# Hacking SandMark

Christian Collberg

January 28, 2002

# Contents


**I Working with SandMark** **7**

**1 Introducing SandMark** **9**
1.1 Introduction . . . 9
Trying out the Watermarker . . . 10
1.2 Installing SandMark . . . 12
Building SandMark . . . 12
1.3 Scripting SandMark . . . 13

**2 The SandMark Code-base** **15**
2.1 SandMark Packages . . . 15
sandmark . . . 15
sandmark.gui . . . 15
sandmark.html . . . 16
sandmark.obfuscate . . . 16
sandmark.obfuscate.setfieldspublic . . . 16
sandmark.obfuscate.splitclass . . . 17
sandmark.obfuscate.util . . . 17
sandmark.optimise . . . 17
sandmark.util . . . 17
sandmark.util.exec . . . 18
sandmark.util.graph . . . 19
sandmark.util.graph.codec . . . 19
sandmark.util.javagen . . . 19
sandmark.watermark . . . 21
sandmark.watermark.CT . . . 21
sandmark.watermark.CT.embed . . . 21
sandmark.watermark.CT.encode . . . 21
sandmark.watermark.CT.encode.ir . . . 22
sandmark.watermark.CT.encode.ir2ir . . . 23
sandmark.watermark.CT.encode.storage . . . 23
sandmark.watermark.CT.recognize . . . 23
sandmark.watermark.CT.trace . . . 24
sandmark.watermark.constantstring . . . 24
sandmark.statistics . . . 24

**3 Extending SandMark** **25**
3.1 Adding an Obfuscator . . . 25
3.2 Adding a Watermarker . . . 26
3.3 Adding a Graph Codec . . . 28

3.4 Documentation . . . . . 30

**II SandMark Algorithms 31**

**4 The Collberg-Thomborson Watermarking Algorithm 33**
4.1 Introduction . . . . . 33
4.2 Annotation . . . . . 33
4.3 Tracing . . . . . 36
An Example . . . . . 37
Choosing `mark()`-Locations . . . . . 38
4.4 Embedding . . . . . 38
Building the Graph . . . . . 38
Splitting the Graph . . . . . 39
Generating Intermediate Code . . . . . 40
Generating Java Code . . . . . 42
Inserting the Java Code . . . . . 43
4.5 Recognition . . . . . 45
4.6 Passing Roots in Formal Parameters . . . . . 46

**5 The ConstantString Static Watermarking Algorithm 47**
5.1 Introduction . . . . . 47
5.2 Embedding . . . . . 47
5.3 Recognition . . . . . 47

**III Appendices 49**

**A Useful Tools 51**
A.1 Examining Java Classfiles . . . . . 51
javap . . . . . 51
Jasmin . . . . . 51
BCEL's Class Construction Kit . . . . . 52
BCEL's listclass . . . . . 52
BCEL's JustIce . . . . . 53
A.2 BCEL . . . . . 53
Getting started . . . . . 53
Editing a Method . . . . . 54
Editing Instructions . . . . . 55
Local variables . . . . . 55
Example 1: Creating a New Class . . . . . 56
Branches . . . . . 57
Exceptions . . . . . 58
Types . . . . . 58
Method Calls . . . . . 60
Example 2: Modifying an Existing Program . . . . . 60
Control Flow Graphs . . . . . 61
Miscellaneous . . . . . 61
A.3 BLOAT . . . . . 61
Introduction . . . . . 61
The Basics . . . . . 62
Editing Class Files . . . . . 62

Committing Edits . . . 65
Instruction Representation . . . 66
Advanced Features . . . 66

**B Working with the SandMark Code-base** **71**
B.1 Coding standard . . . 71
B.2 Using CVS . . . 72
Installing CVS . . . 72
Getting Started . . . 72
Checking out code . . . 73
Updating . . . 74
Deleting files . . . 74
Summary . . . 75

# 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

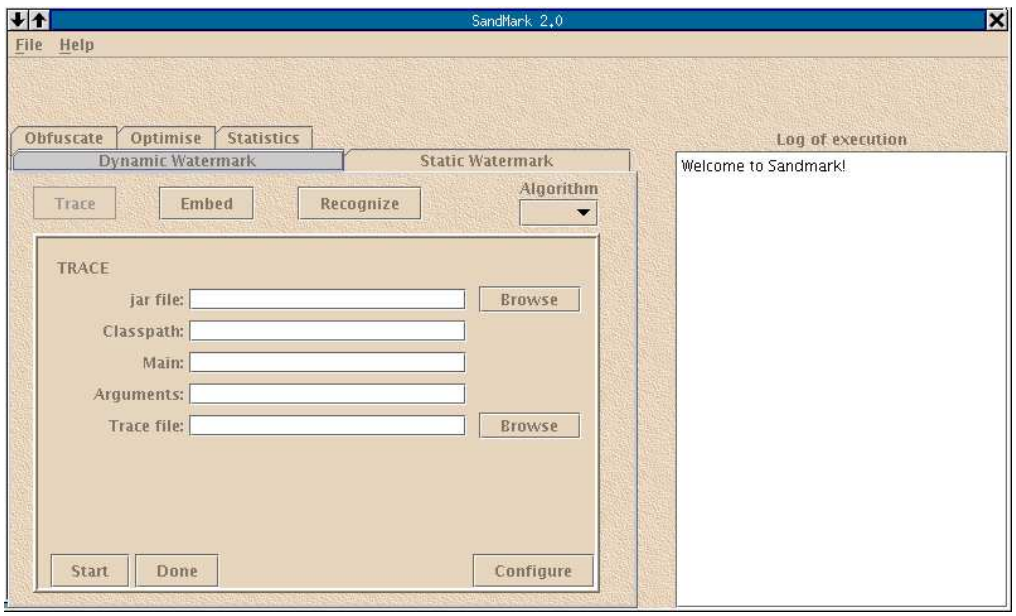
1. code obfuscation,

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:



## 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:

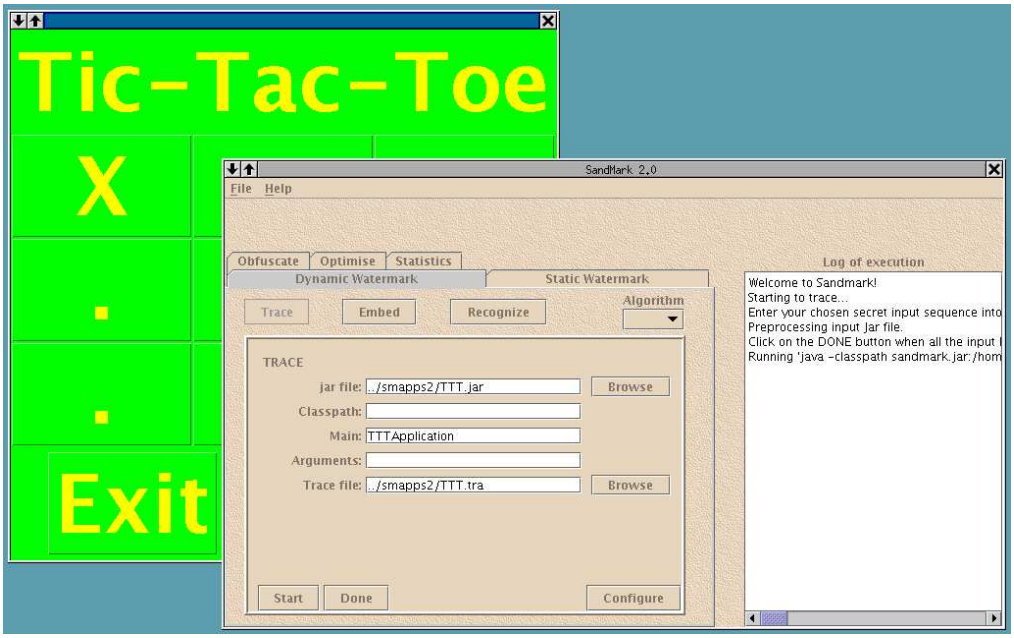
1. In the *trace* pane enter

   **Jar-file to watermark:** `TTT.jar`

   **Main class name:** `TTTApplication`

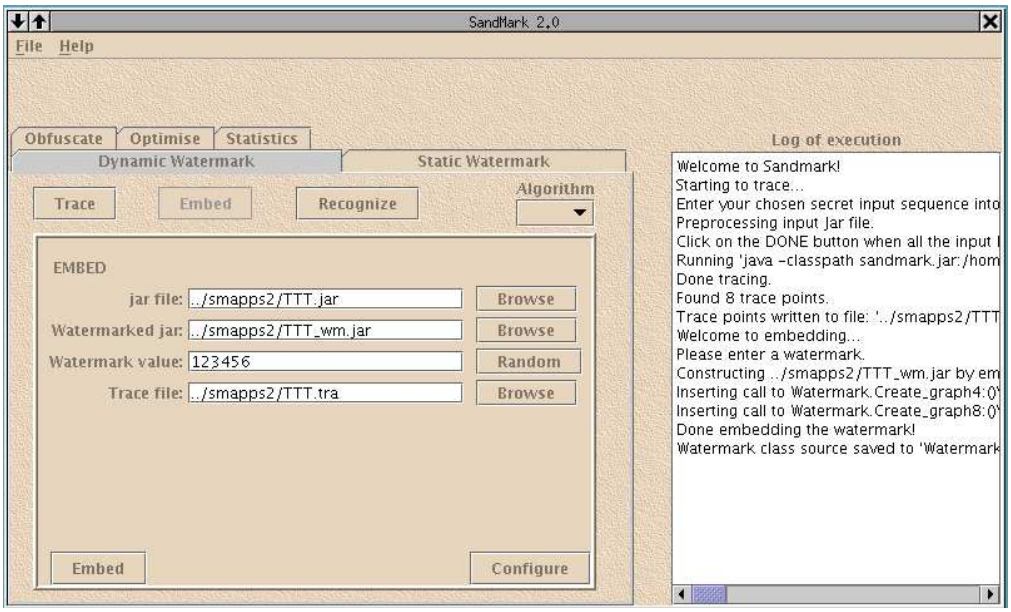
2. Hit START.

