Develop and Evaluate a Security Analyzer for Finding Vulnerabilities in Java programs

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2021

By

Songtao Wang(10612858)

Department of Computer Science

content

Abstract

The main goal of this dissertation is to understand the basic principles of software verification, to understand the software verification tools and their extensions that have been implemented, and to develop and evaluate based on the implementation. Java Bounded Model Checking (JBMC) is a bounded model checker that allows Java programs to be verified. As an efficient software verification tool, it aims to accurately verify and analyze the correctness of Java software according to a given specification. In order to verify whether the determined violation is actually a valid error, an extension based on witness verification was designed and simply implemented. However, there are some shortcomings in this extended tool for software security analysis.

In this thesis, we explain in detail the algorithm structure based on JBMC and the witness verification method, and verify the feasibility of the counterexample witness verification in software verification by reproducing the implementation. Our focus is on ways to improve the verification capabilities of the extension, including improving the shortcomings in the previous implementation, expanding the scope of the verification set, and improving the level of tool automation. We still try using witnesses in the format of GraphML, but more other frameworks are used in this project, including PowerMock and JUnit. In the algorithm chapter, we analyze in detail the algorithm structure applied in the previous implementation, and analyze the points that can be improved. We make appropriate modifications and additions to the original algorithm to achieve better results. Finally, we verify the capabilities of our tools through experimental evaluations, and explain the deficiencies that still exist and the areas that can be improved.

Acknowledgments

First of all, I want to express my gratitude to my MSc project supervisor, Dr Lucas Cordeiro. He provided me with a lot of help and guidance throughout the process of completing the thesis, and at the weekly group meetings, I could get some comments about my tasks from him, which were very helpful.

Second, I would like to thank every single person in the group at weekly meetings. From their presentations, I learned some knowledge in related fields, which is really interesting. I especially want to thank Tong Wu, who has a similar project to mine. He helped me solve some confusing problems, and he also gave me very useful suggestions when I had trouble with my project.

Finally, I also want to say thanks to my parents, because they supported me to finish my MSc degree. Besides, I would thank the University of Manchester for offering me the opportunity to complete a master's degree.

Chapter 1 Introduction

Java is a kind of modern programming language which was first designed by James Gosling et al. at Sun Microsystems, Inc. in 1991. Then the name of this programming language was changed into “Java” in 1995 [1]. It has been becoming more and more popular since it developed, and wildly used in many areas, from enterprise computing to Android apps development. In 2014, it was reported in InfoWorld that about 90 percent of Fortune 500 companies were applying Java. Moreover, as the worldwide No.1 mobile platform, Google Android captured 62% flat panel sales in 2013 [2].

The PYPL Popularity of Programming Language Index shows how often various languages are searched in Google, which can reflect their popularity and trend [3]. The programming language is assumed to be more popular if it is searched more often. As illustrated in Table 1, Java is still one of the most popular languages around the world as of August 2021, which is more than 15% of all the programming languages.

Rank	Change	Language	Share	Trend
1		Python	29.93 %	-2.2 %
2		Java	17.78 %	+1.2 %
3		JavaScript	8.79 %	+0.6 %
4		C#	6.73 %	+0.2 %
5	↑	C/C++	6.45 %	+0.7 %

Table 1: PYPL Popularity of Programming Language [3]

There are many reasons why Java is so popular, one of which is platform independence [4]. Programs can run on different types of devices as long as they have a Java Runtime Environment (JRE) installed. Therefore, when you develop a Java

program once, you can run it almost anywhere at any time. This excellent feature of Java is related to its software development process.

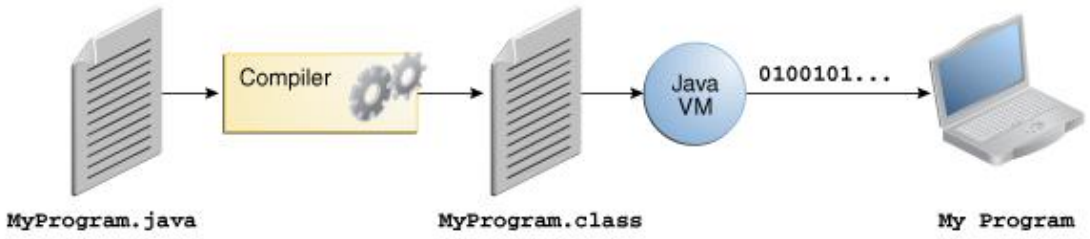

Figure 1.1: An overview of Java software development process [5]

As is shown in Figure 1.1, in a Java program, the source code is written in text files, of which name ends with the .java extension. The .java source file is compiled by the compiler and produce a file ending with .class. A .class file contains bytecodes, which can be executed on the Java Virtual Machine(JVM). Then the program or application can run on the Java launcher tool with the Java Virtual Machine [5].

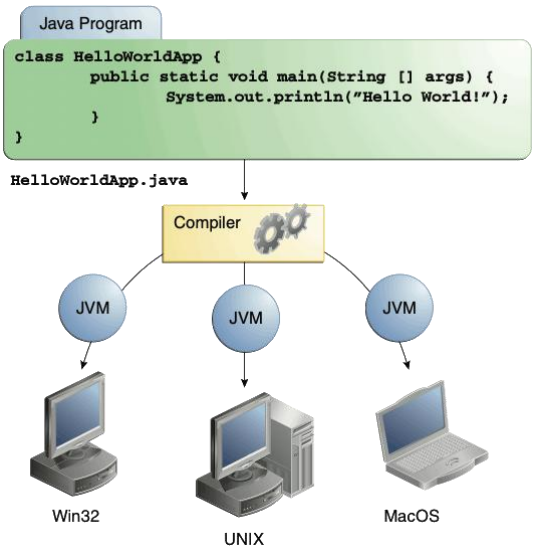

Figure 1.2: Platform independence and JVM [5]

Java is platform-independent because the JVM plays an important role during the compiling and running of the software. As a bridge to connect the javac compiler to

different devices, the JVM can be installed on many kinds of operating systems(OS), and a bytecode file that ends with .class is able to run on the OS with JVM. Therefore, various platforms and operating systems such as Microsoft Windows, Linux and Mac OS can execute the same Java applications without any change [5].

Under normal circumstances the operation of Java software requires the support of class libraries and other resources, and the Java Runtime Environment(JRE) plays such a role. The JRE is a layer to provide the class libraries and some necessary resources for Java programs to run on top of operating systems [6]. The JRE combines Java code written using the JDK with the libraries needed to run it on a JVM, and then generates a JVM instance to run the resultant application.

In the JRE runtime architecture, there are mainly three parts. All classes required to run a Java application are dynamically loaded by the Java ClassLoader. The JRE uses ClassLoaders to automate this operation on demand since Java classes are only loaded into memory when they are needed. Before passing Java code to the interpreter, the bytecode verifier validates its format and correctness. The class will not be loaded if the code violates system integrity or access privileges. The Java interpreter generates an instance of the JVM once the bytecode has loaded successfully, allowing the Java application to run natively on the underlying system [6].

1.1 Problem Description

In the following decades, the security of Web applications is becoming more and more important, because some of them are applied in some of the particular fields, such as sensitive financial and medical data [7]. The Java Development Kit(JDK) is designed with a security architecture, which includes a number of application programming interfaces(APIs) and tools to avoid security vulnerabilities [8]. However, there is still an increasing number of security issues about Java programs.

Java Bounded Model Checking tool(JBMC) is a verification tool for discovering vulnerabilities within Java programs [9]. It is able to verify the Java bytecode based on the CPROVER framework effectively. Nonetheless, several incorrect results, including incorrect TRUE and FALSE, are produced in the Benchmark results of JBMC [10]. Therefore, there is a need to verify the results after executing JBMC, and also, we can make some improvement based on JBMC.

There is an effective verification tool based on JBMC designed by Vi, which contains algorithms about GraphML witness produced by JBMC [11]. It is able to find some existing security vulnerabilities. However, there are still some shortcomings in the design of the algorithm and the choice of the implementation framework, so the coverage and accuracy of the results are not perfect.

In this thesis, the main task we focus on is to learn and try using JBMC [9] as a verification tool to verify Java programs. Furthermore, we can also find potential errors or bugs during using JBMC and the tool designed by Vi, and then we will do some extensions on top of the previous implementation to find valid vulnerabilities in Java programs.

