INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

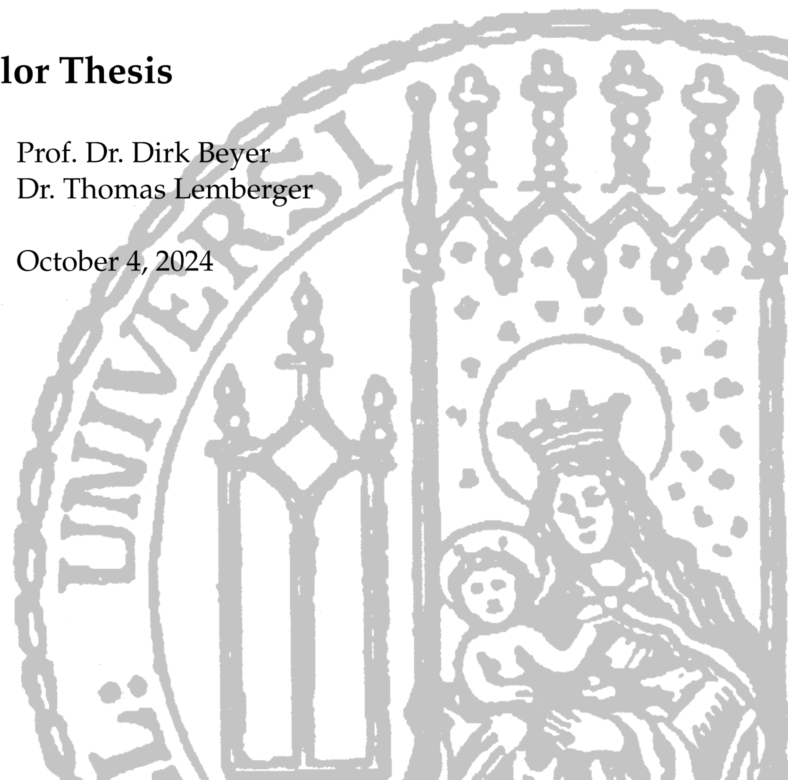
Streamlining Software Verification

A Maven Plugin for Formal Verification of Java Code

Yannick Martin

Bachelor Thesis

Supervisor	Prof. Dr. Dirk Beyer
Mentor	Dr. Thomas Lemberger
Submission Date	October 4, 2024



Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments. I used ChatGPT to improve wordings of single sentences and small paragraphs, and to suggest small snippets of code for evaluation purposes.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich habe ChatGPT verwendet, um Formulierungen für einzelne Sätze und kleine Absätze zu erstellen und zu verbessern, und um kleine Codefragmente für Auswertungen vorzuschlagen.

München, October 4, 2024 Yannick Martin

Abstract

Ensuring the reliability and correctness of software is essential for the safety, security, and robustness of modern systems. However, a significant gap exists in the integration of formal verification tools into mainstream Java development workflows. These tools often require complex configurations and are not easily accessible for typical developers, limiting their adoption. To address this challenge, this thesis introduces the fm-verify plugin, a Maven plugin designed to streamline the formal-verification process for Java applications by integrating JBMC and providing seamless build integration without manual configuration. The plugin was evaluated on 268 open-source Maven projects, with performance impacts measured in terms of runtime and memory usage. The results indicate that fm-verify plugin effectively automates the verification process while maintaining acceptable build performance, making formal verification more accessible in standard Java development environments.

Acknowledgements

First and foremost, I would like to express my gratitude to Prof. Dr. Dirk Beyer, for the opportunity to write this thesis at the Software and Computational Systems Lab.

I would also like to extend my deepest appreciation to my mentor, Dr. Thomas Lemberger. Your unwavering support, constructive criticism, and wealth of knowledge have not only enhanced the quality of this work but also significantly contributed to my personal and academic growth.

Lastly, I would like to thank my family and friends for their constant love and encouragement.

Contents

Contents **vii**

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

2 Related Work **3**

3 Background **5**
3.1 Formal Verification . . . 5
3.2 Bounded Model Checking . . . 5
3.3 CBMC & JBMC . . . 6
3.4 Maven . . . 6
3.5 Tools used in the fm-verify Plugin . . . 8

4 Contribution **9**
4.1 Usage . . . 9
4.2 (Optional) Configuration . . . 9
4.3 Approach . . . 10
4.4 Entry-Point Selection . . . 10
4.5 Running the Verifier . . . 12
4.6 Parsing . . . 12
4.7 Presenting the Results . . . 12

5 Evaluation **15**
5.1 Experimental Setup . . . 15
5.2 Regression Test . . . 15
5.3 Methodology . . . 16
5.4 Quantitative Analysis . . . 17
5.5 Qualitative Analysis . . . 20
5.6 Threats to Validity . . . 23

6 Future Work **25**
6.1 User Study on Development with the fm-verify Plugin . . . 25
6.2 Integration with other build tools . . . 25

6.3 Expanding the fm-verify Plugin 25

7 Conclusion **27**

Bibliography **29**

A List of projects **33**

List of Figures

4.1 Sketch of the architecture 11

5.1 Flowchart of how many repos needed to be verified in order to obtain 100 repositories that worked well with the fm-verify plugin 17
5.2 Increase in runtime caused by the fm-verify plugin 19
5.3 Graph of memory usage during verification of multiple methods 23

List of Tables

5.1 Performance metrics without (w/o) and with (w/) the fm-verify plugin. All values are rounded to three significant places. 18

A.1 List of all 100 projects used for the evaluation 33

Listings

1 Examplary output by the fm-verify plugin 2
2 Java code with a loop 6
3 Unwound code with k=2 6
4 Original Java Code 7
5 SSA Form 7
6 Constraints and Property 7
7 Last part of an examplary output produced by JBMC 7
8 Configuration for the fm-verify plugin 10
9 First example of terminal output by the fm-verify plugin 13
10 Second example of terminal output by the fm-verify plugin 13
11 Summary of all verified methods 21
12 Result of a single method verification 21
13 Excerp from the method in question 22
14 Configuration in the pom.xml file to verify specific methods 23

1 Introduction

As modern software grows increasingly complex, identifying and addressing unwanted behavior (commonly known as spotting bugs) has become a crucial aspect of the software-development lifecycle. Traditional approaches to this problem rely heavily on *testing*, where code is executed with a predefined set of inputs to ensure that the program's output matches the expected results. While effective, testing has a significant limitation: it can only confirm the presence of bugs, not their absence. Additionally, testing for large-scale projects is time-consuming and resource-intensive, with a substantial portion of development time dedicated to this process [12].
In recent years, *formal verification* has garnered more attention within the scientific community as an alternative or complementary approach to traditional testing. Unlike testing, formal verification is the process of using mathematical methods to prove and disprove the correctness of an algorithm. This approach can provide stronger guarantees of software reliability, ensuring that specific properties hold under all possible inputs and conditions, rather than just the scenarios covered by test cases.
Integration and automation are key paradigms in modern software development, yet most formal verifiers are not well integrated with existing build tools. To illustrate, running the Java Bounded Model Checker (JBMC), a formal verifier for Java, involves manually constructing a command-line argument that specifies the location of JBMC's executable, the paths to all relevant Java files, and any configuration parameters, such as the function to verify. An examplary command would be:

```
/pathToJBMC/jbmc-binary -jar /pathToProject/project.jar --function org.example.
    TestProgram.main:([Ljava/lang/String;)V --unwind 5 --trace
```

This process only verifies a single function, meaning a similar command must be constructed repeatedly for each function, leading to substantial manual effort when attempting to verify an entire project.
To address this gap, we contribute the fm-verify plugin, a Maven plugin that seamlessly integrates JBMC into Maven's build lifecycle. This plugin automatically verifies the source code by directing JBMC to the JAR file generated by Maven and initiating one or multiple verification runs. This reduces the command to:

```
mvn verify
```

The last part of the output can be seen in Listing 1. The fm-verify plugin automatically selects main methods as entry points for the verification. It also allows users to set the entry functions through a regular expression. The plugin runs JBMC on all functions that match this regular expression, and reports a verification summary.

```
1 [INFO] Verifying the main method specified in the Manifest
2 [INFO] Found 1 method to verify
3 [INFO] Verifying method: org.example.TestProgram.main(java.lang.String
     [])
4 [INFO] Verification successful
5 [INFO] ***Summary***
6 [INFO] Methods verified: 1, Failures: 0, Unknown: 0
7 [INFO] Verification successful
8 [INFO] For a detailed overview for every verified method, see .../
     target/fm-verify-plugin/verifiedMethods/
```

Listing 1: Examplary output by the fm-verify plugin

By providing an easy-to-use solution, the fm-verify plugin encourages developers to adopt formal verification as an additional tool in their software-development process.
This study aims to address the following research questions:
RQ 1. Is it possible to create a Maven plugin for formal verification using JBMC?
RQ 2. Can the plugin be used on existing Maven projects without requiring manual configuration?
RQ 3. Does the plugin significantly impact build performance in terms of runtime and memory usage?
For answering RQ 1, a modified version of JBMC's regression suite is used. For RQ 2, 268 open-source projects from Github are used. For RQ 3, the same projects as in RQ 2 are used and CPU time and memory usage are measured.