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:

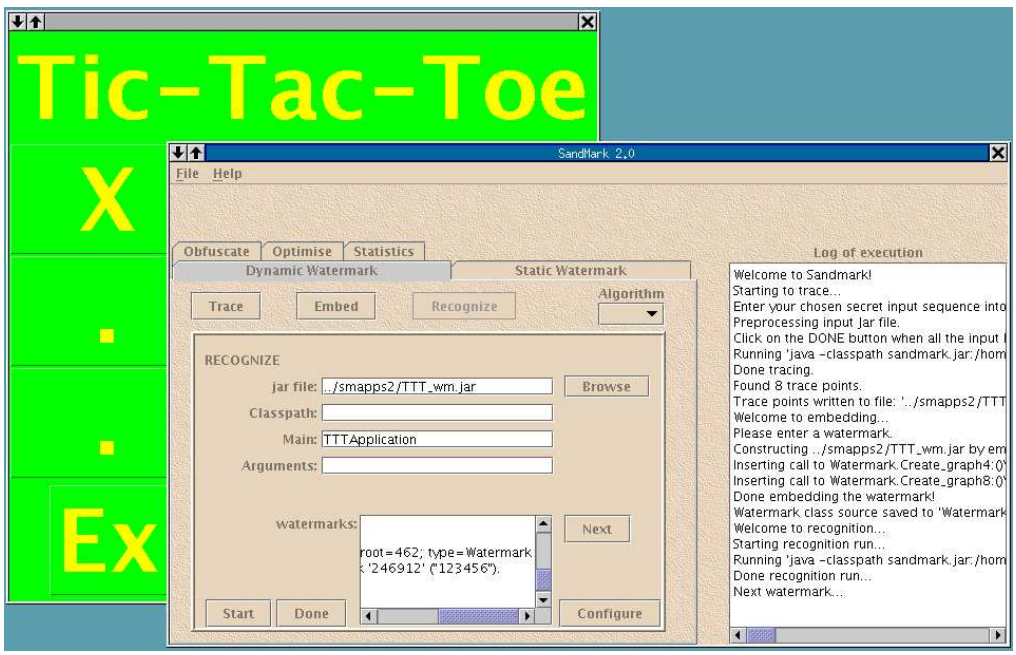



4. Hit DONE.

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




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:




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 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

```
> setenv CVS_RSH ssh
> cvs -d :pserver:anonymous@cvs.cs.arizona.edu:/cvs/wmark login
> cvs -d :pserver:anonymous@cvs.cs.arizona.edu:/cvs/wmark \
   checkout -P smark2 smextern smapps2 smbloat2 smbin
```

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

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

**smbloat2:** 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
> make again
> make again
```

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:

  **DWM_CT_AnnotatorClass:**

  **DWM_CT_Encode_NodeType:**

  **DWM_CT_Encode_ParentClass:**

  **DWM_CT_Encode_ClassName:**

  **DWM_CT_Encode_AvailableEdges:**

**DWM_CT_Encode_StoreWhat:**

**DWM_CT_Encode_StoreMethods:**

**DWM_CT_Encode_StoreLocation :**

**DWM_CT_Encode_ProtectionMethods:**

**DWM_CT_Encode_IndividualFixups:**

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

**DWM_CT_Encode_Components:**

**DWM_CT_Encode_Package:**

**DWM_MaxTracePoints:**

**DWM_CT_Encode_StoreLocation:** One of `formal` or `global`.

**DWM_CT_DumpIR:**

**DWM_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.EditedClass:** A class for maintaining a cache of classes and methods edited with `BCEL`.

**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 method within 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.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:

```
1   package sandmark.obfuscate;
2   public abstract class ClassObfuscator extends GeneralObfuscator {
3   protected ClassObfuscator(String label) {
4       super(label);
5   }
6   abstract public void apply(
7       sandmark.util.ClassFileCollection cfc, String classname)
8       throws Exception;
9   }
10  public String toString() {
11      return "ClassObfuscator(" + getLabel() + ")";
12  }
13  }
```

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

```
1   public class ReorderMethods extends sandmark.obfuscate.ClassObfuscator {
2      public ReorderMethods(String label) {
3         super(label);
4      }
5      public void apply(
6          sandmark.util.ClassFileCollection cfc, String classname) throws Exception {
```

```
7         // Your code goes here!
8      }
9   }
```

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:

```
1   package sandmark.watermark;
2
3   public abstract class StaticWatermarker
4      extends sandmark.watermark.GeneralWatermarker {
5
6   public StaticWatermarker() {}
7
8   /* Embed a watermark value into the program. The props argument
9    * holds at least the following properties:
10   *  <UL>
11   *     <LI> WM_Encode_Watermark: The watermark value to be embedded.
12   *     <LI> WM_Embed_JarInput: The name of the file to be watermarked.
13   *     <LI> WM_Embed_JarOutput: The name of the jar file to be constructed.
14   *     <LI> SWM_Embed_Key: The secret key.
15   *  </UL>
16   */
17  public abstract void embed(
18     java.util.Properties props)
19        throws sandmark.watermark.WatermarkingException,
20               java.io.IOException;
21
22
23  /* Return an iterator which generates the watermarks
24   * found in the program. The props argument
25   * holds at least the following properties:
26   *  <UL>
27   *     <LI> WM_Recognize_JarInput: The name of the file to be watermarked.
28   *     <LI> SWM_Recognize_Key: The secret key.
29   *  </UL>
```

```
30    */
31   public abstract java.util.Iterator recognize(
32      java.util.Properties props)
33         throws sandmark.watermark.WatermarkingException,
34                java.io.IOException;
35
36   }
```

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.

Implementing a dynamic watermarker is more complex, since you have to provide methods for running the application during tracing and recognition:

```
1   package sandmark.watermark;
2
3   public abstract class DynamicWatermarker
4      extends sandmark.watermark.GeneralWatermarker {
5
6   /**
7    * Start a tracing run of the program. Return an iterator
8    * object that will generate the trace points encountered
9    * by the program.
10   */
11  public abstract java.util.Iterator startTracing (
12      java.util.Properties props) throws sandmark.util.exec.TracingException;
13
14  /*
15   * Wait for the program being traced to complete.
16   */
17  public abstract void waitForTracingToComplete()
18     throws sandmark.util.exec.TracingException;
19
20  /**
21   * This routine should be called when the tracing run has
22   * completed. tracePoints is a vector of generated
23   * trace points generated by the iterator returned by
24   * startTracing.
25   */
26  public abstract void endTracing(
27      java.util.Properties props,
28      java.util.Vector tracePoints) throws sandmark.util.exec.TracingException;
29
30  /**
31   * Force the end to a tracing run of the program.
32   */
33  public abstract void stopTracing(
34      java.util.Properties props) throws sandmark.util.exec.TracingException;
35
```

```
36
37  /* Embed a watermark value into the program. The props argument
38   * holds at least the following properties:
39   *  <UL>
40   *     <LI> WM_Encode_Watermark: The watermark value to be embedded.
41   *     <LI> DWM_Embed_TraceInput: The name of the file containing trace data.
42   *     <LI> WM_Embed_JarInput: The name of the file to be watermarked.
43   *     <LI> WM_Embed_JarOutput: The name of the jar file to be constructed.
44   *     <LI> DWM_CT_Encode_ClassName: The name of the Java file that builds the watermark.
45   *  </UL>
46   */
47  public abstract void embed(
48     java.util.Properties props);
49
50  /**
51   * Start a recognition run of the program.
52   */
53  public abstract void startRecognition (
54      java.util.Properties props) throws sandmark.util.exec.TracingException;
55
56  /**
57   * Return an iterator object that will generate
58   * the watermarks found in the program.
59   */
60  public abstract java.util.Iterator watermarks();
61
62  /**
63   * Force the end to a tracing run of the program.
64   */
65  public abstract void stopRecognition(
66      java.util.Properties props) throws sandmark.util.exec.TracingException;
67
68  }
```

## 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`:

```
1  package sandmark.util.graph.codec;
2  public abstract class GraphCodec {
3     public java.math.BigInteger value = null;
4     public sandmark.util.graph.Graph graph = null;
5
6  /**
7   * Codecs should implement this method to convert
8   * the 'value' into 'graph'.
```

```
 9    */
10      abstract void encode();
11
12   /**
13    * Codecs should implement this method to convert
14    * the 'graph' into 'value'. Whenever the decoding
15    * failes (eg. because the graph has the wrong
16    * shape) the codec should simply throw an exception.
17    */
18      abstract void decode() throws sandmark.util.graph.codec.DecodeFailure;
19
20   /**
21    * Constructor to be used when encoding an integer into a graph.
22    *   @param value   The value to be encoded.
23    */
24      public GraphCodec (
25          java.math.BigInteger value) {
26          this.value = value;
27          encode();
28      }
29
30   /**
31    * Constructor to be used when decoding a graph to an integer.
32    *   @param graph   The graph to be decoded.
33    *   @param root    The root of the graph.
34    *   @param kidMap  An array of ints describing which field
35    *                  should represent the first child, the
36    *                  second child, etc.
37    */
38      public GraphCodec (
39          sandmark.util.graph.Graph graph,
40          int kidMap[]) throws sandmark.util.graph.codec.DecodeFailure {
41          this.graph = graph;
42          this.kidMap = kidMap;
43          decode();
44      }
45   }
```

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:

```
 1   package sandmark.util.graph.codec;
 2
 3   public class RadixGraph extends sandmark.util.graph.codec.GraphCodec {
 4
 5       public final static String FULLNAME  = "Radix Graph";
 6       public final static String SHORTNAME = "radix";
 7
 8   // Used when encoding.
 9   public RadixGraph (java.math.BigInteger value)  {
10       super(value);
```

```
11        this.fullName = FULLNAME;
12        this.shortName = SHORTNAME;
13    }
14
15    void encode() { ... }
16
17    // Used when decoding.
18       public RadixGraph (
19           sandmark.util.graph.Graph graph,
20           int kidMap[]) throws sandmark.util.graph.codec.DecodeFailure {
21           super(graph, kidMap);
22           this.fullName = FULLNAME;
23           this.shortName = SHORTNAME;
24       }
25       void decode() throws sandmark.util.graph.codec.DecodeFailure { ... }
26    }
```

## 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

```
1    sandmark.trace.Annotate.mark();
2    String S = ...;
3    sandmark.trace.Annotate.mark(S);
4    long L = ...;
5    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:

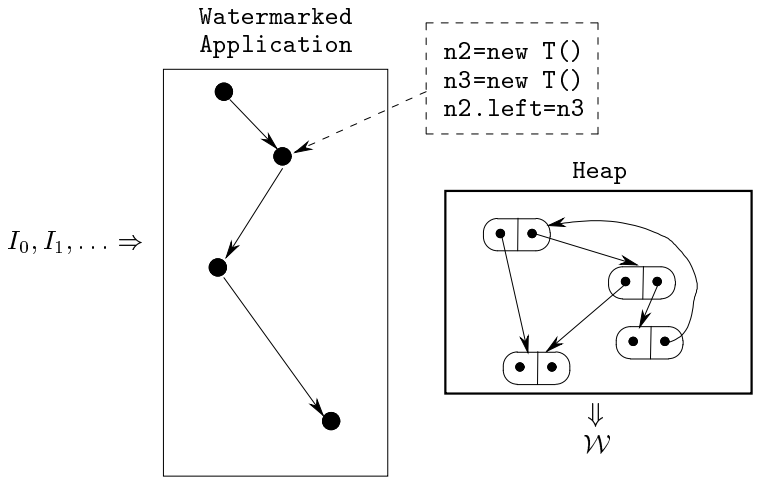



Figure 4.1: Overview of how the CT algorithm recognizes a watermark $\mathcal{W}$. At runtime the watermarked application will – given the special secret input key sequence $I_0, I_1, \ldots$ – traverse certain points in the program. At these points code has been inserted which builds a graph $G_{\mathcal{W}}$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

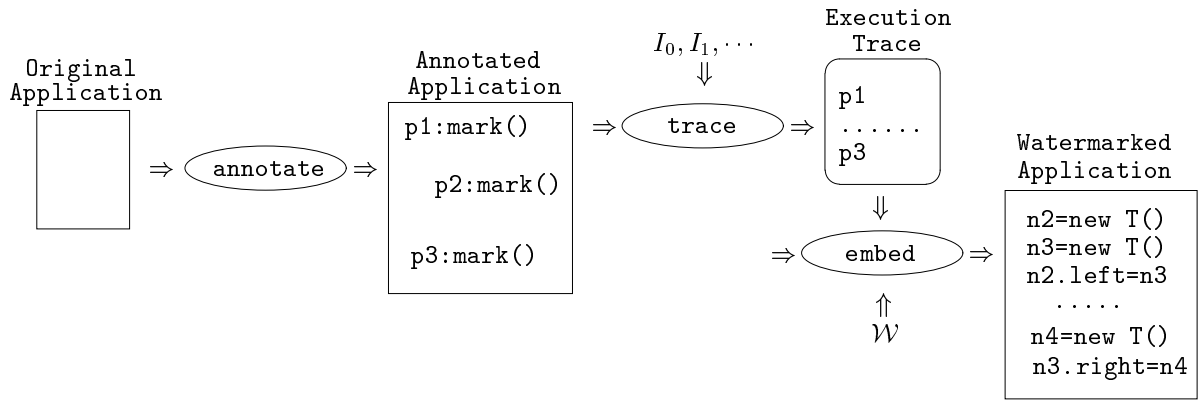



Figure 4.2: Overview of how the CT algorithm watermarks an application. First, the user adds *annotation points* (`mark()`-calls) to the application. These are locations where watermarking code may be inserted. Secondly, the application is run with a secret input sequence, $I_0, I_1, \ldots$ and the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

```
1   public class Simple {
2      static void P(String i) {
3         System.out.println("Hello " + i);
4      }
5      public static void main(String args[]) {
6         P(args[0]);
7      }
8   }
9
10  public class Simple_W {
11     static void P(String i) {
12        if (i.equals("World")) Watermark.Create_G4();
13        System.out.println("Hello " + i);
14     }
15     public static void main(String args[]) {
16        Watermark.Create_G2();
17        P(args[0]);
18     }
19  }
20
21  public class Watermark extends java.lang.Object {
22    public Watermark edge1;
23    public Watermark edge2;
24    public static java.util.Hashtable sm$hash = new java.util.Hashtable();
25    public static Watermark[] sm$array = new Watermark[4];
26
27    public static void Create_G2 () {
28       Watermark n3 = new Watermark();
29       Watermark n2 = new Watermark();
30       Watermark.sm$array[1] = n2;
31       n2.edge1 = n3;
32       n2.edge2 = n3;
33    }
34    public static void Create_G4 () {
35       Watermark n1 = new Watermark();
36       Watermark n4 = new Watermark();
37       Watermark.sm$hash.put(new java.lang.Integer(4), n4);
38       n4.edge1 = n1;
39       Watermark n2 = Watermark.sm$array[1];
40       n1.edge1 = n2;
41       Watermark n3 = (n2 != null)?n2.edge1:new Watermark();
42       n3.edge1 = n1;
43    }
44  }
```

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.

```
1       Node n1 = new Node();
2       Node n2 = new Node();
3       n1.edge = n2;
4       ...
```

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:

```
1       if (Math.random() < 0.5) {
2          ...
3          sandmark.trace.Annotate.mark();
4       }
```

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:

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

```
1   package sandmark.trace;
2   public class Annotator {
3      static String VALUE = "";
4      public static long stackFrameNumber=0;
5      public static void MARK(){}
6      public static void mark() {
7         long sm$stackID = sandmark.trace.Annotator.stackFrameNumber++;
8         VALUE = "----";
9         MARK();
10     }
11     public static void mark(String s) {
12        long sm$stackID = sandmark.trace.Annotator.stackFrameNumber++;
13        VALUE = "\"" + s + "\"";
14        MARK();
15     }
16     public static void mark(long v) {
17        long sm$stackID = sandmark.trace.Annotator.stackFrameNumber++;
18        VALUE = Long.toString(v);
19        MARK();
20     }
21  }
```

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:

```
1      public class SimpleA {
2         static void P(int i) {
3            sandmark.trace.Annotator.mark(6*i+9);
4         }
5         public static void main(String args[]) {
6            P(3);
7         }
8      }
```

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 $\langle value, location \rangle$ 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.

## 4.4 Embedding

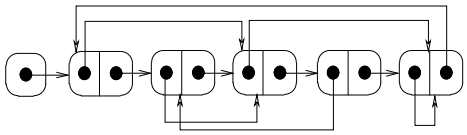
Once the application has been traced we can finally start embedding the watermark. The input to this phase is tracing information (as described in the previous section), a watermark $\mathcal{W}$ to be embedded, and a jar-file containing the classfiles in which to embed the mark. The embedding is divided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

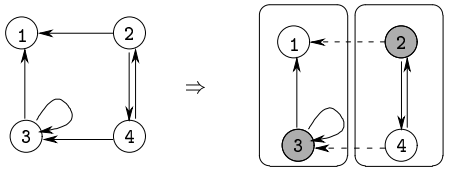
2. The splitting of $G$ should be done in such a way that each subgraph has a root, a special node from which all other nodes in the graph can be reached. This allows us to store only pointers to root nodes to prevent the garbage collector from collecting the subgraphs. (More about this later.)

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(`$n_2 \xrightarrow{\texttt{edge2}} n_3$`, `$G_2$`, `$G_2$`)`

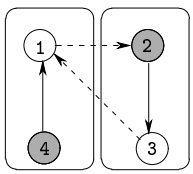



Figure 4.6: Two graph components $G_2$ and $G_4$. Components are named after their (shaded) root nodes.