1.2 Objectives

This project is focused on further development based on the Java Bounded Model Checking tool (JBMC). By learning software model checking to find security vulnerabilities, evaluate existing verification strategies and implement suitable extensions.

The specific aims and objectives are to:

- Figure out the principles and concepts to find vulnerabilities in Java bytecode by

JBMC.

- Evaluate strategies available in JBMC in finding security vulnerabilities.
- Reproduce the extension in the previous thesis, and find out existing limitations and drawbacks.
- Implement suitable extensions on top of previous work and evaluating the results.

1.3 Contribution

The contribution of this MSc project can be roughly divided into two parts. The first part of the implementation contains some Python scripts and a .java file that is called a validation harness. These programs are used to extract the counterexample that may trigger the conditions of a bug from JBMC and inject it into the validation harness. Then an appropriate mocking framework on Java will be used to simulate the bug found by JBMC, and in this way, we can simulate the bug and determine whether it is valid. This part is based on the algorithm in the previous implementation, and we need to modify and optimize some of the details to make up for its original shortcomings. The other part of the project is to complete integration and automated testing, which is made up of some Python scripts and .html files. In order to compare the results of the validation tool and JBMC more efficiently, we develop a simple web application and integrate JBMC and my tool so that we can generate a result table easily.

Witness verification based on Java language has not been paid attention to now and is still in the initial stage, while witness verification based on C language has been extended and implemented by many researchers and institutions. Therefore, another contribution of this project is to explore the potential of witness verification methods in Java programs based on JBMC and the previously implemented extensions. Through our extended verification tool, we tried witness verification as a new method in the verification of Java program vulnerabilities. At the same time, our research and implementation also show that witness verification may also have application prospects in other program verification.

1.4 Organization of Dissertation

The whole dissertation is mainly divided into 6 chapters. It starts with Chapter 1, a general introduction of the total project, which contains an overview of the subject, problem description, objectives, and my contributions. In Chapter 2, we discuss the background knowledge of relevant research fields. The next chapter is Chapter3, which is about the proposed methodology of the project. We discuss and analyze the design and implementation process of the algorithm in detail, and show part of the code for explanation. That is followed by Chapter 4, the experimental evaluation chapter, which is used to display the results of the tests, and the advantages and disadvantages of the implementation and possible reasons are also discussed in this chapter. Then, in Chapter 5, we list some of the related works in this area. The final chapter, Chapter 6, is the concluding chapter, in which we summarize the results and outcomes of the dissertation, think and discuss them, and put forward some ideas and suggestions for future work.

Chapter 2 Background

The main topic of this chapter is to provide enough and necessary background into this thesis, which includes knowledge and concepts related to my project. The main content comes from existing papers and websites, and it is able to introduce the development of relative fields and explain clearly the meaning of some proper nouns, so as to make the content afterwards easier to understand. There are 6 sections in this chapter. Section 2.1 describes the concept of security vulnerabilities and explains the importance and necessity of resolving them in software development. Section 2.2 introduces the concept of software verification and validation, and also introduces common software testing methods, including software testing techniques at various levels. Section 2.3 explains the detailed concept of software model checking and explains its execution process. After that, it further explained the architecture and execution process of JBMC, as well as its specific usage in the software verification process. In section 2.4, a software verification competition(SV-COMP) is introduced. It briefly introduces the content of the competition, and explains the performance of JBMC in recent years. The fifth section of the background chapter is about the previous extensions on JBMC, which is similar to my project. We explain its implementation algorithm and implementation process, and analyze its shortcomings and improvements. In the last section, we introduce some of the Java verification tools including Java Ranger, JPF and JDart.

2.1 Security Vulnerabilities

With the development of internet technology and software engineering, our life is getting more and more convenient because the digital lifestyle is gradually replacing the traditional way. Meanwhile, it means that people are pretty dependent on the internet, and a large number of personal data such as credit cards and phone numbers is under the control of the internet. In addition, a large number of enterprises are also

generally using digital management of their data and information. Therefore, there is a growing concern about information security and software vulnerabilities. A survey conducted by the FBI and Computer Security Institute(CSI) shows that in 2002, at least one vulnerability occurred in more than 50% of databases, and every single vulnerability could cause nearly 4 million dollars in losses [7]. There is another report from IdentityForce showing that over 7.9 billion data records were leaked by accident because of security attacks from January to September in 2019 [12]. Generally, apart from a few factors caused by management, the most notable reason is vulnerabilities in software.

A security vulnerability is defined by ISO 27005 as “a weakness of an asset or group of assets”, which can be exploited by threats [13]. Another definition from the National Institute of Standards and Technology(NIST) says that a vulnerability is a weak point that could be exploited or triggered, which may exist in an information system or software application [14]. A security vulnerability could cause damage to the stakeholders including software users and owners while applying the application [15]. The main factors of the susceptibility could be caused by the technology or users’ behaviors, which means an unexpected bug or a weak password may bring problems [14]. Regardless of the cause of the vulnerability, such a potentially catastrophic factor should be concerned by developers.

Nowadays, a lot of organizations and institutions summarize and list common software vulnerabilities and risks, and publish the list on the website for developers to improve code and program security. The Open Web Application Security Project (OWASP) is one such online community. It produces articles, documentation, tools, and so on in the field of web application security. The "Top Ten" is a standard awareness document of OWASP, which was first published in 2003 and is regularly updated. This document aims to raise awareness by identifying some of the most critical risks and represent a broad consensus about application security so that developers can produce secure code [16]. Top Ten lists ten web application security

risks including injection, XML external entities(XEE) and cross-site scripting(XSS) [17]. The Common Weakness Enumeration(CWE) is a list developed by the community, which is about weakness types including software and hardware. Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses(CWE Top 25) also reports a yearly list of software weaknesses that frequently occur in the projects [18]. In addition to the common risks, code-level software vulnerabilities are also mentioned in this list. For example, out-of-bounds write and read can be found as the top 5 in the ranking. And improper restriction of operations within the bounds of a memory buffer is also mentioned in CWE Top 25. Besides, The Common Vulnerabilities and Exposures (CVE) system, which dates back to 1999, serves as a reference for publicly known information-security vulnerabilities and exposures. Software vulnerabilities of various levels and types and their frequency are displayed and analyzed on the website as charts and graphs by year [19].

As a type-safe programming language, the Java language is designed with some security features, including automatic memory management and garbage collection [8]. These features could reduce the possibility of errors to some extend and the robustness of the program is ensured. Moreover, Java programs are compiled to bytecodes by the compiler, and bytecodes will be checked by a verifier before execution [8]. Java security application programming interfaces(APIs) are applied to provide Java software developers tools to enhance the level of security. Besides, some third-party security frameworks such as Spring Security have been designed and built to control authentication, which also helps to avoid vulnerabilities [20]. However, in the database of CVE, there is a total of 673 vulnerabilities on JRE and 39 of them were reported last year(2020). Therefore, vulnerabilities in Java programs still occur from time to time under the design of various security features of the Java language.

2.2 Software Verification and Validation

Software verification and validation is a critical part in the process of developing software, it allows a group of software developers and testers to guarantee that the product is developed correctly throughout the development life cycle [21]. Software verification and validation operations compare the software to its specifications and requirements to ensure that it performs as expected. The software in a project must be validated and verified by the following aspects [22]:

- ensuring that each piece of software fulfills the specified requirements;
- checking each part of the software before using it as an input to another activity;
- ensuring that, as far as feasible, tests on each software item are performed by someone other than the author;
- guaranteeing that the verification and validation effort is sufficient to demonstrate that each software item is fit for operational usage.

The organization of software verification and validation activities, the establishment of software verification and validation roles, and the assignment of people to those roles are all responsibilities of project management [22].

Regardless of the size of the software, software verification is essential in the software development cycle, because it greatly affects the quality and efficiency of the software. During development, 20 to 50 mistakes per 1000 lines of code are common, while 1.5 to 4 errors per 1000 lines of code persist after system testing [23]. Each of these mistakes has the potential to result in an operational failure or non-compliance with a requirement. The goal of software verification and validation is to keep software mistakes at a minimum. Depending on the software's containment and complexity, the effort required can range from 30 percent to 90 percent of the entire project resources [24].

Verification, in a wide sense, is the same as software testing. There are two basic ways to verification in this case: dynamic verification and static verification.