2 Related Work

The Landscape of Verifiers

In recent decades, formal methods have gained considerable attention as a means of improving software reliability and security [12]. These methods encompass a variety of tools and techniques, with software verification being a key area of interest. The Competition on Software Verification (SV-COMP) [5], the largest competition in the field, showcases a wide range of verifiers. However, the majority of these tools are designed for the C programming language, leaving a noticeable gap in verification tools for other languages. Java is the only other language that is represented in the SV-COMP.

The emphasis on C language verifiers can be attributed to two main factors. First, C's ability to perform direct memory manipulation provides developers with numerous opportunities to introduce memory-related vulnerabilities, making verification crucial. Second, C remains one of the most widely used programming languages, largely due to its execution speed and efficiency. However, while there are numerous tools available for C, there are only nine entries in the SV-COMP for Java. Notable and well-established verifiers for Java are Java PathFinder [20], JayHorn [22], and JBMC [15]. Other verifiers for Java are MLBSE [24], GDart [28], COASTAL [39], SWAT [25], and extensions to Java Pathfinder, namely JDart [27], SPF [32], and Java-Ranger [21].

Verification on Real-World Projects

The SV-COMP typically evaluates tools based on isolated tasks and benchmarks that may not fully represent the challenges of verifying complex, real-world projects.

Previous work has explored the use of formal verification tools such as the C Bounded Model Checker (CBMC) in industrial settings. For example, Amazon's integration of CBMC to ensure the reliability of its Commons Library has been highly successful, resulting in a more efficient and bug-free development process [11]. Similarly, CBMC has been used in industrial train control systems to achieve 100% branch coverage, demonstrating the tool's effectiveness in critical safety applications [2]. Since our proposed tool is based on JBMC, which is a derivative of CBMC, these successes further motivate our goals by showing the potential impact of similar verification efforts in the Java ecosystem. Model Checking in general has been applied in various industrial settings such as aviation [9, 26], automotive [16, 34], and chip manufacturing [4, 33]. There are relatively few papers on verifying industrial Java projects. Pasareanu [31] discusses using Java PathFinder at NASA, but most research focuses on academic case studies.

Lack of Integration
For all nine of the Java verifiers in the SV-COMP, we were unable to find any integration into an IDE or build tool. The core version of Java Pathfinder, the verifier on which three of these competitors are based, has a NetBeans plugin and an Eclipse plugin [17], although support for the latter has already been withdrawn.[1] C verifiers on the other hand are better integrated. CBMC has received direct integration into the compilation process via AWS's One Line Scan [3], while CPAchecker [7] offers a web frontend [6], as well as an Eclipse CDT plugin [35].
Alglave et al. [1] see the seamless integration of software verification as a desirable long-term goal. Szekeres et al. [37] also identify integration as a key to adoption. In a more general context, Nielsen [29] emphasizes the importance of usability in tool design.
This paper aims to address this gap by developing a Maven plugin that integrates JBMC to work seamlessly within full Java projects, thereby extending formal verification capabilities into everyday software-development workflows. Our approach specifically targets the shortcomings of Java verification tools by providing a comprehensive solution for real-world applications.

Similar Tools
There exist other tools that similarly aid developers debug their codebase, but are based on different technologies.
EvoSuite [18] is a well-known tool that automatically generates test cases for Java applications and integrates smoothly with Maven. Although EvoSuite is effective at achieving high code coverage, it is not designed to guarantee exhaustive verification, making it less suitable for applications that require stringent safety assurance.
Spotbugs [36] is a popular tool that uses static analysis to find potential bugs, bad practices, and security vulnerabilities. It relies on pattern matching and heuristic-based techniques to detect these bugs. Again, it does not guarantee the correctness of a program and may miss subtle bugs.
Another related tool is eMOP [38], a Maven plugin that supports runtime verification. Unlike JBMC, which performs static analysis, eMOP extends the power of tests through runtime monitoring but does not provide full code verification.

[1] https://github.com/javapathfinder/jpf-core/wiki/Eclipse-Plugin

3 Background

3.1 Formal Verification

Per Bjesse defines formal verification as "[...] the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness" [8]. Unlike traditional testing methods, which can only prove the presence of bugs for a given input, formal verification can prove their absence. This is done by proving that a program holds the specified property for all possible inputs. This ensures a higher level of confidence in the system's reliability, especially in critical domains such as aerospace, automotive, and software security, where failures can have catastrophic consequences. Several techniques and tools have been developed to facilitate formal verification [19]. These include, but are not limited to, model checking [12], theorem proving [10], and symbolic execution [23]. Each technique varies in complexity, applicability, and the types of systems that can be verified. The importance of formal verification in modern software engineering cannot be overstated, as it provides rigorous assurance of software correctness, safety, and security.
For the purposes of this paper, a *verification task* is defined as a program to be analyzed and the specification to be checked. A *verification run* is the execution of a verifier on a verification task.

3.2 Bounded Model Checking

Model Checking is a technique that originated in the verification of finite-state systems, such as hardware controllers. However, software can inherently have infinite states, as there are loops or recursive function calls that can be designed to run forever. *Bounded Model Checking* (BMC) sets a limit, a bound, on each loop or recursive function call. This is called *k-induction*, where the value of *k* is the maximum number of times a piece of code can be re-executed. This produces a loop-free, finite-state program that can be formally verified.
Listings 2 and 3 show the principle of unwinding. Note that in this scenario, it would take a bound of *k=10* to find a path that violates the assert statement.
This renders BMC an inherently incomplete approach, since property violations can only be found up to a certain bound. An ongoing challenge is to determine a reasonable bound for each project, as it is essentially a tradeoff between runtime and verification completeness. The exponential complexity of BMC is discussed by Clarke et al. [14].

```java
int i = 0;
while (NonDetBool()) {
    i = i + 1;
}
assert(i < 10);
```

Listing 2: Java code with a loop

```java
int i = 0;
if(NonDetBool()) {
    i = i + 1
    if(NonDetBool()) {
        i = i + 1;
    }
}
assert(i < 10);
```

Listing 3: Unwound code with *k*=2

3.3 CBMC & JBMC

CBMC: The C-Bounded Model Checker (CBMC) [13] is a program that implements Bounded Model Checking in the C programming language. It transforms the programming instructions into a logical formula, which can then be checked for violations of safety properties, such as the correctness of pointer constructs, array bounds, and user-provided assertions.
JBMC: Java Bounded Model Checker (JBMC) [15] is an extension of CBMC, which consists of a frontend for parsing Java bytecode and a Java Operational Model (JOM), which is an exact but verification-friendly model of the standard Java libraries. This allows Java programs to be verified in a manner similar to their C counterparts. JBMC works with both '.class' files, which contain compiled Java bytecode, and '.jar' files (called JARs). These are archive files that contain .class files, among other resources and meta-information. Listing 7 shows a typical, exemplary output produced by JBMC. It contains all the checked properties and their status.
JBMC offers a wide range of configuration options, enabling precise control over the verification process. Key options include specifying the path to the Java classes, selecting the class or method to be verified, and setting the k-induction value with *–unwind k*. The unwind parameter is critical in bounded model checking because it controls the unwinding of loops and recursive functions that could theoretically run indefinitely. For advanced users familiar with formal verification, JBMC also provides options to customize features such as choosing a specific SAT solver or restricting certain computations.
Listings 4 to 6 display a simplified version of how JBMC/CBMC operate. The original source code is converted to a Static Single Assignment (SSA) form and then concatenated into logical expressions. These expressions are then given to a SAT solver to verify that the assertions always hold true.

3.4 Maven

Maven is a widely used build automation tool designed primarily for Java projects. It operates based on the concept of a Project Object Model (POM), where project configurations such as dependencies, build phases, and other critical information are defined in a 'pom.xml' file. Maven organizes the build process into a series of well-

```
int y = m * n;
if (y > 10) {
    y = 5;
} else {
    y += 3;
}
assert(y < 12);
```

Listing 4: Original Java Code

```
int y1 = m * n;
int y2 = (y1 > 10) ? 5 : (y1 + 3);
assert(y2 < 12);
```

Listing 5: SSA Form

```
// Constraints (C)
C := (y1 == m * n) ∧ (y2 == ((y1 > 10) ? 5 : (y1 + 3)))

// Property (P)
P := (y2 < 12)
```

Listing 6: Constraints and Property

```
my/petty/examples/Simple.java function java::my.petty.examples.Simple.main:([Ljava/lang/String;)V
[java::my.petty.examples.Simple.main:([Ljava/lang/String;)V.1] line 6 no uncaught exception: SUCCESS
[java::my.petty.examples.Simple.main:([Ljava/lang/String;)V.null-pointer-exception.1] line 6 Null pointer check: SUCCESS
[java::my.petty.examples.Simple.main:([Ljava/lang/String;)V.assertion.1] line 8 assertion at file my/petty/examples/Simple.java line 8 function java::my.petty.examples.Simple.main:([Ljava/lang/String;)V bytecode-index 16: FAILURE
[java::my.petty.examples.Simple.main:([Ljava/lang/String;)V.null-pointer-exception.2] line 8 Null pointer check: SUCCESS
[java::my.petty.examples.Simple.main:([Ljava/lang/String;)V.null-pointer-exception.3] line 8 Null pointer check: SUCCESS

** 1 of 47 failed (2 iterations)
VERIFICATION FAILED
```

Listing 7: Last part of an examplary output produced by JBMC

defined lifecycle phases, such as compilation, testing, packaging, and deployment. Each phase has specific tasks that provide a structured workflow for managing project builds.

