add some more code the project:

```
> emacs interpreter.c
> cat interpreter.c
main() {
  int i;
}
> cvs commit -m "Added more code."
   cvs commit: Examining .
   alice@cvs.cs.arizona.edu's password: alice-pw
   Checking in interpreter.c;
   /cvs/cvs/cs453/cs453bd/ass1/interpreter.c,v  <--  interpreter.c
   new revision: 1.2; previous revision: 1.1
   done
```

OK, so what about Bob? Well, he decides he should also contribute to the project, so he checks out the source:

```
> cvs -d :ext:bob@cvs.cs.arizona.edu:/cvs/cvs/cs453/cs453bd checkout ass1
```

```
   bob@cvs.cs.arizona.edu's password: bob-pw
cvs server: Updating ass1
> cd ass1
> ls
CVS interpreter.c
> cat interpreter.c
main() {
  int i;
}
> emacs interpreter.c
> cat interpreter.c
main() {
  int i=5;
}
> cvs commit -m "Added more stuff to the project."
```

Alice has now gone back to her dorm-room where she wants to continue working on the project on her home computer. She has installed CVS and she has added

```
> setenv CVS_RSH ssh    # Or export CVS_RSH=ssh
```

to her `.cshrc` file to make sure that she runs this command every time. Now she can go ahead and check out the code again, this time on the home machine:

```
> cvs -d :ext:alice@cvs.cs.arizona.edu:/cvs/cvs/cs453/cs453bd checkout ass1
   alice@cvs.cs.arizona.edu's password: alice-pw
> cat ass1/interpreter.c
main() {
  int i=5;
}
```

Notice that she got the code that Bob checked in to CVS!

Alice can continue working on the code from home. When she's done for the day she uses the `commit` command to submit her changes to the cvs database.

Updating

The next day Bob is getting ready to work on the project again. In case Alice has made some changes to the code, he runs the `update` command:

```
> cvs update -d
```

Any files that have changed since the last time Bob worked on the project will be downloaded from the server. Bob makes his edits, then runs `commit` when he is done to upload the changes to the repository.

Deleting files

If Alice needs to delete a file she runs the CVS `rm` command:

```
> rm interpreter.c
> cvs rm interpreter.c
> cvs commit
```

Note that you have to delete the file before you can run the `cvs rm` command.

Summary

These are the most common CVS commands:

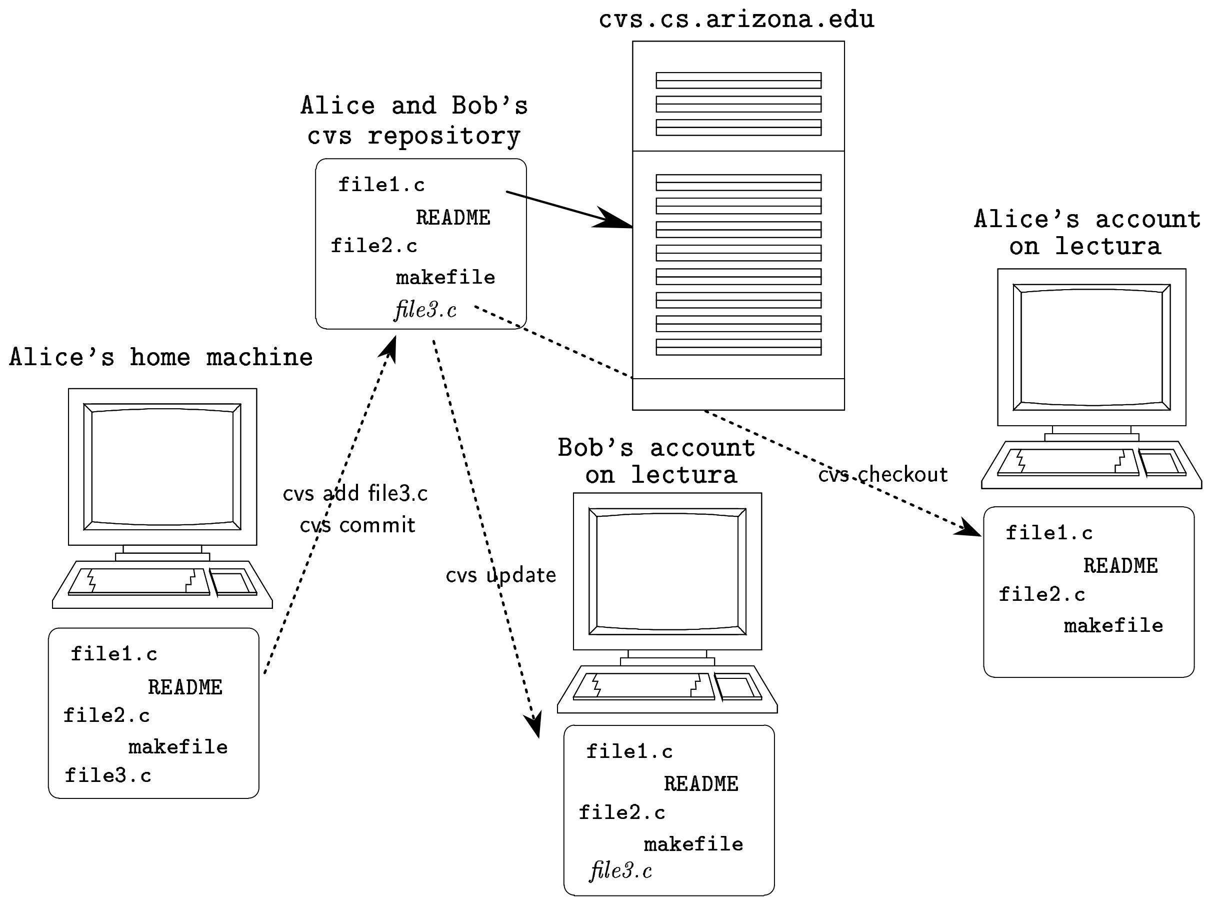
cvs add file Add a new file to the project. The file will not actually be uploaded to the repository until you run the `commit` command.

cvs rm file Remove a file from the project. The file will not actually be removed from the repository until you run the `commit` command.

cvs commit Update the repository with any changed files.

cvs update -d Download any changed files to your local machine.

The figure below describes a typical situation. Alice and Bob have three versions of the code checked out: two on their lectura accounts, and one version in Alice's home machine. Alice adds a new file `file3.c` and checks it in to the repository. To see the new file, Bob has to run the `update` command.


cvs.cs.arizona.edu
Alice and Bob's cvs repository
file1.c
README
file2.c
makefile
file3.c
Alice's account on lectura
file1.c
README
file2.c
makefile
Alice's home machine
file1.c
README
file2.c
makefile
file3.c
cvs add file3.c
cvs commit
cvs update
Bob's account on lectura
file1.c
README
file2.c
makefile
file3.c
cvs checkout