Extending and Evaluating a Control Flow Obfuscation Technique for JVM Applications Utilizing `invokedynamic` with Native Bootstrapping

Bachelor's Thesis Proposal

Andre Blanke

Juli 15, 2022

1 Introduction

Obfuscation techniques are an important tool to make malicious reverse engineering attempts of software harder and more time-consuming. These techniques are often employed to protect trade secrets, intellectual property, and program integrity.

Even higher importance is placed on the usage of obfuscation techniques when the software is deployed in an environment controlled by a trusted user of the software (e.g. to a personal computer as opposed to a microcontroller for example) for the prevention of so-called man-at-the-end (MATE) attacks as coined by (Falcarin et al. 2011).

Programming languages compiled to high-level bytecode—such as Java Virtual Machine (JVM) bytecode instead of machine code—are especially vulnerable to MATE attacks, as JVM bytecode is more structured and thus allows easier modification when compared to machine code.

A variety of obfuscation techniques exist, some of which, such as symbol scrambling (Chan and Yang 2004), remove information in an irreversible way. Other aspects, like the interprocedural control flow graph of a program, are harder to obfuscate. Keeping this information intact allows an attacker to gain great insight into the architecture and inner workings of an application.

Java 7 saw the introduction of the `invokedynamic` instruction which allows the invocation of methods without directly specifying the target call site inside the bytecode, instead delegating execution to a so-called bootstrap method in order to determine the call site at runtime rather than compile-time. The instruction enables dynamic dispatch behavior similar to reflection but in a more performant way because the call site can be accessed directly after its initial extraction from the bootstrap method as illustrated by (Ivanov 2015). These properties have proven useful for call graph obfuscation efforts.

A novel obfuscation technique proposed by (Wood and Azim 2021) utilizing `invokedynamic` implements the bootstrap method inside a native library and replaces regular `invoke*` instructions with `invokedynamic` ones, hindering static analysis attempts. Outsourcing the bootstrap method implementation to native code increases the level of obfuscation and enables the application of further obfuscation methods going beyond techniques available for the JVM. It also prevents analysis of the bootstrap method bytecode with the help of established tools in the ecosystem.

The proposed bachelor's thesis would primarily focus on this obfuscation technique in an aim to create an implementation of it since none is openly available at the moment. Once a basic implementation is finished, it will be extended further and evaluated.

2 Problem

While a proof-of-concept has been developed for the novel obfuscation technique proposed by (Wood and Azim 2021), no implementation of it has been released which hinders further benchmarking attempts that are needed in order to judge its performance impact in a larger context: what are the performance implications when applying this technique to an input program, especially across a broader variety of JVM distributions and JIT compilers? Does the obfuscation lead to an inflation in bytecode size and—if so—to what extend?

In addition to the lack of an implementation, the developed technique does not yet fully exhaust its potential and can be further extended beyond obfuscation of the call graph itself: field access instructions (`getfield`, `putfield`, `getstatic`, and `putstatic`) may be included in the obfuscation effort by introducing additional method invocations, as mentioned in the future works section of the research paper.

A different area that is only briefly mentioned by the paper is the effectiveness of the the technique which should also be evaluated further, seeking to answer research questions such as "How easy is it to identify the used obfuscation technique and understand what transformation have been applied to the obfuscated code?" and "How complex are attempts at bypassing the obfuscation technique, once understood?".

3 Solution Approach

3.1 Implementation

The primary goal of the bachelor's thesis will lay on the implementation of an open-source tool capable of performing the proposed obfuscation on single classes as well as entire JAR files using the ASM bytecode engineering library. (Bruneton 2011)

Figure 1 gives an overview of the three-step obfuscation process. Two inputs will be required by the obfuscation tool: a JAR or class file to be obfuscated and a template for the bootstrap method which is populated using the symbol table built in step 2.

Step 1 is optional and responsible for the inclusion of field access instructions in the obfuscation effort to increase the provided level of indirection, serving as an extension to the technique proposed by (Wood and Azim 2021). Obfuscation of field accesses is achieved through the generation of synthetic getter and setter methods which are invoked in place of the original instruction.
No further steps are necessary when performing this bytecode transformation as the first step before the main obfuscation pass, as the generated `invoke*` instructions will be treated like any other method invocation present in the original bytecode and replaced by

an equivalent `invokedynamic` instruction.
Introducing the additional synthetic methods will lead to an inflation in bytecode size which should be measured by comparing the size of the input file with that of the obfuscated output file.