The core of Maven's functionality is provided through a rich ecosystem of plugins that handle tasks such as dependency management, code compilation, test execution, and application packaging. Maven also allows for the development of custom plugins, offering extensive flexibility to extend its functionality. Plugins in Maven have access to many of Maven's internal variables, including the outputs of previous build phases. This enables plugins to manage inter-phase dependencies and handle outputs in a controlled manner. Moreover, Maven has conventions for where output is written, ensuring consistency and structure within the build process.

Custom plugins can be developed to meet specific project needs and are configured directly within the pom.xml. This configuration makes it possible to define dependencies between plugins or jobs, making it easier to ensure that tasks are performed in the correct order. Maven provides extensive support for plugin configuration, allowing developers to customize plugin behavior to fit their exact requirements.

3.5 Tools used in the fm-verify Plugin

The following is a list of frameworks and tools used by the fm-verify plugin, but it's not important to understand their functionality.

Javassist
Javassist (Java Programming Assistant)[1] is a Java library that simplifies bytecode manipulation, allowing developers to dynamically modify Java classes at runtime or during load time without directly writing bytecode. It's commonly used for tasks like enhancing classes for frameworks or generating proxy classes in Java applications. It is used in the fm-verify plugin to retrieve declared methods from the JAR created by Maven.

Jackson
Jackson[2] is a popular Java library used for processing JSON data, allowing serialization and deserialization between Java objects and JSON with ease. It provides powerful data binding capabilities and supports features like custom serializers, deserializers, and tree models to work with JSON in various formats. It is used in the fm-verify plugin to parse output of JBMC.

[1] https://www.javassist.org/
[2] https://github.com/FasterXML/jackson

4 Contribution

We present the *fm-verify plugin* for Maven, which is designed to automate the verification of Java source code within a project. This plugin uses JBMC to ensure the correctness of the code by analysing it for potential errors.
Sections 4.1 and 4.2 provide a guide to using the fm-verify plugin, while Sects. 4.3 to 4.7 detail the inner workings of the plugin and present the design choices made.

4.1 Usage

To use the fm-verify plugin in a Maven project, simply add the code shown in Listing 8 to the projects pom.xml file.
Maven will automatically download the plugin and JBMC will be automatically installed when the plugin is run. It is executed when invoking the 'verify' goal of Maven, or separately with 'mvn org.sosy_lab:fm-verify-maven-plugin:verify'. Note that the fm-verify plugin requires a packaged JAR, so running it as part of a lifecycle is recommended.

4.2 (Optional) Configuration

The fm-verify plugin is designed to work 'out of the box', i.e. with no configuration required. Nevertheless it has several options to customize it to ones specific needs. These can be set in the configuration section of the plugin in the pom.xml file.
This section presents some, but not all, of the available options:

verifyMethods This argument expects a regular expression that defines all methods to be verified. This expression is matched against the source-code notation of methods. This allows users to verify a specific selection of methods.

verifierArgs Advanced users can pass arguments to JBMC here. These arguments are given to JBMC without modification.

verificationRuntimeLimit Sets the time limit of a single verification run, before the process is forcibly terminated. The default value of 30 seconds is an educated guess to balance responsiveness with the limits of human patience.

ignoreProperties This parameter expects a list of Strings. It allows the users to ignore certain properties. It can also be used to ignore violations that come from a specific method.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.sosy_lab</groupId>
      <artifactId>fm-verify-maven-plugin</artifactId>
      <version>1.0.0</version> <!-- or newest version -->
      <!-- You can add configuration like this
      <configuration>
        <verifierArgs>--unwind 10</verifierArgs>
      </configuration>
      -->
      <executions>
        <execution>
          <goals>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Listing 8: Configuration for the fm-verify plugin

failsOnError Testing frameworks like Junit cause the Maven build to fail if there are test failures. The same behavior can be achieved with this parameter. This is not the default configuration, as there is usually a plethora of violated properties.

traceLevel Sets the trace level for JBMC. Traces can provide valuable insight into non-trivial failed properties. However we have found that setting the trace option (both *–trace* and *–compact-trace*) regularly causes JBMC not to terminate on tasks that would otherwise terminate. Therefore, the default configuration doesn't print traces.

saveResultsForFailuresOnly To avoid cluttering the output, the fm-verify plugin only saves files to the target folder for methods that fail the verification. The parameter can be changed to save results for successfully verified methods as well.

4.3 Approach

As shown in Fig. 4.1, our approach starts with Maven compiling the project into a JAR file. Next, JBMC is run on the JAR, sometimes multiple times with different entry points. Finally, the output from JBMC is parsed and analyzed to detect any property violations.

4.4 Entry-Point Selection

JBMC requires an entry point to begin its analysis of a class file or JAR. Typically, this entry point is a user-specified function or the 'main' method. However, many modern software architectures do not use main methods, especially in projects that

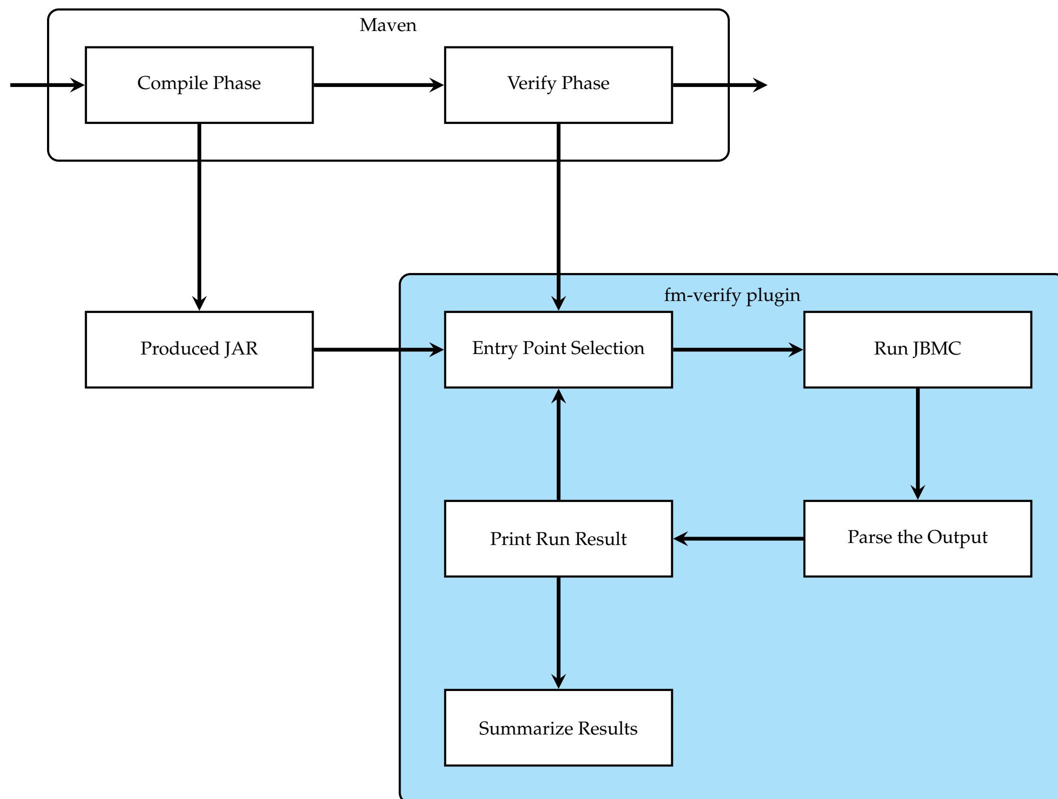


Figure 4.1: Sketch of the architecture

provide APIs with multiple methods but no executable main function. Attempting to run JBMC on such a JAR without a designated entry point results in an error.

To address this, the fm-verify plugin provides a feature to automatically select entry points. If a main class is specified within the JAR, it uses this class as the entry point, as it is likely to be the most commonly used function. If no main class is specified, the fm-verify plugin uses Javassist to detect any ‘public static void main(String[])‘ methods. It then uses these methods as entry points. If no main methods are found, it defaults to using every public method as an entry point.

The last part is a fallback. In larger projects, where there may be hundreds or thousands of public methods, this has a significant impact on runtime. To refine this process, the fm-verify plugin provides the ’**verifyMethods**’ parameter, which allows users to specify a regular expression for filtering method names. This ensures that only relevant methods are considered as entry points. In the absence of main methods, users are encouraged to set this parameter to include all critical methods.

In Chapter 5, the impact of entry point selection on project verification will be explored.

