CS266 Software Reverse Engineering (SRE)
Applying Anti-Reversing Techniques to Java Bytecode

Teodoro (Ted) Cipresso
Senior Software Engineer, IBM
SRE Guest Lecture

The information in this presentation is taken from the thesis "Software reverse engineering education" available at http://scholarworks.sjsu.edu/etd_theses/3734/ where all citations can be found.

Applying Anti-Reversing Techniques to Java Bytecode

Introduction and Motivation for Anti-Reversing

- If Java bytecode is not protected through obfuscation it's straightforward for a reverser to decompile the bytecode, make changes, and recompile.
- If our software sells for money, then it's natural to offer a trial version.
- Trial versions employ guards to ultimately earn your business:
  - Time-bomb: stops working after *n* days.
  - Usage limit: stops working after *n* minutes or *n* operations.
  - Crippleware: Critical functionality is disabled (e.g, can't save you work).
    - This is hard to get right without annoying a potential customer.

Applying Anti-Reversing Techniques to Java Bytecode

Introduction and Motivation for Anti-Reversing

- Applying anti-reversing techniques to Java bytecode (or binaries in general) can prevent a reverser from:
  - Removing trialware guards in software (types on previous slide).
  - Recovering algorithms that make the software valuable.
- While anti-reversing techniques cannot completely prevent software from being reversed, they act as a deterrent by increasing the challenge for the reverser.
- [5] states “It is never possible to entirely prevent reversing” and “What is possible is to hinder and obstruct reversers by wearing them out and making the process so slow and painful that they give up.”

Applying Anti-Reversing Techniques to Java Bytecode

Introduction and Motivation for Anti-Reversing

- Apply anti-reversing techniques to source code, machine code, or bytecode can have adverse effects on a program's size, efficiency, and maintainability.
  - Therefore, it’s important to evaluate whether a particular program warrants the cost of protecting it.
- Anti-reversing techniques are meant to be applied post-production, after the coding for an application is complete and tested.
  - These techniques obscure data and logic and therefore are difficult to implement while also working on the functionality of the application.
- Even worse than the general confusion of interleaving anti-reversing techniques with regular coding is the possibility of creating a dependency between the actual application logic and the anti-reversing techniques used.

Applying Anti-Reversing Techniques to Java Bytecode

Basic Anti-Reversing Techniques

- Eliminating Symbolic Information: Render unrecognizable, all symbolic information in machine code or bytecode because such information can be quite useful to a reverse engineer.
- Obfuscating the Program: Includes removing symbolic information but goes much further. Care must be taken with the following techniques to ensure that the original program functionality remains intact.
  - Modify the layout of a program (move things around).
  - Introducing confusing non-essential logic or control flow.
  - Storing data in difficult to interpret organizations or formats.
    - Obfuscate data structures, encrypt strings etc..

Applying Anti-Reversing Techniques to Java Bytecode

Basic Anti-Reversing Techniques

- Embedding Antidebugger Code: Live analysis is how most reversers accomplish their objective. Therefore it is common for developers to implement guards against binary debuggers.
  - Live analysis of machine code is accomplished using an interactive debugger-disassembler where you attach to a running programming and step instructions to observe the program at points of interest.
  - Static analysis of machine code is usually carried out using a disassembler and heuristic algorithms that attempt to understand the structure of the program.
  - Interactive debugging of Java bytecode can be accomplished using debugger interfaces implemented by the JVM per the Java Platform Debugger Architecture (JPDA).

Applying Anti-Reversing Techniques to Java Bytecode

Java Bytecode Anti-Reversing Considerations

- While it is most often the case that we cannot recover the original Java source code from the bytecode, the results will be functionally equivalent.
- When new features are added to the Java language, new bytecode instructions are not always added or needed. For example:
  - Support for generics in collections is implemented by carrying additional information (in the constants pool of the class) that describes the type of object a collection should contain.
    - This information can then be used at execution time by the JVM to validate the type of each object in the collection.

Applying Anti-Reversing Techniques to Java Bytecode

Java Bytecode Anti-Reversing Considerations

- The strategy of having newer Java language constructs result in compatible bytecode with optionally-utilized metadata provides the benefit of allowing legacy Java bytecode to run on newer JVMs.
  - If a decompiler doesn't know to look for the metadata, some information is lost. For example:
    - The fact that a program used generics would not be recovered and all collections would be of type *Object* (with cast statements of course).

Applying Anti-Reversing Techniques to Java Bytecode

Java Bytecode Anti-Reversing Tools

- Since Java bytecode is standardized there are many obfuscation tools available on the Internet which perform transformations directly on the Java bytecode instead of on the Java source code itself.
  - SandMark: A Tool for the Study of Software Protection Algorithms [27]
  - RetroGuard for Java Obfuscation [28] (defunct, CLI only)
  - ProGuard [29]
  - Zelix Klassmaster [26] (not free)
  - Extensive list of related or alternative tools


sandmark340.zip


retroguard-v231.zip


proguard5.2.zip

Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information

- Variable, class, and method names, are all left intact when compiling Java source code to Java bytecode.
- Oracle's Java compiler javac provides an option to leave out debugging information in Java bytecode: specifying javac -g:none will exclude information on line numbers, the source file name, and local variables.
  - This option offers little to no help in fending off a reverser as none of the remaining variable names, methods names, and constants are obfuscated.
- [26] states that a high-level of protection can be achieved for Java bytecode by applying three transformations: Name Obfuscation, String Encryption, and Flow Obfuscation.
  - There does not appear to be a free anti-reversing tool that can perform all three of these transformations to Java bytecode.

Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information

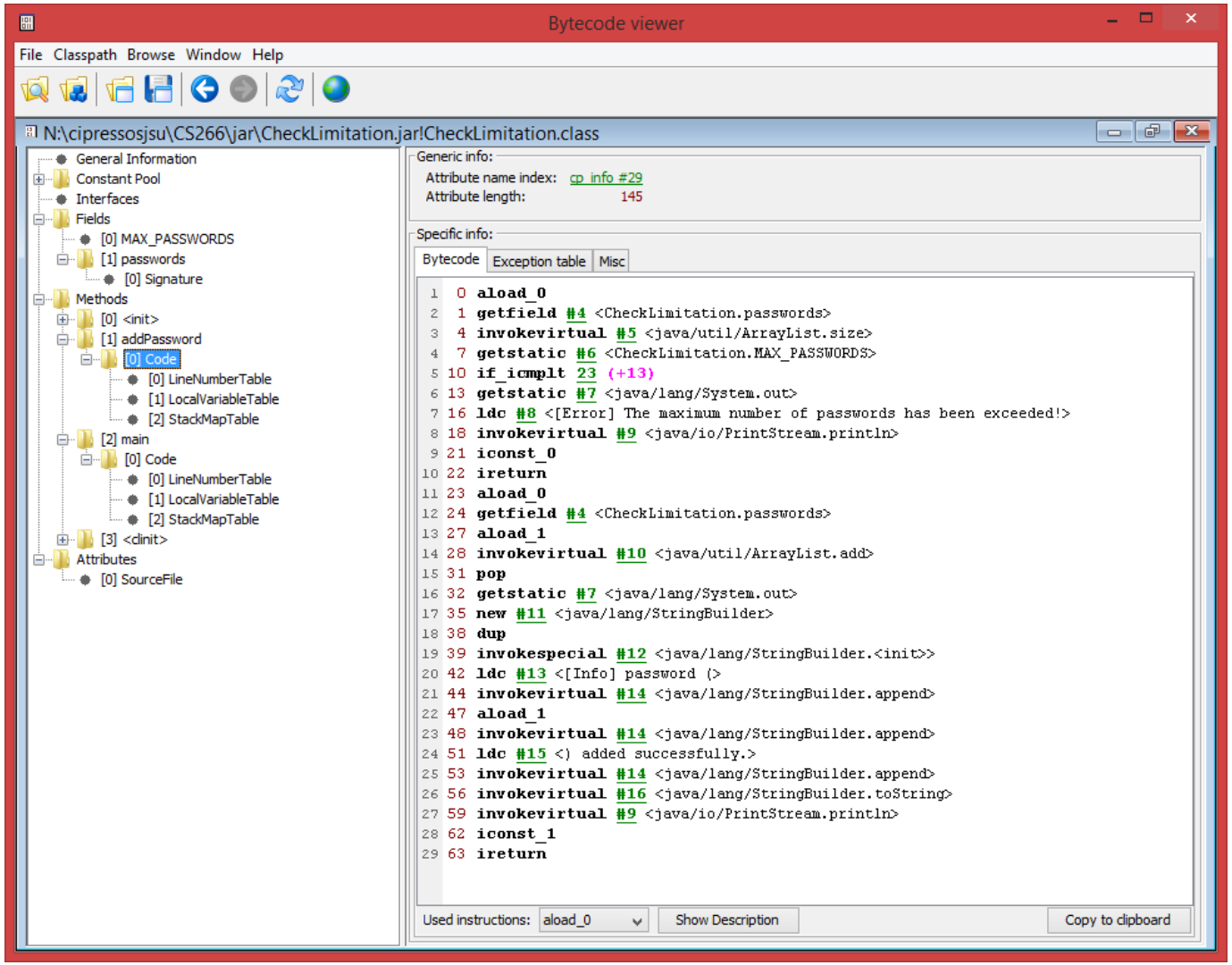
```java
import java.util.ArrayList;

public class CheckLimitation {
    private static int MAX_PASSWORDS = 5;
    private ArrayList<String> passwords;
    public CheckLimitation() {
        passwords = new ArrayList<String>();
    }
    public boolean addPassword(String password) {
        if (passwords.size() >= MAX_PASSWORDS) {
            System.out.println("[Error] The maximum number of passwords has been exceeded!");
            return false;
        } else {
            passwords.add(password);
            System.out.println("[Info] password (" + password + ") added successfully.");
            return true;
        }
    }
    public static void main(String[] arguments) {
        CheckLimitation store = new CheckLimitation();
        boolean loop = true;
        for (int i = 0; i < arguments.length && loop; i++) {
            if (!store.addPassword(arguments[i])) {
                loop = false;
            }
        }
    }
}
```

Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information

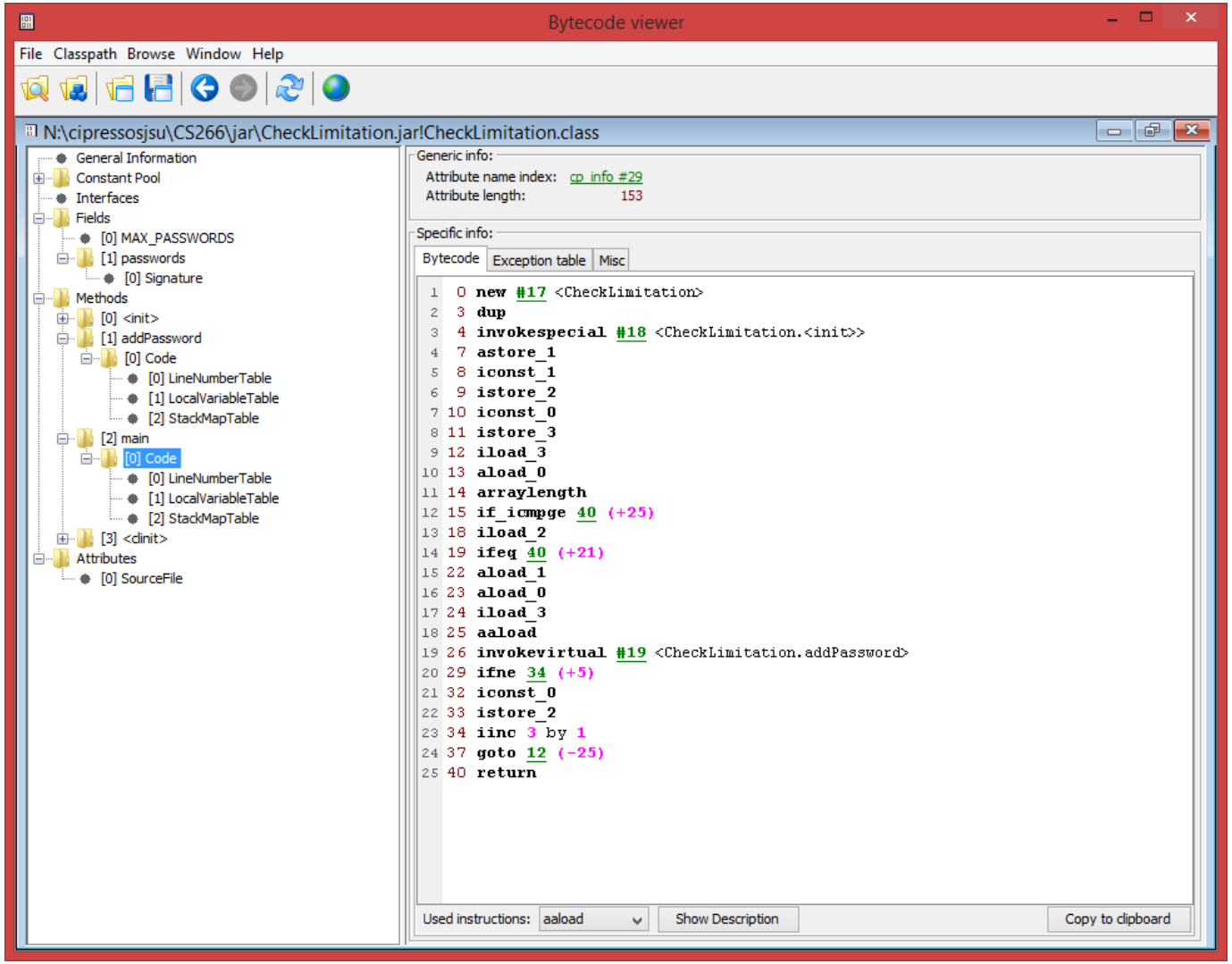
CheckLimitation.jar

jclasslib_win64_4_3_1.zip



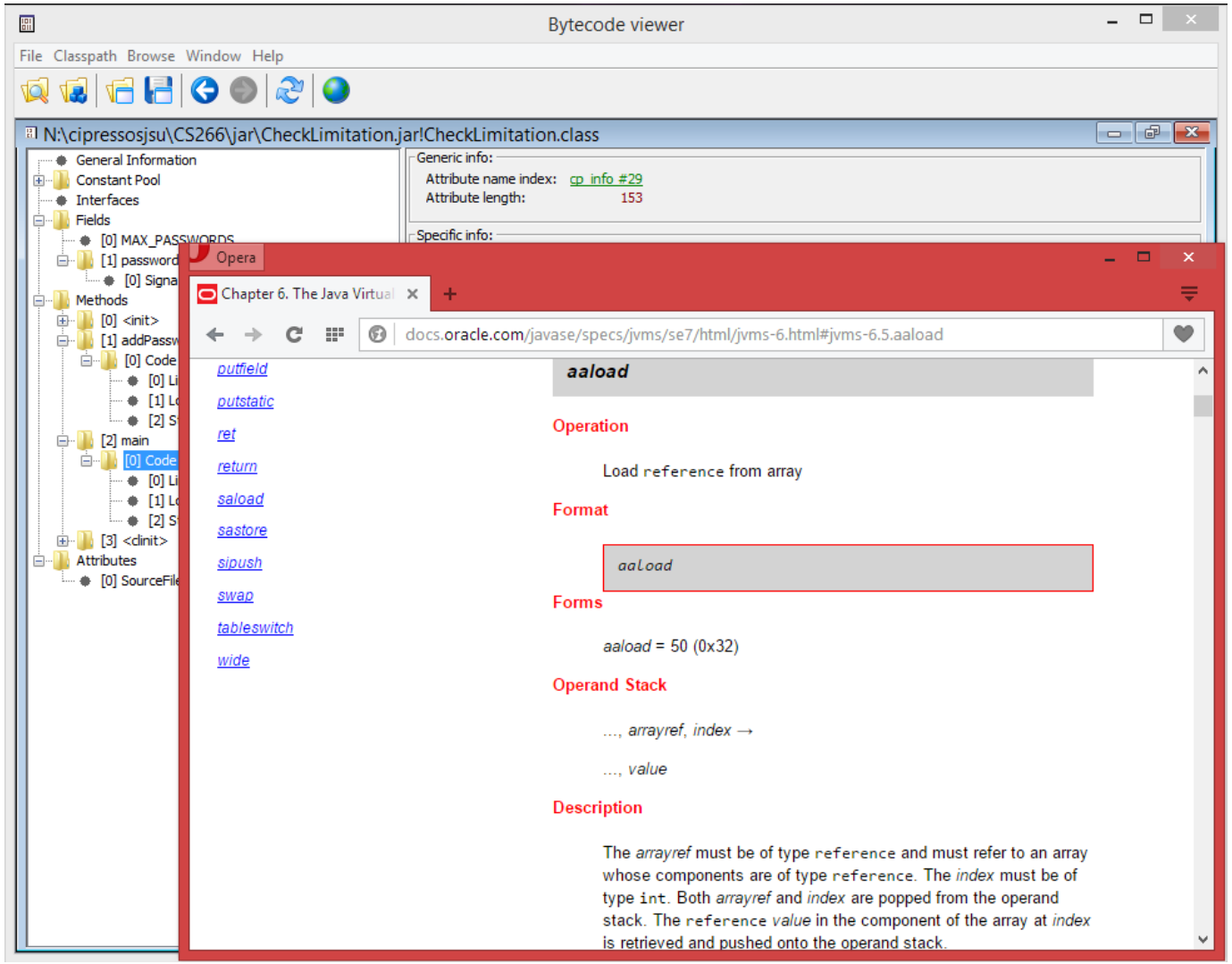
Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