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

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(`$G_i$`,` $G_i$`,` $n \xrightarrow{\texttt{edge}} m$`)`-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 $m \xrightarrow{\texttt{edge}} n$ 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

```
1       Watermark n2 = Watermark.sm$array[1];
2       Watermark n3 = (n2 != null)?n2.edge1:new Watermark();
3       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:

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

; or

```
1  public class Watermark extends java.lang.Object {
2    public Watermark edge1;
3    public Watermark edge2;
4    public static java.util.Hashtable sm$hash;
5    public static Watermark[] sm$array;
6
7    public static void Create_G2 () {
8       Watermark n3 = new Watermark();       // n3 = CreateNode(G2)
9       Watermark n2 = new Watermark();       // n2 = CreateNode(G2)
10      Watermark.sm$array[1] = n2;           // SaveNode(n2, G2, 'n2:Array/global')
11      n2.edge1 = n3;                        // AddEdge(G2, G2, n2-edge1->n3)
12      n2.edge2 = n3;                        // AddEdge(G2, G2, n2-edge2->n3)
13   }
14
15   public static void Create_G4 () {
16      Watermark n1 = new Watermark();       // n1 = CreateNode(G4)
17      Watermark n4 = new Watermark();       // n4 = CreateNode(G4)
18      Watermark.sm$hash.put(
19          new java.lang.Integer(4), n4);    // SaveNode(n4, G4, 'n4:Hash/global')
20      n4.edge1 = n1;                        // AddEdge(G4, G4, n4-edge1->n1)
21      Watermark n2 = Watermark.sm$array[1]; // n2 = LoadNode(G4, 'n2:Array/global')
22      n1.edge1 = n2;                        // AddEdge(G4, G2, n1-edge1->n2)
23      Watermark n3 =                        // n3 := FollowLink(G2, n2-edge1->n3)
24         (n2 != null)?n2.edge1:new Watermark();
25      n3.edge1 = n1;                        // AddEdge(G2, G4, n3-edge1->n1)
26   }
27 }
```

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`:

```
1     Watermark n2 = Watermark.sm$array[1];
2     if (n2 != null) {
3        Watermark n3 = n2.edge1;
4        n3.edge1 = n1;
5     }
```

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<br>1. `static java.util.Hashtable sm$hash = new java.util.Hashtable()`;<br>2. `static Watermark sm$array = new Watermark[`$m$`]`;<br>3. `static java.util.Vector sm$vec = new java.util.Vector(`$m$`); sm$vec.setSize(`$m$`)`;<br>4. `static Watermark sm$n1,sm$n2,...;`<br>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<br>1. `Watermark` $n$ `= (Watermark) sm$hash.get(new java.lang.Integer(`$k$`))`;<br>2. `Watermark` $n$ `= Watermark.sm$arr[`$k-1$`]`;<br>3. `Watermark` $n$ `= (Watermark) sm$vec.get(`$k-1$`)`;<br>4. `Watermark` $n$ `= Watermark.sm$n`$k$<br>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<br>1. `sm$hash.put(new java.lang.Integer(`$k$`), `$n$`)`;<br>2. `Watermark.sm$arr[`$k-1$`] =` $n$;<br>3. `(Watermark) sm$vec.set(`$k-1$`,` $n$`)`;<br>4. `Watermark.sm$n`$k$ `=` $n$<br>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

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

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

```
1       if (expr==value)
2          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:

```
1   package java.lang;
2   public class Object {
3       public static long objCount = 0;
4       public long allocTime;
5       public Object() {allocTime = objCount++;}
6   }
```

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):

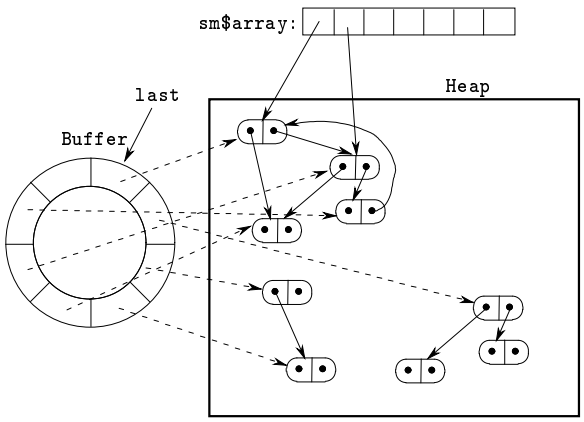



Figure 4.8: A view of memory during recognition. A circular linked buffer holds the last allocated objects. The recognizer examines the objects in reverse allocation order and extracts the subgraph reachable from each object. This is decoded into the watermark.

```
1     static int kidMaps[][] = {{1,2},{2,1},{1,3},{3,1},{2,3},{3,1}};
2     for every object O on sandmark.util.CircularBuffer, starting with the last allocated do {
3         G := the graph consisting of the nodes reachable from O;
4         for every graph decoder D do {
5            for every kidmap K do {
6               W := decode G using D, assuming K;
7               print W;
8            }
9         }
10    }
```

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:

```
1      String jarInput  = props.getProperty("WM_Embed_JarInput");
2      String jarOutput = props.getProperty("WM_Embed_JarOutput");
3      String watermark = props.getProperty("WM_Encode_Watermark");
4
5      sandmark.util.ClassFileCollection cfc =
6         new sandmark.util.ClassFileCollection(jarInput);
7
8      java.util.Iterator classes = cfc.classes();
9      String className = (String) classes.next();
10     de.fub.bytecode.classfile.JavaClass origClass = cfc.getClass(className);
11     de.fub.bytecode.generic.ClassGen cg = new de.fub.bytecode.generic.ClassGen(origClass);
12
13     de.fub.bytecode.generic.ConstantPoolGen cp = cg.getConstantPool();
14     int stringIndex = cp.addString("sm$watermark" + "=" + watermark);
15
16     de.fub.bytecode.classfile.JavaClass newClass = cg.getJavaClass();
17     cfc.addClass(newClass);
18     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"`:

```
1      String jarInput  = props.getProperty("WM_Recognize_JarInput");
2      sandmark.util.ClassFileCollection cfc =
3          new sandmark.util.ClassFileCollection(jarInput);
4      java.util.Iterator classes = cfc.classes();
5      while (classes.hasNext()) {
6         String className = (String) classes.next();
```

```
7         de.fub.bytecode.classfile.JavaClass clazz = cfc.getClass(className);
8         de.fub.bytecode.classfile.ConstantPool cp = clazz.getConstantPool();
9         for (int i=0; i<cp.getLength(); i++) {
10           de.fub.bytecode.classfile.Constant c = cp.getConstant(i);
11           if (c instanceof de.fub.bytecode.classfile.ConstantString) {
12               de.fub.bytecode.classfile.ConstantString s =
13                  (de.fub.bytecode.classfile.ConstantString) c;
14               String v = (String)s.getConstantValue(cp);
15               if (v.startsWith("sm$watermark")) {
16                  String w = v.substring("sm$watermark".length()+1);
17                  // w is the watermark
18               }
19           }
20        }
21     }
22     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:

```
1       .class public HelloWorld
2       .super java/lang/Object
3
4       ;
5       ; standard initializer (calls java.lang.Object's initializer)
6       ;
7       .method public <init>()V
8          aload_0
9          invokenonvirtual java/lang/Object/<init>()V
10         return
11      .end method
12
13      ;
14      ; main() - prints out Hello World
15      ;
16      .method public static main([Ljava/lang/String;)V
17         .limit stack 2   ; up to two items can be pushed
18
19         ; push System.out onto the stack
20         getstatic java/lang/System/out Ljava/io/PrintStream;
21
22         ; push a string onto the stack
23         ldc "Hello World!"
24
25         ; call the PrintStream.println() method.
26         invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
27
28         ; done
29         return
30      .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 BCEL

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` (formerly `JavaClass`) allows you to load a class, iterate through the methods and fields, change methods, add new methods and fields, etc. `BCEL` also makes it easy to create new classes from scratch.

`BCEL` is here: `http://jakarta.apache.org/bcel/index.html`. There is a manual, and an on-line API description at `http://bcel.sourceforge.net/docs/index.html`. Unfortunately, neither are particularly informative.

SandMark contains a lot of `BCEL` code that might be worth studying:

- `sandmark.util.javagen` is a package for building up bytecode classes from scratch. Essentially, it allows you to build a Java AST (abstract syntax tree) and compile it down to bytecode. `javagen` is built on top of `BCEL`.
- `sandmark.util.BCEL` contains `BCEL` utility routines. The intention is that this class should grow in the future.
- The package `sandmark.watermark.CT.embed` contains several classes that manipulate Java classfiles using `BCEL`.

### Getting started

To open a class file for editing you do the following:

```
1       sandmark.util.ClassFileCollection cfc = ...;
2
3       cfc = new sandmark.util.ClassFileCollection(jarFile);
4       de.fub.bytecode.classfile.JavaClass jc = cfc.getClass(className);
5       de.fub.bytecode.generic.ClassGen cg = new de.fub.bytecode.generic.ClassGen(jc);
6       de.fub.bytecode.generic.ConstantPoolGen cp =
7          new de.fub.bytecode.generic.ConstantPoolGen(jc.getConstantPool());
```

`jarFile` is the jar-file that contains the class file and `className` is the name of the class (possibly with a package prefix). `sandmark.util.ClassFileCollection` is the repository used within SandMark for loading and saving class files.

A `de.fub.bytecode.classfile.JavaClass`-object represents the parsed class file. You can query this object to get any information you need about the class, for example its methods, fields, code, super-class, etc.

A `de.fub.bytecode.generic.ClassGen`-object represents a class that can be edited. You can add methods and fields, you can modify existing methods, etc.

A `de.fub.bytecode.generic.ConstantPoolGen`-object represents a constant pool to which new constants can be added.

When you are finished editing the class you must close it and return it to the repository maintained by `sandmark.util.ClassFileCollection`:

```
8       de.fub.bytecode.classfile.JavaClass jc1 = cg.getJavaClass();
9       jc1.setConstantPool(cp.getFinalConstantPool());
10      cfc.addClass(jc1);
```

### Editing a Method

`de.fub.bytecode.classfile.Method` represents a method that has been read from a class file, while `de.fub.bytecode.generic.MethodGen` is the `BCEL` class that represents a method being edited:

```
1       de.fub.bytecode.generic.ConstantPoolGen cp = ...;
2       de.fub.bytecode.generic.ClassGen cg = ...;
3
4       de.fub.bytecode.classfile.Method method =
5          cg.containsMethod(methodName, methodSignature);
6       de.fub.bytecode.generic.MethodGen mg =
7          new de.fub.bytecode.generic.MethodGen(method, className, cp);
```

Note that to open a method with `containsMethod` you need to know the class in which it is declared, as well as its name as well as its signature. The reason is that a Java class can have several methods with the same name (they're overloaded).

Alternatively, SandMark also provides a class `sandmark.util.EditedClass.open` than can be used to maintain a cache of classes and methods that have been opened by `BCEL`:

