Java Obfuscator and Optimizer

David Armanious

Table of Contents

Purpose

The purpose of this project is to create a tool that, when invoked, will optimize and heavily obfuscate Java bytecode. One of the greatest advantages of Java is that its developers can "compile once, run everywhere" as long as each client has a JVM installed, which is accomplished through Java bytecode. One of the greatest disadvantages, however, is that one can very easily reverse engineer Java bytecode–there are a plethora of automated tools that do just this with a high success rate. The purposes of obfuscation in this project are manifold: primarily to hinder and possibly prevent the reverse engineering of compiled .class or .jar files. A secondary purpose of this project is to optimize the code so that it runs faster and mitigates some of the speed slowdown that results from certain types of obfuscation. Anyone would be able to use this tool, so that when their product is released, automatic decompilation is virtually impossible and manual reverse engineering on a large scale infeasible.

Framework

The majority of time spent on this project was debugging the underlying logic for the following classes that compose the framework of the entire project. For example, one seemingly working bug fix would break another feature but only be discovered weeks later.

The DataManager class

- Instantiates various "helper" maps to efficiently lookup related items:
    - Class internal name (String) to ClassNode instance
    - Field internal name to ClassNode owner instance
    - Method internal name to ClassNode owner instance
    - MethodNode instance to MethodInformation instance
- Constructs the class hierarchy tree
    - Uses basic ClassHierarchyTreeNode class with a Map<String, ClassHierarchyTreeNode> GLOBAL to facilitate lookups
    - Falls back on reflection when necessary
- Determines which FieldNodes are modified by which methods and the exact instructions
- Constructs the method call graph
    - Includes constructors, abstract methods, interface methods, and static initializers (these are all treated the same as bytecode)
    - Falls back on reflection when necessary
- Creates a MethodInformation instance for each non-native, non-abstract method

The MethodInformation class

- Computes the graph composed of BasicBlock and BlockEdge instances
    - Instructions serve as the delimiters of the BasicBlock
    - Adds a LabelNode psuedoinstruction at the beginning and end of its method to ensure a unique entrance (sometimes loop go back to the first instruction) and exit block
- Labels the graph depth-first and *simultaneously computes iDominance* (code)
    - Link to code

    - Dominance relationships can be inferred from iDominance relationships by simply traversing the graph through each node's immediate dominator
    - TREE edges are simply iDominance relationships
    - BACK edges do not affect iDominance
    - FORWARD and CROSS edges are set to the closest ancestor of the two blocks
- Computes the loop graph
    - Pretty straightforward segment of code; the only hitch was determining the unique parent of each loop
    - This was solved by keeping track of the previously instantiated LoopEntry and passing it to the constructor, which then loops through parents until it finds its own parent (as opposed to its sibling)
- Symbolically executes the method
    - Breadth-first traversal of the basic block graph to ensure each predecessor has been executed (excluding back edges) to properly account for PhiTemporaries
    - Saves the state (JavaStack, and Temporary[] for locals) associated with each instruction after execution
- Determines whether its method has side effects
    - If method sets any field (irrespective of how many times), it causes side effects
    - If any method it calls has side effects, it too has side effects
        - Expandable array of known side-effect-free library methods
            - Room for improvement: evaluate these dynamically; however, I was worried about the runtime necessary to do so

The Temporary abstract class and its subclasses

- Temporaries for each type of operation that can push values onto the stack performed by the JVM
    - Special cases (marked by a Type of VOID): PUTFIELD, PUTSTATIC, void methods
- PhiTemporary used to merge locals – more on the instantiation later
- Each Temporary has a Type value associated with it

Temporary#cloneOnInstruction()

- Method for referencing temporaries while maintaining all relevant information with that temporary; necessary to introduce a parentTemporary field to accomplish this

Temporary#getContiguousBlockSorted()

- Recursive method that generates a list of each instruction used in the calculation of this temporary
- Returns null for non-contiguous blocks

Temporary#getCriticalTemporaries()

- Returns a list of all Temporary oeprands; related to getContiguousBlocksSorted()

Temporary#getConstancy()

- Beyond standard constancy determination, constancy can be forced on a temporary and it's parents by the forceConstancy(int) method
    - Necessary for IINC instructions and expression(bool ? val1 : val2) semantics

