Java: Making reverse engineering difficult

Java is a programming language invented by Sun Microsystems in 1995, who were later acquired by Oracle[1] in 2009. Java is an interesting language, as it is compiled (i.e. requires an intermediary step between written code → executable code), but is not native (i.e. produces a platform-specific direct executable, eg. in the ELF format). Instead, Sun opted to invent their own micro-instruction-set, which are executable on a virtual machine, uncreatively named the "Java Virtual Machine", or JVM.

When you compile java source code, it is then continuously analyzed and translated into what is known as the class-file-format, a very specifically laid out format used by the JVM to execute programs (relatively analogous to an ELF). It also leaves a large amount of debug information in the binary, such as line numbers, labels, variable names, etc.

Since Java executes off a heavily RISC (Reduced Instruction-Set Computer) inspired instruction-set, with only ~115 unique instructions[2], it is a much easier feat to reverse engineer JVM Bytecode back into valid Java (compared to, for example, x86 which is a CISC[3] design) which produces the same result - i.e. decompilation.

This has been known for a while, so many automatic decompilers exist nowadays, which are nowadays able to effectively produce almost identical source files from just the compiled class. This poses a problem for people who do not want their source code to be exposed, which will be talked on later.

To solve this problem, obfuscators were born in the Java ecosystem - programs whose sole purpose is to edit other class file binaries such that they become as difficult to decompile as possible. This includes: stripping debug information (as mentioned earlier), encrypting strings, changing control flow to large complex switches with goto's to delinearize the program, etc.

Some of these changes can be reversed (tools to fix goto delinearization exist, etc.) but some changes cannot (eg. once variable names are changed, there is no way to get the original names back). However, with sufficient human intervention, it is possible to sit down and slowly figure out how a program works, naming variables yourself as you see fit.

There is one interesting target of exploitation in obfuscation, which I took a keen interest in for my Something Awesome - disparities between the rules of Java and the rules of the JVM - such as that identifiers aren't allowed special characters in Java, but when compiled, the JVM is perfectly happy with the contrary. Exploiting these mechanics often causes decompilers to behave in unexpected ways due to their assumptions, even if the program still executes normally.

---

[1] *shudders*

[2] Not counting single byte variants, as of SE8 which this SA is based on

[3] Complex Instruction-Set Computer

My Something-Awesome Obfuscator

Located: https://github.com/insou22/sa-obf

My SA-Obfuscator takes in an input class (compiled) file, and outputs another obfuscated class-file which should execute identically to the input, but is much harder to reverse engineer. I have used multiple techniques to achieve this, ordering in complexity:

- Line number removal
- “This” overwrite manipulation
- Symbol name manipulation:
    - Method names
    - Field names
    - Method parameters
    - Method local variables
- “The Fuckinator”

I will cover each individually.

Line Number Removal

```java
package co.insou.obfuscator.transformers;

import org.objectweb.asm.tree.AbstractInsnNode;
import org.objectweb.asm.tree.ClassNode;
import org.objectweb.asm.tree.LineNumberNode;

import java.util.stream.Stream;

public class LineNumberObfuscator implements ClassTransformer {

    @Override
    public ClassNode transform(ClassNode node)
    {
        node.methods.forEach(method ->
        {
            Stream.of(method.instructions.toArray())
                    .filter(this::isLineNumberInstruction)
                    .forEach(method.instructions::remove);
        });

        return node;
    }

    private boolean isLineNumberInstruction(AbstractInsnNode insn)
    {
        return insn instanceof LineNumberNode;
    }

}
```

The line number removal implementation is quite simple, but it was the initial steps to get there that were quite difficult. Before this obfuscation was performed, the class is loaded as a byte array from the file, parsed and stored into a tree.

Line-number obfuscation is a very simple technique, but still has an effect. Decompilers no longer use line number markers to figure out which lines go where, they simply use their context and surroundings to generate the most likely source lines to have compiled to that particular bytecode.

However, the JVM uses line-number markers to display stack-traces after an exception has occurred. If all the line-number markers have been stripped from the binary, it can no longer mark where it came from. This can make reverse-engineering a binary more difficult.

Examples follow on the next page.