```
1        sandmark.util.ClassFileCollection cfc = ...;
2        sandmark.util.EditedClass ec = sandmark.util.EditedClass.open(cfc, className);
3       java.util.Iterator methods = ec.methods();
4       while (methods.hasNext()){
5           sandmark.util.MethodID m = (sandmark.util.MethodID) methods.next();
6           de.fub.bytecode.generic.MethodGen mg = ec.openMethod(m);
7           ....
8       }
```

A `sandmark.util.MethodID`-object represents a method within a Java application. It is essentially a tuple

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

### Editing Instructions

Once you have opened a method for editing you can get its list of instructions:

```
1      de.fub.bytecode.generic.MethodGen mg = ...;
2
3      de.fub.bytecode.generic.InstructionList il = mg.getInstructionList();
4      de.fub.bytecode.generic.InstructionHandle[] ihs = il.getInstructionHandles();
5      for(int i=0; i < ihs.length; i++) {
6         de.fub.bytecode.generic.InstructionHandle ih = ihs[i];
7         de.fub.bytecode.generic.Instruction instr = ih.getInstruction();
8            ....
9      }
```

`de.fub.bytecode.generic.InstructionLists` can be manipulated by appending or inserting new instruction, deleting instructions, etc.

All bytecode instructions in `BCEL` are represented by their own class. `de.fub.bytecode.generic.IADD` is the class that represents integer addition, for example. This allows us to use `instanceof` to classify instructions:

```
1      de.fub.bytecode.generic.Instruction instr = ...;
2      de.fub.bytecode.generic.InstructionHandle ih = ...;
3
4      if (instr instanceof de.fub.bytecode.generic.INVOKESTATIC) {
5         de.fub.bytecode.generic.INVOKESTATIC call =
6           (de.fub.bytecode.generic.INVOKESTATIC) instr;
7         String className  = call.getClassName(ec.cp);
8         String methodName = call.getMethodName(ec.cp);
9         String methodSig  = call.getSignature(ec.cp);
10
11        ih.setInstruction(new de.fub.bytecode.generic.NOP());
12     }
```

`setInstruction` is used to replace an original instruction with a new one.

### Local variables

Local variables are quite complicated in Java bytecode. A Java method can have a maximum of 256 local variables. Note, however, that

- In a virtual (non-static) method local variable zero ("slot #0") is `this`.
- Formal arguments take up the first slots. In other words, in a virtual method with 2 formal parameters, slot #0 is `this`, slot #1 is the first formal, slot #2 is the second formal, and any local variables are slots #3,#4,...,#255.
- Local variables can be reused within a method. For example, consider the following method:

```
void P() {
   int x= 5;
```

```
        System.out.println(x);
        float y=5.6;
        System.out.println(y);
    }
```

A clever compiler will allocate both `x` and `y` to the same slot, since they have non-overlapping lifetimes.

For this reason, `BCEL` requires that we indicate where a new local variable is accessible:

```
1   de.fub.bytecode.generic.Type T = ...;
2
3   de.fub.bytecode.generic.LocalVariableGen lg =
4      mg.addLocalVariable(localName, T, null, null);
5   int localIndex = lg.getIndex();
6
7   // push something here
8   de.fub.bytecode.generic.Instruction store =
9      new de.fub.bytecode.generic.ASTORE(localIndex);
10
11  de.fub.bytecode.generic.InstructionHandle start = ...;
12  lg.setStart(start);
```

`de.fub.bytecode.generic.LocalVariableGen` creates a new local variable. The two `null` arguments are the locations within the method where we want the variable to be visible. We start these locations out as `null` (unknown) and fill them in using the `setStart` method. It takes an `de.fub.bytecode.generic.InstructionHandle` as input, namely the first instruction where the variable should be visible.

### Example 1: Creating a New Class

Here's a longer example that shows how you create a class from scratch.

We start by creating a new `de.fub.bytecode.generic.ClassGen` object representing the class. We provide the name of the class, the name of the parent class, the interfaces it implements, and its access flags:

```
1   int access_flags = de.fub.bytecode.Constants.ACC_PUBLIC;
2   String class_name = "MyClass";
3   String file_name = "MyClass.java";
4   String super_class_name = "java.lang.Object";
5   String[] interfaces = {};
6
7   de.fub.bytecode.generic.ClassGen cg =
8      new de.fub.bytecode.generic.ClassGen(
9         class_name, super_class_name, file_name,
10        access_flags, interfaces);
11  de.fub.bytecode.generic.ConstantPoolGen cp = cg.getConstantPool();
```

We can then add a field `field1` of type `int` to the new class:

```
12  de.fub.bytecode.generic.Type field_type =
13     de.fub.bytecode.generic.Type.INT;
14  String field_name = "field1";
15
16  de.fub.bytecode.generic.FieldGen fg =
17     new de.fub.bytecode.generic.FieldGen(
18        access_flags, field_type, field_name, cp);
19  cg.addField(fg.getField());
```

Next, we create a method `method1`:

```
20      int method_access_flags =
21         de.fub.bytecode.Constants.ACC_PUBLIC |
22         de.fub.bytecode.Constants.ACC_STATIC;
23
24      de.fub.bytecode.generic.Type return_type =
25         de.fub.bytecode.generic.Type.VOID;
26
27      de.fub.bytecode.generic.Type[] arg_types =
28         de.fub.bytecode.generic.Type.NO_ARGS;
29      String[] arg_names = {};
30
31      String method_name = "method1";
32      String class_name = "MyClass";
33
34      de.fub.bytecode.generic.InstructionList il =
35         new de.fub.bytecode.generic.InstructionList();
36      il.append(de.fub.bytecode.generic.InstructionConstants.RETURN);
37
38      de.fub.bytecode.generic.MethodGen mg =
39         new de.fub.bytecode.generic.MethodGen(
40            method_access_flags, return_type,  arg_types,
41            arg_names, method_name, class_name, il, cp);
42
43      mg.setMaxStack();
44      cg.addMethod(mg.getMethod());
```

The method has no arguments, returns `void`, and contains only one instruction, a `return`.

Note the call to `mg.setMaxStack()`. With every method in a class file is stored the maximum stack-size is needed to execute the method. For example, a method whose body consists of the instructions ⟨`iconst_1`, `iconst_2`, `iadd`, `pop`⟩ (i.e. compute $1 + 2$ and throw away the result) will have max-stack set to two. We can either compute this ourselves or let `BCEL`'s `mg.setMaxStack()` do it for us.

### Branches

As is always the case with branches, we need to be able to handle forward jumps. The typical way of accomplishing this is to create a branch with unspecified target:

```
1       de.fub.bytecode.generic.InstructionList il = ...;
2
3       de.fub.bytecode.generic.IFNULL branch =
4          new de.fub.bytecode.generic.IFNULL(null);
5       il.append(branch);
6       ...
```

When we have gotten to the location we want to jump to we add a (bogus) `NOP` instruction which will serve as our branch target. The `append` instruction returns a handle to the `NOP` and we use this handle to set the target of the branch instruction:

```
7       de.fub.bytecode.generic.InstructionHandle h =
8          il.append(new de.fub.bytecode.generic.NOP());
9       branch.setTarget(h);
```

### Exceptions

Here is the generic code for building a `try-catch`-block:

```
1      de.fub.bytecode.generic.InstructionList il = ...;
2      de.fub.bytecode.generic.MethodGen mg = ...;
3      String exception = "java.lang.Exception";
4
5      de.fub.bytecode.generic.InstructionHandle start_pc =
6         il.append(new de.fub.bytecode.generic.NOP());
7
8      // The code that builds the try-body goes here.
9
10     // Add code to jump out of the try-block when
11     // no exception was thrown.
12     de.fub.bytecode.generic.GOTO branch =
13        new de.fub.bytecode.generic.GOTO(null);
14     de.fub.bytecode.generic.InstructionHandle end_pc =
15          il.append(branch);
16
17     // Pop the exception off the stack when entering the catch block.
18     de.fub.bytecode.generic.InstructionHandle handler_pc =
19         il.append(new de.fub.bytecode.generic.POP());
20
21     // The code that builds the catch-body goes here.
22
23     // Add a NOP after the exception handler. This is
24     // where we will jump after we're through with the
25     // try-block.
26     de.fub.bytecode.generic.InstructionHandle next_pc =
27         il.append(new de.fub.bytecode.generic.NOP());
28     branch.setTarget(next_pc);
29
30     de.fub.bytecode.generic.ObjectType catch_type =
31        new de.fub.bytecode.generic.ObjectType(exception);
32
33     de.fub.bytecode.generic.CodeExceptionGen eg =
34        mg.addExceptionHandler(start_pc, end_pc, handler_pc, catch_type);
```

`start_pc` and `end_pc` hold the beginning and the end of the `try`-body, respectively. `handler_pc` holds the address of the beginning of the `catch`-block.

### Types

Types (such as method signatures) are defined by the class `de.fub.bytecode.generic.Type`:

```
1      de.fub.bytecode.generic.Type return_type =
2         de.fub.bytecode.generic.Type.VOID;
3      de.fub.bytecode.generic.Type[] arg_types   =
4         new de.fub.bytecode.generic.Type[] {
5            new de.fub.bytecode.generic.ArrayType(
6               de.fub.bytecode.generic.Type.STRING, 1)
7         };
```

A source-level Java type is encoded into a classfile type descriptor using the following definitions:

| BaseType | Type | Interpretation |
|---|---|---|
| **B** | byte | signed 8-bit integer |
| **C** | char | Unicode character |
| **D** | double | 64-bit floating point value |
| **F** | float | 32-bit floating point value |
| **I** | int | 32-bit integer |
| **J** | long | 64-bit integer |
| **L**<*classname*>**;** | reference | an instance of class <*classname*> |
| **S** | short | signed 16-bit integer |
| **Z** | boolean | `true` pr `false` |
| [ | reference | one array dimension |
| **V** | | void |
| (*BaseType**)*BaseType* | | method descriptor |

`BCEL` has quite a number of methods that convert back and forth between Java source format (such as `"java.lang.Object[]"`), bytecode format (such as `"[Ljava/lang/Object;"`), and `BCEL`'s internal format (`de.fub.bytecode.generic.Type`). Unfortunately, these are by and large poorly documented. We will describe some of the more common methods here.

`de.fub.bytecode.generic.Type.getType` converts from Java bytecode format to Java source format. In other words, the following code