2.2.1 Dynamic Verification

Dynamic verification, often known as experimentation or just testing, is a process that checks the behavior of software as it is being executed; it is commonly referred to as the Test phase. The goal of software dynamic verification is to identify mistakes caused by a single activity or by the repeated execution of one or more activities. We can divide tests into four groups based on their scope: unit testing, integration testing, system testing, and acceptance testing [25].

Unit testing is the most basic level of testing. It tests the smallest testable bit of software, the basic unit of software, and is referred to as a "unit," "module," or "component" interchangeably [25]. Unit testing is commonly referred to as a white box test. That is, it is oriented toward examining and assessing code in its current state, rather than assessing compliance with a set of rules [26].

Integration testing is performed when two or more tested units are merged into a bigger structure. If the quality feature of the larger structure cannot be judged from its components, the test is frequently performed on both the interfaces between the components and the larger structure being produced [25]. Integration testing is a method of building a program's structure while also testing for interfacing issues. The goal is to use unit tested components to create a program structure that has been dictated by design. Integration testing can be done in two ways: top down or bottom up [26].

System testing is used to ensure that the complete system is of high quality from beginning to end. System testing is usually focused on the system's functional/requirement specification. Non-functional quality factors like dependability, security, and maintainability are also scrutinized [25]. Software or hardware system testing is testing done on a complete, integrated system to see if it complies with the system's requirements. System testing is considered black box testing, and as such, it

should not necessitate any understanding of the code's or logic's inner workings. The basic goal of system testing is to completely exercise the computer-based system. Despite the fact that each test has a different goal, they all aim to ensure that system pieces have been correctly integrated and are performing their assigned roles [26].

When the completed system is turned over from the developers to the customers or users, acceptance testing is performed. Its goal is to instill trust in the system's functionality rather than to uncover flaws [25]. User acceptance testing is a sort of testing used to determine whether a product has been designed in accordance with industry standards and criteria, and if it meets all of the customer's expectations [27]. When a product is developed externally by another company, this form of testing is usually done by a user/customer. Acceptance testing is a type of black box testing in which the user is less concerned with the system's internal workings/coding and instead evaluates the system's general functionality and compares it to the requirements they have given. Before the system is fully delivered or given over to the end user, user acceptability testing is considered to be one of the most significant tests performed by users. Validation testing, final testing, QA testing, factory acceptance testing, and application testing are all terms used to describe acceptance testing. Acceptance testing can be done at two levels in software engineering: one at the system provider level and another at the end user level [26].

2.2.2 Static Verification

Static verification is the process of analyzing the code before it runs to ensure that it fits the requirements. Static verification covers a wide range of topics, including code conventions, bad practices (anti-pattern) detection, software metrics calculation, formal verification, and so on. Verification via investigation, mathematical calculations, logical evaluation, and computations using traditional textbook methods or widely established general-purpose computer methods are all covered by the analysis verification method. To establish conformance with requirements, analysis

comprises sampling and connecting measured data and observed test results with calculated expected values [28].

Static Analysis is concerned with a variety of methods for determining or estimating software quality without relying on actual executions [27]. Static verification is the examination of computer code without the need to run it to check that standard coding practices have been followed. Some versions of the source code are subjected to analysis, which allows programmers to debug new code and uncover potential flaws in compiled code [29].

Static verification is the examination of computer code without the need to run it to check that standard coding practices have been followed. Some versions of the source code are subjected to analysis, which allows programmers to debug new code and uncover potential flaws in compiled code. Data flow analysis, model checking, abstraction interpretation, and assertion usage are some examples of static verification implementation methodologies [29].

2.3 Java Bounded Model Checking

Because of the increasing threat of software vulnerabilities, more powerful and formal software verification are designed. Formal verification tools can guarantee the absence of specified defects in a design [30]. In this chapter, model checking and bounded model checking will be introduced as two formal verification tools. We explore their concepts and principles and introduce the knowledge about Java bounded model checking(JBMC), which is a new formal verification tool for Java programs based on them.

2.3.1 Model Checking and Bounded Model Checking

The concept of model checking was first declared by Clarke and Emerson in 1981. In the theory, by checking exhaustively within all reachable states, the correctness property can be certain. A counterexample will be returned if the property does not hold, and it is related to a violated state [31].

Model checking is an algorithmic method for verifying whether a system model meets a set of correctness specifications [31]. A program model is made up of states, transitions and a specification or property which is a logical formula. The program counter, all program variable values, and stack and heap settings are all evaluated in a state. Transitions explain the process through which a program progresses from one state to the next. Model checking techniques check all of the possible states in a program. If the state space is finite, this method is guaranteed to end. If a state is discovered that violates a correctness property, a counterexample (an execution trace indicating the error) is generated. Partially specified qualities, such as safety or liveness, are checked using model checking techniques. Safety attributes describe the inaccessibility of bad states, such as those in which an assertion violation, null pointer dereference, or buffer overflow has occurred, or API usage contracts, such as the order of function calls, have been broken. Liveness properties represent that something good finally happens, such as the condition that requests must be served eventually or that a program must eventually terminate [30].

Bounded model checking(BMC) builds on the concept of model checking. This method searches the state space in a depth-bounded exhaustive manner. In the semiconductor business, bounded model checking is one of the most widely used formal verification tools. Propositional SAT solvers' outstanding capacity is responsible for the technique's success. Biere et al. introduced the concept of BMC in 1999 [32]. It's named bounded because it only looks at states that can be reached in a finite number of steps, such as k. The design under verification is unwound k times and linked with a property to generate a propositional formula, which is then submitted to an SAT solver. If and only if there is a trace of length k that refutes the

property, the formula is satisfied. Many bugs have been identified by BMC that would otherwise have gone unnoticed [30].

2.3.2 Java Bounded Model Checking(JBMC)

Java Bounded Model Checking (JBMC) is a bounded model checker that allows Java programs to be verified [9]. It is based on Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT). JBMC inherits its sibling tool CBMC's memory model, symbolic execution engine, and SAT/SMT backends, and it includes a frontend for parsing Java bytecode as well as a Java operational model (JOM), which is an accurate but verification-friendly model of the standard Java libraries. As a result, JBMC supports Java bytecode and can use CBMC's verification engine to test programs that use classes, inheritance, polymorphism, arrays, bit-level operations, and floating-point arithmetic. Array bound violations, unintentional arithmetic overflows, and other types of functional and runtime errors are all handled by JBMC [10].

BMC is at the core of JBMC, where a program is unrolled and checked to see if a specific state of the program can be reached in a certain number of steps, generally referred to as the upper bound and denoted by k. We want to determine if a bad condition, which indicates a vulnerability, may be realized during runtime within the known upper constraint k, in JBMC [9].

As a kind of open-source verifier, it is desired to be used to find out inconspicuous bugs in Java programs by verifying Java bytecode [9]. JBMC tool can be executed easily by the command line, and the input can be a JAR archive file or a Java bytecode class file. The command below can be used to run JBMC from the command line interface [33]. If a bug is found, it will show “VERIFICATION FAILED” in the output, or the output will be “VERIFICATION SUCCESSFUL” to indicate that there are not any bugs in the program.

<some-directory>$ <path-to-jbmc>/jbmc <filename> <additional properties (optional)>

Only class files with the.class file extension and Java archive (JAR) files with the.jar file extension are accepted by JBMC. It may also allow for the additional properties to be further specified. The following table shows part of the common optional properties [33]:

Option	Description
--property id	only check one specific property
--stop-on-fai	stop analysis once a failed property is detected
--trace	give a counterexample trace for failed properties
--show-parse-tree	show parse tree
--show-goto-functions	show loaded goto program
--classpath dir/jar	set the classpath to load additional jar files or class files
--unwind nr	unwind nr times
--graphml-witness filename	write the witness in GraphML format to filename

Table 2: Optional properties of JBMC [33]

Here is an example of using some of the properties to execute JBMC:

<some-directory>$ <path-to-jbmc>/jbmc <filename> --unwind 5 --classpath <path-to-jbmc>/core-models.jar:.

The command line is to run in the directory <some-directory> since JBMC uses a similar strategy as JVM when searching for class files. The core-models.jar refers to JBMC's model of the Java runtime library. A specific upper bound k is determined in this command line, and k is determined to be ten here.