Source code:

```
class Test {

    public static void main(String[] args)
    {
        throw new RuntimeException("lol u thought");
    }

}
```

Compiled with javac and run:

```
…/IdeaProjects/obfuscator-asm/l_example 100%
› java Test
Exception in thread "main" java.lang.RuntimeException: lol u thought
        at Test.main(Test.java:5)
```

Obfuscated, and run:

```
…/IdeaProjects/obfuscator-asm/l_example 100%
› java -jar ../target/obfuscator-asm_v1.0.jar -i Test.class -o obf/Test.class -l
Successfully obfuscated line numbers
Size before: 449
Size after:  407
…/IdeaProjects/obfuscator-asm/l_example 100%
› cd obf
…/obfuscator-asm/l_example/obf 100%
› java Test
Exception in thread "main" java.lang.RuntimeException: lol u thought
        at Test.main(Test.java)
```

Key observation: Exception before obfuscation shows:
“at Test.main(Test.java:5)”

Exception after obfuscation:
“at Test.main(Test.java)”

Disassembled before obfuscation:

```
…/IdeaProjects/obfuscator-asm/l_example  100%
 javap -c -l Test
Compiled from "Test.java"
class Test {
  Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 1: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   LTest;

  public static void main(java.lang.String[]);
    Code:
       0: new           #2                  // class java/lang/RuntimeException
       3: dup
       4: ldc           #3                  // String lol u thought
       6: invokespecial #4                  // Method java/lang/RuntimeException."<init>":(Ljava/lang/String;)V
       9: athrow
    LineNumberTable:
      line 5: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  args   [Ljava/lang/String;
}
```

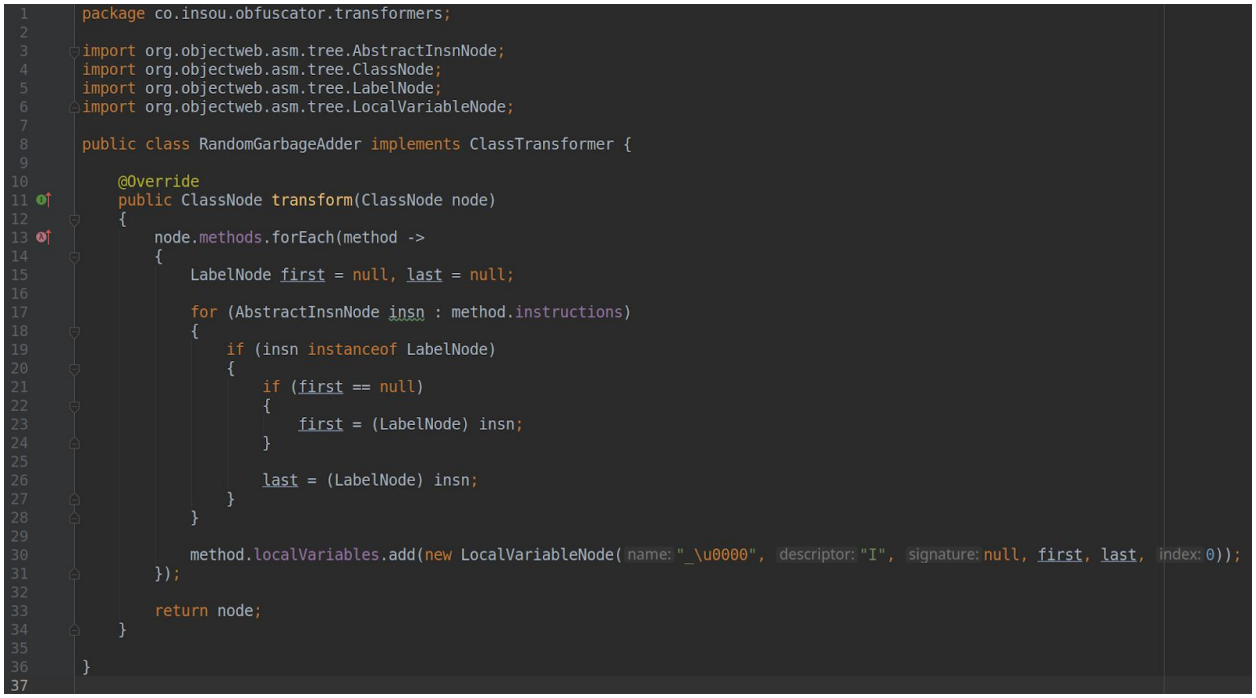
Disassembled after obfuscation:

```
…/obfuscator-asm/l_example/obf  100%
 javap -c -l Test
Compiled from "Test.java"
class Test {
  Test();
    Code:
       0: aload_0
       1: invokespecial #9                  // Method java/lang/Object."<init>":()V
       4: return
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   LTest;

  public static void main(java.lang.String[]);
    Code:
       0: new           #15                 // class java/lang/RuntimeException
       3: dup
       4: ldc           #17                 // String lol u thought
       6: invokespecial #20                 // Method java/lang/RuntimeException."<init>":(Ljava/lang/String;)V
       9: athrow
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  args   [Ljava/lang/String;
}
```

Key observation:
LineNumberTable marking lines with code locations only exists before obfuscation.