4.5 Running the Verifier

JBMC requires the classpath to the JAR and the selected entry point, which is handled by the plugin. Additional arguments can be configured in the project's pom.xml under the plugin's configuration section.
If no specific arguments are provided by the user, the unwind parameter is set to a default value of 5. The choice of this parameter is a trade-off. Setting no value may result in JBMC not terminating, while setting a high value can significantly increase the verification time, especially if the loop body contains a lot of code. However, using a low value may prevent the detection of certain violations that a higher value would detect.
Each verification run is limited to a maximum runtime of 30 seconds. If no result is obtained within this time, the method being verified is marked as inconclusive.

4.6 Parsing

JBMC proves to be helpful and cooperative in parsing its output, as it clearly communicates its status via its exit code. This means that interpreting its output can be done with minimal performance overhead by a simple switch case.
The fm-verify plugin extends JBMC's capabilities by providing additional features when parsing JBMC's output.
One of these features is the ability to ignore certain failed properties. For example, it is common in microservices to have contracts that guarantee non-null values as parameters. However, JBMC may still detect potential null pointer exceptions as violations of these properties. If a developer determines that these particular violations are not relevant or important, they can choose to ignore them using the plugin's functionality.
Because property names in JBMC include both type and location, it is also possible to ignore property violations that arise from specific methods or packages. This is especially helpful, when Maven builds a so-called 'fat JAR' that also contains the project's dependencies. In such a case, JBMC may find unsafe code of a dependency. Fixing such a problem is usually beyond the scope of a developer, so they might want to ignore these warnings.
In case the user passes the '–json-ui' argument to JBMC, a backend parser is used that accepts input in the JSON format. This parser does not provide the same functionality as the default parser.

4.7 Presenting the Results

The resulting verdict is then presented to the user in two ways. Immediately visible in the terminal, as well as in text files for permanent storage.

```
1  [INFO] --- fm-verify:1.0.0:verify (default) @ myproject1 ---
2  [INFO] Verifying Methods that fit this regex: .*(main|test).*
3  [INFO] Found 2 methods to verify
4  [INFO] Verifying method: com.example.TestProgram.main(java.lang.String
       [])
5  [INFO] Verification successful
6  [INFO] Verifying method: com.example.TestProgram.testSomething()
7  [INFO] Verification successful
8  [INFO] ***Summary***
9  [INFO] Methods verified: 2, Failures: 0, Unknown: 0
10 [INFO] Verification successful
11 [INFO] For a detailed overview for every verified method, see .../
       target/fm-verify-plugin/verifiedMethods/
```

Listing 9: First example of terminal output by the fm-verify plugin

```
1  [INFO] --- fm-verify:1.0.0:verify (default) @ myproject6 ---
2  [INFO] Verifying Methods that fit this regex: .*(main|test|
       sameMethodName).*
3  [INFO] Found 7 methods to verify
4  [ERROR] ***Summary***
5  [ERROR] Methods verified: 7, Failures: 3, Unknown: 0
6  [ERROR] Verification failed
7  [ERROR] Failed methods:
8  [ERROR] com.example.TestProgram.testSomething1Fail()
9  [ERROR] com.example.TestProgram.testSomething2Fail()
10 [ERROR] com.example.TestProgram.testSomething4Fail()
11 [INFO] For a detailed overview for every verified method, see .../
       target/fm-verify-plugin/verifiedMethods/
```

Listing 10: Second example of terminal output by the fm-verify plugin

Terminal
The terminal serves as a direct means of communication with the user. This means, that it can be used to display the system status for better understanding. Information such as JBMC being installed, the number of methods being verified, or any warnings and errors are displayed here. When verifying 3 or less methods, each method name and its individual verification verdict is displayed. This is not done for 4 or more methods, as this would clutter the terminal too much. The cutoff point is also configurable. In addition, a summary is printed at the end, listing all failed and unknown methods, if there are any. Two examples can be seen in Listings 9 and 10.

Files
The summary is also printed in a subdirectory of Mavens target folder. In addition, a more detailed result is printed for each failed method as a text file, listing all failed properties and their source locations in an own format, mostly . These files are located in a subdirectory, that is named as the method verified in this verification task. This allows for reviewing the results in a more organized manner. Additionally the fm-verify plugin can be configured to print JBMC's complete output for each verification run.

5 Evaluation

5.1 Experimental Setup

To address the research questions, we conduct a series of experiments to evaluate the performance of the fm-verify plugin. All benchmarks are executed on a machine running Ubuntu 22.04.4 LTS, with an Intel Core i7-8565U CPU and 16 GB of RAM. The following tools are used:

- BenchExec: version 3.23
- Java: OpenJDK 21.0.4
- JBMC: 5.72.1 (cbmc-5.72.1-22-g1ab48b9654-dirty)
- Maven: 3.9.8

5.2 Regression Test

To provide an initial evaluation addressing RQ1 and verify the plugin's effectiveness in identifying unsafe code, a suitable benchmark is required for comparison. Fortunately, JBMC has a regression suite. [1] [2]

The fm-verify plugin, being a plugin for Maven, requires a different environment than the one provided in the regression suite. A custom Python script is used to create a Maven project for each test, run it, and check that the output of the fm-verify plugin matches the output described in the test.

Of the 727 tests in the regression suite, 121 have to be skipped for various reasons. Some of these include the fact that certain functionality is already covered by the fm-verify plugin, such as class paths and proper usage. In addition, several projects fail to compile due to various issues, and a number of tests are skipped because these test cases cover known bugs in JBMC. There are also other factors that contributed to the need to skip these tests that aren't fully listed here.

Of the 605 tests, only one results in a different output from the fm-verify plugin than is described in the test. Technically, 7 result in a different output, but 6 of them are caused by erroneous test cases. In these 6 cases, the provided Java source code and byte code files are not equivalent, leading to this different output.

The only real mismatch is due to the –unwind value being set by the fm-verify plugin, when no arguments are given. Ironically, setting an unwind value of 6 would produce the desired result. We could have adjusted the default value of

[1] A large amount of tests to check that new versions don't break existing functionality

[2] https://github.com/diffblue/cbmc/tree/develop/jbmc/regression/jbmc

this parameter to 6, so that the entire regression suite would run successfully, but we decided to leave the default value at 5, as it serves as a good reminder, that low k-induction values won't find all problems. A more detailed overview of this regression test, including the script used to modify the regression tests can be found in the plugin's repository. [3]
With this test done, we consider Research Question 1 to be successfully and positively answered.

5.3 Methodology

To answer RQ 2 and RQ 3, we conduct an empirical study on 268 open-source projects that utilize Maven. GitHub serves as the source for these projects. A custom-built crawler automates the verification process by performing the following steps:

1. Collect a list of open-source projects.
2. Clone each project's repository.
3. Initially compile the projects to ensure functionality.
4. Compile each project multiple times to establish baseline metrics, including time and memory usage.
5. Integrate the fm-verify plugin into each project's pom.xml file.
6. Recompile each project multiple times with the plugin enabled to measure the corresponding metrics.

Project Selection
Projects are selected using a random sampling approach to minimize bias and ensure that the evaluation was not skewed toward projects that are inherently compatible with the plugin. We aim to include a diverse mix of project sizes to reflect a broad range of real-world scenarios. In earlier testing we found that randomly sampling projects on Github tends to select many small, personal repositories.
To address this, repositories with no star or a single star are filtered out and only repositories with two or more stars are considered. We also found, that sorting projects by their most recent updates on GitHub provided a more balanced mix of project sizes, as larger projects tend to be updated more frequently, either by maintainers or automation tools such as maintenance bots. This method also reduces the likelihood of selecting outdated projects that might use deprecated Java versions or dependencies.
In addition, the repository size limit is to 1 GB, as in earlier testing we encountered some repositories that were unnecessarily large.

Initial Run
For each project, a compilation run is performed prior to the measured runs. This serves two purposes. First, this new project may require dependencies via Maven that are not installed and need to be downloaded. If there was no initial run, this would negativly affect the first measured run without the plugin. Second, it serves as

[3] https://gitlab.com/sosy-lab/software/ba-maven-plugin/-/tree/main/regression-test?ref_type=heads

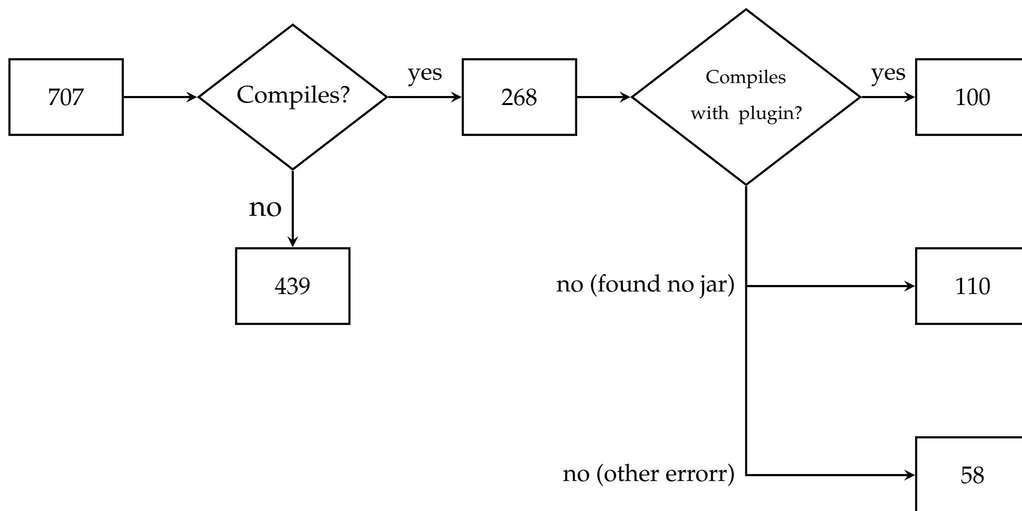


Figure 5.1: Flowchart of how many repos needed to be verified in order to obtain 100 repositories that worked well with the fm-verify plugin

a check on which projects compile and which do not. This serves as an unintentional reflection of the current state of open-source projects.