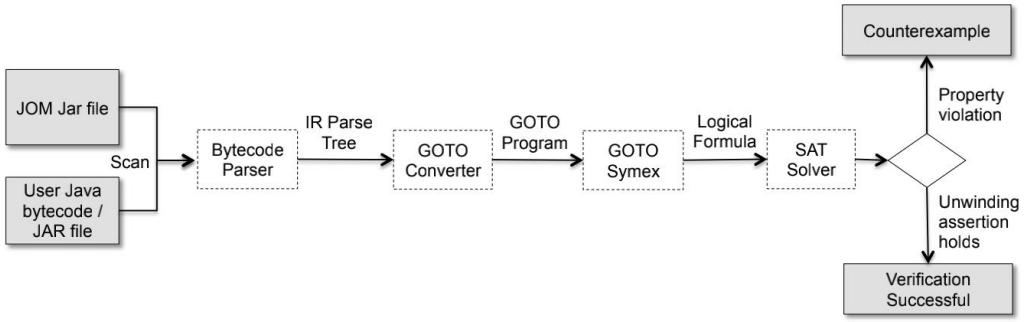

Fig. 2.1: JBMC Architecture [34]

JBMC's architecture is illustrated in Figure 2.1. There are 3 main parts in the total process, the first of which is the input section, the gray rectangles on the left of the picture. The white rectangles in the middle represent the steps of verification, including bytecode parser, GOTO converter etc. On the right-hand side of the illustration, it shows the output of the architecture.

We can set Java bytecode class files or JAR files as input in JBMC. And also JOM is needed, which parses the Java bytecode and converts it to the CPROVER control-flow graph representation, also known as a GOTO program. This transformation simplifies the Java bytecode representation while also lowering exceptional control flow. After the transformation, the program is ready for verification, and it is passed to the GOTO Symex, which is in charge of unwinding loops and unfolding recursive function calls in accordance with the upper bound, k. The GOTO Symex component manages dynamic memory allocation, encoding of virtual method dispatch, unrolling of loops, and unfolding of recursive method calls by performing a symbolic execution of the program. The result of that operation is then sent to a SAT or SMT solver, which will determine whether any bugs have been discovered. As a result, it provides an output at the end of the verification lifetime [9].

2.4 Software Verification Competition

The software-verification tool competition (SV-COMP), which takes place at TACAS, is a driving force for the development of innovative methods, technologies, and tools in the field of software verification. It solves the problem of there being no widely distributed benchmark suite of verification tasks and most concepts being proven only in research prototypes. For comparing software verifiers, a set of verification tasks has been constructed, and the tools are available on the SV-COMP website. The competition is valuable for comparing numerous new and powerful software-verification tools, presenting the most recent application of research results in our community, and rewarding academics and students who spend significant time building verification algorithms and software packages [35].

The goal of the competition is to [35]:

- Provide the community with a snapshot of the state-of-the-art in software verification. That is, different verification tools are compared in terms of precision and performance, regardless of particular paper projects and approaches.
- Increase the visibility and credit given to tool creators. That is, to create a forum for the presentation of tools and debate of the latest technologies, as well as to allow students to publish about their development work.
- Create a community-wide set of software verification benchmarks. This entails creating and maintaining a set of programs with explicit properties to verify, as well as making those programs publicly available for academics to utilize in performance comparisons when evaluating new techniques.

As a valuable verification tool, JBMC performed well in SV-COMP 2019 [36]. However, there are still some weaknesses in JBMC. For instance, regexes string can not be supported by JBMC, and some features such as lambda expressions, reflection can not be proved. Besides, similar to CBMC, JBMC can only deal with bounded programs, which contain a known upper bound [34]. In SV-COMP 2021, JBMC got 603 points out of 693 points, and finished 423 of 473 benchmark tasks, which means

it is still a powerful verification tool on Java programs [37].

2.5 Completed Research and Extension

Since JBMC was designed and published, there were some research and extensions developed on top of it. Last year, a student of the University of Manchester named Vi Lynn Tan designed and implemented an extension based on JBMC, which used the technique of witness validation. This is an interesting attempt, and I study related concepts and work on it. In this section, we will introduce some key concepts used in the extension, and explain the algorithm applied in the project.

2.5.1 Witness Validation

Model checking is a popular automated verification approach that has a wide range of applications. It's a particularly effective bug-hunting strategy: if a property is violated, a counterexample is presented that witnesses the violation. This is why they are frequently referred to as witnesses. Witness validation is the process of verifying that a witness generated by a software model checker is indeed a witness demonstrating that the concrete program violates the property. Producers are software model checkers such as CBMC, JBMC, and others that generate witnesses, whereas validators are software tools that do witness validation [38].

Witness validation is a new idea in software verification and model checking that was created to handle the problem of software verifiers occasionally producing false alarms [39]. In 2015, the first two validators were submitted to SV-COMP, with a total of six validators submitted by 2020 [40]. In SV-COMP 2021, six independently developed witness-based result validators and one witness linter were used to justify the result [37]. The goals of witness validation are to increase the reliability of software verifiers' verification results and to use an adaptable format to express witness data [39].

A violation witness or a correctness witness file can be found in a witness file. Along with the counterexample, a violation witness comprises witness data that describes error paths or violations found by the program verifier. A correctness witness, on the other hand, comprises witness data that explains correctness proof in which the software verifier has discovered no potential event in which a violation might occur. With a common witness format in place, verification results may be independently evaluated, allowing for the use of several verification techniques [39].

GraphML is one of the proposed formats for usage as an exchangeable format for witness validation. GraphML is an XML-based format that was created to represent and preserve graph structures [41]. The relative simplicity of reading from and writing to GraphML files was one of the key reasons for its selection as the preferred exchangeable file format. GraphML is expandable by nature, allowing for the definition and storage of customer data. This allows for the representation and storage of specific witness information such as error pathways. As a result, witness validation can be done in a much more direct manner if verifiers adopt a changeable format like GraphML [39].

2.5.2 Implemented Extensions

An extension related to witness validation on top of JBMC was implemented. The aim of witness validation is to improve the accuracy of the results, which is produced by software verification [39]. GraphML is one of the appropriate data formats for witness validation, and it can be easily read or written.

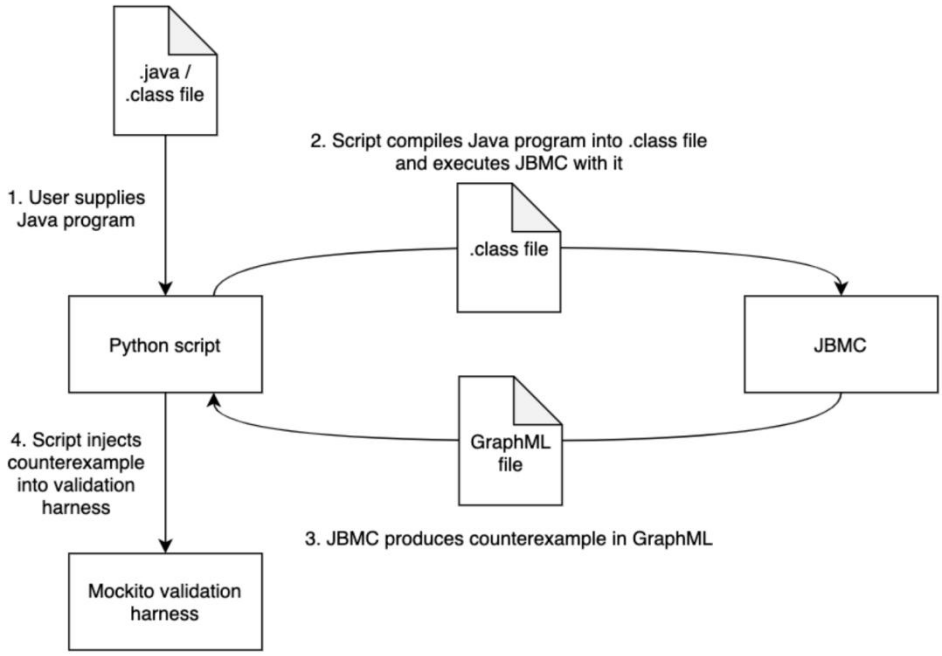

Fig. 2.2: Extension Architecture [11]

Figure 2.2 illustrates the JBMC extension tool architecture, which contains four main sections. In the implementation, the process is controlled through a python file. The target Java file is compiled to bytecode class file first, and then run the JBMC tool with the class file. In the next step, JBMC produces a GraphML file with the result of execution. If there is any bug found in the program, a counterexample will be found in GraphML. Then the process is to parse the GraphML, extract the counterexample and corresponding value type, and inject them into the validation harness. The final step is to execute the harness program, verify the counterexample by Mockito framework, and then the result can be shown in the terminal [11].