This overwrite manipulation



```java
package co.insou.obfuscator.transformers;

import org.objectweb.asm.tree.AbstractInsnNode;
import org.objectweb.asm.tree.ClassNode;
import org.objectweb.asm.tree.LabelNode;
import org.objectweb.asm.tree.LocalVariableNode;

public class RandomGarbageAdder implements ClassTransformer {

    @Override
    public ClassNode transform(ClassNode node)
    {
        node.methods.forEach(method ->
        {
            LabelNode first = null, last = null;

            for (AbstractInsnNode insn : method.instructions)
            {
                if (insn instanceof LabelNode)
                {
                    if (first == null)
                    {
                        first = (LabelNode) insn;
                    }

                    last = (LabelNode) insn;
                }
            }

            method.localVariables.add(new LocalVariableNode( name: "_\u0000",  descriptor: "I",  signature: null, first, last,  index: 0));
        });

        return node;
    }

}
```

“this” overwrite manipulation is a technique I came up with.

```java
public class Test {

    public static void main(String[] args)
    {
        new Test(3, 4);
    }

    private int multiplier = 2;
    private int adder = 5;
    private int times = 3;

    public Test(int x, int y)
    {
        int multiplied = this.multiplier * x * y;
        int added = this.adder + multiplied;

        printTimes(added, this.times);
    }

    private void printTimes(int number, int times)
    {
        int i = 0;

        while (i < times)
        {
            System.out.println(number);

            i++;
        }
    }

}
```

Standard javac compilation:

```
…/IdeaProjects/obfuscator-asm/t_example  100%
javac -g Test.java
…/IdeaProjects/obfuscator-asm/t_example  100%
java Test
29
29
29
```

After obfuscation (still runs identically):

```
…/IdeaProjects/obfuscator-asm/t_example  100%
java -jar ../target/obfuscator-asm_v1.0.jar -i Test.class -o obf/Test.class -t
Successfully obfuscated this names
Size before: 915
Size after:  951
…/IdeaProjects/obfuscator-asm/t_example  100%
cd obf
…/obfuscator-asm/t_example/obf  100%
java Test
29
29
29
```

Before obfuscation:

```
public Test(int, int);
  LineNumberTable:
    line 13: 0
    line 8: 4
    line 9: 9
    line 10: 14
    line 14: 19
    line 15: 28
    line 17: 36
    line 18: 46
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0      47     0  this   LTest;
        0      47     1     x   I
        0      47     2     y   I
       28      19     3 multiplied   I
       36      11     4 added   I

private void printTimes(int, int);
  LineNumberTable:
    line 22: 0
    line 24: 2
    line 26: 7
    line 28: 14
    line 30: 20
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0      21     0  this   LTest;
        0      21     1 number   I
        0      21     2 times   I
        2      19     3     i   I
```

After obfuscation:

```
public Test(int, int);
  LineNumberTable:
    line 13: 0
    line 8: 4
    line 9: 9
    line 10: 14
    line 14: 19
    line 15: 28
    line 17: 36
    line 18: 46
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0      47     0  this   LTest;
        0      47     1     x   I
        0      47     2     y   I
       28      19     3 multiplied   I
       36      11     4 added   I
        0      47     0     _   I

private void printTimes(int, int);
  LineNumberTable:
    line 22: 0
    line 24: 2
    line 26: 7
    line 28: 14
    line 30: 20
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0      21     0  this   LTest;
        0      21     1 number   I
        0      21     2 times   I
        2      19     3     i   I
        0      21     0     _   I
```

Key observation:
There is an extra variable at the end of the LocalVariableTables with a name of '_' + a null byte after it, which is placed in slot 0 in the obfuscated version.

This conflicts with what the "this" variable is stored in (always slot 0). The jvm will essentially ignore this extra unused variable at runtime, but decompilers will generally run through the binary and read everything in the local variable table, overwriting whatever came before it as all the slots *should* be unique. This often causes every reference of "this" to be renamed to underscore null byte, causing lovely undefined behaviour.