Temporary#equals(Temporary)

- Two temporaries equal each other iff they are of the same class, and all operands are also equal to each other and the variables of the class are equal to each other

Optimizations

- A note about optimizations: Sun's built-in JIT (just-in-time) compiler performs the optimizations that make Java a feasible language
- The following optimization strategies impact the runtime minimally, but in special cases (specially constructed loop test case, for example), the optimized version runs about about 5-8% faster than the optimized version, so there are indeed results

Loop Invariant Optimizations

- Generates a map of variant locals (maps to innermost LoopEntry in which it is variant)
- Moving from innermost loop outward, selects the Temporary instances that are eligible to be moved outside of the loop
- For each eligible temporary, replaces the contiguous block of instructions with a corresponding XLOAD VAR instruction
- Inserts the instructions necessary to compute each temporary before the first child of the loop's parent, in this way ensuring all LoopEntry siblings have the local variable defined before execution (test case demonstrates this)

Constant Folding Optimizations

- A note about the style of this class: after two weeks of debugging my constant folder that corrupted the bytecode, I rewrote the implementation using a totally different code style: Java's pseudo-functional programming with Streams and lambdas
    - After realizing the clarity of the rewritten code, it would be a worthwhile objective to refactor preexisting code to follow this same pattern as it is much more readable and much easier to debug
- For each eligible temporary (marked as CONSTANT and has a contiguous block of instructions with length greater than 1 and is not of Type VOID):
- select the most general/longest form of that temporary
    - Example: select ((100 * 12) / 30) instead of just (100 * 12)
- fold that temporary by replacing the equivalent instructions with a corresponding load constant instruction

Obfuscations

Name Obfuscations

Randomly Generating Names

- Uses a number system of base n with n being the number of unique characters desired in the pattern; each BigInteger within the range corresponds to a generated name
- To prevent collisions, keeps track of used names in a HashSet<BigInteger> and simply randomly generates numbers within the range until one that has not been used is found

- Best way to understand the name generation is to look at the code

Implementing Changings

- Four HashMap<String, String> instances with formats:
    - packageNameRemapping: “org” -> “ABCD”, “armanious" -> “DCBA”, etc.
    - classNameRemapping: “org/armanious/csci1260/Entry” -> “ABCD”
    - fieldNameRemapping: “org/armanious/csci1260/Entry.run_output” -> “ABCD”
    - methodNameRemapping: “org/armanious/csci1260/Entry.reset()V” -> “ABCD”
- Necessary to maintain inheritance relationships: superclasses, inner classes, method overloading, package structure, accessing superclass’s fields, etc.
- Also replaces type information in not-so-obvious places: exception types, FrameNode pseudoinstruction types for the stack and locals, signatures
- After populating the maps using names supplied by the NameGenerator instance, simply goes through each ClassNode and corresponding methods and fields and renames
- Supports basic String-literal reflection with the following methods (overloads included):
    - ClassLoader#loadClass
    - ClassLoader#defineClass
    - ClassLoader#findClass
    - ClassLoader#findLoadedClass
    - Class.forName
    - Class.getMethod
    - Class.getDeclaredMethod
    - Class.getField
    - Class.getDeclaredField

Stack Manipulation Obfuscations

- Has a RATE_OF_STACK_MANIPULATION field (the default 0.25 is relatively high)
- Generates a list of Temporary instances that are eligible to be obfuscated via stack manipulation (instruction block length greater than three; greatest non-overlapping; has critical temporaries)
    - I will refer to a Temporary’s contiguous block of instructions simply as “block”

Insert, Swap, and Pop

- Inserts instructions before the block, and if necessary, inserts temporaries to restore the stack after the block
- Inserts 0, 1, or 2 stack words before, each of which has a hardcoded set of instructions
    - Immediate area of improvement: randomly generate the insertion instructions, while keeping track of various maximum stack depth changes (framework for doing so is already implemented)
    - 2 stack words maximum to guarantee the ability to restore the stack after the block
- Inserts instructions after the block that correspond to the number of stack words inserted before the block
- Example