```
1       String S = "[Ljava/lang/Object;";
2       de.fub.bytecode.generic.Type T =
3          de.fub.bytecode.generic.Type.getType(S);
4       System.out.println(T);
```

prints out ⌜`java.lang.Object[]`⌝.

The method `de.fub.bytecode.classfile.Utility.getSignature` converts a type in Java bytecode format to Java source format. The code

```
1       String type = "java.lang.Object[]";
2       String S = de.fub.bytecode.classfile.Utility.getSignature(type);
3       System.out.println(S);
```

prints ⌜`[Ljava/lang/Object;`⌝.

The methods `de.fub.bytecode.generic.Type.getArgumentTypes` and `de.fub.bytecode.generic.Type.getReturnType` take a type in Java bytecode format as argument and extract the array of argument types and the return type, respectively:

```
1       String S = "(Ljava/lang/String;I)V;";
2       de.fub.bytecode.generic.Type[] arg_types =
3          de.fub.bytecode.generic.Type.getArgumentTypes(S);
4       de.fub.bytecode.generic.Type return_type =
5          de.fub.bytecode.generic.Type.getReturnType(S);
6       String M = de.fub.bytecode.generic.Type.getMethodSignature(return_type, arg_types);
7       System.out.println(M);
```

`de.fub.bytecode.generic.Type.getMethodSignature` converts the return and argument types back to a Java bytecode type string.

The method `de.fub.bytecode.generic.Type.getSignature` converts a `BCEL` type to the equivalent Java bytecode signature. This code

```
1       de.fub.bytecode.generic.Type T =
2          de.fub.bytecode.generic.Type.STRINGBUFFER;
3       String M = T.getSignature();
4       System.out.println(M);
```

will print out ⌜Ljava/lang/StringBuffer;⌝.

As a convenience, SandMark provides the method `sandmark.util.javagen.Java.typeToByteCode`, that converts a `type` in Java source format to a `BCEL de.fub.bytecode.generic.Type`:

```
1       String S = de.fub.bytecode.classfile.Utility.getSignature(type);
2       return de.fub.bytecode.generic.Type.getType(S);
```

### Method Calls

To make a static method call we push the arguments on the stack, create a constant pool reference to the method, and then generate an `INVOKESTATIC` opcode that performs the actual call:

```
1       String className  = ...;
2       String methodName = ...;
3       String signature  = ...;
4
5       de.fub.bytecode.generic.InstructionList il = ...;
6       de.fub.bytecode.generic.ConstantPoolGen cp = ...;
7
8       // Generate code that pushes the actual arguments of the call.
9
10      int index = cp.addMethodref(className,methodName,signature);
11      de.fub.bytecode.generic.INVOKESTATIC call =
12          new de.fub.bytecode.generic.INVOKESTATIC(index);
13      il.append(call);
```

Making a virtual call is similar, except that we need an object to make the call through. This object reference is pushed on the stack prior to the arguments:

```
1       String className  = ...;
2       String methodName = ...;
3       String signature  = ...;
4
5       de.fub.bytecode.generic.InstructionList il = ...;
6       de.fub.bytecode.generic.ConstantPoolGen cp = ...;
7
8       // Generate code that pushes the object on the stack.
9
10      // Generate code that pushes the actual arguments of the call.
11
12      int index = cp.addMethodref(className,methodName,signature);
13      de.fub.bytecode.generic.INVOKEVIRTUAL s =
14          new de.fub.bytecode.generic.INVOKEVIRTUAL(index);
15      il.append(call);
```

### Example 2: Modifying an Existing Program

Consider the following program:

```
1      public class C {
2         public static void main(String args[]) {
3            int a = 5;
4            int b = 6;
5            int c = a + b;
6            System.out.println(c);
7         }
8      }
```

### Control Flow Graphs

The `JustIce` verifier that is built on top of `BCEL` has functions that construct control flow graphs for methods. Unfortunately, they don't construct real basic blocks: each instruction is in a block by itself.

### Miscellaneous

`sandmark.util.javagen.Java.accessFlagsToByteCode` takes a list of string modifiers as argument and returns the corresponding access flag:

```
1      String[] attributes = {"public","static"};
2      int access_flags =
3         sandmark.util.javagen.Java.accessFlagsToByteCode(attributes);
```

## A.3 BLOAT

### Introduction

#### What is BLOAT?

BLOAT is a Java optimizer written entirely in Java. By optimizing Java byte code, code improvements can occur regardless of the compiler that compiled the byte code or the virtual machine on which the byte code is run. BLOAT performs many traditional program optimizations such as constant/copy propagation, constant folding and algebraic simplification, dead code elimination, and peephole optimizations. Additionally, it performs partial redundancy elimination of arithmetic and field access paths.

Beyond this BLOAT can also be used for class creation as well as method creation. In combination with byte code level editing BLOAT is can also be used as a framework for Java Class File obfuscation.

#### What to expect?

This how-to is in no way related to the creators of BLOAT. It was written by Richard Smith, a computer science undergraduate at the University of Arizona, to aid in research efforts by Christian Collberg. This guide does not demonstrate the optimization capabilities of BLOAT, but does demonstrate its ability to be used as a framework for byte code obfuscation. From this knowledge you can easily infer how to do optimizations. For further questions about this how-to email: `rsmith@cs.arionza.edu`. You may also look at The BLOAT Book which can be found at `http://www.cs.purdue.edu/s3/projects/bloat`. At the time of writing this how-to this book referred to BLOAT-0.8a which lacks many of the features of BLOAT-1.0.

A number of examples of how BLOAT is used can be found in the cvs directory `smbloat2`. Section 1.2 describes how to check out this directory.

## The Basics

### Architecture

BLOAT is a very complicated beast. This will be a brief overview, but will be enough to begin with. The first you must understand is the concept of a context. A context supplies a means of loading and editing classes. All contexts implement the `EditorContext`. This interface supplies the generic means of committing edits, editing classes, and a utility to load classes. There are five major contexts:

1. `BloatContext`,
2. `InlineContext`,
3. `EditorContext`,
4. `PersistentBloatContext`, and
5. `CachingBloatContext`.

The `InlineContext` and the `EditorContext` are both interfaces. The `InlineContext` and the `EditorClass` are both implemented by the abstract class `BloatContext`. The `PersistentBloatContext` and the `CachingBloatContext` are both subclasses of the `BloatContext`. The two contexts that you should worry about are the `PersistentBloatContext` and the `CachingBloatContext`. These contexts are used for the same purpose, which is to act as a repository for all of the edits made to java class files. The difference between the two contexts is that the `PersistentBloatContext` keeps every peace of data relevant to your work, the `CachingBloatContext` on the other hand manages your edits so that when a editor is no longer dirty it drops the instance of the editor.

Editors consist of the `ClassEditor`, `MethodEditor`, and `FieldEditor`. The names of these class should give evidence enough of their subjects of manipulation. It is through these editors that you will make actual changes to byte code. It should be noted that the `ClassEditor` constructor takes a `BloatContext` and all other editors accepts a `ClassEditor`. In some ways this is the use of a decorator pattern or wrapper methodology.

## Editing Class Files

### Where to Start?

BLOAT seems to have endless constructors that depend on other objects which in turn depend on other objects. In truth unless you know where to begin you will wrap yourself around the API for a few hours trying to find the object you need create in order to create the object you want to create.

The first thing that you need in BLOAT is a `ClassFileLoader`. Unlike most classes in BLOAT the constructor for this class takes no arguments. The `ClassFileLoader` provides you with tools to manipulate the system CLASSPATH, open zip/jar archives, and create new classes. Overall these tools will not be useful to you. They can be used as opposed to the process I am about to demonstrate to you. However, if you chose to do so you will not be using contexts as the central point of your BLOAT manipulations which was fore mentioned as the core of the BLOAT architecture.

After creating a `ClassFileLoader` the next step is to create either a `PersistentBloatContext` or a `CachingBloatContext`. The choice between these two contexts depends on the resources available to you. In most case you should chose the `PersistentBloatContext` as memory is cheap.

Once you have a BLOAT context you can load the class files you wish to edit. This is done by using the `loadClass` method of the BLOAT context. You could also use the same method in the `ClassFileLoader`, however again this does not take advantage of the context architecture.

After loading the class file the `loadClass` method returns a `ClassInfo` object. This is really an interface implemented by the `ClassFile` object so you will want to cast the `ClassInfo` object to a `ClassFile` to take advantage of some more advanced features found in the `ClassFile` object. The `ClassFile` object, as one might think, contains all of the relevant information about the class you load. This includes information

```
1   EDU.purdue.cs.bloat.file.ClassFileLoader classFileLoader = new
2      EDU.purdue.cs.bloat.file.ClassFileLoader();
3
4   EDU.purdue.cs.bloat.context.PersistentBloatContext bloatContext =
5      new EDU.purdue.cs.bloat.context.PersistentBloatContext(
6          classFile.loader());
7
8   String className = "ClassFileToTest";
9   EDU.purdue.cs.bloat.reflect.ClassInfo info = null;
10
11  try {
12     info = bloatContext.loadClass(className);
13  } catch (ClassNotFoundException ex) {
14     System.err.println("Couldn't find class: " +ex.getMessage());
15     System.exit(1);
16  }
17
18  EDU.purdue.cs.bloat.file.ClassFile classFile =
19        (EDU.purdue.cs.bloat.file.ClassFile)info;
```

Figure A.1: Loading a class file.

pertaining to fields and methods as well as more byte code specific information. Figure A.1 will illustrate this process so far.

**The `ClassEditor`**

Once you created a `ClassFile` object you can begin byte-code manipulation. The next step is to create a `ClassEditor`. There are two ways to go about this, create a `ClassEditor` from scratch or create a `ClassEditor` from an existing `ClassFile`.

To create a `ClassEditor` from an existing `ClassFile` simply use the method `editClass` supplied by the `BloatContext`. This method returns a `ClassEditor` for the `ClassFile` passed as a parameter:

```
1   EDU.purdue.cs.bloat.editor.ClassEditor classEditor =
2                bloatContext.editClass(classFile);
```

To create a `ClassEditor` from scratch is a little different. BLOAT supplies a constructor for the `ClassEditor` to do so however the API advises you not to use this. Instead you should use the method `newClass` provided by the `BloatContext`:

```
newClass(
   intmodifiers,
   java.lang.StringclassName,
   TypesuperType,
   Type[]interfaces);
```

The `newClass` method takes a modifier, a String, a Type, and a Type array. The modifier specifies whether the class will be public, private, or protected. The correct parameter to use for the modifier is a constant from the modifier interface found in the `EDU.purdue.cs.bloat.reflect` package:

```
EDU.purdue.cs.bloat.reflect.Modifier.PUBLIC
EDU.purdue.cs.bloat.reflect.Modifier.PRIVATE
EDU.purdue.cs.bloat.reflect.Modifier.PROTECTED
```

The String parameter specifies the name of the class being created. The Type passed to the constructor indicates from where the class extends i.e. java.lang.Object. Again this is specified by a constant from the `EDU.purdue.cs.bloat.editor` package:

```
EDU.purdue.cs.bloat.editor.Type.OBJECT
```

The Type array specifies all of the interfaces that the class is to implement. Again these can be found in the `EDU.purdue.cs.bloat.editor` package and include interface such as Serializable and Comparable. Here's a complete example of the use of this method:

```
1  EDU.purdue.cs.bloat.editor.ClassEditor classEditor =
2     bloatContext.newClass(
3        EDU.purdue.cs.bloat.reflect.Modifiers.SUPER,
4        "MyNewClass",
5        EDU.purdue.cs.bloat.editor.Type.OBJECT,
6        null);
```

Note that if you intend to implement an interface or override a method that is inherited you will have to create those methods from scratch as well. The section on the method editor will demonstrate method creation from scratch.

The `ClassEditor` provides manipulation of data such as the class modifiers, the constant pool, and the interfaces implemented by the class. All of these uses are pretty intuitive with the exception of constant pool manipulation. You should refer to `ChangeFieldNames.java`, provided in the directory examples-1.0, for an example of constant pool manipulation.

### The `MethodEditor`

If you wish to edit a method in an existing class you will again need to refer to the `BloatContext`. The `BloatContext` provides a method, `editMethod`, to modify methods. The `editMethod` method takes a `MethodInfo` object as a parameter. You can retrieve this information from a `ClassEditor` with the method,`methods`, that returns an array of `MethodInfo` objects:

```
1  EDU.purdue.cs.bloat.reflect.MethodInfo[] methods =
2     classEditor.methods();
3  EDU.purdue.cs.editor.MethodEditor methodEditor =
4     bloatContext.editMethod(methods[0]);
```

This array contains the method information for all of the methods contained in the `ClassFile` related to the class editor including the constructor. Note that a `MethodInfo` object is really an interfaced used by a `Method` object and therefore you can cast a `MethodInfo` object to a `Method` object.

There is most likely a specific method that you wish to edit. You can retrieve a specific method by obtaining its name with the name method in the `MethodEditor` object. To create a new method for an existing class or a class you creating from scratch you must create a new `MethodEditor` object. Note that unlike creating a `ClassEditor` from scratch this is not done through the BLOAT context. To create a new `MethodEditor` use the constructor provided below:

```
MethodEditor(
   ClassEditoreditor,
   intmodifiers,
   TypereturnType,
   java.lang.StringmethodName,
   Type[]paramTypes,
   Type[]exceptionTypes);
```

The constructor first takes a `ClassEditor` for the class you wish to create a new method in. Then you must provide a modifier such as public, private, or protected. This can be specified by a constant in the `EDU.purdue.cs.reflect.Modifier` interface. The next parameter specifies the return type of the method. This can be specified using a constant from the `EDU.purdue.cs.editor.Type` interface. The String parameter is used to name the method you have chosen to create. Then the parameters of the method are specified by constants from the Type interface. The final parameter of the constructor specifies exceptions thrown by the method. Again this can be specified using constants found in the Type interface.

Once you have created a method you can begin adding instructions to the method. This done by calling the method `addInstruction` in the `MethodEditor`. The method takes an opcode constant as a parameter. These constants can be found in the `EDU.purdue.cs.bloat.editor.Opcode` interface. There is an opcode in this interface corresponding to each opcode in Java byte-code. It should be noted that valid Java byte-code must be entered or you will receive a Java runtime error.

#### The `FieldEditor`

The `FieldEditor` is used to edit attributes of a class. To edit fields of an existing class simply call the `editField` method of the `BloatContext`. This method takes a `FieldInfo` object. This object can be obtained from the `ClassEditor` by calling the fields method. This method returns an array of `FieldInfo` objects. This example demonstrates how to put these things together to create a `FieldEditor`:

```
1   EDU.purdue.cs.bloat.editor.FieldInfo[] fieldInfo =
2      classEditor.fields();
3   EDU.purdue.cs.bloat.editor.FieldEditor fieldEditor =
4      bloatContext.editField(fieldInfo[0]);
```

To create a new field you must use a constructor found in the `FieldEditor` class. The constructor is illustrated below:

```
FieldEditor(
   ClassEditoreditor,
   intmodifiers,
   java.lang.Classtype,
   java.lang.Stringname);
```

The constructor takes the `ClassEditor` for the class you wish to add an attribute to. Like the constructor to create a new method the constructor for the `FieldEditor` also takes a constant to specify a Modifier to specify whether the attribute is public, private, or protected. The constructor also takes a Type constant to indicate whether the attribute is an int, String, or other possibility. Finally the constructor accepts a String to name the attribute.

### Committing Edits

#### One Rule To Follow

There is one rule to follow in BLOAT that will save you a lot of pain, after you edit commit. This seems pretty obvious but there are allot of methods to commit edits in BLOAT. For instance the `ClassEditor`, `FieldEditor`, `MethodEditor`, and the `BloatContext` all have several commit methods. The rule can be applied as followed. If you make an edit with a `FieldEditor`, or any other editor, call the commit method of the editor when you have completed editing with that specific editor. The `BloatContext`'s

commit method can be used to commit all edits to all class files the `BloatContext` has knowledge of, i.e. any class that you have loaded or created through the `BloatContext`.

## Instruction Representation

### Opcodes

The folks who created BLOAT were nice enough to create a constant to represent every opcode in the Java byte-code language. These opcodes can be found in the `EDU.purdue.cs.bloat.editor.Opcode` interface.

### Instructions

You can create an instruction by simply passing an Instruction object an opcode constant form the `EDU.purdue.cs.bloat.editor.Opcode` interface.

```
1   EDU.purdue.cs.bloat.editor.Instruction instruction =
2      new EDU.purdue.cs.bloat.editor.Instruction(
3          EDU.purdue.cs.bloat.editor.Opcode.opc_astore);
```

There are several methods in the Instruction class that test whether the instruction the class represents is a jump or a switch as well as many other things. Each of these methods returns a boolean value based on their validity.

Each instruction can be add to a method or block in a control flow graph. You can do this by calling the `addInstruction` method for either of these objects and passing it an Instruction object.

## Advanced Features

### Inlining

Inlining is a fairly straight forward process. Simply create a `BloatContext` and load your classes as usual. You will now need to create an Inline object. The constructor for the object takes a `InlineContext` and an integer as a parameter. There are no special steps for obtaining an `InlineContext` the `BloatContext` already implements this interface. The integer determines the maximum number of instructions a method can grow to.

```
1   EDU.purdue.cs.bloat.inline.Inline inline =
2      new EDU.purdue.cs.bloat.inline.Inline(bloatContext,20);
```

To actually inline a method you must call the inline method of the Inline class. This method takes a `MethodEditor` as a parameter. Refer to the section on `MethodEditor`s if you do not already know how to create one. Once the method is called the inlining is complete

```
1   EDU.purdue.cs.bloat.inline.Inline inline =
2      new EDU.purdue.cs.bloat.inline.Inline(bloatContext,20);
3   inline.inline(methodEditor);
```

There are some extra features that can be used during inlining. Using the method `maxCallDepth` you can specify with an integer the maximum number of nested method calls inlined. The inlining process can be rather time consuming. Bloat attempts to remedy this by providing a method, `setMaxInlineSize`, that sets the maximum size method that will be inlined. This size is set by passing an integer to the `setMaxInlineSize` method.You may also specify whether to inline method that throw exceptions or not. This is simply set passing a boolean value to the method `setInlineExceptions`. Note that if you need to use one these features you must call one of these methods prior to calling the inline method.

```
1   int size = 25;
2   int depth = 3;
3   boolean flag = true;
```

```
4   inline.setMaxInlineSize(size);
5   inline.setMaxCallDepth(depth);
6   inline.setInlineExceptions(flag);
7   inline.inline(methodEditor);
```

### Class Hierarchy

It maybe of some interest to understand or manipulate the class hierarchy of a class. The first step in this process is to load all classes in the hierarchy using your current `BloatContext`. The `BloatContext` must first have knowledge of all of the class relevant to the hierarchy. Once the classes are load you can extract information on the hierarchy by calling `getHierarchy` method found in the `BloatContext`. The method returns a `ClassHierarchy` object. The `ClassHierarchy` object allows you to view all of the classes and interfaces in a hierarchy as well as edit the hierarchy by adding or removing classes an interfaces.