Decompiler behaviour:
jd-gui (most popular java decompiler) - completely breaks, instead opts to print bytecode as it cannot decompile at all. Also breaks some formatting in the GUI.

```
5   public static void main(int _\000) { new Test(3, 4); }



    private int adder;

    private int times;


    public Test(int x, int y) { // Byte code:
      //   0: aload_0
      //   1: invokespecial <init> : ()V
      //   4: aload_0
      //   5: iconst_2
      //   6: putfield multiplier : I
      //   9: aload_0
      //   10: iconst_5
      //   11: putfield adder : I
      //   14: aload_0
      //   15: iconst_3
      //   16: putfield times : I
      //   19: aload_0
      //   20: getfield multiplier : I
      //   23: iload_1
      //   24: imul
      //   25: iload_2
      //   26: imul
      //   27: istore_3
      //   28: aload_0
      //   29: getfield adder : I
      //   32: iload_3
      //   33: iadd
      //   34: istore #4
      //   36: aload_0
      //   37: iload #4
      //   39: aload_0
      //   40: getfield times : I
      //   43: invokespecial printTimes : (II)V
      //   46: return
      // Line number table:
      //   Java source line number -> byte code offset
      //   #13   -> 0
      //   #8    -> 4
      //   #9    -> 9
      //   #10   -> 14
      //   #14   -> 19
      //   #15   -> 28
      //   #17   -> 36
      //   #18   -> 46
      // Local variable table:
```

Fernflower (second-most popular): Is tricked by the overwrite, prints box character for null byte, breaks main's args parameter, but is still able to decompile.

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//

public class Test {
    private int multiplier = 2;
    private int adder = 5;
    private int times = 3;

    public static void main(String[] _□) { new Test( x: 3,  y: 4); }

    public Test(int x, int y) {
        int multiplied = _□.multiplier * x * y;
        int added = _□.adder + multiplied;
        _□.printTimes(added, _□.times);
    }

    private void printTimes(int number, int times) {
        for(int i = 0; i < times; ++i) {
            System.out.println(number);
        }

    }
}

```

Procyon (not very popular, but most powerful decompiler): Is not tricked.

```
public class Test
{
    private int multiplier;
    private int adder;
    private int times;

    public static void main(final String[] args) {
        new Test(3, 4);
    }

    public Test(final int x, final int y) {
        this.multiplier = 2;
        this.adder = 5;
        this.times = 3;
        final int multiplied = this.multiplier * x * y;
        final int added = this.adder + multiplied;
        this.printTimes(added, this.times);
    }

    private void printTimes(final int number, final int times) {
        for (int i = 0; i < times; ++i) {
            System.out.println(number);
        }
    }
}

```

Symbol name manipulation

I will be using the same example as previously for the “this overwrite manipulation”, and using all obfuscation methods mentioned until now in parallel (i.e. line numbers, this overwrite, symbol names).
The source code for this module is much longer, so I will not paste a photo of it here, but it is viewable on git in the SymbolRenamer class. Essentially, it renames all method names, field names, method params, and local variable names to a combination of underscores, null bytes and zero-width spaces (pure evil).

The same example is run with -l, -t, and -s flags to enable all the modules, and it is clear it executes identically.

```
…/IdeaProjects/obfuscator-asm/s_example  100%
 java -jar ../target/obfuscator-asm_v1.0.jar -i Test.class -o obf/Test.class -l -t -s
Successfully obfuscated line numbers
Successfully obfuscated symbol names
Successfully obfuscated this names
Size before: 915
Size after:  875
…/IdeaProjects/obfuscator-asm/s_example  100%
 cd obf
…/obfuscator-asm/s_example/obf  100%
 java Test
29
29
29
```

On the next page, the newly obfuscated version is disassembled with javap.

```
…/obfuscator-asm/s_example/obf  100%
 javap -p -c -l
Error: no classes specified
…/obfuscator-asm/s_example/obf  100%
 javap -p -c -l Test
Compiled from "Test.java"
public class Test {
  private int ;

  private int ;

  private int ;

  public static void main(java.lang.String[]);
    Code:
       0: new           #2                  // class Test
       3: dup
       4: iconst_3
       5: iconst_4
       6: invokespecial #15                 // Method "<init>":(II)V
       9: pop
      10: return
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      11     0     _   [Ljava/lang/String;
          0      11     0    _   I

  public Test(int, int);
    Code:
       0: aload_0
       1: invokespecial #21                 // Method java/lang/Object."<init>":()V
       4: aload_0
       5: iconst_2
       6: putfield      #23                 // Field "":I
       9: aload_0
```

Note: fields at top of class have no names in javap

Continued on the next page.