Measured Runs

BenchExec is used to measure real-world elapsed time (referred to as wall time), CPU time (the time a processor core was in use), and the maximum amount of memory used by each run. We focus on wall time over CPU time, as JBMC runs on a single thread and thus CPU time scales linearly with wall time. This paper does not introduce a new or enhanced algorithm. Instead, it emphasizes the user-centric experience of the plugin. Said user will be more concerned with the real time elapsed rather than how hard their processor was working.

Each repository is compiled 10 times to reduce the likelihood of an outlier negatively skewing the average. 'mvn clean verify' is used as an argument. This call executes two goals; the *clean* goal, which removes the target folder to ensure the same environment for each subsequent run, and the *verify* goal, the lifecycle in which the fm-verify plugin operates. A time limit of 10 minutes is set for the execution of each command.

5.4 Quantitative Analysis

5.4.1 Project Filtering and Metrics

Figure 5.1 shows the distribution of crawled repositories. Of the 707 projects that use Maven as their build tool, only 268 compile successfully. Common problems included missing dependencies and test failures. Of the 268 projects that compile, 168 are incompatible with the fm-verify plugin. A common issue is that the plugin requires a packaged JAR, and 110 projects do not produce one.

Table 5.1: Performance metrics without (w/o) and with (w/) the fm-verify plugin. All values are rounded to three significant places.

	Mean	**Median**	**SD**	**Min**	**Max**
Wall time w/o plugin (in s)	29.1	9.77	61.2	2.27	396
Wall time w/ plugin (in s)	51.4	27.0	74.2	3.17	438
CPU time w/o plugin (in s)	80.9	36.7	141	8.11	1 070
CPU time w/ plugin (in s)	106	59.4	148	11.4	1 100
Memory w/o plugin (in MB)	661	438	740	198	6 560
Memory w/ plugin (in MB)	1 780	750	2 150	230	9 320

Initially, this seems to indicate a limitation in the applicability of the plugin. However, further investigation of 30 such projects reveal that 26 of them have a parent-child structure, where the root pom.xml manages multiple sub-projects that produce output artifacts, mostyly JAR files. By adding the fm-verify plugin to the child pom.xml files, these sub-projects are successfully verified. The remaining 4 of these 30 projects produce a '.war' (Web Application Resource) file, which the fm-verify plugin currently does not support.
For the 58 projects where other errors occurr after adding the fm-verify plugin, common issues include:

- 28 are caused by the Maven Enforcer plugin.[4]
- 11 are caused by an I/O-related bug in the fm-verify plugin that has since been fixed.
- 9 exceed the 10-minute runtime limit set in this experiment.

The remaining 10 issues were diverse, involving plugins such as Spotless, Surefire, and Checkstyle.
The full list of the 100 compatible projects can be found in Appendix A.
Despite these challenges, we argue that the plugin can be used effectively without manual configuration, especially when developers familiar with their project's architecture know which pom.xml file to modify. Therefore, we consider Research Question 2 to be answered in the affirmative.

5.4.2 Performance Evaluation

To answer Research Question 3, we measure wall time, CPU time, and maximum memory usage. CPU time can exceed wall-time due to multithreading, where multiple CPU cores are used in parallel.
Table 5.1 presents descriptive statistics of the measurements. Visual inspection suggests a notable difference in the metrics. However, statistical tests are performed to confirm this.
Each repository provides 10 measurements with the fm-verify plugin and 10 without it. Six lists of 1,000 elements each were created, pairing each measurement by index. To test whether the means of two lists are statistically different, the usual approach is a t-test, which requires normally distributed data. We use the Shapiro-Wilk-Test

[4]The Enforcer plugin provides goals to control certain environmental constraints such as Maven version, JDK version, and OS family along with many more built-in and user-created rules.

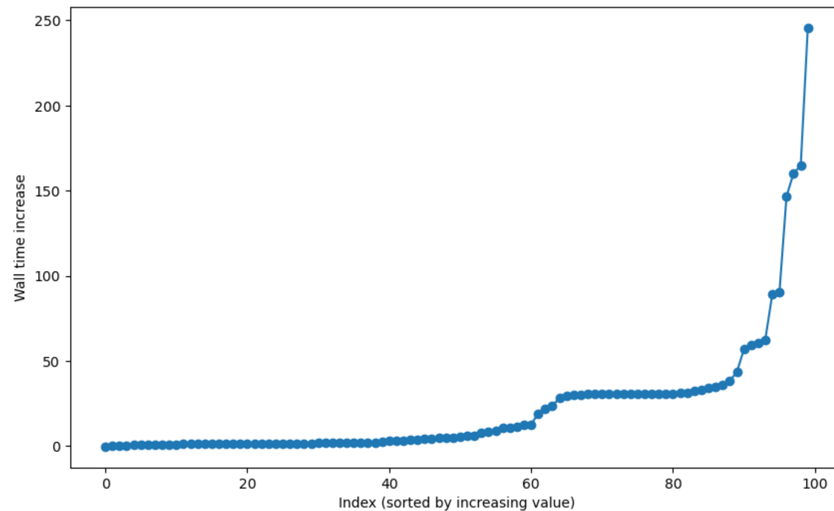


Figure 5.2: Increase in runtime caused by the fm-verify plugin

to test for normality. By convention, we consider values below 0.05 as significant, with values below 0.01 as highly significant.
These are the p-values from the Shapiro-Wilk-Test:

- Wall time w/o (seconds): 0.0
- Wall time w/ (seconds): 2.11×10^{-42}
- CPU time w/o (seconds): 0.0
- CPU time w/ (seconds): 7.01×10^{-45}
- Memory w/o (megabytes): 0.0
- Memory w/ (megabytes): 2.32×10^{-39}

Since the data is not normally distributed, we use the Wilcoxon Signed-Rank Test to evaluate the differences. The resulting p-values are:

- Wall time: 3.94×10^{-162}
- CPU time: 4.24×10^{-148}
- Memory usage: 5.05×10^{-125}

These results are quite clear: the fm-verify plugin significantly increases resource consumption.
However, this does not account for the full complexity of the data. There is a notable difference between the Mean and the Median of each list, as can be seen in Table 5.1. In addition, the Standard Deviation (SD) is always higher than the Mean. This is typically indicative of data with high variance. Figure 5.2 plots the increase in runtime (with plugin minus without plugin) for each repository. Notably, about half the repositories exhibit an almost negligible runtime increase of under 5 seconds, while a few outliers at the high end greatly influence this shift in the mean.
We also observe a distinct step at the 30-second mark on the y-axis, which likely corresponds to the 30-second runtime limit for verification tasks. Additional clusters around 60 and 90 seconds suggest that serveral methods in these repositories are hitting the same limit. One such example is discussed in Section 5.5.2.

Overall, the fm-verify plugin significantly increases build time and memory usage, and Research Question 3 is considered answered.

5.5 Qualitative Analysis

This section presents two case studies,namely the open-source projects Buji-pac4j and SystemDS, to illustrate the functionality of the fm-verify plugin and its application in identifying and fixing software bugs. Both projects are selected based on their ranking among the top 10 repositories by GitHub stars, each having aquired over 400 stars. Repositories with many stars are often indicative of high code quality, active maintenance, and a strong community, making them good candidates for evaluation.
The first project shows minimal performance overhead, with a wall-time increase of less than 5 seconds while verifying multiple methods. The second project represents the opposite, a case where the 30-second runtime limit is hit on its main method. In addition, this project has the most Lines of Code (LOC) according to our measurements (although these may not be accurate) and ranks as the third largest in physical size.

5.5.1 Project buji-pac4j

The first case study examines the buji-pac4j project[5], which serves as a bridge between two security frameworks for web applications: Apache Shiro and pac4j. This integration enables Shiro's authentication processes to be managed by pac4j, a framework specialized in connecting to various external identity providers and supporting a wide range of protocols, including OAuth, OpenID Connect, SAML, CAS, and others.
Since Buji-pac4j is an API, it does not contain a main class or main methods. The fm-verify plugin analyzes 33 public methods in this project and detecting property violations in 13 of them. Listing 11 shows the output generated by the plugin, which is displayed on the console for immediate feedback and stored in target/fm-verify-plugin/overallResult.txt for subsequent review. However, simply identifying the methods that failed verification is insufficient without understanding the specific cause and location of the failure. To address this, the fm-verify plugin generates detailed output files within directories named after the respective methods.
Listing 12 shows the result of the second failed method, where two potential NullPointerExceptions are identified on lines 56 and 58. Listing 13 displays the referenced method. These exceptions likely occur because the code calls SecurityUtils.getSubject().login() without checking whether getSubject() returns a non-null object. This hypothesis can be confirmed by configuring the fm-verify plugin to display the JBMC trace, which is disabled by default, as discussed in Sect. 4.2. By assigning the Subject in a separate step and checking for null values, this method was subsequently verified as safe by JBMC.