Applying Anti-Reversing Techniques to Java Bytecode

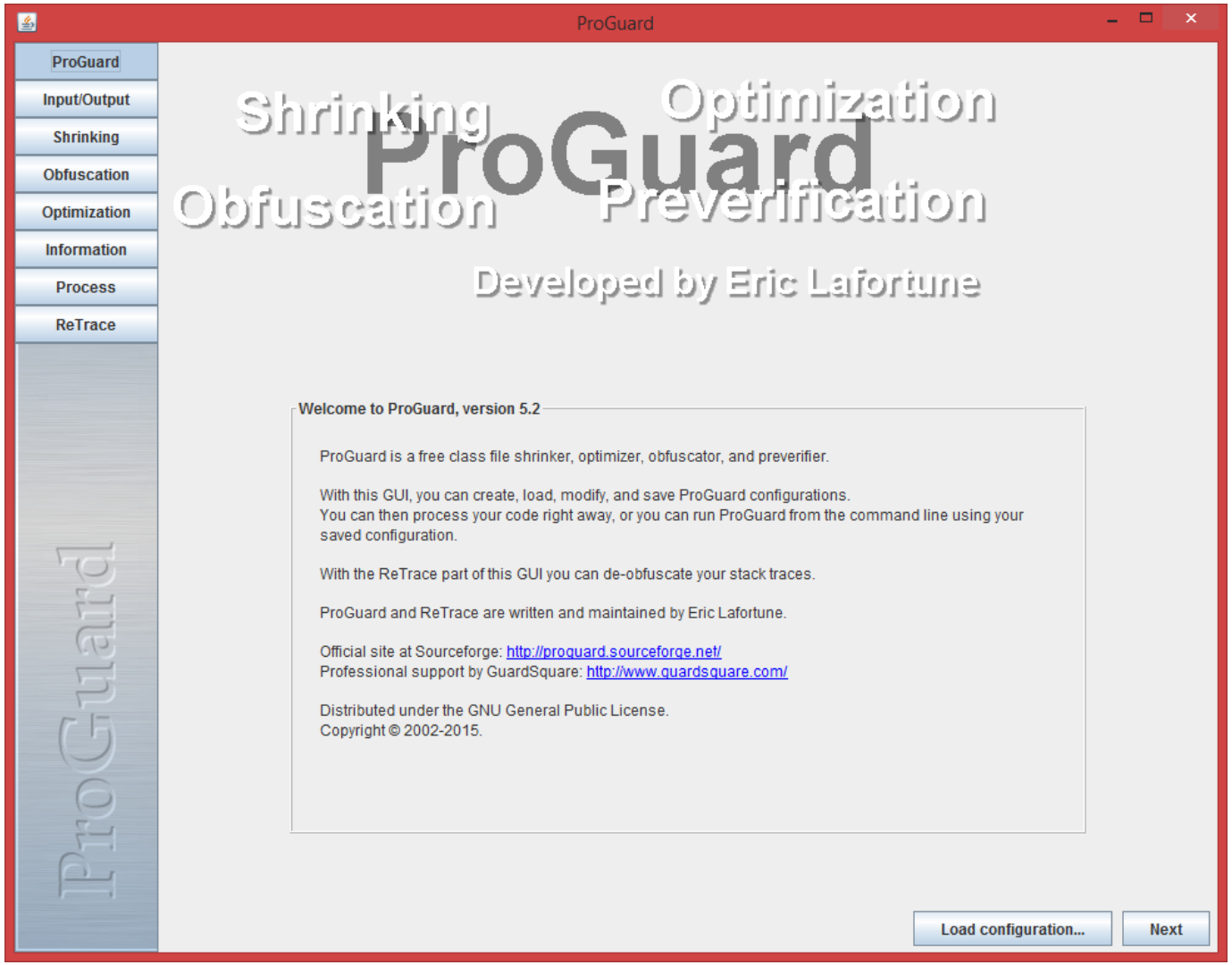
Eliminating Symbolic Information



Applying Anti-Reversing Techniques to Java Bytecode

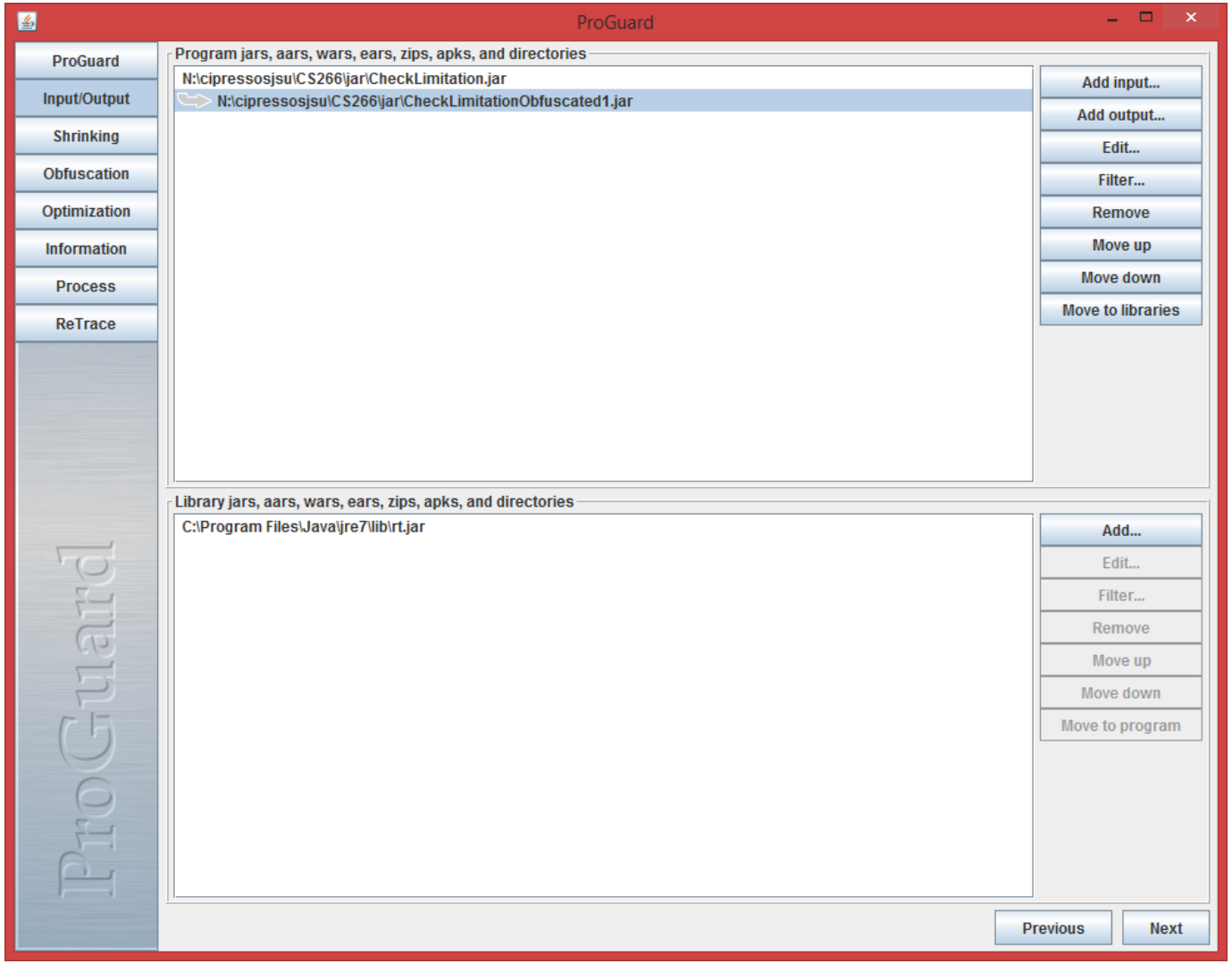