```
public Test(int, int);
  Code:
     0: aload_0
     1: invokespecial #21                 // Method java/lang/Object."<init>":()V
     4: aload_0
     5: iconst_2
     6: putfield      #23                 // Field "":I
     9: aload_0
    10: iconst_5
    11: putfield      #25                 // Field "":I
    14: aload_0
    15: iconst_3
    16: putfield      #27                 // Field "":I
    19: aload_0
    20: getfield      #23                 // Field "":I
    23: iload_1
    24: imul
    25: iload_2
    26: imul
    27: istore_3
    28: aload_0
    29: getfield      #25                 // Field "":I
    32: iload_3
    33: iadd
    34: istore        4
    36: aload_0
    37: iload         4
    39: aload_0
    40: getfield      #27                 // Field "":I
    43: invokespecial #30                 // Method "":(II)V
    46: return
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0      47     0  this   LTest;
        0      47     1     _   I
        0      47     2     _   I
       28      19     3     _   I
       36      11     4     _   I
        0      47     0    _   I

private void (int, int);
  Code:
     0: iconst_0
     1: istore_3
     2: iload_3
     3: iload_2
     4: if_icmpge     20
     7: getstatic     #38                 // Field java/lang/System.out:Ljava/io/PrintStream;
    10: iload_1
    11: invokevirtual #44                 // Method java/io/PrintStream.println:(I)V
    14: iinc          3, 1
    17: goto          2
    20: return
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
        0      21     0  this   LTest;
        0      21     1     _   I
        0      21     2     _   I
        2      19     3     _   I
        0      21     0    _   I
```

Based on the disassembly above, it is clear that the variable tables are completely magled. Everything has the same name of an _, and variables are conflicting with each other.

However - it still executes as per normal, simply making it harder to reverse engineer.

Decompiler behaviour:
jd-gui (most popular java decompiler) - not a chance.

```
public class Test {
  private int \000\000\000\000\u200B\000\000\000\u200B\u200B;

  private int \u200B\000\000\u200B\u200B\u200B\u200B\000\000\000;

  private int \000\u200B\u200B\u200B\000\000\u200B\000\u200B\u200B;

  public static void main(int _\000) { new Test(3, 4); }

  public Test(int _, int _) { // Byte code:
    //   0: aload_0
    //   1: invokespecial <init> : ()V
    //   4: aload_0
    //   5: iconst_2
    //   6: putfield \000\000\000\000\u200B\000\000\000\u200B\u200B : I
    //   9: aload_0
    //   10: iconst_5
    //   11: putfield \u200B\000\000\u200B\u200B\u200B\u200B\000\000\000 : I
    //   14: aload_0
    //   15: iconst_3
    //   16: putfield \000\u200B\u200B\u200B\000\000\u200B\000\u200B\u200B : I
    //   19: aload_0
    //   20: getfield \000\000\000\000\u200B\000\000\000\u200B\u200B : I
    //   23: iload_1
    //   24: imul
    //   25: iload_2
    //   26: imul
    //   27: istore_3
    //   28: aload_0
    //   29: getfield \u200B\000\000\u200B\u200B\u200B\u200B\000\000\000 : I
    //   32: iload_3
    //   33: iadd
    //   34: istore #4
    //   36: aload_0
    //   37: iload #4
    //   39: aload_0
    //   40: getfield \000\u200B\u200B\u200B\000\000\u200B\000\u200B\u200B : I
    //   43: invokespecial \u200B\u200B\000\u200B\u200B\u200B\000\u200B\u200B\000 : (II)V
    //   46: return
    // Local variable table:
    //   start length    slot name descriptor
    //   0     47   0    this LTest;
    //   0     47   1    _    I
    //   0     47   2    _    I
    //   28    19   3    _    I
    //   36    11   4    _    I
    //   0     47   0    _\000     I }

  private void \u200B\u200B\000\u200B\u200B\u200B\000\u200B\u200B\000(int _, int _) {
    for (int _ = 0; _ < _; _++)
      System.out.println(_);
  }
}
```

Fernflower (second-most popular): Is able to decompile but gets fully jeffed.

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//

public class Test {
    private int □□□□□□□ = 2;
    private int □□□□□ = 5;
    private int □□□□ = 3;

    public static void main(String[] _□) { new Test( □ 3, □ 4); }

    public Test(int _, int _) {
        int _ = _□.□□□□□□□ * _ * _;
        int _ = _□.□□□□□ + _;
        _□.□□□(_, _□.□□□□);
    }

    private void _□□□/* $FF was: □□□*/(int _, int _) {
        for(int _ = 0; _ < _; ++_) {
            System.out.println(_);
        }

    }
}
```