The extension runs well, because there are not any different outputs between JBMC and the script python program, which indicates that they both produced correct outcomes of the test cases in benchmarks [11]. However, there are still several obvious limitations and drawbacks in the project. First, she had to modify the verifier code and each program code in the benchmark to meet the use of Mockito, and for this reason, it is not available to run the benchmark in one go. Second, only the programs which have one verify type can be proved by the method, multiple verifiers

and no verifier type programs can not be solved. Third, problems exist in the string type verification, because some counterexamples of string type are unexpected. Besides, there should be a few incorrect results for JBMC, but in Vi's implementation, none of them was found.

2.6 Other Java verification tools

In addition to JBMC, there are many other Java verification tools, among which the more famous ones are Java Ranger, Java Pathfinder (JPF), Jdart, etc. These verification tools have a good performance in the annual SV-COMP. This section introduces the background of various Java verification tools, and briefly discusses their working principles and performance in software verification competitions.

2.6.1 Java Ranger

Java Ranger is an extension for the popular Symbolic Pathfinder program, and the veritesting technique for symbolic execution of Java bytecode is extended in it. When compared to SPF, Java Ranger reduces the running time and number of execution routes by a total of 38% and 71% in a set of nine benchmarks [52].

Java Ranger has a setup that is fairly similar to that of SPF. Because Java Ranger is only an extension of SPF, the Java Ranger directory can be supplied as a legitimate JPF jpf-symbc extension [53].

Java Ranger competed in a static verification competition at a top theoretical conference, where state-of-the-art Java verifiers were among the competitors. JR took first place in the Java verification track of the competition, scoring 630 out of 693 points and completing 427 out of 473 tasks [37].

2.6.2 JPF

The NASA Ames Research Center created Java PathFinder (JPF), a model checker for Java programs that won NASA's TGIR Award for Engineering Innovation in 2003.

JPF is made up of a proprietary Java Virtual Machine (JVM) that interprets bytecode and a search interface that allows you to investigate the whole behavior of a Java program. JPF is written in Java, and its architecture is designed to allow for quick prototyping of new features. JPF is an explicit-state model checker, in that it enumerates all visited states and so suffers from the state-explosion problem that plagues large-scale program analysis. It's best for analyzing programs with less than 10kLOC, but it's also been used to discover faults in concurrent applications with up to 100kLOC [54].

2.6.3 JDart

Since 2010, CMU and NASA Ames Research Center have been working on JDart, a dynamic symbolic analysis framework for Java. The primary purpose of JDart development was to provide a dynamic symbolic analysis tool that could be used on industrial-scale software, such as sophisticated NASA systems.

The Executor and the Explorer are the two most important parts of JDart. The Executor runs the analyzed program and keeps track of data values' symbolic constraints. It is currently implemented as a Java PathFinder framework plugin. The Explorer chooses the exploring strategy that will be used. It makes use of the JConstraints constraints library as an abstraction layer for efficiently encoding symbolic path constraints and provides an interface for various constraint solvers [55].

In SV-COMP 2021, the score of JDart is 623 out of 693, which is second only to Java Ranger. It completed the most task solving on the Java track, 437 out of 473 tasks,

which also reflects its excellent ability [37].

Chapter 3 Proposed Methodology

In this chapter, we introduce the proposed methodology of the implemented extension based on JBMC and the verification tool in the previous thesis. We proposed an improved algorithm compared to the previous one, and re-adjusted the appropriate framework selection during the implementation process. In the first section, we describe the overall structure of the project. We introduce the role of each part of the system and the entire process of the project from the beginning to the result generation by means of flowcharts and words. In section 3.2, we introduce the framework and technology used in the project, and explain their features and role in the project in detail. Then the main part of this chapter is section 3.3, in which we explain the implementation details in the project, especially the implemented algorithms. In this section, we use a lot of code to explain, and at the same time, we also evaluated the complexity of the algorithm. In the last section, some illustrative examples are provided to show the result of our implementation.

3.1 Architecture

Figure 3.1 briefly shows the overall architecture design and execution process of the project. The execution of the application starts from the web front-end interface. The user loads the file and selects the execution option on the interface, then some python scripts control and execute the process, and finally display the input results to the user on the web page in the form of a table. Below we describe the structure of the project in detail, explaining the role of each part of the project and the exact process of its execution.

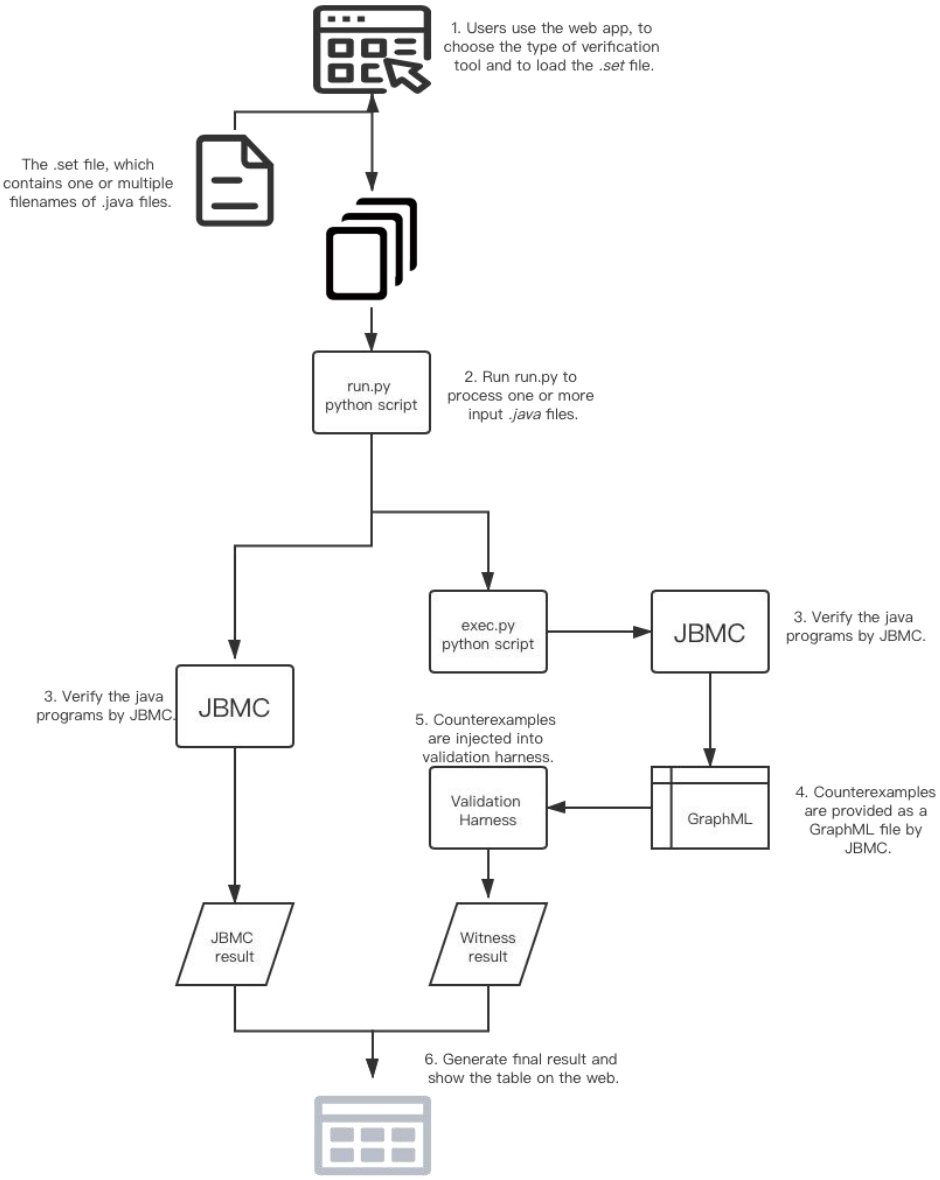

Fig. 3.1: The architecture of the project

The first part is the entrance to the application, which is a simple interface located on the web. The user is required to provide a file with a .set extension as input. This input file is the test set that needs to be verified, and its content contains one or more paths and names of benchmark files. These benchmarks can be obtained from the SV-COMP official website, and they usually end in .java. Then the user can choose the verification option, JBMC, my extended Witness tool, or both of them. After the above is completed, click the execute button to start running the project.