Eliminating Symbolic Information

proguard5.2.zip



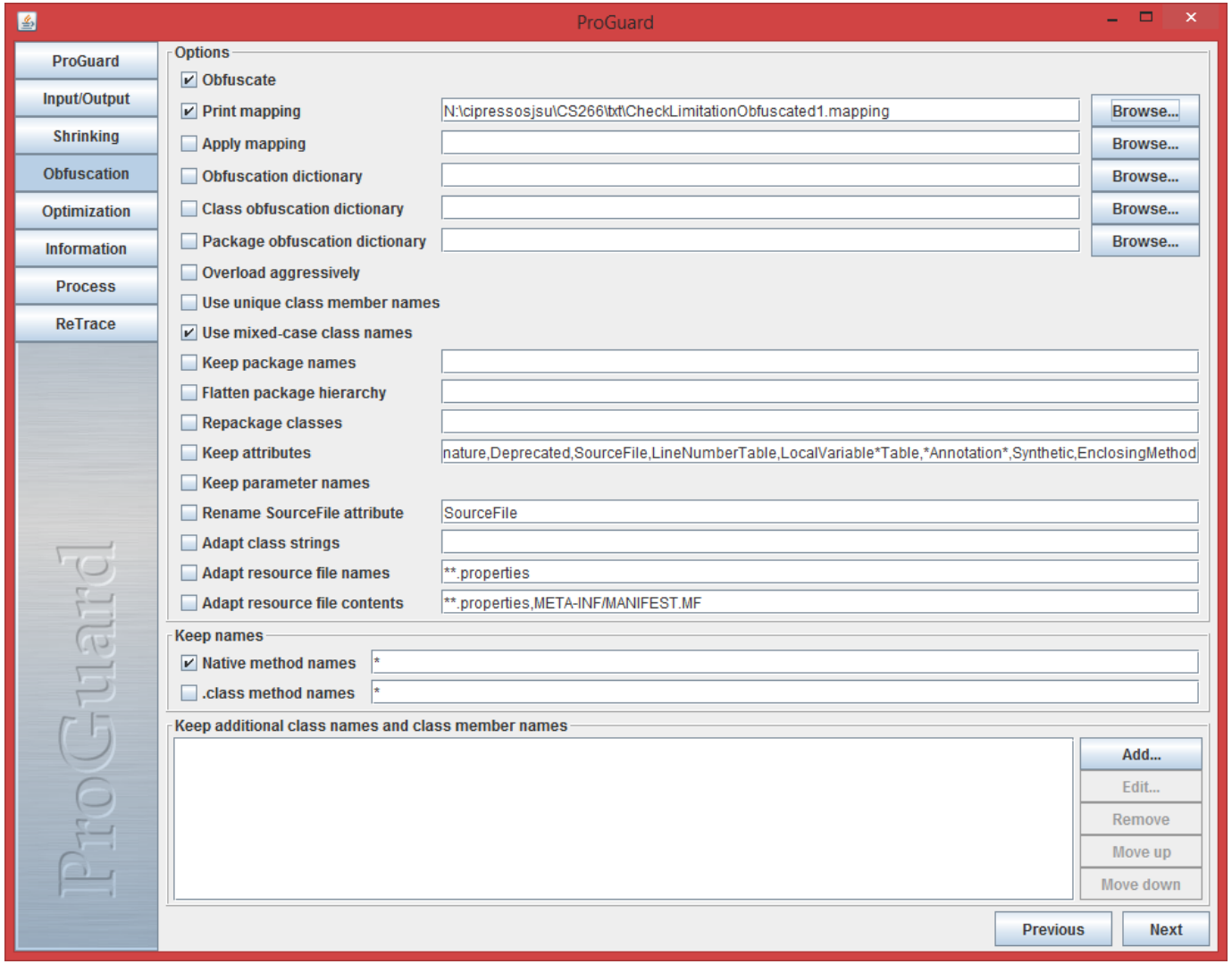
Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



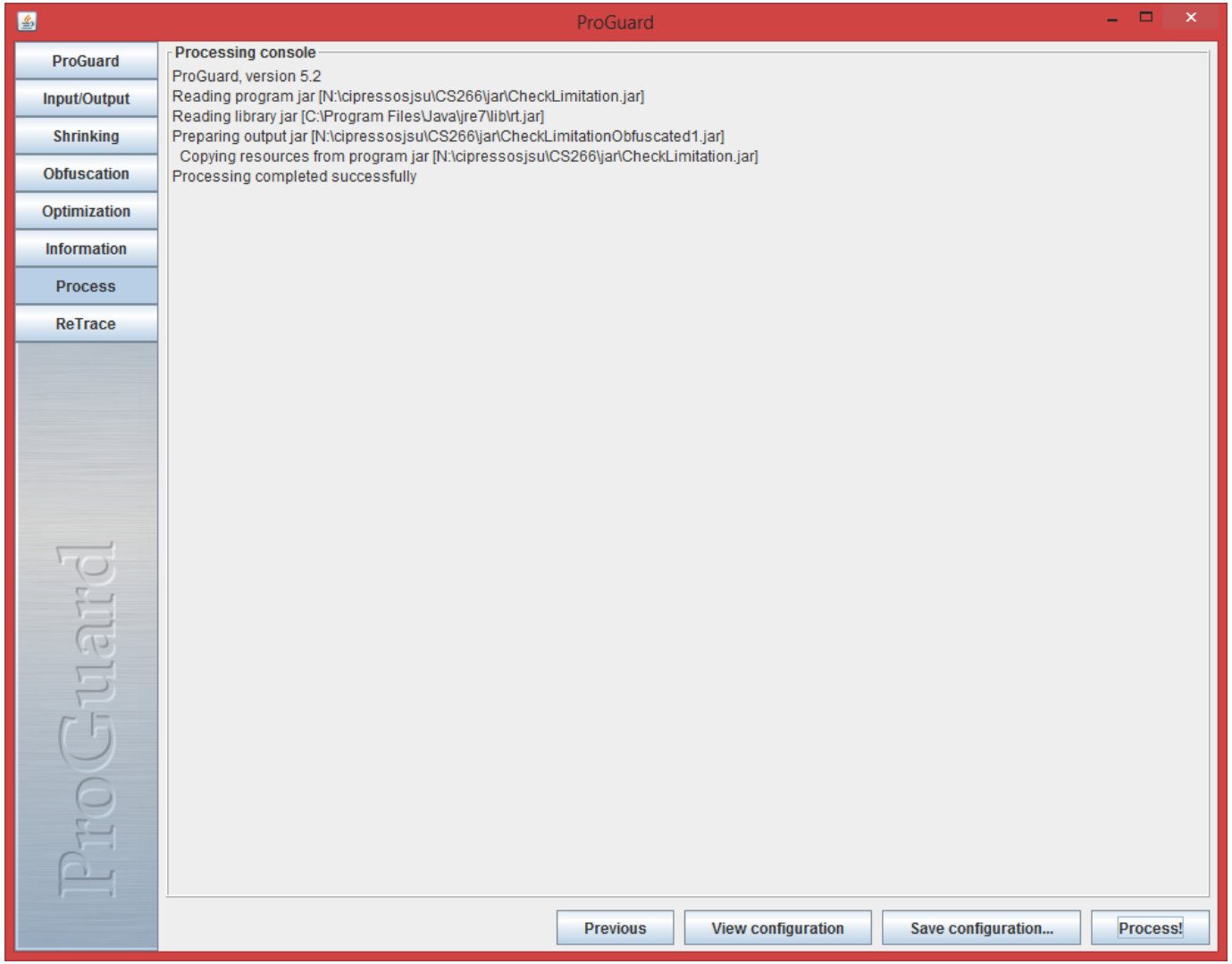
Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