Procyon (not very popular, but most powerful decompiler): The most successful at decompiling it, still very hard to read. Also didn't seem to expect method parameters having identical names.

```
public class Test
{
    private int \u0000\u0000\u0000\u0000\u200b\u0000\u0000\u0000\u200b\u200b;
    private int \u200b\u0000\u0000\u200b\u200b\u200b\u0000\u0000\u0000\u0000;
    private int \u0000\u200b\u200b\u200b\u0000\u0000\u200b\u0000\u200b\u200b;

    public static void main(final String[] _) {
        new Test(3, 4);
    }

    public Test(final int _, final int _) {
        this.\u0000\u0000\u0000\u0000\u200b\u0000\u0000\u0000\u200b\u200b = 2;
        this.\u200b\u0000\u0000\u200b\u200b\u200b\u0000\u0000\u0000\u0000 = 5;
        this.\u0000\u200b\u200b\u200b\u0000\u0000\u200b\u0000\u200b\u200b = 3;
        final int _2 = this.\u0000\u0000\u0000\u0000\u200b\u0000\u0000\u0000\u200b\u200b * _ * _;
        final int _3 = this.\u200b\u0000\u0000\u200b\u200b\u200b\u200b\u0000\u0000\u0000 + _2;
        this.\u200b\u200b\u0000\u200b\u200b\u200b\u0000\u200b\u200b\u0000(_3, this.\u0000\u200b\u200b\u200b\u0000\u0000\u200b\u0000\u200b\u200b);
    }

    private void \u200b\u200b\u0000\u200b\u200b\u200b\u0000\u200b\u200b\u0000(final int _, final int _) {
        for (int _2 = 0; _2 < _; ++_2) {
            System.out.println(_);
        }
    }
}
```

Before moving onto **The Fuckinator**, it is interesting to demonstrate the effects of this obfuscation on a larger file. This is a quadratic equation solver in Java I wrote over 4 years ago: https://gist.github.com/insou22/16a8feefdbba88842d1795beda30c37d
It is not written very well, but serves to illustrate the effects of obfuscation in a larger project. This is Quad.class decompiled without any obfuscation:

```
133         if (!this.simplify) {
134             System.out.print("Would you also like to simplify? ");
135             if ((new Scanner(System.in)).nextBoolean()) {
136                 this.simplify(a, b, c);
137             }
138         }
139 
140     }
141 
142     private List<Integer> factors(int n) {
143         if (n < 0) {
144             n *= -1;
145         }
146 
147         int upperlimit = (int)Math.sqrt((double)n);
148         List<Integer> factors = new ArrayList();
149 
150         for(int i = 1; i <= upperlimit; ++i) {
151             if (n % i == 0) {
152                 factors.add(i);
153                 factors.add(i * -1);
154                 if (i != n / i) {
155                     factors.add(n / i);
156                     factors.add(n / i * -1);
157                 }
158             }
159         }
160 
161         Collections.sort(factors);
162         return factors;
163     }
164 
165     private boolean hasHcf(int x, int y) { return this.gcd(x, y) != 0; }
168 
169     private int gcd(int x, int y) {
170         BigInteger b1 = BigInteger.valueOf((long)x);
171         BigInteger b2 = BigInteger.valueOf((long)y);
172         BigInteger gcd = b1.gcd(b2);
173         return gcd.intValue();
174     }
175 
176     private void debug(String text) {
177         if (this.debug) {
178             this.print(">> " + text);
179         }
180 
181     }
182 
183     private void print(String text) { System.out.println(text); }
186 }
187 
```

Looks like it easily could have been the exact source.