The overall running logic of the project is controlled by the python script run.py. First, the files in the .set file are processed by a loop logic, so that the benchmarks are verified one by one. Secondly, the selected verification mode is processed by the branch statement to determine whether to execute the corresponding code. Because the object of bounded model checking by JBMC is bytecode file, that is, the file with the extension of .class, we need to compile the input .java file with the javac compiler. It is also allowed to directly input the .class type file, but this requires the user to provide the original data type of the specific Verifier method at the same time so that it can be called in the subsequent verification process. Now, make sure that the file type entered in the next step is bytecode, which meets the needs of JBMC.

The method of running JBMC is very simple. The command is called directly through the command line interface, and the parameters are the bytecode file name obtained in the previous step and the original data type of the variable applied by the bounded model checking. There are three results from executing the command line: if the result is "VERIFICATION FAILED", it means that JBMC has verified that there are vulnerabilities in the program; if the result is "VERIFICATION SUCCESSFUL", it means that no errors in the program were detected by JBMC; the last kind of result is the execution timeout.

In my extension tool on Witness, the main process is controlled by the python script named exec.py. The first step executed in the script is similar to the one mentioned earlier, using the command line to execute the JBMC validator. The difference here is that the optional property parameter --graphml-witness provided by JBMC is used here, and the result generated by the validator is stored in GraphML, which is an XML-based file format used to represent graphs. As one of the features of JBMC, we can judge whether there is an assertion violation in the program through the results generated by GraphML, and provide a counterexample for each violation when there is a violation.

If no violations are found, it means that no vulnerabilities have been detected by the JBMC verification tool; if violations are found, one or more counterexamples in GraphML will be injected into the designed verification tool. This verification tool is implemented by the Java language, using a suitable mock framework (such as Mockito, etc.) to mock the original program, and then substituting the counterexample for execution. At the same time, we need to rely on other testing tools to identify whether the counterexample is an effective cause wrong value, so as to determine whether the program has vulnerabilities. In this project, we mainly verify the counterexamples found in violations, so the main research object is the output GraphML file obtained after "verification failed".

The last step is to integrate the results from the JBMC verification with the results obtained by the Witness verification tool, and display the final results in the form of a table on the web page, and display the three results of "success", "failure" and "timeout" using different colours to distinguish them for comparison.

In Section 3.3, we will explain the implementation details more specifically through the code in the program.

3.2 Techniques

In this section, we introduce some of the relatively new technologies applied in the project and analyze the necessity of choosing them. The techniques includes mock frameworks Mockito and PowerMock, Java program testing framework JUnit5 and web page front-end common language HTML.

3.2.1 Mockito and PowerMock

Mock objects are very useful and necessary in the project, especially for the injection

and execution of the validation harness. In the validation harness program, we use mock objects to simulate the behavior of the Verifier class in a controlled way, and to provide an exact value when one of the methods of the class is called. In this way, we can run the test program to determine whether the counterexample is valid.

Mockito is one of the common libraries dominating Java mocking. Comparing with other frameworks it offers a simpler and intuitive approach, that is, we can verify what we want. Furthermore, the APIs of Mockito are slim, which means it is easy to use [43]. In Vi's dissertation, Mockito is one of the core technologies used in the programs, and it plays a vital role [11].

In this project, using Mockito is not a perfect choice, because some of the shortcomings of the original Mockito framework have been exposed in the previously implemented extension. In the code of Vi's program, we need to modify all the static methods in the Verifier class by removing the static modifier. And in the original benchmark code provided by SV-COMP, we have to modify all the code that calls the static methods in the Verifier class and change it to an instantiate Verifier object and call its corresponding methods [11]. The root cause of this problem is that Mockito does not support mocks for static methods. We have found an alternative mock framework applied to solve this problem: PowerMock [42].

PowerMockito is a PowerMock's extension API to support Mockito. It enables users to work with the Java Reflection API in a simple way, overcoming Mockito's limitations, such as the inability to mock final, static, or private methods [44]. Mocking of static methods, final classes and methods, private methods and more is possible with PowerMock due to a custom classloader and bytecode manipulation. Because the entire expectation API is the same for static methods and constructors, PowerMock is simple to use [42]. Therefore, PowerMock is a better choice for mocking in this project.

3.2.2 JUnit5

JUnit is a unit testing framework for the Java programming language. JUnit 5 is made up of three sub-projects, each with its own set of modules. The JUnit Platform is a foundation for testing frameworks that run on the JVM. It also defines the TestEngine API, which can be used to create a testing framework for the platform. In JUnit 5, JUnit Jupiter combines the new programming model with the extension concept for writing tests and extensions. A TestEngine is provided by the Jupiter sub-project for performing Jupiter-based tests on the platform. JUnit Vintage comes with a TestEngine that allows us to run JUnit 3 and JUnit 4 tests on the platform [45].

In the implementation of the project, Junit5 is used in conjunction with the mock framework, and through the use of several appropriate annotations, it can achieve the verification function together with the PowerMock framework.

3.2.3 HTML and Flask

HTML, or HyperText Markup Language, is the standard markup language for texts that are intended to be viewed on a web browser. Web browsers receive HTML documents from a web server or locally stored files and convert them to multimedia web pages. HTML originally featured cues for the document's look and described the structure of a web page logically. Images and other objects, such as interactive forms, can be embedded in the produced page using HTML techniques. HTML allows you to create organized documents by indicating structural semantics for text elements like headers, paragraphs, lists, links, quotations, and other elements [46].

A front-end framework is used in the project, which is Flask. Flask is a Python microweb framework that doesn't require any special tools or libraries. It doesn't have a database abstraction layer, form validation, or any other components that rely on third-party libraries to do typical tasks, and provides extensions that can be used to

extend the functionality of an application [47]. Flask is a framework for building lightweight web applications. It's built to make getting started simple and quick, with the flexibility to scale up to more sophisticated projects. It has grown in popularity as one of the most widely used Python web application frameworks [48].

In order to finish a user-friendly extension tool, we use HTML and Flask framework to implement a simple front-end interface, so that we can run the verification of benchmarks just by selecting input file and clicking the button, instead of executing command line one by one. In the same way, we show our result in a table on the web page, which is also implemented by the techniques, and this can make it more intuitive for users to get the information from the final result table.

3.3 Algorithms

We focus on explaining the algorithms designed and implemented in the project in this section. The code part of the project is mainly divided into three sections, the program control module composed of python scripts, the verification module where the witness file is injected into the mock verification program, and the front-end interface implementation module. Each part corresponds to a subsection of this section. Besides, in subsection 3.3.4, we analysis the complexity and some other features of our programs.

3.3.1 Python scripts

There are a total of 3 python scripts in the project, two of which control the execution logic of the program. We describe the two scripts in detail in this subsection, and the other is related to the front-end, which we describe in subsection 3.3.3.

run.py

```
1. with open("result.html", 'w') as r:
2.     r.write("""
3. <html>
4. <table border="1">
5. <tr>
6. <th>NO</th>
7. <th>Test suite</th>
8. <th>Title name</th>
9. <th>Type</th>
10. <th>Correct output</th>""")
```

Listing 3.1: Result table header settings in run.py

The code in Listing 3.1 mainly uses HTML elements to construct the final result table result.html, which contains the attribute name and border width of the header. The header has a number, test suite name, test benchmark name, data type and expected result, etc.

```
1. set_path = sys.argv[1]
2. option = sys.argv[2]
3. if(option == "JBMC" or option == "Both"):
4.     with open("result.html", 'a') as r:
5.         r.write("""<th>JBMC output</th>""")
6. if(option == "Witness" or option == "Both"):
7.     with open("result.html", 'a') as r:
8.         r.write("""<th>Witness output</th>""")
```