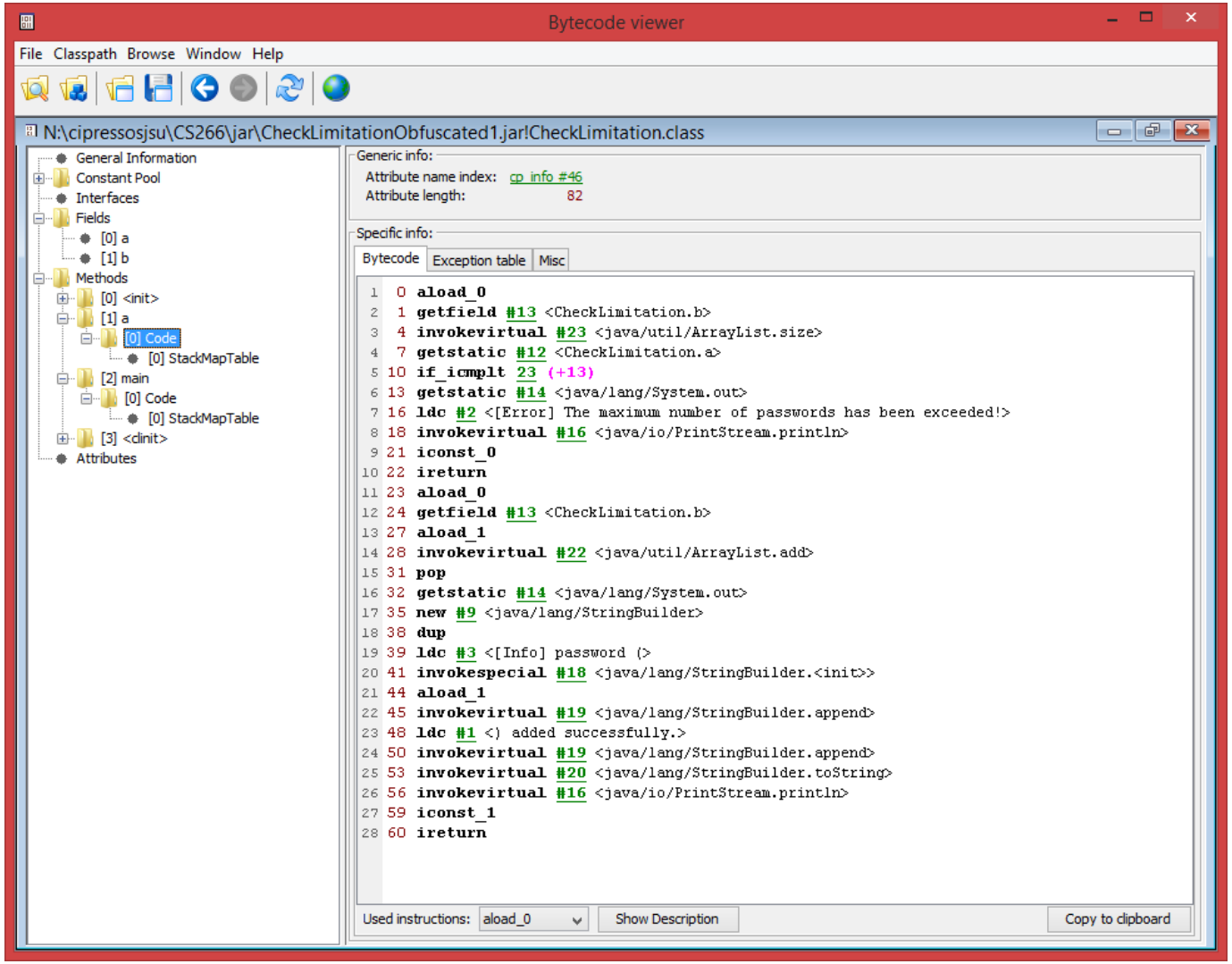
Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



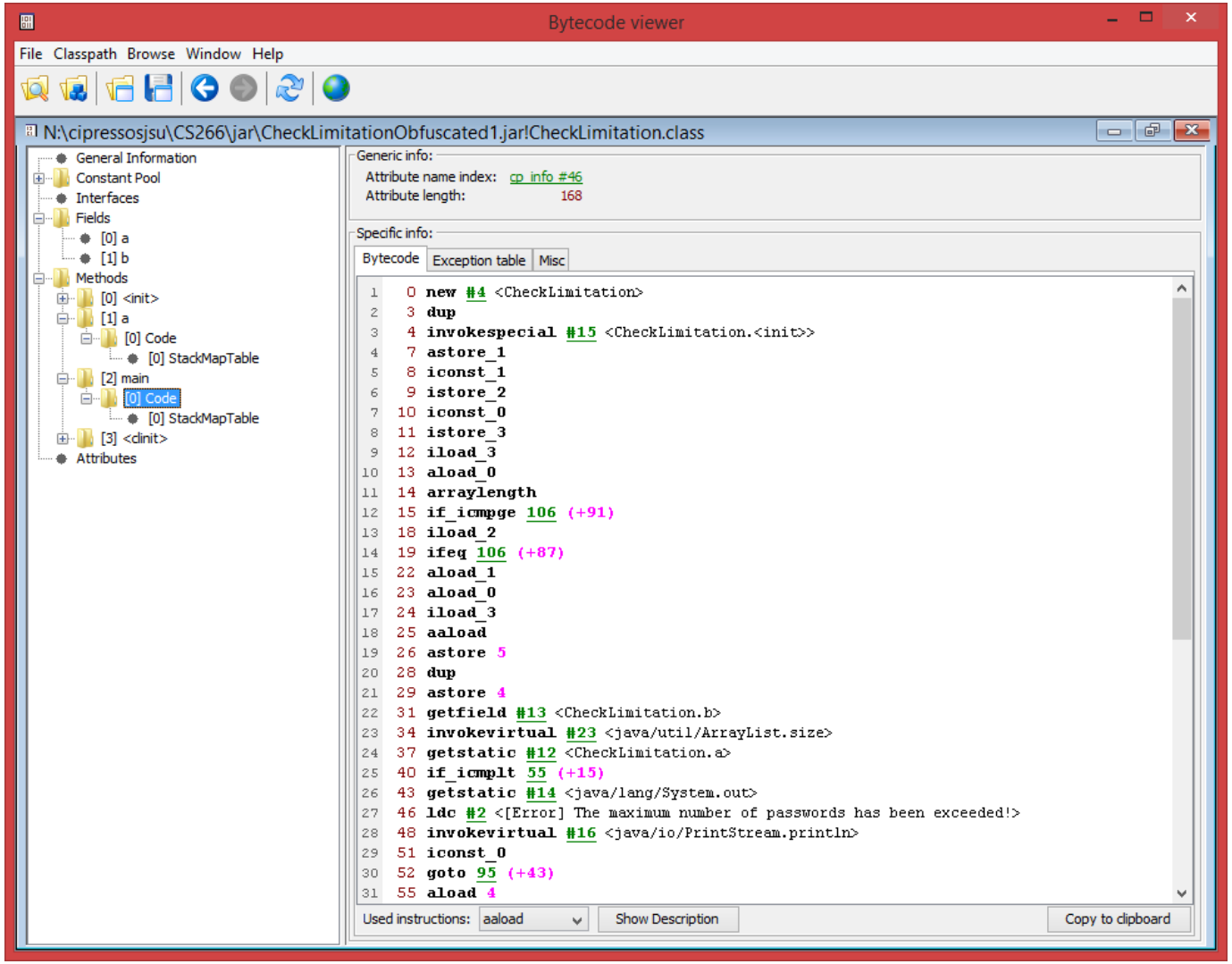
Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information