Here is Quad.class decompiled after my obfuscations:

```
133         if (!_□.□□□□□□) {
134             System.out.print("Would you also like to simplify? ");
135             if ((new Scanner(System.in)).nextBoolean()) {
136                 _□.□□□□□□(_, _, _);
137             }
138         }
139 
140     }
141 
142     private List<Integer> _□□□□□/* $FF was: □□□□□□*/(int _) {
143         if (_ < 0) {
144             _ *= -1;
145         }
146 
147         int _ = (int)Math.sqrt((double)_);
148         List<Integer> _ = new ArrayList();
149 
150         for(int _ = 1; _ <= _; ++_) {
151             if (_ % _ == 0) {
152                 _.add(_);
153                 _.add(_ * -1);
154                 if (_ != _ / _) {
155                     _.add(_ / _);
156                     _.add(_ / _ * -1);
157                 }
158             }
159         }
160 
161         Collections.sort(_);
162         return _;
163     }
164 
165 @   private boolean _□□□□□/* $FF was: □□□□□*/(int _, int _) {
166         return _□.□□□□□(_, _) != 0;
167     }
168 
169     private int _□□□□□/* $FF was: □□□□□*/(int _, int _) {
170         BigInteger _ = BigInteger.valueOf((long)_);
171         BigInteger _ = BigInteger.valueOf((long)_);
172         BigInteger _ = _.gcd(_);
173         return _.intValue();
174     }
175 
176     private void _□□□□□□/* $FF was: □□□□□□□*/(String _) {
177         if (_□.□□□) {
178             _□.□□(">> " + _);
179         }
180 
181     }
182 
183     private void _□□/* $FF was: □□*/(String _) { System.out.println(_); }
186 }
187 
```

Almost unusable.

The important thing to remember is that these binaries execute exactly the same in the JVM, and the binary size is very comparable, so there is very little cost to making these transformations. Unfortunately however, this assertion does not follow for…

The Fuckinator

This is by-far the most ambitious and crazy transformation I added to my obfuscator.
The Fuckinator transforms your class by:

1. Applying the Line Number Removal obfuscation
2. Applying the This Overwrite Manipulation obfuscation
3. Applying the Symbol Name Manipulation obfuscation
4. Converting the class to a single stream of raw bytes
5. Encrypting all the raw bytes
6. Building a new class from scratch which decrypts, loads, injects and calls the class bytes
7. Replacing your class with this wrapper class
8. Finally, going back to step 1 and repeating all these steps continually until the byte streams become too long to even store in a class file.

The Fuckinator turns your class into an onion, where your binary becomes a class which decrypts and loads an encrypted class which decrypts and loads an encrypted class which decrypts and loads an encrypted class… which eventually decrypts and loads your base class and executes it.

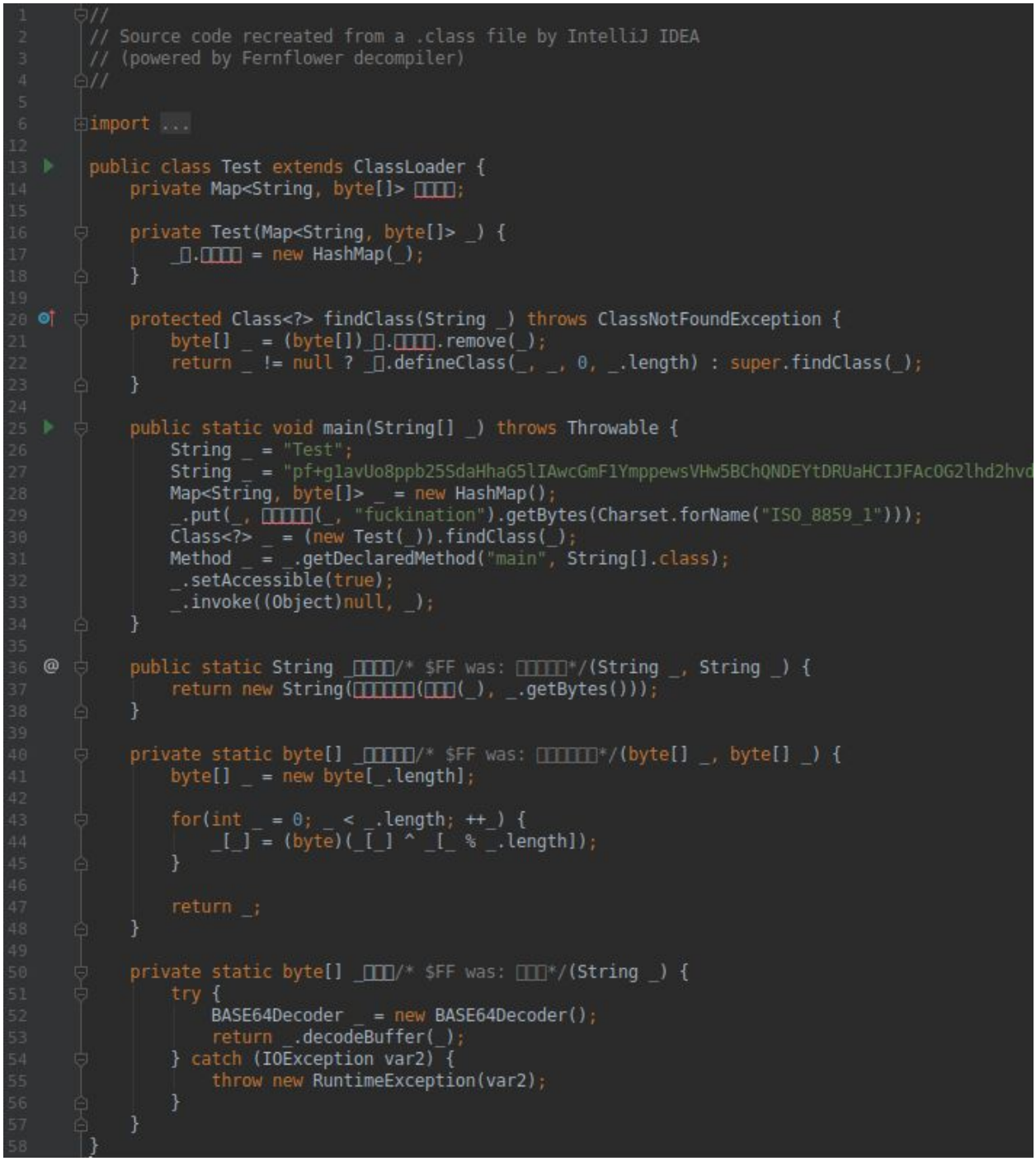
This makes it essentially unfeasible to ever reverse engineer your code without using highly specialized tools such as a custom JVM designed for heavy internal debugging, or writing an in-house tool to try and unwrap the Fuckinator Onion™.