Following the optional pre-processing of field instructions in the first step, obfuscation **step 2** is responsible for the replacement of `invoke*` instructions with `invokedynamic` ones, making up the largest amount of bytecode transformation operations.
A unique name is assigned to each encountered method invocation instruction and the association is stored in the symbol table. The unique name is present in the bytecode of the `invokedynamic` instruction and is used by the bootstrap method implementation to return the correct `CallSite` that was associated with it in the original bytecode.
The symbol table built from this obfuscation step is passed to the input bootstrap method template using a template engine for the generation of the bootstrap method source code. Automatic compilation of the processed template is outside the scope of the tool due to reasons outlined in section 5.

Step 3 is the last obfuscation step and adds the bootstrap method definition to the main application class, along with code responsible for loading the native library containing the bootstrap method implementation. The library loading code is prepended to `<clinit>` to ensure that it is invoked before any other application code which might be obfuscated (and thus assume the bootstrap method implementation is already available).

3.2 Evaluation

Open research questions regarding the performance implications of the obfuscation technique on input JAR files will be approached through the use of benchmarks using the Java Microbenchmark Harness which is part of (OpenJDK 2013). Care must be taken when selecting the input JAR files to use for the benchmark, as they and their dependencies must be compiled for newer Java versions. Reasons for this are outlined in section 5. While no concrete libraries of applications have been chosen for their usage in benchmarks yet, reusing ones picked for benchmarking purposes by other obfuscation techniques might allow some level of comparability.

The ease of identification will be judged by comparing the original bytecode with its obfuscated version. To determine the complexity of an attack on the technique, an attempt at bypassing it will be made using dynamic analysis. As the bootstrap method implementation resides in native code which presents a harder target than code running on the JVM, it seems natural to target the interface between the two in search for a weakness.