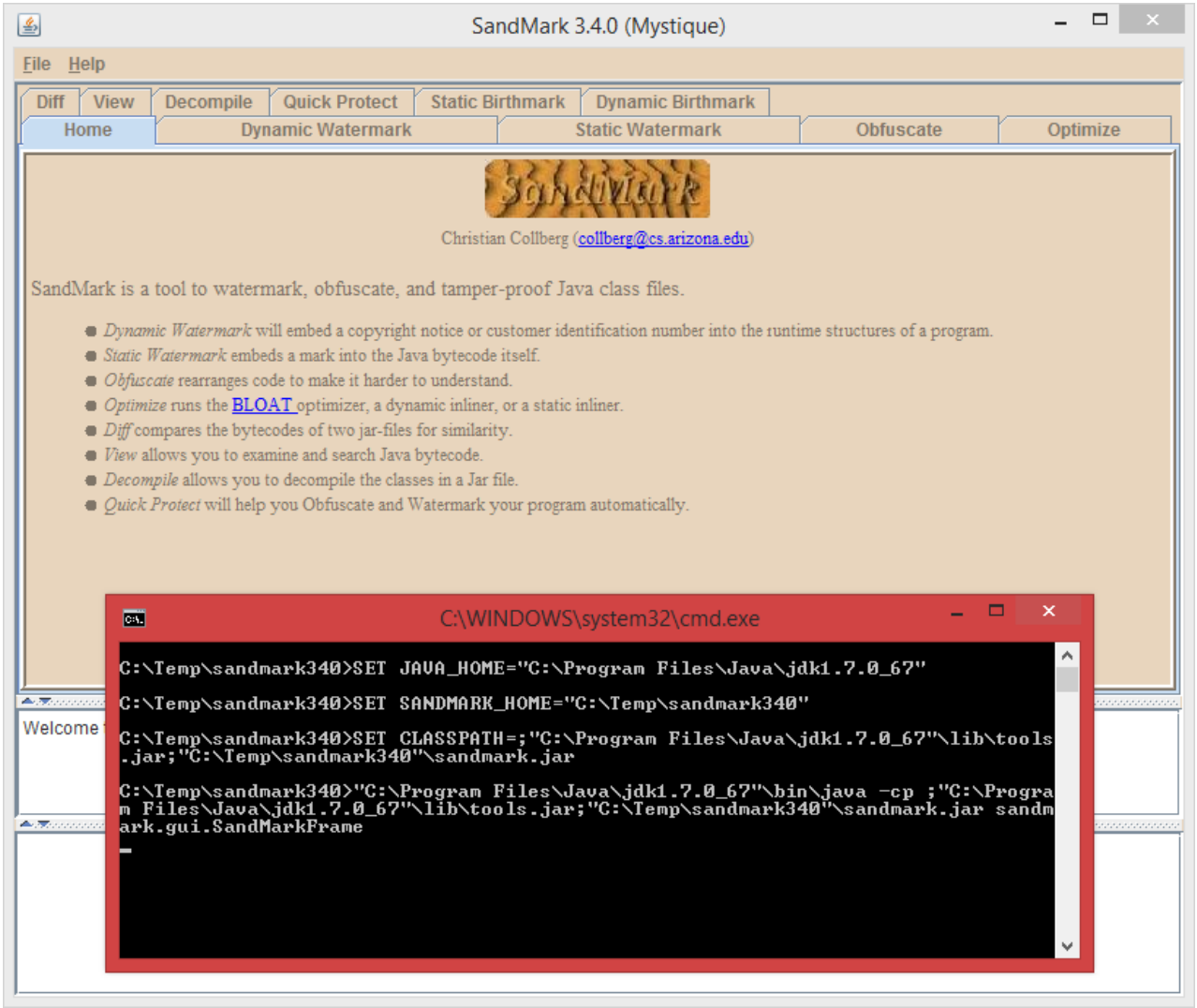
fernflower.jar

CheckLimitationObfuscated1.jar

```java
import java.util.ArrayList;
public class CheckLimitation {
   private static int a = 5;
   private ArrayList b = new ArrayList();
   private boolean a(String var1) {
      if(this.b.size() >= a) {
         System.out.println("[Error] The maximum number of passwords has been exceeded!");
         return false;
      } else {
         this.b.add(var1);
         System.out.println("[Info] password (" + var1 + ") added successfully.");
         return true;
      }
   }
   public static void main(String[] var0) {
      CheckLimitation var1 = new CheckLimitation();
      boolean var2 = true;
      for(int var3 = 0; var3 < var0.length && var2; ++var3) {
         String var5 = var0[var3];
         boolean var10000;
         if(var1.b.size() >= a) {
            System.out.println("[Error] The maximum number of passwords has been exceeded!");
            var10000 = false;
         } else {
            var1.b.add(var5);
            System.out.println("[Info] password (" + var5 + ") added successfully.");
            var10000 = true;
         }
         if(!var10000) {
            var2 = false;
         }
      }
   }
}
```

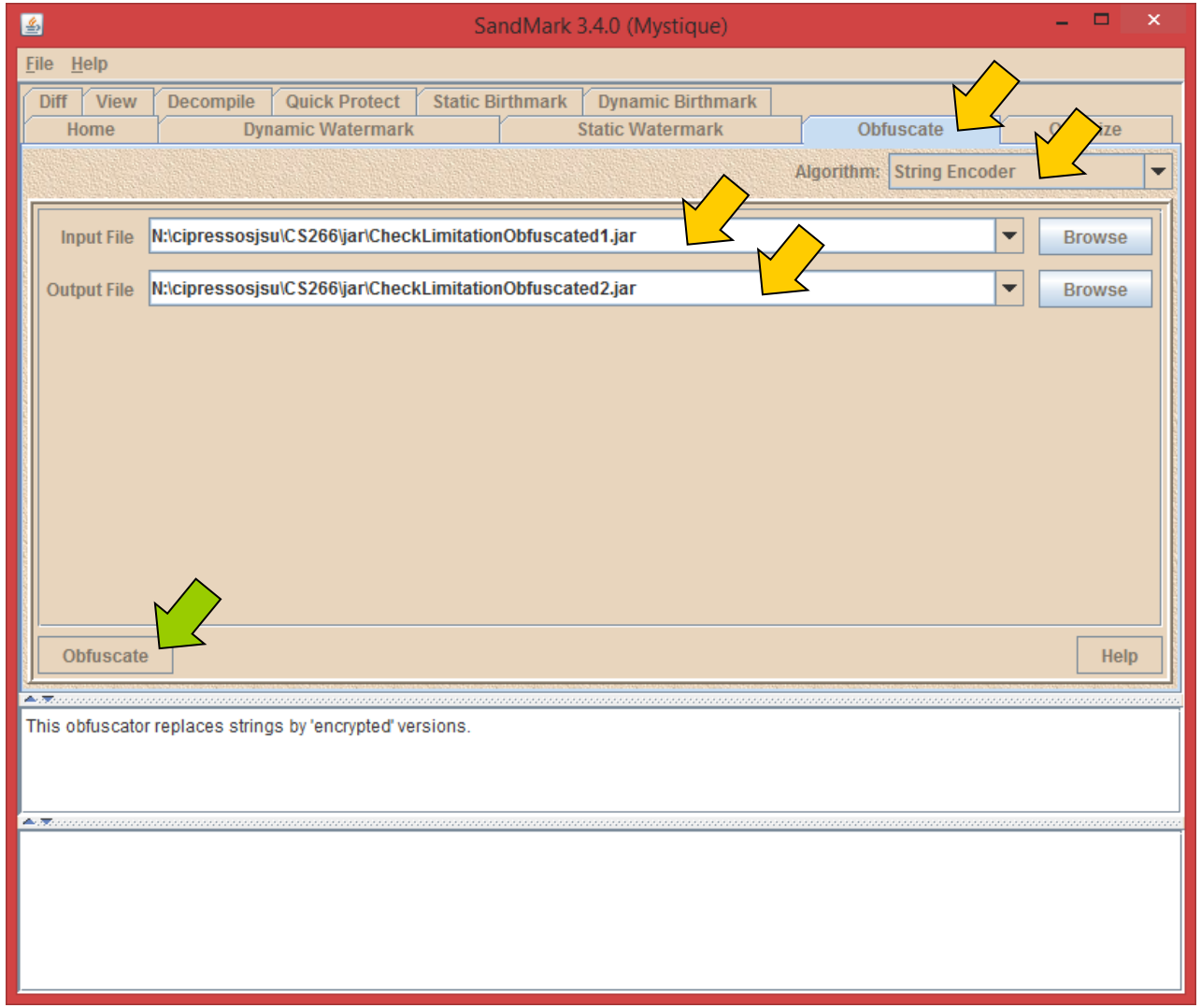
Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