[5] https://github.com/bujiio/buji-pac4j

```
1  ***Summary***
2  Methods verified: 33, Failures: 13, Unknown: 0
3  Verification failed
4  Failed methods:
5  org.pac4j.framework.adapter.FrameworkAdapterImpl.
       applyDefaultSettingsIfUndefined(org.pac4j.core.config.Config
       )
6  io.buji.pac4j.subject.Pac4jSubjectFactory.createSubject(org.
       apache.shiro.subject.SubjectContext)
7  io.buji.pac4j.subject.Pac4jPrincipal.getProfile()
8  io.buji.pac4j.subject.Pac4jPrincipal.equals(java.lang.Object)
9  io.buji.pac4j.subject.Pac4jPrincipal.toString()
10 io.buji.pac4j.subject.Pac4jPrincipal.getName()
11 io.buji.pac4j.subject.Pac4jPrincipal(java.util.List,java.lang.
       String).<init>
12 io.buji.pac4j.profile.ShiroProfileManager.removeProfiles()
13 io.buji.pac4j.context.ShiroSessionStore.getSessionId(org.pac4j.
       core.context.WebContext,boolean)
14 io.buji.pac4j.context.ShiroSessionStore.get(org.pac4j.core.
       context.WebContext,java.lang.String)
15 io.buji.pac4j.context.ShiroSessionStore.set(org.pac4j.core.
       context.WebContext,java.lang.String,java.lang.Object)
16 io.buji.pac4j.context.ShiroSessionStore.destroySession(org.
       pac4j.core.context.WebContext)
17 io.buji.pac4j.token.Pac4jToken.getCredentials()
```

Listing 11: Summary of all verified methods

```
1  Verification failed
2  [java::io.buji.pac4j.util.ShiroHelper.populateSubject:(Ljava/
       util/LinkedHashMap;)V.null-pointer-exception.4] line 56 Null
        pointer check: FAILURE
3  [java::io.buji.pac4j.util.ShiroHelper.populateSubject:(Ljava/
       util/LinkedHashMap;)V.null-pointer-exception.7] line 58 Null
        pointer check: FAILURE
```

Listing 12: Result of a single method verification

NullPointerExceptions are common in verification of real-world projects, often due to contracts that implicitly guarantee non-null returns without formally declaring this constraint. To handle such cases, the fm-verify plugin allows the user to ignore certain properties. Configuring the plugin to ignore NullPointerExceptions reduces the number of methods with failed verifications from 13 to 5. The remaining failures are due to dynamic cast checks, which occured because JBMC can not infer the inheritance hierarchy of objects from the pac4j core library.
By ignoring these specific properties, all methods were eventually considered safe. However, it is important to note that this only indicates the absence of exceptions; it does not guarantee that the system behaves as expected. This limitation is due to the scarcity of assert statements in the project, which would otherwise validate the correctness of individual values at various stages of execution.

```java
51 public static void populateSubject(final LinkedHashMap<String, UserProfile> profiles) {
52   if (profiles != null && profiles.size() > 0) {
53     final List<UserProfile> listProfiles = ProfileHelper.flatIntoAProfileList(profiles);
54     try {
55       if (IS_FULLY_AUTHENTICATED_AUTHORIZER.isAuthorized(null, null, listProfiles)) {
56         SecurityUtils.getSubject().login(new Pac4jToken(listProfiles, false));
57       } else if (IS_REMEMBERED_AUTHORIZER.isAuthorized(null, null, listProfiles)) {
58         SecurityUtils.getSubject().login(new Pac4jToken(listProfiles, true));
59       }
60     } catch (final HttpAction e) {
61       throw new TechnicalException(e);
62     }
63   }
64 }
```

Listing 13: Excerp from the method in question

5.5.2 Project SystemDS

The second case study examines the Apache SystemDS project[6], an open-source machine-learning system designed for the end-to-end data science lifecycle.

When attempting to verify the main methods specified in the JAR file, the process times out. This result is not unexpected, as the project contains 268,862 lines of code, making comprehensive verification through a single entry point challenging. Initially, reducing the unwind value to 1 by manually specifying it in the verifier arguments still results in a timeout within the 30-second limit. Increasing this value to an arbitrarily large number caused another issue: after more than 80 minutes and over 76 GB of memory usage (16 GB RAM and 60 GB swap partition), the verification process was terminated manually to avoid exhausting system resources. As Daniel Kroening notes in [30]: "CBMC isn't designed to scale." For a project that provides a framework for machine learning, a field of computer science that is itself resource-intensive, it is not surprising that the verification of such a framework is so resource-intensive.

However, this does not render the verification process futile. The fm-verify plugin allows users to manually specify methods for verification. Developers who are familiar with their codebase can identify key entry points that cover critical sections of code without triggering long-running processes. An an example, we verify the *Utils* package using a regular expression to match all public methods within that package, as shown in Listing 14.

This configuration results in 350 methods being individually verified. Although we did not take precise measurements, the total runtime increased from approximately 30 seconds to 12 minutes. The runtime scales linearly with the number of methods verified. However, the memory consumption remains relatively stable as many small tasks are performed instead of a single large one. Figure 5.3 illustrates the fluctuations in memory usage, with the peak memory consumption increasing from approximately 2.4 GB without the fm-verify plugin to 3.0 GB with it.

These results suggest that a verification or testing team could use the plugin to iteratively verify their codebase, progressively identifying and fixing potential problems.

[6] https://github.com/apache/systemds

```
<groupId>org.sosy_lab</groupId>
<artifactId>fm-verify-maven-plugin</artifactId>
<version>1.0.0</version>
<configuration>
  <verifyMethods>.*public  .*org.apache.sysds.utils.*</
      verifyMethods>
  <verifierArgs>--unwind 1</verifierArgs>
</configuration>
<executions>
  <execution>
    <goals>
      <goal>verify</goal>
    </goals>
  </execution>
</executions>
```

Listing 14: Configuration in the pom.xml file to verify specific methods

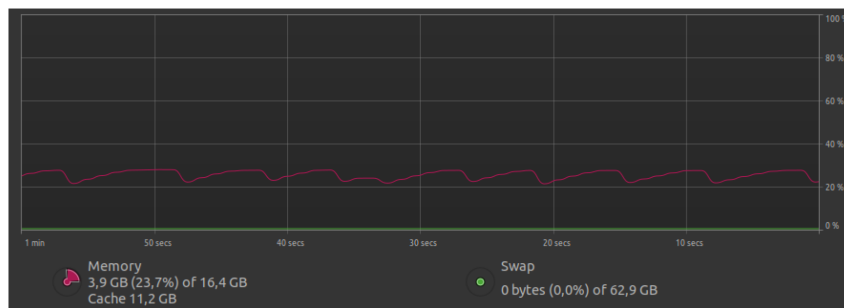


Figure 5.3: Graph of memory usage during verification of multiple methods

5.6 Threats to Validity

Internal Valididy

There is a possibility that measurement inaccuracies occurred during our experiments. To mitigate this risk and ensure accurate and reproducible results, we used BenchExec, a widely recognized benchmarking tool, to measure metrics such as wall time, CPU time, and memory usage of system calls. BenchExec provides precise and controlled execution environments, which reduces the likelihood of measurement noise and bias. Furthermore, we executed each experiment 10 times, observing only marginal deviations between runs, ensuring the stability and consistency of our measurements. However, the small deviations, while minimal, could still introduce minor uncertainty into the final results.

When conducting the regression suite for JBMC (as described in Sect. 5.2), not all aspects of the provided test specifications were considered. Specifically, some tests required JBMC to produce a particular error message or an exact output for failed properties, but we only checked whether the overall classification (i.e., "Safe" or "Unsafe") was consistent with expected outcomes. This approach might overlook nuanced failures that do not affect the overall classification but still indicate behavior changes in the system. While this limitation does not affect the primary goal of

evaluating verification correctness, it may reduce sensitivity to subtle behavioral variations or improvements in the tool's feedback. However we consider these variations as improbable, as our work does not affect the inner workings of JBMC.