Swap and Rearrange

- Randomly selects two critical temporaries that are no more than four stack words apart (inclusive) to ensure the stack can be restored without excessive spilling to local variables
    - Note: one scenario, when the operands to be swapped are of size one and the operand between them is of size two, requires spilling into a local variable
- Swaps the two blocks in the instructions: very simple operation given that the instructions are stored as a doubly-linked list
- After the end of block of the second operand that was swapped: insert the stack restoration instructions (int[][][][][] array containing the appropriate stack-restoration opcodes; two scenarios are special-cased, one of which was noted above)

String Literal Obfuscations

- Has a RATE_OF_NEW_CIPHER field (the default 0.25 is rather high)
- Walks through each instruction of each method of each class; for each LDC String:
- Retrieves the cipher to be used, which may or may not be a newly generated one
    - Two variables randomly generated for each cipher: one for the modulus operator and and one for the add operator
    - Tests for the String s: s.equals(decipher(cipher(s)))
        - If true, use it
        - If false (which occurs sometimes but at a low enough rate to be negligible for this simple application), generates a new cipher until one works
    - If it's a new cipher, randomly insert it into the given classes
        - Can even insert into interface class files
- Replace the constant in the LDC instruction with the ciphered String
- Insert an instruction immediately after the LDC with a call to the corresponding decipher method which returns the desired String
- Example

Compile-Time Data Compression

- Relevant class to look at it: CustomClassLoader.java (in the default package)
- Generates a JAR file with three components:
    1. Manifest pointing to the obfuscated name of CustomClassLoader, which in addition to being the ClassLoader will be the jar's entry point
    2. The CustomClassLoader class data
        - All obfuscations are run on this class before it is written
    3. A single "inner" file (with a random name) that is a ZIP file containing all the class data supplied to the program that will be read and mapped by the CustomClassLoader at startup
        - The classes are then defined as needed

Testing

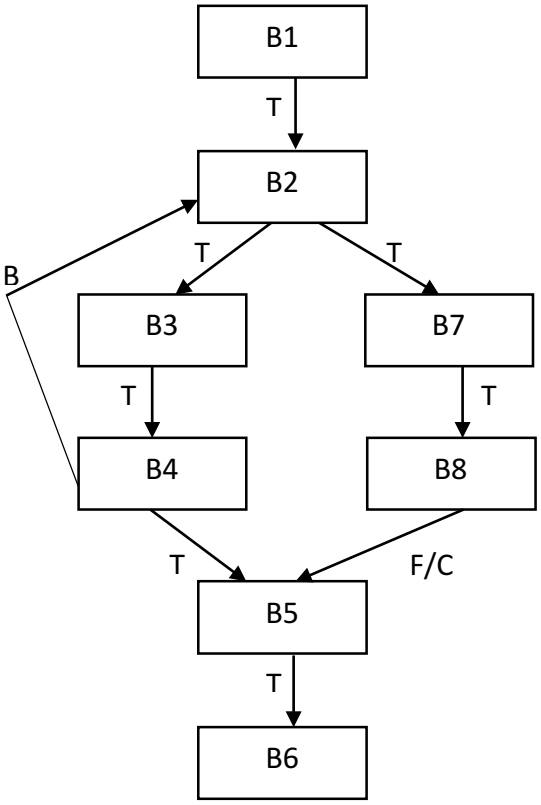
- The best form of verification to ensure the bytecode is valid is simply loading the classes: by default, the Sun JVM passes all bytecode being loaded as a class through a Verifier that will catch any bugs: corrupted stack, loading undefined local variable, etc.
- My personal test is rather convoluted and has no specific purpose besides trying to be convoluted in order to find as many edge cases as possible and to determine as many bugs as possible in my program
- I translated all the supplied decaf test programs into Java (or an approximation thereof; ansi.decaf does not work as intended, but still prints a recognizable and predictable String that is replicatable)
- Other miscellaneous testing included previous projects of mine written in Java as well as other programs written by other computer science students; they all passed

Examples

DFS Edge-Labeling and Computing iDominance

iDominance Relationships as performed by algorithm:
B1 – null
B2 – B1
B3 – B2
B4 – B3
~~B5 – B4~~
B6 – B5
B7 – B2
B8 – B7
B5 – ancestor(B4, B8) = B2
Done