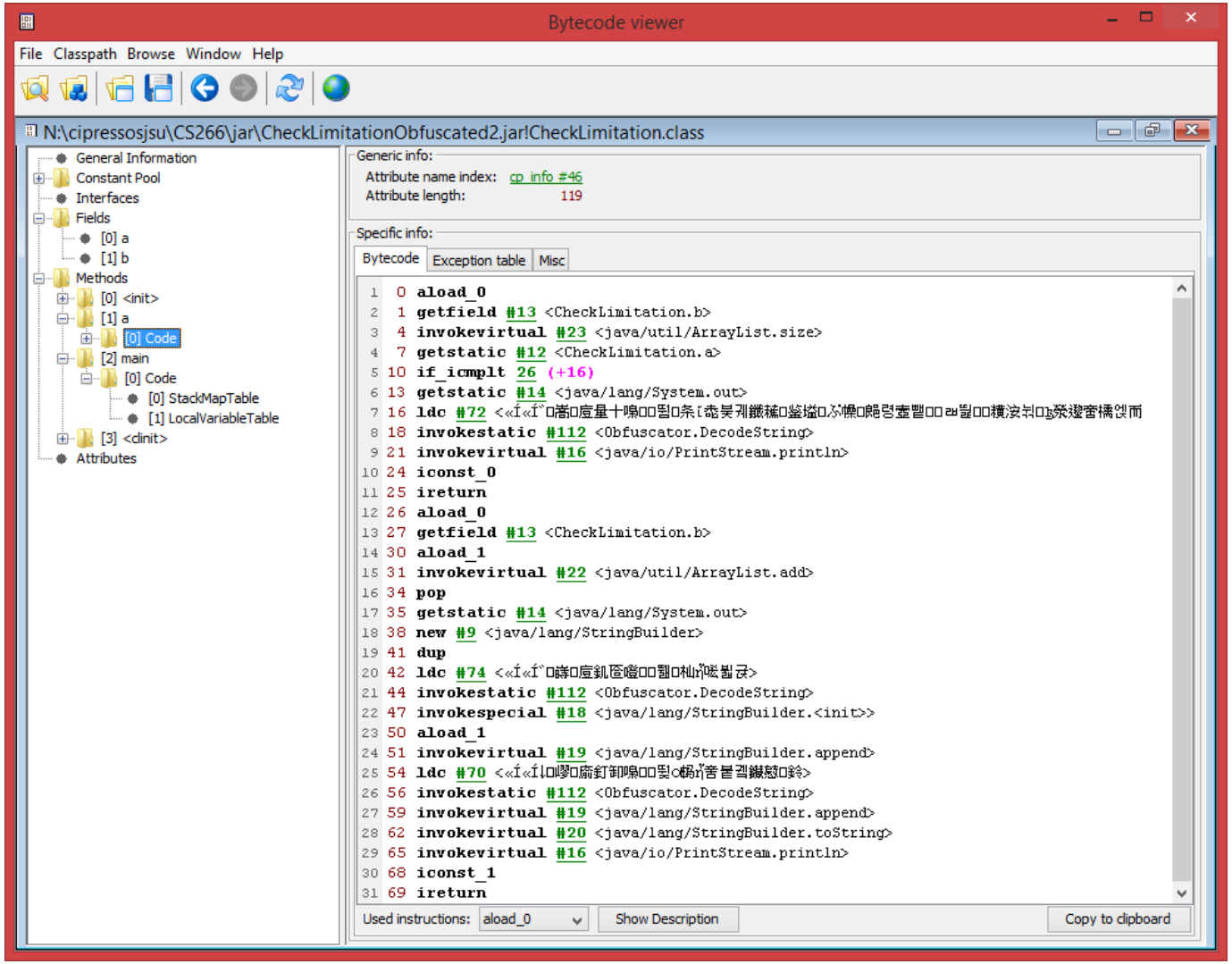
Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information



Applying Anti-Reversing Techniques to Java Bytecode

Eliminating Symbolic Information

Jad.zip

CheckLimitationObfuscated2.jar

```
1 import java.io.PrintStream;
2 import java.util.ArrayList;
3
4 public class CheckLimitation {
5     public CheckLimitation() {
6         b = new ArrayList();
7     }
8     private boolean a(String arg0) {
9         if(b.size() >= a) {
10            System.out.println(Obfuscator.DecodeString(
              "\253\315\253\315\uFF9E\u2A3D\u5D69\u2AA5\u3884\u91CF\u5341\u5604\uDF5B\uA902\uB6C8\u
              5\u0EBA\uDDF4\u313C\uB751\uFA9D\u166C\u42A3\u6D1D\uB25A\uA15E\u026E\u6ECE\u908C\u557B
11            return false;
12        } else {
13            b.add(arg0);
14            System.out.println((new StringBuilder(Obfuscator.DecodeString("\253\315\253\315\uFF9E
              arg0).append(Obfuscator.DecodeString("\253\315\253\315\uFFEC\u2A58\u5D7A\u2AB3\u388F\
15            return true;
16        }
17    }
18    public static void main(String arg0[]) {
19        CheckLimitation checklimitation = new CheckLimitation();
20        boolean flag = true;
21        for(int i = 0; i < arg0.length && flag; i++) {
22            String s = arg0[i];
23            CheckLimitation checklimitation1 = checklimitation;
24            if(checklimitation.b.size() >= a) {
25                System.out.println(Obfuscator.DecodeString(
                  "\253\315\253\315\uFF9E\u2A3D\u5D69\u2AA5\u3884\u91CF\u5341\u5604\uDF5B\uA902\uB6
                  uBE95\u0EBA\uDDF4\u313C\uB751\uFA9D\u166C\u42A3\u6D1D\uB25A\uA15E\u026E\u6ECE\u90
26                if(false)
27                    continue;
28            } else {
29                checklimitation1.b.add(s);
30                System.out.println((new StringBuilder(Obfuscator.DecodeString("\253\315\253\315\u
                  append(s).append(Obfuscator.DecodeString("\253\315\253\315\uFFEC\u2A58\u5D7A\u2AB
                  toString());
31                if(true)
32                    continue;
33            }
34            flag = false;
35        }
36
37    }
38    private static int a = 5;
39    private ArrayList b;
40 }
```

Applying Anti-Reversing Techniques to Java Bytecode

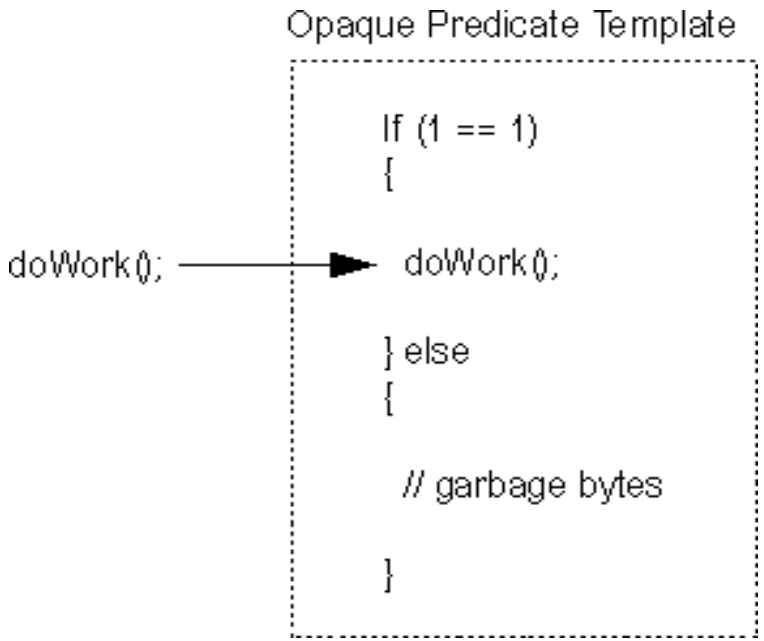
Obfuscating The Program

- One of the most popular, and fragile, techniques for preventing decompilation involves the use of *opaque predicates*:
  - By introducing false ambiguities into the control flow of a program we may trick a decompiler into traversing garbage bytes that are masquerading as logic contained in an *else* clause.
  - `if ( 1 == 1 )` and `if ( 1 == 2 )` are opaque predicates because the first always evaluates to true, and the second always to false.
    - "Branches that appear to be conditional but are really not." [5]
  - The key to preventing decompilation with opaque predicates is to insert invalid instructions in the *else* branch of an always-true predicate or the if-body of an always false predicate.