External Validity
We aimed to include a representative mix of projects to ensure that our results generalize to the broader Java ecosystem. However, the randomness of our project selection introduces a potential sampling bias. Despite our efforts, we cannot guarantee that our sample captures the full diversity of the Java ecosystem, especially considering differences in project size, domain, and coding practices. For example, our selection criteria of projects with at least two stars and a size of less than 1 GB may inadvertently exclude very large, complex, or high-stakes projects (such as those in enterprise environments or with large teams), which could have different verification challenges or patterns. We classified these larger projects as outliers, focusing instead on projects that reflect average development practices. However, this does limit the external validity to smaller-scale projects.
Moreover, we only used publicly available repositories to conduct our evaluation. This may lead to a selection bias, as enterprise or proprietary projects, which are often not hosted on public platforms, could have significantly different code structures, architectural patterns, or testing practices. The exclusion of these types of projects means that our results might not generalize well to codebases commonly found in closed-source environments. Proprietary systems may place different demands on verification tools, including larger codebases, different concurrency patterns, or stricter safety-critical requirements.
The default configuration settings for the plugin (e.g., an unwind value of 5, and a 30-second timeout for verification runs) may not represent the optimal configuration for every project or environment. While these values were chosen to strike a balance between precision and performance across various projects, they may not generalize to all types of verification tasks. Different projects could require different unwind limits or timeout values, and adjusting these parameters could yield more precise verification outcomes in specific contexts. Nonetheless, we argue that there are no universally optimal settings for these parameters, and that our chosen values represent a reasonable trade-off for evaluating JBMC's performance on a wide range of typical use cases.

6 Future Work

In this chapter, we propose potential future work, both scientific and practical in nature.

6.1 User Study on Development with the fm-verify Plugin

In this paper, in Sect. 5.5 we have provided a retrospective analysis of how verification can aid in development and bug-chasing. We did not provide a real-world evaluation of how a developer might use the plugin in a real-world scenario and whether it would improve debugging and overall progress. Such a study is beyond the scope of this paper and may prove difficult to implement. Providing a controlled environment while also representing a large portion of the population and their development routines seems difficult. However we believe it would provide great insights in understanding and improving the general development workflow.

6.2 Integration with other build tools

Maven, which is one of the, if not the largest build tool for Java application, now has an easy to use plugin. However, there is another popular build tool, Gradle, which is becoming increasingly popular. Especially larger projects use it because of its increased flexibility and performance over Maven. At the time of writing, a corresponding project is being developed at SoSy-Lab. We could envision a collaboration to create an improved plugin that is compatible with both Maven and Gradle and that benefits from the strengths of each individual project.

6.3 Expanding the fm-verify Plugin

More Verifiers
The architecture of the fm-verify plugin would allow for multiple verifiers. As the SV-COMP demonstrates, different verifiers have different strengths and weaknesses [5]. Implementing them in a single plugin and allowing the user to choose which verifier to use, is undoubtedly an improvement.
However a significant amount of development time for this plugin was spent on the JBMC-specific runner and parser. We would expect a similar, perhaps slightly less, effort when adding other verifiers.

File-based Configuration
We haven't encountered the need for this, but we assume it might be desirable for the fm-verify plugin to be able to read its configuration from a file. When having a long list of ignored properties, such as methods where the failed property is accepted or handled via a contract, writing the list into the pom.xml file may make it unnecessarily cluttered. This feature would also allow for quick configuration changes, such as having one configuration that quickly verifies some set methods during development, but then having another configuration that exhaustively verifies the entire project after a merge.

Automatic Verification of Sub-projects
The inconvenience we encountered in Sect. 5.4.1 could be mitigated by a feature that automatically detects sub-projects. However, this is probably more useful in the specific case of crawling through many different public repositories, than as a developer on a single project.

Individual Verification for Different Methods
At this point in time, all settings, such as the runtime limit or arguments to JBMC, are used equally for each verification of a method. It is not currently possible to verify one set of methods with one set of arguments, then another set of methods with different arguments. This would allow to fine-tune the configuration, so that different methods can be verified with a different bound.

7 Conclusion

The fm-verify plugin for Maven significantly enhances the formal verification process for Java applications. By leveraging JBMC, this plugin offers an automated solution that simplifies the detection of potential safety violations within Java codebases. The evaluation across 100 open-source projects demonstrated the plugin's ability to integrate seamlessly with Maven and provide valuable feedback on unsafe code without introducing substantial performance overheads. Ultimately, the fm-verify plugin represents a crucial step towards making formal verification more accessible and practical for mainstream Java development.

Bibliography

[1] J. Alglave, A. F. Donaldson, D. Kroening, and M. Tautschnig. Making software verification tools really work. In *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, pages 28–42. Springer, 2011.

[2] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina. Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *J. Autom. Reason.*, 45(4):397–414, 2010.

[3] AWS labs. One-line-scan, 2022. `https://github.com/awslabs/one-line-scan` Last Accessed: 2024-04-14.

[4] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at IBM. *Formal Methods Syst. Des.*, 22(2):101–108, 2003.

[5] D. Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In B. Finkbeiner and L. Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, volume 14572 of *Lecture Notes in Computer Science*, pages 299–329. Springer, 2024.

[6] D. Beyer, G. Dresler, and P. Wendler. Software verification in the google app-engine cloud. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 327–333. Springer, 2014.

[7] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. *CoRR*, abs/0902.0019, 2009.

[8] P. Bjesse. What is formal verification? *SIGDA Newsl.*, 35(24):1–es, dec 2005.

[9] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels. Model checking flight control systems: The airbus experience. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 18–27. IEEE, 2009.

[10] J. C. Cherniavsky. Review of "automated theorem proving: a logical basis" by d. w. loveland. north-holland publishing co. 1977. *SIGACT News*, 11(1):18, 1979.

[11] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Code-level model checking in the software development workflow. In G. Rothermel and D. Bae, editors, *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, pages 11–20. ACM, 2020.

[12] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[13] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[14] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[15] L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík. JBMC: A bounded model checking tool for verifying java bytecode. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 183–190. Springer, 2018.

[16] P. Filipovikj, N. Mahmud, R. Marinescu, C. Seceleanu, O. Ljungkrantz, and H. Lönn. Simulink to UPPAAL statistical model checker: Analyzing automotive industrial systems. In J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 748–756, 2016.

[17] F. A. Fontana, C. Raibulet, I. Rigo, and L. Ubezio. An eclipse plug-in for the java pathfinder runtime verification system. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*, pages 142–152. IEEE Computer Society, 2006.

[18] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419. ACM, 2011.

[19] C. A. González and J. Cabot. Formal verification of static software models in MDE: A systematic review. *Inf. Softw. Technol.*, 56(8):821–838, 2014.

[20] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *Int. J. Softw. Tools Technol. Transf.*, 2(4):366–381, 2000.

[21] S. Hussein, Q. Yan, S. McCamant, V. Sharma, and M. W. Whalen. Java ranger: Supporting string and array operations in java ranger (competition contribution). In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 553–558. Springer, 2023.

[22] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. Jayhorn: A framework for verifying java programs. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 352–358. Springer, 2016.

[23] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[24] X. Li, Y. Liang, H. Qian, Y. Hu, L. Bu, Y. Yu, X. Chen, and X. Li. Symbolic execution of complex program driven by machine learning based constraint solving. In D. Lo, S. Apel, and S. Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 554–559. ACM, 2016.

[25] N. Loose, F. Mächtle, F. Sieck, and T. Eisenbarth. SWAT: modular dynamic symbolic execution for java applications using dynamic instrumentation (competition contribution). In B. Finkbeiner and L. Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, volume 14572 of *Lecture Notes in Computer Science*, pages 399–405. Springer, 2024.

[26] G. Lüttgen and V. Carreño. Analyzing mode confusion via model checking. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings*, volume 1680 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 1999.

[27] M. Mues and F. Howar. Jdart: Dynamic symbolic execution for java bytecode (competition contribution). In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 398–402. Springer, 2020.

[28] M. Mues and F. Howar. Gdart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution). In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 435–439. Springer, 2022.

[29] J. Nielsen. *Usability engineering*. Academic Press, 1993.

[30] P. W. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In A. Dawar and E. Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 13–25. ACM, 2018.

[31] C. S. Pasareanu. Using symbolic (java) pathfinder at nasa. *Nasa*, 2010.

[32] C. S. Pasareanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *ASE*, pages 179–180. ACM, 2010.

[33] S. Ray, N. Ghosh, R. J. Masti, A. K. Kanuparthi, and J. M. Fung. Formal verification of security critical hardware-firmware interactions in commercial socs. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 43. ACM, 2019.

[34] B. Schlich, F. Salewski, and S. Kowalewski. Applying model checking to an automotive microcontroller application. In *IEEE Second International Symposium on Industrial Embedded Systems, SIES 2007, Hotel Costa da Caparica, Lisbon, Portugal, July 4-6, 2007*, pages 209–216. IEEE, 2007.

[35] M. N. Seghir and D. Kroening. A visual studio plug-in for cprover. In *2013 3rd International Workshop on Developing Tools as Plug-Ins (TOPI)*, pages 43–48, May 2013.

[36] SpotBugs Team. SpotBugs. `https://spotbugs.github.io/` Last Accessed: 2024-08-23.

[37] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 48–62. IEEE Computer Society, 2013.

[38] A. Yorihiro, P. Jiang, V. Marqués, B. Carleton, and O. Legunsen. emop: A maven plugin for evolution-aware runtime verification. In *RV*, volume 14245 of *Lecture Notes in Computer Science*, pages 363–375. Springer, 2023.

