Hacking SandMark

Christian Collberg

January 28, 2003

Contents

Chapter 1

Extending SandMark

1.1 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.

1.2 Obfuscation Configuration

To customize Obfuscation Algorithm Application and Obfuscation Level Settings for a whole jar file ,or a class file and its methods or a method alone the configure button in the obfuscation panel is to be used .On clicking The Configure button in the obfuscate panel , a new window pops up which is known as Obfuscation Dialog ,which helps the users to select specific obfuscation algorthims and obfuscation level .

Method Summary

```
2   /*  Find if a particular algorithm is marked for obfuscation in the case
3    *  of a class or jar file . Default is that all algorithms are to be applied
4    *  for all classes and methods in the jar file .
5    */
6       public boolean IsMarked(String algo,String class_or_jarname);
7
8   /*  Find if a particular algorithm is marked for obfuscation in the case
9    *  of a method . Please specify the  signature enclosed in brackets.
10   */
11      public boolean IsMarked(String algo,String classname,String methodname,
12           String signature);
13
14  /*  Obtain the obfuscation level for a class or jar file. Possible String
15   *  values are high , medium, low,none. Default is high .
16   */
17      public String getLevel(String class_or_jarname) ;
18
19  /*  Obtain the obfuscation  level for a method .The  signature should
20   *   be enclosed in brackets .Possible values are high ,medium,low,none.
21   *   Default is high .
22   */
23      public String getLevel(String classname,String methodname,String signature)
```

1.3 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  }
```

1.4 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  }
```

1.5 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`.

Chapter 2

Working with the SandMark Code-base

2.1 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.
- We make an effort to write methods that pass the "hand test". The "hand test" passes if a method is short enough to be completely covered by the programmer's hand placed horizontally with fingers together on the screen. ("Horizontally" means that the fingers are pointed to a side of the monitor, not toward the top or bottom.)

2.2 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:

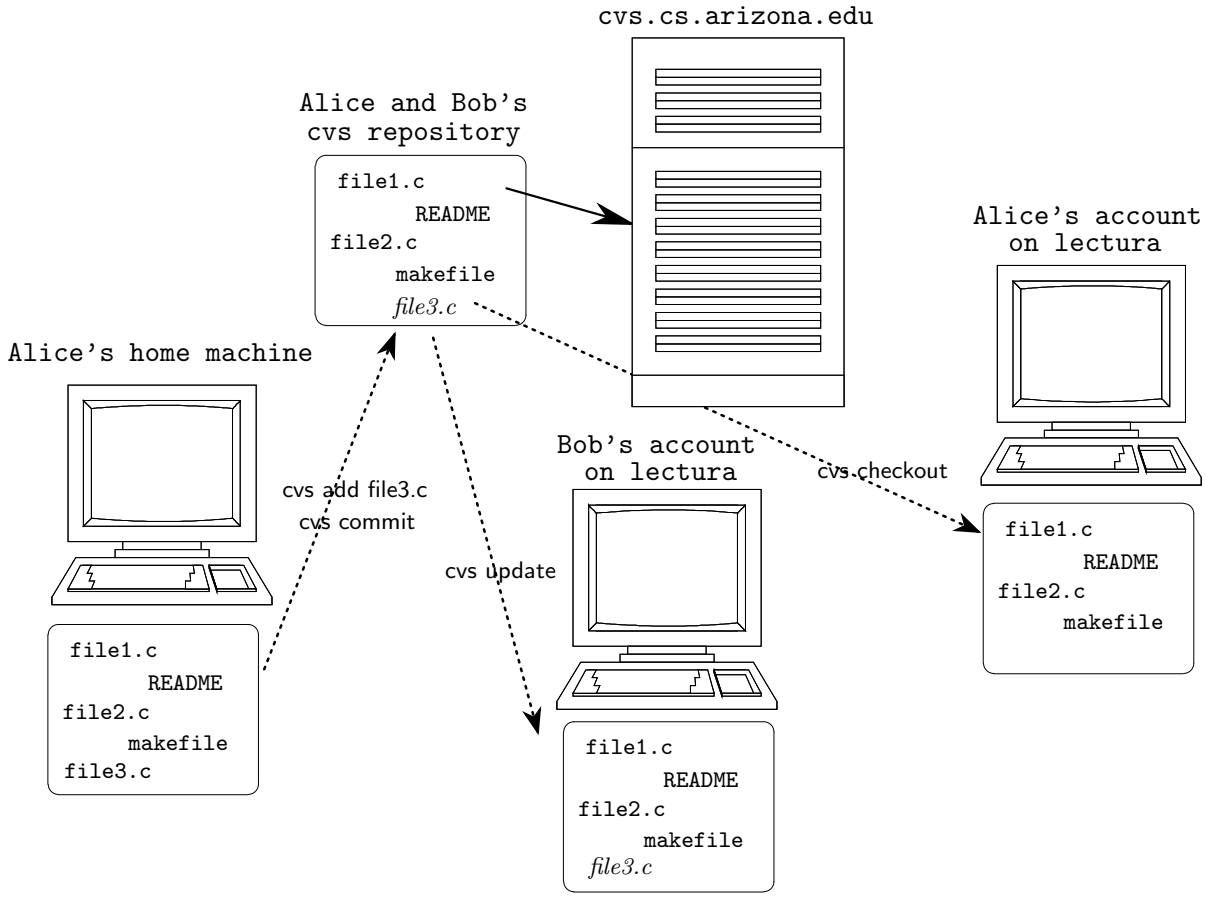
cvs add file Add a new file to the project. The file will not actually be uploaded to the repository until you run the `commit` command.

