Byteboy : A Byteman Companion application

Alex P. T. Macdonald
apmacdon[at]sfu.ca
Simon Fraser University

Ian Pun
ipun[at]sfu.ca
Simon Fraser University

ABSTRACT

As computer programs grow larger and more complex, the likelihood of catastrophic issues may increase over time. Although many bugs can be discovered by using modern testing frameworks, some bugs may go unrecognized and appear in production builds.

Byteman is a powerful open-source tracing and debugging tool for Java, that injects Java code into application methods without the need of recompilation, repackaging, or even redeploying the application. However, the Byteman scripting language syntax is not currently supported by IDEs, and learning how to write Byteman rules is a completely manual and potentially tedious process. Currently, learning Byteman requires the user to either read the User's Manual, or search for tutorial blog posts that can be adapted to fit their needs.

This paper presents Byteboy, a companion application suite for Byteman which includes a manual, semi-automatic and automatic rule builder, a VS Code language extension, and a basic fuzz test generator for the Byteman scripting language.

1 INTRODUCTION

Writing bug free software is often easier said than done. Modern compilers and testing frameworks will often help developers minimize the number of bugs that make it into production environments, but there is no guarantee that the end result will be completely error free. For example, a Java program with poor exception handling coverage may pass a syntax check, but will crash if an unexpected exception type is thrown in response to an unexpected input. As a result, there is a need for tools that can diagnose and repair software bugs in runtime environments, and can interact with the program code in order to either create a temporary fix or reveal the root cause of the error such that a proper fix can be developed.

An example of a tool that is capable of diagnosing and repairing errors in Java programs is Byteman[1]. Byteman is a bytecode injection tool, that allows the user to write code using the Byteman scripting language, which will be injected into a running application. Currently, there is no IDE language support for the Byteman scripting language, and the best methods of learing how to write rules includes reading the Programmer's Guide [2], reading various tutorial blog posts[3], or by watching a Byteman tutorial YouTube video[4]. As a result, when looking at Byteman from the perspective of a first time user it may present a large learning curve, especially to a developer that is accustomed to the modern luxuries of language learning that include code snippets, hints, generators[5], and guided learning.

In this paper, we propose Byteboy[7], a companion application for Byteman. Byteboy provides functionality for manual, semi-automatic, and automatic rule generation, fuzz test generation, and a Visual Studio Code Byteman rule language extension. Byteboy was designed to improve the Byteman user experience from the point of rule inception, through to actually implementing and shipping the proposed bug-fixing rule file.

2 BACKGROUND

2.1 Byteman

Byteman is a tool for monitoring, debugging, and testing Java code[1]. It is a Java Virtual Machine Tool Interface (JVMTI) agent, which allows it to be installed via the command line to retrieve and interact with data from the Java Virtual Machine (JVM). Byteman modifies the bytecode of an application at runtime, and can inject into any Java methods (including the JDK runtime and libraries), and can even interact with private methods[1]. Unlike many other bytecode transformers, Byteman operates at the level of Java (not bytecode), so you provide Byteman with rules which specify the Java code you want to be executed, and Byteman works out how to rewrite the bytecode such that it behaves as if it were native to the source code[1].

2.2 Byteman Rule Syntax

The general syntax of a Byteman rule is as follows:

```
RULE <unique-name>
CLASS <classname>
METHOD <method-name>
AT <entry/exit>
IF <condition>
DO <action>
ENDRULE
```

`RULE` specifies the name of the rule, which will be used when unloading the rule, and should be unique for best practices.

`CLASS` specifies the Java class the rule wants to target.

`METHOD` specifies the target method within the selected class.

`AT` specifies the timing of which the rule logic should take place relative to the method. For example, `AT ENTRY` causes the rule logic to execute when the target method is entered, and `AT EXIT` can be used to execute code when the method has completed execution.

`IF` specifies a condition in which the rule logic will execute. This can be based on variable input, or conditions of interest, or could be left as `TRUE` to always execute.

`DO` specifies the logic of the rule, in which actions will be executed. This is where the rule can bind to existing variables, create new variables (such as counters), or use traceln statements to print information to the console.

`ENDRULE` specifies the end of the rule content. A single Byteman script may contain many rules, so ensuring separation using the `RULE` and `ENDRULE` are important for passing syntax checks, but also for ensuring that individual rules can be unloaded at will without having to remove all of the rules.

2.3 Using Byteman

Byteman can be used on an application that is currently executing, or at start time. Byteman is a JVMTI agent, which allows it to be supplied to the `javaagent` argument when running a Java application.