[39] F. Yulianto, M. Wibowo, A. Yananto, D. H. F. Perdana, E. A. Wiguna, Y. Prabowo, N. Rahili, A. Nurwijayanti, M. Y. Iswari, E. Ratnasari, A. Rusdiutomo, S. Nugroho, A. S. Purwoko, H. Aziz, and I. Fachrudin. Coastal vulnerability assessment using the machine learning tree-based algorithms modeling in the north coast of java, indonesia. *Earth Sci. Informatics*, 16(4):3981–4008, 2023.

A List of projects

Table A.1: List of all 100 projects used for the evaluation

Name	URL	Stars	Size (kb)
abreuapps-core	https://github.com/NewtonxD/abreuapps-core	4	8169
adif-processor	https://github.com/urbancamo/adif-processor	4	38042
adventure-engine	https://github.com/finley243/adventure-engine	3	2824
algorithms-competitive-programming	https://github.com/FredAlissonx/algorithms-competitive-progra	2	5414
april-zh-hotspot-push	https://github.com/april-projects/april-zh-hotspot-push	9	27945
Axolotl	https://github.com/SoraVWV/Axolotl	2	1785
bing-wallpaper	https://github.com/niumoo/bing-wallpaper	1892	7988
bing-wallpaper	https://github.com/acc8226/bing-wallpaper	3	863
bing-wallpaper	https://github.com/v5tech/bing-wallpaper	66	1877
buji-pac4j	https://github.com/bujiio/buji-pac4j	485	503
camel-language-server	https://github.com/camel-tooling/camel-language-server	46	3068
carta	https://github.com/avereon/carta	2	2450
Charging-Stations-in-Slovenia	https://github.com/zigad/Charging-Stations-in-Slovenia	2	22723
chatgpt-java	https://github.com/PlexPt/chatgpt-java	3538	435

closure-templates	https://github.com/google/closure-templates	637	76009
cloud-sql-jdbc-socket-factory	https://github.com/GoogleCloudPlatform/cloud-sql-jdbc-socket-factory	229	3713
COM2545StubbedAssignments	https://github.com/Yeshiva-University-CS/COM2545StubbedAssignments	4	1825
comment_backups	https://github.com/sudojia/comment_backups	2	344
coreLang	https://github.com/mal-lang/coreLang	10	1044
CoreProtect	https://github.com/PlayPro/CoreProtect	611	948
csr-signing-using-kms	https://github.com/aws-samples/csr-signing-using-kms	2	16
DisplayEntityUtils	https://github.com/PZDonny/DisplayEntityUtils	3	210
dive-into-spring-ai	https://github.com/qifan777/dive-into-spring-ai	35	2265
dog-box	https://github.com/llnancy/dog-box	6	773
dropwizard-correlation-id	https://github.com/dhatim/dropwizard-correlation-id	6	87
DSLearn	https://github.com/ViniciusStabile/DSLearn	3	200
easyAi	https://github.com/lifejwang11/easyAi	52	7412
ErrorWindowAddon	https://github.com/FlowingCode/ErrorWindowAddon	9	348
evm-abi-decoder	https://github.com/osslabz/evm-abi-decoder	54	267
forwardproto	https://github.com/gwttk/forwardproto	2	1759
foundation	https://github.com/paritytrading/foundation	11	160
fujiWeekly	https://github.com/zas023/fujiWeekly	4	4057
FunctionalUtils	https://github.com/VassilisSoum/FunctionalUtils	5	94
gaokao_bot	https://github.com/HerbertGao/gaokao_bot	3	334
github-statistics	https://github.com/tanjeffreyz/github-statistics	7	1038

glassfish-build-maven-plugin	https://github.com/eclipse-ee4j/glassfish-build-maven-plugin	3	512
glassfish-logging-annotation-processor	https://github.com/eclipse-ee4j/glassfish-logging-annotation-processor	2	91
HamburgueRia	https://github.com/pedrozardetti/HamburgueRia	2	24216
HXDD	https://github.com/Lemon-King/HXDD	14	7155
iCRFGenerator	https://github.com/aderidder/iCRFGenerator	9	1776
integrations-api	https://github.com/cryptomator/integrations-api	11	271
iron-verifiable-credentials	https://github.com/filip26/iron-verifiable-credentials	13	1509
itext-pdfsweep-java	https://github.com/itext/itext-pdfsweep-java	37	29949
jamal	https://github.com/verhas/jamal	57	38189
jar-analyzer	https://github.com/jar-analyzer/jar-analyzer	869	81956
javalin-pac4j	https://github.com/pac4j/javalin-pac4j	37	277
jboss-invocation	https://github.com/jbossas/jboss-invocation	14	601
job4j_tracker	https://github.com/ArtemPolshchak/job4j_tracker	2	220
jscep	https://github.com/jscep/jscep	114	9102
just-the-dip	https://github.com/puddingspudding/just-the-dip	10	7385
KettraShop	https://github.com/kettraworld/KettraShop	4	187
laohuangli_bot	https://github.com/HerbertGao/laohuangli_bot	3	234
light-chaser-server	https://github.com/xiaopujun/light-chaser-server	6	117
LolDamageCalculator	https://github.com/PauHPMCBR/LolDamageCalculator	5	7034
MessageSync	https://github.com/MinecraftProgrammingTeam/MessageSync	4	72

minecraft-stress-test	https://github.com/PureGero/minecraft-stress-test	95	59
MtgDesktopCompanion	https://github.com/nicho92/MtgDesktopCompanion	156	476119
multicrew-artillery-plot	https://github.com/star1954/multicrew-artillery-plot	3	6521
naikan-maven-plugin	https://github.com/enofex/naikan-maven-plugin	2	186
native-spring-boot	https://github.com/alina-yur/native-spring-boot	31	359
negotiator	https://github.com/BBMRI-ERIC/negotiator	4	7280
neohabitat	https://github.com/frandallfarmer/neohabitat	228	11484
opeo-maven-plugin	https://github.com/objectionary/opeo-maven-plugin	8	5177
parking-domain	https://github.com/cezarysanecki/parking-domain	16	2625
passay	https://github.com/vt-middleware/passay	272	436587
PathWatcher	https://github.com/Cantara/PathWatcher	2	209
PeerBanHelper	https://github.com/PBH-BTN/PeerBanHelper	841	3041
pipeline-steps-doc-generator	https://github.com/jenkins-infra/pipeline-steps-doc-generator	15	17495
plugin-for-Tentorium	https://github.com/turtlegamerw/plugin-for-Tentorium	2	129
Program-Practice	https://github.com/karishma502/Program-Practice	2	242
ref-GemLibPki	https://github.com/gematik/ref-GemLibPki	14	1258
RIAP-old-code-with-apis	https://github.com/OscarNavarrolol/RIAP-old-code-with-apis	3	5031
script-control-panel	https://github.com/szymciogrosik/script-control-panel	3	1671
sdk-java	https://github.com/openfeed-org/sdk-java	4	1207
shmupCC	https://github.com/AnnaSaysHi/shmupCC	3	9745

Simple_user_-management	https://github.com/farzadafi/Simple_user_management	2	141
spark-pac4j-demo	https://github.com/pac4j/spark-pac4j-demo	36	113
spring-cqs	https://github.com/prisma-capacity/spring-cqs	8	488
spring-security-jee-pac4j-boot-demo	https://github.com/pac4j/spring-security-jee-pac4j-boot-demo	34	163
spring-security-pac4j	https://github.com/pac4j/spring-security-pac4j	268	550
spring-swt-word	https://github.com/Himmreich/spring-swt-word	2	24
spring-webflux-pac4j	https://github.com/pac4j/spring-webflux-pac4j	3	162
spring-webmvc-pac4j-boot-demo	https://github.com/pac4j/spring-webmvc-pac4j-boot-demo	43	233
state-machine-bug-finder	https://github.com/assist-project/state-machine-bug-finder	2	1739
systemds	https://github.com/apache/systemds	1029	374340
TechMaps-Back-v2	https://github.com/yellowisk/TechMaps-Back-v2	2	252
Ter-WebServer	https://github.com/21xiaoye/Ter-WebServer	2	443
ThirstBar	https://github.com/Vanderis-Team/ThirstBar	3	40880
tlcsdm-common	https://github.com/tlcsdm/tlcsdm-common	3	317
todomvc	https://github.com/selenide-examples/todomvc	5	229
TrainingMaterials	https://github.com/devopsjaba/TrainingMaterials	2	54193
undertow-pac4j-demo	https://github.com/pac4j/undertow-pac4j-demo	7	145
vertx-pac4j-demo	https://github.com/pac4j/vertx-pac4j-demo	27	315
visuale	https://github.com/Cantara/visuale	9	17382
VPL-Core	https://github.com/sepp87/VPL-Core	18	34293
wavefront-sdk-java	https://github.com/wavefrontHQ/wavefront-sdk-java	6	801

Whydah-OAuth2Service	https://github.com/Cantara/Whydah-OAuth2Service	2	1998
WhymerCraft	https://github.com/NickGaultney/WhymerCraft	2	130
wvp-GB28181-pro	https://github.com/648540858/wvp-GB28181-pro	4741	45421
zol-collector	https://github.com/zhugezifang/zol-collector	19	15879