Listing 3.2: The execution option judgment logic in run.py

In Listing 3.2, the code shows how to handle the parameters from the front-end, especially the choice of test options. There are two parameters in the input when executing the program, as is shown in lines 1 and 2, the first of which is the path of the .set file and the second is the option. If the option parameter is “JBMC”, we execute the JBMC test only, or we execute the Witness tool if the option is “Witness”, or we execute both of them.

```
1. with open(set_path, 'r') as sets:
```

```
2.  for set_name in sets:
3.      ymls = glob.glob('/'.join(set_path.split('/')[:-1]) + '/' + set_name.split()[0])
4.      ymls.sort()
5.  for yml in ymls:
6.      print(yml)
7.      with open(yml, 'r') as y:
8.          yml_dic = yaml.load(y, Loader = yaml.FullLoader)
9.          if(yml_dic['properties'][0]['expected_verdict'] == True):
10.             correct_result = "True"
11.         else:
12.             correct_result = "False"
13.         path = '/'.join(yml.split('/')[:-1]) + '/' + yml_dic['input_files'][1]
```

Listing 3.3: Code for processing input .set file in run.py

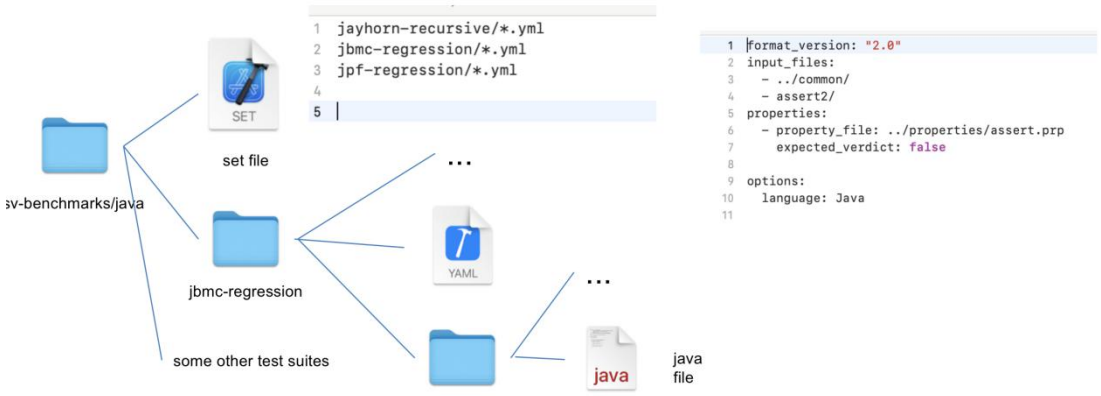

Fig. 3.2: The structure of sv-benhmarks folder

The code fragment in Listing 3.3 shows how to deal with the input file. The folder of original benchmarks is called sv-benchmarks/java, which we can obtain from https://sv-comp.sosy-lab.org/2021 [35]. The structure of the benchmarks folder is illustrated in Fig. 3.2. The folder contains a set file and some folders of test suites such as jbmc-regression, jayhorn-recursive and so on. In each test suites folder, there are many YAML files and corresponding benchmark files that make up the entire test set. Therefore, in this way, we only need to provide a modified set file to test all benchmarks in the test set folder.

In Listing 3.3, in lines 1-4 of the code, first we parse the input set file. The content of this set file contains the YAML file corresponding to the target benchmarks we want to test. We extract the YAML file paths and names as a list and use the sort() function to sort it alphabetically. After that, lines 5-8 indicate that the YAML files in the list are traversed through the YAML loader. In lines 9-12, we extract the expected verdicts of the benchmarks from the YAML files, which have two values, True and False. The 13th line of code is used to find the path of the corresponding benchmark files through the YAML files to prepare for the subsequent operation.

```
1. # Compile
2.         java_list = os.listdir(path)
3.         for file_name in java_list:
4.             if os.path.isfile(path + file_name):
5.                 shutil.copy(path + file_name, os.getcwd())
6.             elif os.path.isdir(path):
7.                 print(path+file_name)
8.                 shutil.copytree(path + file_name, os.getcwd() + '/' + file_name)
9.         subprocess.Popen(['javac', "Main.java"]).wait()
```

Listing 3.4: Code for compiling .java files in run.py

Listing 3.4 is about compiling a java program into a bytecode file before using JBMC or other tools. The key in this code is to use the following command line on the ninth line:

javac Main.java

This command line can compile the java program Main.java into its corresponding bytecode file Main.class through javac. The other parts are used for file path operations.

```
1. # Execute JBMC
2.         if(option == "JBMC" or option == "Both"):
3.             timeout = False
4.             fjout = open("log/" + yml.split('/')[-1].split('.')[0] + "JBMC_file_out.log", 'w')
5.             fjerr = open("log/" + yml.split('/')[-1].split('.')[0] + "JBMC_err_out.log", 'w')
6.             try:
```

```
7.                 subprocess.call(["jbmc", "Main", "--stop-on-fail"], stdout=fjout, stderr=fjerr, timeout
    = 10)
8.             except subprocess.TimeoutExpired as e:
9.                 print("Time out!\n")
10.                timeout = True
11.            with open("log/" + yml.split('/')[-1].split('.')[0] + "JBMC_file_out.log", 'r') as f:
12.                lines = f.readlines()
13.                if "FAIL" in lines[-1]:
14.                    jbmc_result = "False"
15.                elif "SUCCESSFUL" in lines[-1]:
16.                    jbmc_result = "True"
17.                else:
18.                    jbmc_result = "Unknown"
```

Listing 3.5: Code for executing JBMC tool in run.py

Listing 3.5 shows how we call the JBMC verification tool in run.py and process the verification results. Lines 4-5 specify the location of the output file for subsequent correctness analysis. There are two output files. One is a log file, which records the log during execution, and it is saved as JBMC_file_out.log; the other is an error file, which records the error log during execution, and it is saved as JBMC_err_out.log. Line 6-10 of the code is the command line to run JBMC. The key command line to call is:

jbmc Main --stop-on-fail

We manually set an execution timeout time. When the execution time expires, the value of the variable timeout is set to True. Lines 11-18 are the analysis of the generated log file, so that the execution result is saved as a string variable, and finally displayed in the result table. There are three execution results in this step. True means that JBMC verification has no vulnerability, False means that JBMC verification has vulnerabilities, and Timeout indicates that JBMC execution has timed out.

```
1. # Check type
2.             data_type = ''
3.             flag = False
4.             with open("Main.java", "r") as fin:
5.                 for line in fin:
```

```
            if line.find('Verifier.') >= 0 and data_type != '':
                flag = True
                break
            index = line.find('Verifier.')
            if index != -1:
                data_type = line[index + 15:].lower().split('(')[0]
```

Listing 3.6: Code for checking verifier type in run.py

Before running the Witness-based verification tool, we need to determine the data type generated by Verifier class so that we can make mocking by modifying the Validation Harness program. The code in Listing 3.6 shows this checking. The main logic of this code is to find where the Verifier class calls the static method in the Java program and determine the data type. 8 data types such as int, long, and reference type String can be called in the Verifier class.

```
# Execute witness tool
    if(option == "Witness" or option == "Both"):
        fsout = open("log/" + yml.split('/')[-1].split('.')[0] + "script_file_out.log", 'w')
        fserr = open("log/" + yml.split('/')[-1].split('.')[0] + "script_err_out.log", 'w')
        try:
            subprocess.call(["python3", "exec.py", "Main.java"], stdout=fsout, stderr=fserr, timeout=10)
        except subprocess.TimeoutExpired as e:
            print("Time out!")
        with open("log/" + yml.split('/')[-1].split('.')[0] + "script_file_out.log", 'r') as f:
            slines = f.readlines()
            if len(slines)>3:
                if "FAIL" in slines[-3]:
                    script_result = "False"
                elif "OK" in slines[-2]:
                    script_result = "True"
                else:
                    script_result = "Unknown"
            else:
                script_result = "Unknown"
```

Listing 3.7: Code for executing Witness tool in run.py

In Listing 3.7, the code shows the detailed process of executing my Witness

verification tool. The detailed algorithm design of the Witness tool is explained in Listing 3.9 and later, here we just use the following command line to execute.