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.



Chapter 3

Useful Tools

3.1 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
```

3.2 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);
```

3.3 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 **??** 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 piece 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.

```
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 3.1: Loading a class file.

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 pertaining to fields and methods as well as more byte code specific information. Figure **??** 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 al lot 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 methods 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.

Trees

Trees were previously introduced in the section on `FlowGraph`. `Tree` represents the expression tree of a basic block. It consists of an operand (expression) stack comprised of expressions and a list of statements contained in the block. The blocks, as previously mentioned, can be obtained from the `FlowGraph`. `Block` contains a method called `tree` that takes no arguments and returns a `Tree`. `Tree` allows you to manipulate the expression tree to obtain a `Tree`. Instructions may be added with the methods below.

```
1  // adds an instruction that does not change control flow
2  addInstruction(Instruction inst)
3  // adds an instruction to jump to another Block in the FlowGraph
4  addInstruction(Instruction inst, Block next)
5  // add an instruction such as ret or astore to invoke a subroutine
6  addInstruction(Instruction inst, Subroutine sub)
```

A branch in the CFG maybe labeled using the `addLabel(Label)` method. To create a label follow the example below. The index used in the example represents the labels in the instruction array. The boolean in the constructor specifies whether the `Label` starts a block or not. You may also add a comment to a `Label` by calling the `setComment` method which takes a string to represent the comment.

```
2  // The index represents represents a unique offset, .i.e. its offset
3  // in the instruction array
```

```
4   int index = 23;
5   EDU.purdue.cs.bloat.editor.Label label = new EDU.purdue.cs.bloat.editor.Label(index,false);
6   label.setComment("Hello");
7   System.out.println(label);
8   tree.addLabel(label);
```

An additional `Stmt` can also be added to a `Tree` using the method `addStmt(Stmt stmt)`. A `Stmt` is the super class of `JumpStmt`, `ExprStmt`, and `PhisStmt`. A `JmpStmt` is a super class of the following: `GotoStmt`,`IfStmt`,`JsrStmt`. A `GotoStmt` represents an unconditional branch to a basic block. The constructor takes a `Block` as an argument which represents the target of the `GotoStmt`. An `IfStmt` is a super class of statements in which some expression is evaluated and one of two branches is taken. Its subclasses are `IfCmpStmt` and `IfZeroStmt`. To create an `IfCmpStmt` the constructor takes 5 arguments. The first is an integer that represents the comparison operator the if statement. Next it takes a left `Expr` and right `Expr`. Finally it takes a `Block` to jump to when the if condition is true and then a `Block` to jump to if the condition is false. An `IfZeroStmt` evaluates an expression and executes one of its two branches depending on whether or not the expression evaluated to zero. To create a `IfZeroStmt` you simply use the same arguement sequence as supplied to a `IfCmpStmt`. You should take note that a `Stmt` can not be nested as `Expr` can. These are not all of the `Stmt` supplied by BLOAT. This is only an introduction into what they offer. `Stmt`s can be removed from a `Tree` using the method `removeStmt(Stmt stmt)`.

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);
```

Type-Based Alias Analysis (TBBA)

Type-Based Alias Analysis (TBAA) is used to determine if one expression can alias another. An expression may alias another expression if there is the possibility that they refer to the same location in memory. BLOAT models expressions that may reference memory locations with `MemRefExprs`. There are two kinds of "access expressions" in Java: field accesses (`FieldExpr` and `StaticFieldExpr`) and array references (`ArrayRefExpr`). Access paths consist of one or more access expressions. For instance, `a.b[i].c` is an access expression.

TBAA uses the `FieldTypeDecl` relation to determine whether or not two access paths may refer to the same memory location.

A static method in the `EDU.purdue.cs.bloat.tbaa.TBAA` class may be used to do Type-Based Alias Analysis.

```
1    EDU.purdue.cs.bloat.tbaa.TBAA.canAlias(EditorContext context, Expr a, Expr b)
```

Where the `EditorContext` represents the `BloatContext` and the `Expr` parameters represent expressions. `EDU.purdue.cs.bloat.tree.Expr` is a an abstract superclass. Its known subclasses are listed below.

```
1  ArithExpr
2  ArrayLengthExpr
3  CallExpr
4  CastExpr
5  CatchExpr
6  CheckExpr
7  CondExpr
8  ConstantExpr
9  DefExpr
10 NegExpr
11 NewArrayExpr
12 NewExpr
13 NewMultiArrayExpr
14 ReturnAddressExpr
15 ShiftExpr
16 StoreExpr
```