Suppose `BYTEMAN_HOME` is an environment variable that points to the current installed or downloaded version of Byteman, a script (`rule.btm` in this case) can be loaded along side running the Java program `HelloWorld` with the following command:

```
java -javaagent=$BYTEMAN_HOME
/lib/byteman.jar=script:rule.btm HelloWorld
```

Please see the Byteman Programmer's Guide[2] for advanced syntax, use cases, and examples.

3 MOTIVATION

This section presents an example to illustrate how Byteman can be used to trace and debug bugs in Java programs.

3.1 Example Application

Consider the following Java program:

```
public class InfLoop {

    private void doWork(int i) {
        while(i == 0) {
            work();
        }
    }

    private static void work() {}

    public static void main(String[] args) {
        InfLoop inf  = new InfLoop();
        inf.doWork(0);
    }
}
```

The above Java application contains a bug which will cause it to become stuck in an infinite loop. The main method instantiates a new InfLoop object, and decides to put the object to work. The InfLoop object has it's `doWork()` method invoked and is passed the number zero, which allows entry into the while loop and causes `work()` to be called. However, the developer of this program forgot to write an exit condition for the loop, and now the program will loop infinitely. While this program is simplistic for the sake of being an example, the behaviour of this object is not completely unbelievable; it would be completely understandable to have an object that enters a loop after calling a function.

Compiling and running the above program will not throw any errors, and upon executing this program in the terminal, the user will be greeted with a blinking prompt while the program is stuck in execution. With no definitive indication of if or where an error is occurring, this is a good time to construct some Byteman rules.

3.2 Tracing the Error

The first step in debugging the above application will be to trace the error, and try to gather some information about the current program behaviour. Based on looking at the source code, a good point to start tracing may be the entry point of the `work()` method, to see if the application is actually doing any work. In order to confirm or deny the activity of `work()`, the following Byteman rule will be constructed:

```
RULE Traceln when starting work
CLASS InfLoop
METHOD work
AT ENTRY
IF TRUE
DO traceln("- We're starting work.");
ENDRULE
```

The above rule targets the entry point of the `work()` method, and when called, will output a string to the console.

After loading this Byteman rule into the running application, the console will be flooded with strings indicating the constant and relentless calling of the `work()` method. At this point, it can be deduced that the program may be stuck in execution, and it may be a good time to consider a Byteman script that can relieve the application.

3.3 Debugging the Error

Now that the error has been traced and verified, a Byteman rule may be able to be constructed with the purpose of exiting the loop. Going back to the source code, it is evident that the reason `work()` is being called so frequently is because it is being called from within a loop that will only exit if the variable i is changed such that it no longer equals zero. The variable i is an argument that is passed to `doWork()` from the main method, and can be altered by the following Byteman rule:

```
RULE Exit The Loop
CLASS InfLoop
METHOD doWork
AT ENTRY
IF TRUE
DO
$1 = 1;
ENDRULE
```

The above rule targets the entry point of the `doWork()` method, and uses the $1 to bind to the first argument in the function declaration (which is i in this case), and reassigns the value to 1.

As a result, when running this program with the above rule injected, the program will not enter the infinite loop, and will allow the program to terminate.

3.4 Introducing Byteboy

As illustrated using the above Java program and Byteman rules, writing these rules requires knowledge of the target application and access to it's source code in order to design rules that accomplish a specific task. The above example involved a short Java program with a glaring and obvious bug, but what happens if a developer encounters an error in their application that spans thousands of lines of code, and maybe aren't intimately familiar with all aspects of the code base? Furthermore, modern IDEs have no support for the Byteman scripting language, which causes developers to write their scripts without the modern luxuries that language extensions offer. Finally, at the moment, the quickest way of learning to write Byteman rules would be from reading one of the many helpful blog posts[3] or YouTube videos[4] that target beginner and first time users, but no tool currently exists that can help guide a user through the creation of their first scripts.

Here, we introduce Byteboy - a companion application for Byteman. Byteboy was designed to enhance the Byteman user experience by aiding the development of rules starting from the point of rule inception, through to the result of creating in a production-quality script.

The primary functionality of Byteboy is it's rule generator, which can be used in a manual, semi-automatic, and automatic mode. The manual rule generation using Byteboy is reminiscent of the Yeoman Generator[5], in which the program will guide new and first-time users through the process of writing rules. The semi-automatic rule generation actually performs an execution of the target program, and collects and aggregates information about the application performance and code structure to generate a set of plausible rules. The automatic rule generator creates a set of rules that can be used to trace the activity of a program in execution.