Note: FORWARD and CROSS edges are obviously different; for simplicity, both are shown as equivalent here as it initiates the same iDominance calculation strategy




```
private void dfsAndIdom(BasicBlock b){
    pre[b.index] = precount++;
    for(BlockEdge e : b.successors){
        BasicBlock s = e.b2;
        if(pre[s.index] == 0){
            e.classification = BlockEdge.Classification.TREE;
            iDominanceMap.put(s, b);
            dfsAndIdom(s);
        }else if(post[s.index] == 0){
            e.classification = BlockEdge.Classification.BACK;
            //ignore iDominance; no effect
        }else if(pre[b.index] < pre[s.index]){
            e.classification = BlockEdge.Classification.FORWARD;
            iDominanceMap.put(s, ancestor(iDominanceMap.get(s),
                iDominanceMap.get(b)));
        }else{
            e.classification = BlockEdge.Classification.CROSS;
            iDominanceMap.put(s, ancestor(iDominanceMap.get(s),
                iDominanceMap.get(b)));
        }
    }
    post[b.index] = postcount--;
}
```

Constant Folding Optimizations

Java code equivalent: System.out.println((100 * 12) / 30)

Bytecode before:
GETSTATIC System.out
LDC 100
LDC 12
IMUL
LDC 30
IDIV
INVOKEVIRTUAL System.out.println(I)V

Bytecode after:
GETSTATIC System.out
LDC 4
INVOKEVIRTUAL System.out.println(I)V

Note: there are various constant instructions (BIPUSH, SIPUSH, ICONST_X, etc; for simplicity, they are all represented here by LDC, which is a valid instruction, albeit the most efficient. In the program itself, the best instruction possible is inserted, not just LDC (for details, look at ConstantFolder.getConstantInstruction(Object)).

Random Name Generation

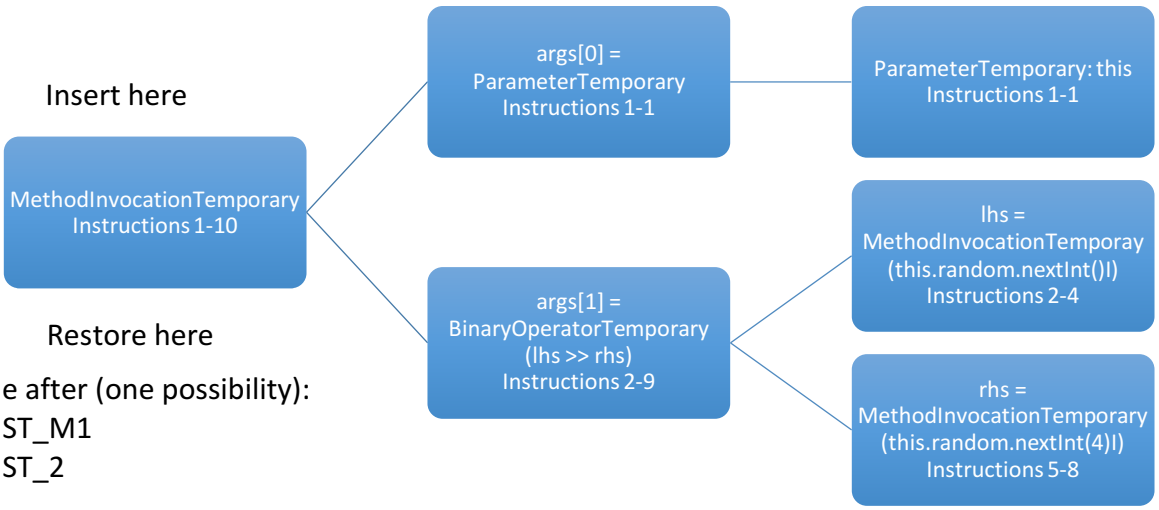
```
public String getNext(){ //pseudocode; actual code is more complex
    BigInteger value = BigDecimal.valueOf(random.nextDouble())
                       .multiply(maximumAsDecimal).toBigInteger();
    if(used.add(value)){
        return convertBigIntegerToString(value);
    }
}

private String convertBigIntegerToString(BigInteger value){
    final char[] string = new char[length];
    int idx = length - 1;
    while(true){
        string[idx] = validChars[value.mod(base).intValue()];
        value = value.divide(base);
        if(value.signum() == 0 || idx == 0){
            break;
        }
        idx--;
    }
    while(idx != 0){ //pad beginning of String with the minimum value
        string[--idx] = validChars[0];
    }
    return new String(string);
}
```