Applying Anti-Reversing Techniques to Java Bytecode

Obfuscating The Program - Opaque Predicates

- Since the invalid instructions will never be reached during normal operation of the program there is no impact on the program's operation.
- A naïve decompiler will evaluate both possibilities of an opaque predicate and fail on attempting to decompile the invalid, unreachable “instructions”.



Applying Anti-Reversing Techniques to Java Bytecode

Obfuscating The Program – Java Bytecode Verifier

- Unfortunately, opaque predicates, often used in protecting machine code from disassembly, cannot be used with Java bytecode because of the presence of the Java Bytecode Verifier in the JVM.
  - The JVM verifies bytecode before execution via single-pass static analysis.
  - Therefore invalid instructions/garbage bytes are likely to be caught.
- [31] documents the following checks made by the Java Bytecode Verifier:
  - Type correctness: arguments of an instruction, whether on the stack or in registers, should always be of the type expected by the instruction.
  - No stack overflow or underflow*:* instructions which remove items from the stack should not do so when the stack is empty. Also, instructions should not attempt to push items on the stack when the stack is full.

Applying Anti-Reversing Techniques to Java Bytecode

Obfuscating The Program – Java Bytecode Verifier

- Register initialization: Within a single method any use of a register must come after the initialization of that register. That is, there should be at least one store operation to that register before a load operation on that register.
- Object initialization: Creation of object instances must always be followed by a call to one of the possible initialization (constructor) methods for that object before it can be used.
- Access control: Method calls, field accesses, and class references must always adhere to the Java visibility policies for that method, field, or reference (e.g., private, protected, public).

- Recall that bytecode targeted to run on a JVM may have been generated by something other than the accompanying Java compiler (even dynamically).
  - This actuality highlights the need for bytecode verification.

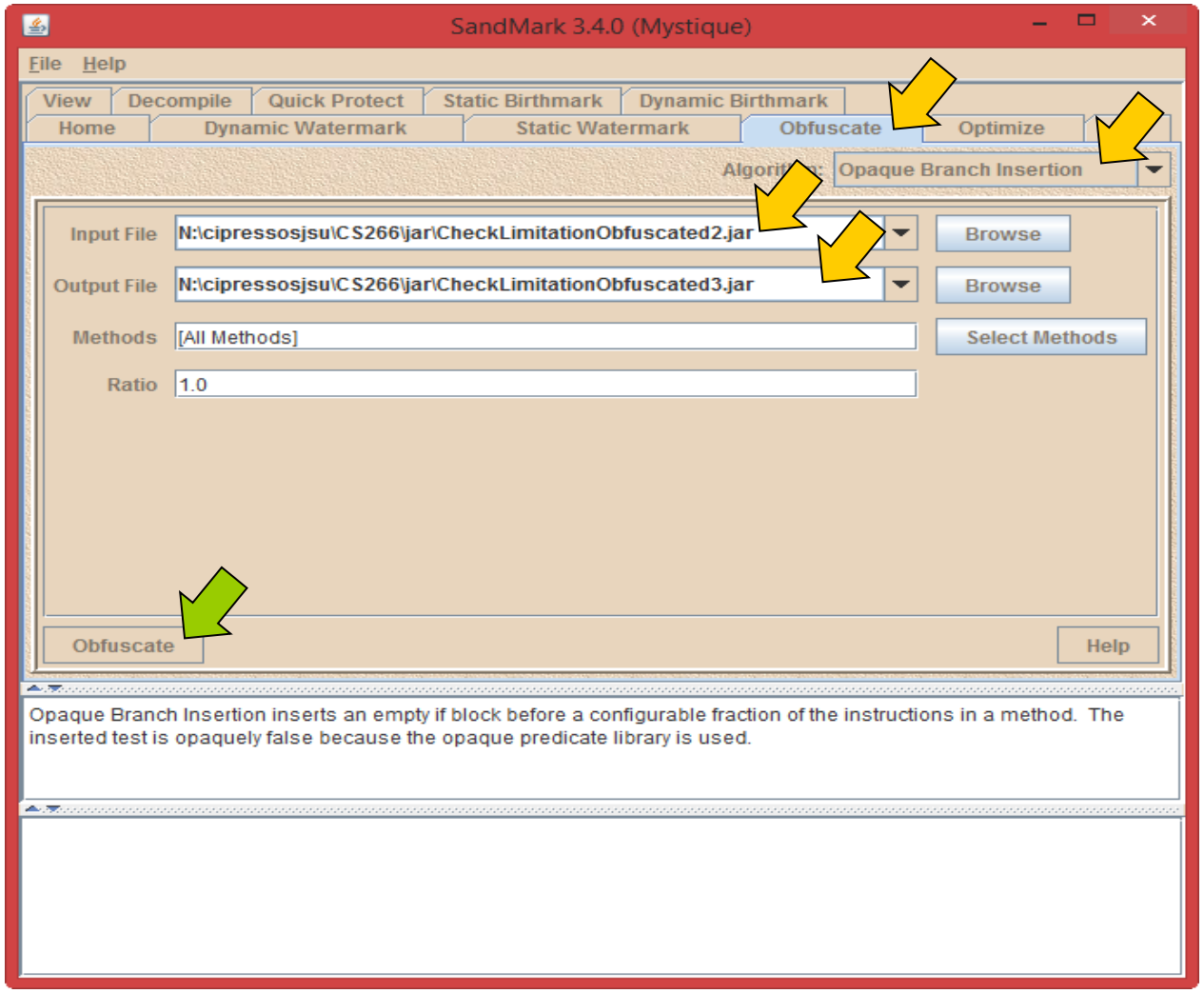
Applying Anti-Reversing Techniques to Java Bytecode

Obfuscating The Program - Opaque Predicates

- Opaque predicates do remain viable for machine code, though there is some evidence that good disassemblers, such as IDA Pro, do check for rudimentary opaque predicates [5].
- SandMark's creators claim that the presence of opaque predicates in bytecode, without garbage bytes of course, makes decompilation more difficult.
- SandMark implements several different algorithms for sprinkling opaque predicates throughout bytecode.
  - For example, the opaque branch insertion obfuscation claims to "insert jumps into a method via opaque predicates so that the control flow graph is irreducible. This inhibits decompilation."

Applying Anti-Reversing Techniques to Java Bytecode

Obfuscating The Program - Opaque Predicates



Applying Anti-Reversing Techniques to Java Bytecode

Obfuscating The Program - Opaque Predicates

Jad.zip

CheckLimitationObfuscated3.jar

CheckLimitation.jad - Notepad

File Edit Format View Help

```
    public static void main(String arg0[])
    {
        CheckLimitation checklimitation;
        int i;
        boolean flag = false;
        checklimitation = new CheckLimitation();
        i = 1;
        false;
        int k;
        k = i - 1;
        k = i * k;
        k %= 2;
        if((((long)k != (lon
            k++;
        int j;
        j;
_L11:
        j;
        arg0.length;
        int l = i - 1;
```

Command Prompt

```
n:\cipressosjsu\CS266\jar>jar -xvf CheckLimitationObfuscated3.jar
 inflated: META-INF/MANIFEST.MF
 inflated: CheckLimitation.class
 inflated: Obfuscator.class

n:\cipressosjsu\CS266\jar>jad -v CheckLimitation.class
Parsing CheckLimitation.class... Generating CheckLimitation.jad
Method <init>
Method a
Method main
Couldn't fully decompile method main
Method <clinit>

n:\cipressosjsu\CS266\jar>notepad CheckLimitation.jad

n:\cipressosjsu\CS266\jar>
```

End