Another feature of Byteboy is it's fuzz test generator, in which it performs similarilty to the automatic functionality, but instead of suggesting rules that could be used to trace the code, it generates rules with the aim of producing unexpected behaviour in the source program.

Lastly, Byteboy ships with a language extension for Visual Studio Code which provides support for the Byteman scripting language.

4 APPROACH

This section describes our approach for Byteboy. Here, we will primarily discuss the approach to the semi-automatic rule generator because it shares the same approach with the fuzz test generator, and the manual rule generator is just a simplified version of this approach. The approach is comprised of three phases to generate Byteman rules: Instrumentation, Analysis, and Synthesis.

4.1 Instrumentation

In order for us to discover useful information in our Java application, we opted to use HProf. As a JVM native agent library, HPROF is dynamically loaded through a command line option, at JVM startup, and becomes part of the JVM process[10]. This allows users to request relevant information such as CPU usage, method calls and their call frequency, and their stack traces. The results are then ranked, parsed, and organized through the analysis phase.

4.2 Analysis

Majority of the work required for Byteboy is analyzing the results from HPROF and presenting it back to the reader in a useful manner. We decided to use Python to handle this task as it was the most accessable and lightweight language available.

Byteboy uses HPROF to generate a `java.hprof.txt` file after the Java application completes its run. It handles applications with an infinite loop by instructing the user to use the `Ctrl+C` shortcut to terminate the process. This will still generate the proper `java.hprof.txt` file which Byteboy will interpret. The resulting `java.hprof.txt` file gives us information of all the methods called in the JVM, their call count, CPU usage, and their stack traces. Byteboy will only use the top 5 method calls that are not internal java libraries with the most CPU usage for its semi-automatic, automatic, and fuzzer features.

Unfortunately, HPROF does not generate information about method argument types, which is necessary in order to include fuzzer functionality. `Javap` is a Java class file disassembler that can print out the package, protected, and public fields and methods passed into it[11]. Using Javap, we are able to create an additional analysis of the java application which strictly gives us method argument types. Byteboy will then combine the outputs of HPROF and `javap` in order to produce the top 5 method calls and their respective argument types.

4.3 Synthesis

Byteboy currently offers four different types of rule construction: manual, semi-automatic, automatic, and fuzz test generator.

In manual, Byteboy goes through the required components of Byteman rule syntax, and prompts the user for input for each of the required fields. The end result is a rule constructed based solely on the information provided by the user, and is reminiscent of the Yo Code generator[6].

In semi-automatic, the process is the same as manual, with the addition of information gathered from the analysis phase. Each Byteman rule component (with the exception of the DO block) can be auto-filled by simply pressing the enter key, and Byteboy will supply the highest ranking choice based on the analysis. However, the user is able to freely enter information here as they please, and use the displayed analytics to craft their own rules based on the result of their program's execution.

In automatic, Byteboy creates a set of rules to trace application behaviour. A set of rules are created that alert the console when methods are called, which will provide real-time status reports to the user when running their program with these injected rules.

In fuzz test generator, Byteboy creates a set of rules that are intended to test the durability and resilience of the source program. Byteboy uses the analysis data, and figures out what the data type of each functions parameters are. Next, a rule is constructed that replaces the variable in the function declaration with a randomized value provided by one of our many helper functions. We currently have functionality to randomize the information of int, long, float, double, boolean, and String data types. The goal of these generated rules are to see if randomized inputs can be used to crash the source program, or cause ill effects.

4.4 VS Code Language Extension

A feature that many developers take for granted is the language support extension for their favourite IDE. Popular IDEs such as Eclipse and Visual Studio Code offer support for various language extensions, and there are currently extensions available for the majority of modern languages. Despite this, Byteman currently has no publicly available language extension.

In an attempt to remedy this situation, Byteboy ships with a Visual Studio Code language extension, that offers support for the Byteman scripting language and a basic rule generation snippet. The extension was initialized using the Yo Code generator[6], and uses the TextMate language grammar format[9] to provide functionality to the IDE. Our TextMate grammar file is based on the Byteman syntax as described in the Byteman Programmer's Guide[2], and includes coverage for the majority of the keywords and behaviours.

The language extension is not yet currently available through the Extensions Marketplace, but future work efforts will be put forward towards a public release. Currently, the only method of installation would be to grab the `byteman-language-extension` folder from the Byteboy repository[7], and copying it into the extensions folder located in the VS Code user settings folder. On Linux, this can be accomplished using the following command:

```
cp -r byteman-language-extension
      ~/.vscode/extensions
```

More instructions and details of usage can be found on the language-extension page of the Byteboy repository[8].