Insert, Swap, and Pop Stack Manipulation

Original code:
this.someMethodReturningBoolean((random.nextInt() >> random.nextInt(4))

Bytecode:
WILL INSERT THREE INSTRUCTIONS HERE
 1: ALOAD 0
 2: ALOAD 0
 3: GETFIELD random
 4: INVOKEVIRTUAL nextInt()I
 5: ALOAD 0
 6: GETFIELD random
 7: ICONST_4
 8: INVOKEVIRTUAL nextInt(I)I
 9: ISHR
10: INVOKEVIRTUAL someMethodReturningBoolean(I)Z
WILL SWAP AND POP STACK HERE



Bytecode after (one possibility):
 1: ICONST_M1
 2: ICONST_2
 3: IMUL
 4: ALOAD 0
 5: ALOAD 0
 6: GETFIELD random
 7: INVOKEVIRTUAL nextInt()I
 8: ALOAD 0
 9: GETFIELD random
10: ICONST_4
11: INVOKEVIRTUAL nextInt(I)I
12: ISHR
13: INVOKEVIRTUAL someMethodReturningBoolean(I)Z
14: SWAP
15: POP

Swap and Rearrange Stack Manipulation

Most convoluted scenario:
Swapping two single-word operands with a double-word operand in between

Java source: someMethod(int1, double, int2)

Bytecode:
ILOAD 0 [int1]
DLOAD 1 [int1, double1, double2] (double takes up two stack words)
ILOAD 3 [int1, double1, double2, int2]
INVOKESTATIC someMethod(IDI)V []

After swapping:
ILOAD 3 [int2]
DLOAD 1 [int2, double1, double2]
ILOAD 0 [int2, double1, double2, int2]
INVOKESTATIC someMethod(IDI)V //int arguments reversed, we need to correct this

To restore the stack:
ILOAD 3 [int2]
DLOAD 1 [int2, double1, double2]
ILOAD 0 [int2, double1, double2, int1]
DUP_X2 [int2, int1, double1, double2, int1]
POP [int2, int1, double1, double2]
DUP2_X2 [double1, double2, int2, int1, double1, double2]
POP2 [double1, double2, int2, int1]
SWAP [double1, double2, int1, int2]
ISTORE 4 [double1, double2, int1] //we cannot ensure that each Temporary will already be stored in a local variable, so we must fall back to spilling and store into the next available local variable index to ensure valid bytecode in all situations
DUP_X2 [int1, double1, double2, int1]
POP [int1, double1, double2]
ILOAD 4 [int1, double1, double2, int2]
INVOKESTATIC someMethod(IDI)V //int arguments have been corrected

String Literal Encryption

Java source:
System.out.println(this.append(“Hello”, “ World”));

Output: “Hello World”

Original bytecode:
GETFIELD System.out
ALOAD 0 (this) [System.out, this]
LDC “Hello” [System.out, this, “Hello”]
LDC “ World” [System.out, this, “Hello”, “ World”]
INVOKEVIRTUAL append() [System.out, “Hello World”]
INVOKEVIRTUAL println()

Obfuscated bytecode:
GETFIELD System.out
ALOAD 0 (this) [System.out, this]
LDC “¨„æœç” [System.out, this, “¨„æœç”]
INVOKESTATIC _2349834958() [System.out, this, “Hello”]
LDC “³Žá¦¶š” [System.out, this, “Hello”, “³Žá¦¶š”]
INVOKESTATIC _9871298432() [System.out, this. “Hello”, “ World”]
INVOKEVIRTUAL append() [System.out, “Hello World”]
INVOKEVIRTUAL println()

Decompiled obfuscated java source:
System.out.println(this.append(_2349834958(“¨„æœç”), _9871298432(“³Žá¦¶š”));

Obfuscated output: “Hello World”