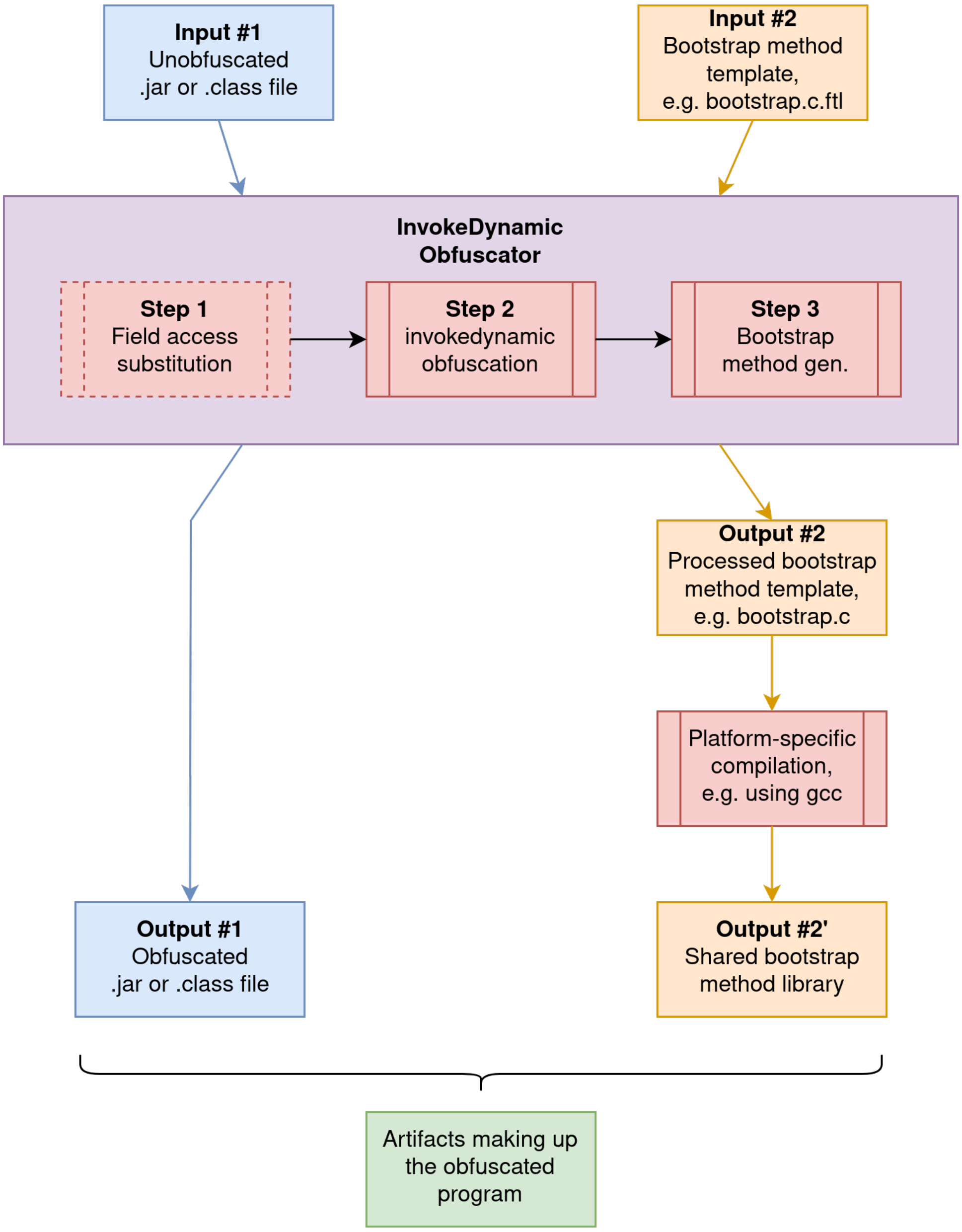


Figure 1: High-level overview of the obfuscation process.

4 Related Work

An attempt at hiding the interprocedural control flow graph of JVM applications has been made by (Pizzolotto and Ceccato 2018) through the translation of selected portions of program bytecode into native libraries but introduced a performance penalty between 5% and 21% by doing so, as the method relies heavily on reflective calls which introduce a certain overhead.

(Fukuda and Tamada 2014) proposes an obfuscation technique intended to confuse debugging tools through the addition or removal of arguments from `MethodHandle` instances within the bootstrap method.

Various implementations of obfuscation techniques making use of the substitution of `invoke*` instructions with `invokedynamic` exist, both open-source (superblaubeere27 2018) and commercial (Zelix Pty Ltd. 1997), but none of them implement the bootstrap method in native code. No associated research papers seem to exist for either of the two projects.

5 Objectives and Limitations

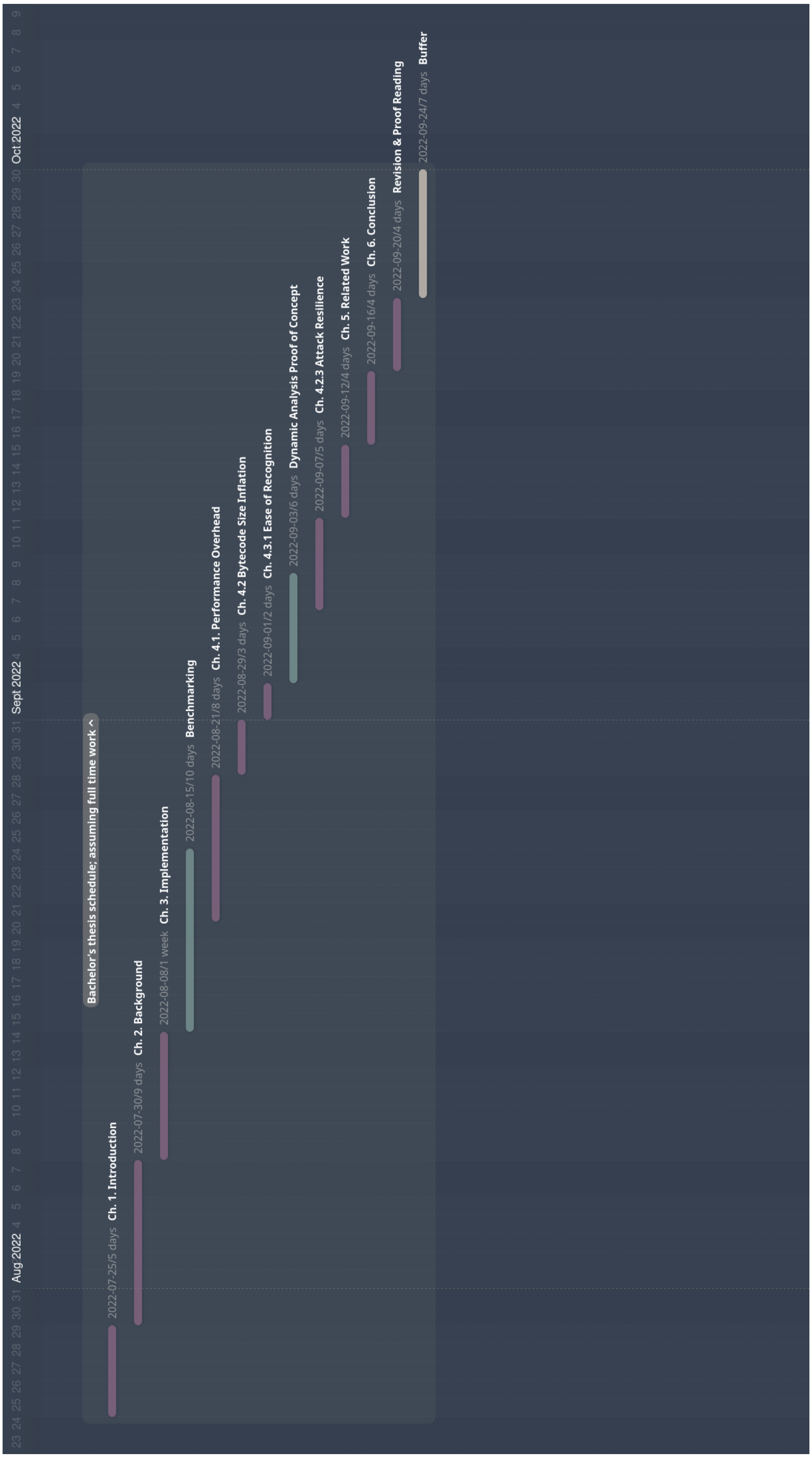
For simplicity reasons, the obfuscation tool will likely be limited to support JVM bytecode emitted by newer versions of Java compilers (class file major version ≥ 50), as bytecode targeting older versions of the class file format does not necessarily contain stack map frames. While these class files can still be correctly loaded and verified by JVMs, the missing stack map frames pose an issue when `invokedynamic` instructions are introduced, as that requires raising the bytecode major version to 51 or higher, making stack map frames mandatory. ASM is capable of computing the stack map frames but requires loading the classes using `Class.forName` in certain situations, which in turn would require loading the input JAR or class file and any potential dependencies using a custom class loader, complicating the implementation.