5 EVALUATION & OBSERVATIONS

We evaluate Byteboy using the previously described `InfLoop` program, and four ”real world” examples. We will also discuss the results of using Byteboy on the test cases, and the observations that were made as a result. Lastly, we will discuss the current shortcomings and limitations of Byteboy, and how they can potentially be overcome.

5.1 Test Cases

The test cases for Byteboy include 4 buggy programs, including the `InfLoop` example that has already been introduced. The other four programs are ”real world” buggy programs, and we will describe the emphasis of the quotation marks here. Frankly, finding bugs in software that are easily and consistently reproducible is a lot harder than it sounds. Many popular applications will likely have had their obvious bugs patched already (assuming they made it through the review process in the first place), and many bugs that do exist may be difficult to reproduce or observe due to concurrency. Initially, we had attempted to use the GitHub search filters to find repositories containing Java applications with bugs that may be easily reproducible. However, we were instead met with a massive list of potential bugs (which is assuring), but the vast majority of them had poor documentation regarding how to reproduce the errors, let alone how to run the program itself (that’s not so assuring).

After investing a considerable amount of time into finding bugs and ending up with nothing of use, we fell back to Google in an attempt to find at least some buggy programs we could use as test cases. By using the Google search engine, we used the following statement as our search criteria for finding buggy programs: ”What’s wrong with this Java code?”. Without digging too far through the pages that Google returned, we evaluated all of the links from first page of returned search query, and selected the programs that would compile. As a result, we gathered 3 buggy programs that we could use, and while they may not be substantial in size or complexity, they are representative of Byteboy’s potential user base: developers (of any skill level) who need to debug their Java programs. The programs range in functionality from a basic Binary Search implementation, to an average marks calculator program, to a GUI ActionListener-based program.

5.2 Results

This section will focus on the outcomes of running the automatic rule generation and fuzz test generation functionality on the test programs. The following table showcases Byteboy’s ability to create valid rules based on the selected action:

Program	Auto. Generator	Fuzz Test Generator
A	Yes	Yes
BinarySearch	Yes	Yes
InfLoop	Yes	Yes
Question3	Yes	Yes

In all cases, Byteboy was able to create a set of rules which provided tracing functionality for the given program. These traces outlined the current program behaviour, and could be of use to detect control flow errors, or unexpected program behaviour.

In all cases, Byteboy was able to produce a set of rules which could be used to potentially alter program behaviour. For the most part, these rules had little affect on the program’s execution, which means that the program is robust enough to deal with unexpected input. In the case of `InfLoop.java`, the generated test actually caused the previously infinite loop to exit, ceasing execution of the program. While we are pleased with the initial results of the fuzz test generator functionality, there are many cases that cannot be properly support, and we will explain these situations in the Limitations part of this paper.

5.3 Observations

Each of the four programs were able to have at least some sort of their execution behaviour traced by using the set of rules which were automatically generated by Byteboy. The majority of the four program’s don’t exhibit complex function interactions, so the traceln results aren’t as verbose as they could be in an application of larger scale. However, for an average developer looking to see where their program may be getting into trouble, these generated rules should be of some use.

The fuzz test rules varied in their range of effectiveness, and their results will be explained individually.

`A.java`: In the case of this program, the variable supplied to the `A()` function is passed into a constructor for `JFrame()`, and as a result, substituting the value of this variable has little to no effect on the application’s behaviour. In this case, the fuzz test rule works only in the sense that it can be run and changes the title of the produced JFrame, but has no effect on the rest of the program's execution.

`BinarySearch.java`: This program has a bug in the way it outputs it's results, and while the execution is able to sustain the fuzz test randomized values, the effect on program behaviour cannot be observed.

`InfLoop.java`: In this program, an infinite loop is entered because the `doWork()` function gets passed a variable with the value of 0, which causes a while loop to start. This while loop has no exit condition, and as a result, the program loops forever. Fortunately, the fuzz test re-assigns the value of the variable which the loop in `doWork()` evaluates in order to start looping. In the case that the fuzz test generates the value of 0, then the program will infinitely loop as previous. However, it is highly likely that a non-zero value will be produced from the fuzz test randomizer, which will allow the program to bypass the infinite loop completly, thus ceasing execution of the program.

`Question3.java`: This program accepts 10 integers as input, and attempts to calculate the average. The fuzz test generator targets the function responsible for calculating the average, but unfortunately the program throws an `ArrayOutOfBoundsException` and crashes before the fuzzed value can be substituted. Unfortunately, the generated fuzz test rules will not have a chance to operate in this example, because of the nature of the error.