Even once you've unwrapped the onion the whole way, you would still have to deal with all the other obfuscation techniques as highlighted above, and this can be combined with other obfuscators to make it even harder to reverse engineer.

Fuckinating the previous example once more absolutely explodes the binary size, but still executes as normal after being wrapped in an onion of 7 layers:

```
…/IdeaProjects/obfuscator-asm/fuck_example 100%
 java -jar ../target/obfuscator-asm_v1.0.jar -i Test.class -o obf/Test.class -l -t -s -fuck
Successfully obfuscated line numbers
Successfully obfuscated symbol names
Successfully obfuscated this names
Successfully fuckinated 7 times
Size before: 915
Size after:  64630
…/IdeaProjects/obfuscator-asm/fuck_example 100%
 cd obf
…/obfuscator-asm/fuck_example/obf 100%
 java Test
29
29
29
```

Decompiled with fernflower, note: line 27 is 62,000 characters long.



```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//

import ...

public class Test extends ClassLoader {
    private Map<String, byte[]> □□□□;

    private Test(Map<String, byte[]> _) {
        _□.□□□□ = new HashMap(_);
    }

    protected Class<?> findClass(String _) throws ClassNotFoundException {
        byte[] _ = (byte[])_□.□□□□.remove(_);
        return _ != null ? _□.defineClass(_, _, 0, _.length) : super.findClass(_);
    }

    public static void main(String[] _) throws Throwable {
        String _ = "Test";
        String _ = "pf+g1avUo8ppb25SdaHhaG5lIAwcGmF1YmppewsVHw5BChQNDEYtDRUaHCIJFAcOG2lhd2hvd
        Map<String, byte[]> _ = new HashMap();
        _.put(_, □□□□□(_, "fuckination").getBytes(Charset.forName("ISO_8859_1")));
        Class<?> _ = (new Test(_)).findClass(_);
        Method _ = _.getDeclaredMethod("main", String[].class);
        _.setAccessible(true);
        _.invoke((Object)null, _);
    }

    public static String _□□□□/* $FF was: □□□□□*/(String _, String _) {
        return new String(□□□□□□(□□□(_), _.getBytes()));
    }

    private static byte[] _□□□□□/* $FF was: □□□□□□*/(byte[] _, byte[] _) {
        byte[] _ = new byte[_.length];

        for(int _ = 0; _ < _.length; ++_) {
            _[_] = (byte)(_[_] ^ _[_ % _.length]);
        }

        return _;
    }

    private static byte[] _□□□/* $FF was: □□□*/(String _) {
        try {
            BASE64Decoder _ = new BASE64Decoder();
            return _.decodeBuffer(_);
        } catch (IOException var2) {
            throw new RuntimeException(var2);
        }
    }
}
```

The source code for The Fuckinator is worth reading as it is all written in raw JVM bytecode, which is no small undertaking - comparable to writing 300 lines of assembly.
It can be found here:
https://github.com/insou22/sa-obf/blob/master/src/main/java/co/insou/obfuscator/transformers/TheFuckinator.java