A different area that is out of scope for the obfuscation tool is covering the entire functionality of the toolchain required for the compilation of the native library containing a bootstrap method implementation. This includes the invocation of a compiler from within the tool. Instead, the tool should provide templating capabilities to allow for the production of valid source files in a user-chosen language by combining a template file with information obtained in the obfuscation pass. The last necessary compilation step would be left to the user of the obfuscation tool.

6 Preliminary Outline

1. Introduction
2. Background
   - 2.1. Obfuscation Techniques
   - 2.2. Java Native Interface
   - 2.3. InvokeDynamic
   - 2.4. Proposed Technique
3. Implementation
   - 3.1. Extension
4. Evaluation
   - 4.1. Performance Overhead
   - 4.2. Bytecode Size Inflation
   - 4.3. Obfuscation Level
     - 4.3.1. Ease of Recognition
     - 4.3.2. Attack Resilience
5. Related Work
6. Conclusion

7 Schedule and Risk Analysis



References

Bruneton, Eric. 2011. *ASM 4.0 a Java Bytecode Engineering Library*. https://asm.ow2.io/asm4-guide.pdf.

Chan, Jien-Tsai, and Wuu Yang. 2004. "Advanced Obfuscation Techniques for Java Bytecode." *Journal of Systems and Software* 71 (April): 1–10. https://doi.org/10.1016/S0164-1212(02)00066-3.

Falcarin, Paolo, Christian Collberg, Mikhail Atallah, and Mariusz Jakubowski. 2011. "Guest Editors' Introduction: Software Protection." *IEEE Software* 28 (March): 24–27. https://doi.org/10.1109/MS.2011.34.

Fukuda, Kazumasa, and Haruaki Tamada. 2014. "An Obfuscation Method to Build a Fake Call Flow Graph by Hooking Method Calls." In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 1–6. https://doi.org/10.1109/SNPD.2014.6888726.

Ivanov, Vladimir. 2015. "Invokedynamic: Deep Dive." Oracle Corp. https://cr.openjdk.java.net/~vlivanov/talks/2015-Indy_Deep_Dive.pdf.

OpenJDK. 2013. "Java Microbenchmark Harness." https://github.com/openjdk/jmh.

Pizzolotto, Davide, and Mariano Ceccato. 2018. "Obfuscating Java Programs by Translating Selected Portions of Bytecode to Native Libraries." In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. https://doi.org/10.1109/scam.2018.00012.

superblaubeere27. 2018. "A Java Obfuscator." https://github.com/superblaubeere27/obfuscator.

Wood, Bradley, and Akramul Azim. 2021. "A Novel Technique for Control Flow Obfuscation in JVM Applications Using InvokeDynamic with Native Bootstrapping." In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, 232–36. USA: IBM Corp.

Zelix Pty Ltd. 1997. "Zelix KlassMaster." https://www.zelix.com/klassmaster/index.html.

Supervisor (Title First name Last name)	Student (First name Last name)