5.4 Limitations

Currently, Byteboy has several limitations that confines it's use cases. For starters, Byteboy accepts a Java program with the extension `.java` or `.class` as input, however, many larger applications are not simply launched through the command-line using the `java` command. Many popular Java applications may be packaged in `.jar` format as an executable, or rely on larger build systems such as Maven and Gradle which makes the situation trickier.

The rules generated by Byteboy currently provide tracing and fuzz test capabilities, and this is reflective of our current product knowledge of Byteman and not the capabilities of Byteman as a tool. There are many more intricate and complicated types of rules that could be created, and using the data from our analysis it may allow for these rules to be created. However, it's difficult to create a code generator for a language when we aren't masters of the language ourselves, so this limited functionality should be seen as a proof of concept, and open the doors to future improvements in this application.

Lastly, the fuzz test generator only handles primitive data types at the moment, and doesn't work with arrays. The ability to fuzz arrays would be an interesting addition of functionality for the future, and allowing for the fuzzing of Java-specific data types (or even custom data types) would be an interesting topic of research as well.

6 RELATED WORK

6.1 ByteBuddy

ByteBuddy is a popular code generation and manipulation library for creating and modifying Java classes during the runtime of a Java application[12]. The bytebuddy library allows a user to inject classes into a running JVM dynamically. Since it is a Java library, it is easier and more intuitive than Byteman rules. However, we do believe that the simplicity of Byteman rules are perferred compared to learning a new library.

6.2 AspectJ

Aspect-oriented programming (AOP) is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing common concerns[13]. When we have to use methods (ex. logging) that are used throughout different methods that span different classes, it can lead to code tangling or scattering. AspectJ is an extension to java that allows for the AOP paradigm to work alongside with object-oriented programming.

Similar to ByteBuddy, there doesn't seem to be any work on recommendations of what classes to modify. Both tools require a decent amount of knowledge of the target source code. Byteboy could eventually, using its HPROF and `javap` analysis, support ByteBuddy and AspectJ synthesis as well. However, code synthesis for both of these tools would prove to be a different challenge on its own.

7 CONCLUSION & FUTURE WORK

Byteman is a powerful tool for tracing and debugging Java programs by injecting code into running application methods, but currently lacks the luxuries of modern programming languages. This paper has presented Byteboy, a companion application suite for Byteman that includes a manual, semi-automatic, and automatic rule generator, a fuzz test generator, and a VS Code language extension. Byteboy has shown that it can produce rules that may be of use to developers (for tracing and fuzzing purposes), but will require additional work towards broadening and strengthening it's functionality.

Future work may be conducted on the instrumentation phase of our approach, as it would be interesting to consider the results of performing different dynamic analysis algorithms on the target program in an attempt to generate a more comprehensive set of Byteman rules. Additionally, more work will be done to improve the grammar file of the language extension, and it will hopefully be available for download via the Visual Studio Marketplace in the near future. Support for other existing Java tools, such as AspectJ and ByteBuddy, could be useful as developers would not need to learn an additional language (Byteman).

References

[1] Byteman: Simplify Java tracing, monitoring and testing with Byteman.
`http://byteman.jboss.org/`

[2] Byteman Programmer's Guide, 3.0.10, Apr 27, 2017
`http://downloads.jboss.org/byteman/3.0.10/ProgrammersGuide.html`

[3] Dinn, Andrew. (2011). A Byteman Tutorial. *JBoss Developer*. Retrieved from:
`https://developer.jboss.org/wiki/ABytemanTutorial`

[4] Dinn, Andrew.
Monitoring Application-Specific Behavior Using Thermostat and Byteman YouTube. 2016. URL:
`https://www.youtube.com/watch?v=teL7qnulUTM`

[5] Yeoman: The web's scaffolding tool for modern webapps.
`http://yeoman.io/`

[6] Yo Code - Extension Generator
`https://code.visualstudio.com/docs/extensions/yocode`

[7] Byteboy: A Byteman Companion Application.
`https://github.com/aptmac/byteboy`

[8] Byteman Language Extension.
`https://github.com/aptmac/byteboy/tree/master/byteman-language-extension`

[9] Textmate: Language Grammars
`https://manual.macromates.com/en/language_grammars`

[10] HPROF: A Heap/CPU Profiling Tool
`https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`

[11] javap - The Java Class File Disassembler
`https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html`

[12] ByteBuddy
`http://bytebuddy.net/`

[13] Introduction to aspectj
`https://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html`