```
1   // Add a class
2   EDU.purdue.cs.bloat.editor.ClassHierarchy classHierarchy =
3      bloatContext.getHeirarchy();
4   String className = "Test";
5   classHierarchy.addClassNamed(Test);
6   // extract the Type of classes in the hierarchy:
7   Collection classes =
8      classHierarchy.classes();
9   // extract Type of classes that implement a specific interface:
10  Collection classes =
11     classHierarchy.implementors(
12        EDU.purdue.cs.bloat.editor.Type.SERIALIZABLE);
13  // find interfaces a Type of object implements
14  Collection interfaces =
15     classHierarchy.interfaces(
16        EDU.purdue.cs.bloat.editor.Type.CLASS);
17  // extract subclasses of a Type of object
18  Collection subclasses =
19     classHierarchy.subclasses(EDU.purdue.cs.bloat.editor.Type.OBJECT);
```

The `ClassHierarchy` may also be used to determine what method will be invoked at any depth in the hierarchy. Therefore you can determine from where a method is inherited or overwritten.

```
// simulates dynamic dispatching:
MemberRef methodInvoked(
   Type receiver,
   NameAndType method);
```

The method takes two arguments a Type, which we are already familiar with from previous examples, and a `NameAndType`. The Type argument determines the receiver of the call. The `NameAndType` class simply describes an a method by its name and descriptor. The name is simply a String and the descriptor describes the return type of the method using the Type class.

```
EDU.purdue.cs.bloat.editor.NameAndType(
   java.lang.String name,
   Type type);
```

The `MemberRef` returned by `methodInvoked` represents the method you are interested in and class from where it was invoked.

```
1   //Putting it all together:
2   EDU.purdue.cs.bloat.editor.ClassHierarchy classHierarchy =
3      bloatContext.getHeirarchy();
4   EDU.purdue.cs.bloat.editor.NameAndType nameAndType =
5      new EDU.purdue.cs.bloat.editor.NameAndType(
6         "test",
7         EDU.purdue.cs.bloat.editor.Type.STRING);
8   MemberRef memRef =
9      classHeirarchy.methodInvoked(
10        EDU.purdue.cs.bloat.editor.Type.OBJECT,
11        nameAndType);
12  System.out.println(memRef);
```

Using the `ClassHierarchy` you may also determine if a method has been overwritten. Note the previous example could be used to determine where it was overwritten. In order to determine if a method has been overwritten you must call the `methodIsOverridden` method:

```
boolean methodIsOverridden(
   Type classType,
   NameAndType method);
```

To use this method you must pass it the class Type of where the method resides and a `NameAndType` that identifies the method and its return type. The method will return a boolean value based on whether the method has been overwritten for that class Type.

```
1   //Putting it all together
2   EDU.purdue.cs.bloat.editor.ClassHierarchy classHierarchy =
3      bloatContext.getHeirarchy();
4   EDU.purdue.cs.bloat.editor.NameAndType nameAndType =
5      new EDU.purdue.cs.bloat.editor.NameAndType(
6         "test",
7         EDU.purdue.cs.bloat.editor.Type.STRING);
8   boolean value =
9      classHeirarchy.methodIsoverridden(
10        EDU.purdue.cs.bloat.editor.Type.OBJECT,
11        nameAndType);
```

### Control Flow Graph (CFG)

Bloat allows a developer to create a control flow graph for any given method. To do so you simply need to construct a `FlowGraph` object. To create a `FlowGraph` you must provide a `MethodEditor` for the constructor the `FlowGraph`.

```
EDU.purdue.cs.bloat.cfg.FlowGraph(
   MethodEditor method);
```

To construct the `FlowGraph` object does not setup the cfg. You must call the initialize method of the `FlowGraph` object to create the graph. Once the graph is created you can extract the basic blocks using a variety of methods. Most of these methods produce a List or Collection of blocks that can be iterated across. There also several print methods to print the cfg in various orderings.

```
1   // printing cfg in different orders
2   EDU.purdue.cs.bloat.cfg.FlowGraph cfg =
```

```
3      new EDU.purdue.cs.bloat.cfg.FlowGraph(methodEditor);
4   cfg.preOrder();
5   cfg.postOrder();
6   cfg.print();
7   cfg.printGraph();
```

```
1   // extracting blocks and manipulating blocks:
2   EDU.purdue.cs.bloat.cfg.FlowGraph cfg =
3      new EDU.purdue.cs.bloat.cfg.FlowGraph(methodEditor);
4   //Blocks returned in trace order:
5   java.util.List list =
6      cfg.trace();
7   for(int i=0;i<list.size();i++){
8      EDU.purdue.cs.bloat.tree.Tree tree =
9         list.tree();
10     EDU.purdue.cs.bloat.editor.Instruction instruction =
11        new EDU.purdue.cs.bloat.editor.Instruction(
12           EDU.purdue.cs.bloat.editor.Opcode.ops_aaload);
13     tree.addInstruction(instruction);
14  }
```

The above example extracts the blocks from a cfg. It then accesses the Tree for each Block. A Tree represents the expression tree for each block. Using the tree you can view the instructions contained with in a Block as well as edit them by adding and deleting instructions. The example above demonstrates how to add an instruction.

### SSA (Value Graphs)

A `FlowGraph` object may also be used to create a value graph. The SSA graph (also called the value graph) represents the nesting of expression in a control flow graph. Each node in the SSA graph represents an expression. If the expression is a definition, the it is labeled with the variable it defines. Each node has directed edges to the nodes representing its operands.

`SSAGraph` is a representation of the definitions found in a CFG in the following form: Each node in the graph is an expression that defines a variable (a `VarExpr`, `PhiStmt`, or a `StackManipStmt`). Edges in the graph point to the nodes whose expressions define the operands of the expression in the source node.

This class is used primarily get the strongly connected components of the SSA graph in support of value numbering and induction variable analysis.

Nate (One of the Authors of BLOAT) warns: Do not modify the CFG while using the SSA graph! The effects of such modification are undefined and will probably lead to nasty things occurring.

```
1   EDU.purdue.cs.bloat.ssa.SSAGraph ssaGraph =
2      new EDU.purdue.cs.bloat.ssa.SSAGraph(cfg);
```

# 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.

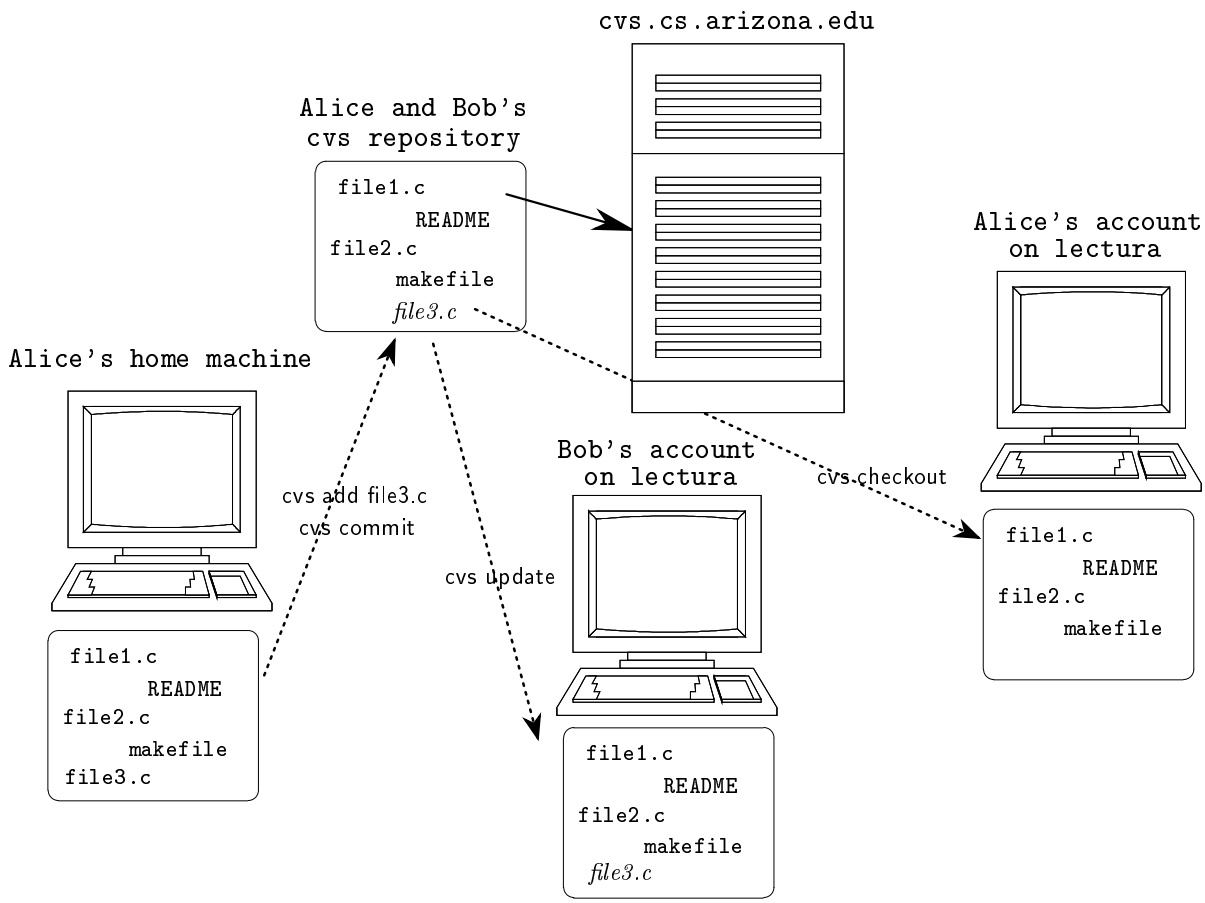
## Summary

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.