python3 exec.py Main.java

Similar to the structure of the code in Listing 3.5, lines 3-4 are also used to save the log and error log generated after the program is executed. The log generated by the Witness tool is saved in two log files, script_file_out.log and script_err_out.log. Lines 5-8 of the code is also about the execution process, using the command line to execute exec.py. Lines 9-19 are parsing log files to output execution result information. The Witness verification tool has three possible results. True means that vulnerabilities has been found in the program, and False means that no vulnerabilities has been found, and Unknown means that the tool cannot be used for verification of this benchmark. There are many situations where the tool cannot be used to verify a certain benchmark, such as a null pointer error in the benchmark, or the JBMC execution timeout does not generate a Witness file. Special circumstances will be recorded in the comment column of the result table, see Table 3.8 for details.

```
1. # Add comments
2.         comment = ''
3.         if "Null pointer" in lines[-5]:
4.             comment = "Null pointer exception"
5.         if data_type == '':
6.             comment = "No verifier type"
7.         if flag:
8.             dada_type = ''
9.             comment = "Multiple verifiers"
10.        if timeout:
11.            comment = "Execution time out"
```

Listing 3.8: Code for adding comments in run.py

Listing 3.8 is the code to add comment at the end. Comments are used to supplement the test results of the benchmarks that we test. The common ones are as follows. “Null pointer exception” indicates that there is a null pointer exception in the program, so JBMC execution reports an error, and the witness file cannot be generated; “No

verifier type" and "Multiple verifiers" indicate that there are no or multiple verifier generated variables in the benchmark; "Execution time out" indicates that the JBMC execution times out and no witness is generated. File. These comments are displayed in the comment column of the final result table for users to view.

exec.py

```
#Extract class name and Compile Java program
classArray = sys.argv[1].split('.')
classname = classArray[0]
subprocess.Popen(['javac', sys.argv[1]]).wait()

#Run JBMC
cmd = 'jbmc ' + classname + ' --stop-on-fail --graphml-witness witness'
try:
    result = subprocess.check_output(cmd, shell=True)
except subprocess.CalledProcessError as e:
    result = e.output

#Check for violation
witnessFile = nx.read_graphml("witness")
violation = False
for violationKey in witnessFile.nodes(data=True):
    if 'isViolationNode' in violationKey[1]:
        violation = True
```

Listing 3.9: Code for some steps in exec.py

Listing 3.9 describes some steps of the witness verification script exec.py, including extracting class name and compiling Java program, executing JBMC and finding violations. The basic algorithm of these steps refers to the implementation of Vi [11].

The first step is to extract the class name and compile the Java program. First we use the split() function to separate its class name and extension from the input parameter file name, and extract the class name from them. Then we compile the input Java file with the javac compiler and call it through the command line. This step is the same as the compilation process in Listing 3.4. The second step of executing the JBMC tool is

similar to the process in Listing 3.5. The difference is that an optional property is used. The command line is as follows:

jbmc Main --stop-on-fail --graphml-witness witness

In the command line, --graphml-witness indicates that the system writes the witness in GraphML format to the file name, and the file name is the parameter witness behind. The third step is about the violation, which is mainly to find whether the violation exists through the witness file generated by the above steps. We use the function nx.read_graphml() in the special library to read the witness file and determine whether there is a violation based on the nodes in the file.

```
#Extract counterexample
    counterexamples = []
    for data in witnessFile.edges(data=True):
        if 'assumption' in data[2]:
            str = data[2]['assumption']
            #Get counterexample between ' = ' and ';'
            #E.g. "anonlocal::1i = 1000;"
            if str.startswith('anonlocal'):
                counterexamples.append(str.split(' = ')[1][:-1])

    #Create validation harness from template
    with open("ValidationHarnessTemplate.txt", "rt") as fin:
        lines = []
        if(len(counterexamples) == 0):
            exit(1)
        for line in fin:
            line = line.replace('ClassName', classname)
            lines.append(line)
        for index in range(0, len(types)):
            type = types[index]
            counterexample = counterexamples[index]
```

Listing 3.10: Code for extracting counterexamples in exec.py

The core code in the exec.py script is explained in Listing 3.10. First we extract counterexamples from witness, and create a program to verify the validity of them based on the violations and their data type through the designed Validation Harness

template. This step is called injecting the counterexamples into the Validation Harness.

Lines 2-9 in Listing 3.10 are the extraction of counterexamples. We mainly read the witness file, find the keywords "assumption" and "anonlocal" describing the counterexamples on their nodes, and split it by the split() function to get the counterexamples. Because some benchmarks have multiple Verifier values, here we use a list to store multiple counterexamples. The code in lines 12-21 is part of the counterexample injection. We need to replace the ClassName variable in the Validation Harness template with the class name extracted in Listing 3.9. Then we inject the counterexample into the template according to the data type. An example is shown in Listing 3.11.

```
1. if type == 'int':
2.                 for i in range(0,len(lines)):
3.                     if "Verifier.nondetInt()" in lines[21]:
4.                         lines[21] = lines[21].replace(";\n", ".thenReturn(" + counterexample + ");\n")
5.                         flag = 0
6.                 if flag:
7.                     lines.insert(-3, "    PowerMockito.when(Verifier.nondetInt()).thenReturn(" + cou
    nterexample + ");\n")
```

Listing 3.11: Code for checking data type in exec.py

Listing 3.11 shows an example of injecting a counterexample of a data type into the Validation Harness. Here we use integer data. If we find the "Verifier.nondetInt()" code segment in the benchmark code, then we need to mock the behavior of Verifier in the template and replace it with a counterexample. There are two situations here. If the Verifier type appears for the first time, we need to add the following Java statement to the template:

PowerMockito.when(Verifier.nondetInt()).thenReturn(counterexample);

If the Verifier type does not appear for the first time, just add the following statement after the above Java statement:

.thenReturn(counterexample)

```
#Compile validation harness
subprocess.Popen(['javac', 'ValidationHarness.java']).wait()

#Execute validation harness
subprocess.Popen(['java', '-ea', 'org.junit.runner.JUnitCore' ,'ValidationHarness']).wait()
```

Listing 3.12: Code for compiling and executing validation harness in exec.py

The last part of the exec.py script is the compilation and execution of Validation Harness. The modified Validation Harness is essentially a Java program that needs to be run. During the execution of the program, we need to use JUnit for testing. If the counterexample is valid, the program will report an error, so that it can be displayed by JUnit. In this step, we use the command line interface to run the following two commands:

javac ValidationHarness.java

Java -ea org.junit.runner.JUnitCore ValidationHarness

3.3.2 Witness and Validation Harness

```
<edge source="18.115" target="19.116">
  <data key="originfile">Main.java</data>
  <data key="startline">13</data>
  <data key="threadId">0</data>
  <data key="assumption">anonlocal::1i = 1000;</data>
  <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
```

Listing 3.13: Code of the violation in witness

The witness file is stored in GraphML format, which is similar to XML. Listing 3.13 is an example that shows the important part of the witness generated after executing the exec.py script. We can clearly see in the fifth line that a counterexample integer data 1000 is generated in this program.

```
@RunWith(PowerMockRunner.class)
public class ValidationHarness {
  @Test
  @PrepareForTest(Verifier.class)
  public void testCallStaticMethod() {
    PowerMockito.mockStatic(Verifier.class);
    PowerMockito.when(Verifier.nondetInt()).thenReturn(1000);
    Main.main(new String[0]);
  }
}
```

Listing 3.14: Code of the validation harness

Listing 3.14 is the Validation Harness program generated after executing exec.py. Corresponding to 3.13, we replaced the placeholder ClassName with this class name, and replaced the placeholder Counterexample with the exact counterexample 1000. In addition, because we use the JUnit framework for verification, we use a lot of JUnit-related annotations here. The first and fourth lines of the code use two annotations @Runwith and @PrepareForTest, and their function is to jointly represent mocking the method of Verifier.class using PowerMock. @Test represents JUnit's unit test method, which is used to verify possible errors after injecting counterexamples.

3.3.3 Front-end Implementation

```
@app.route('/index')
def inxdex():
    return render_template('index.html')

@app.route('/result', methods=['post'])
def result():
    name = request.files['fileUpload'].filename
    options = request.form.get('options')
    subprocess.call(["python3", "run.py", "sv-benchmarks/java/" + name, options])
    return render_template('result.html')
```

Listing 3.15: Code for setting routes in app.py

In this project, we also designed the third python program app.py, which is used to implement the front-end using the Flask framework. Part of the code of this program is shown in Listing 3.15, whose main function is to bind functions to routes. Our front-end implementation is not complicated, there are only two pages for interaction and display. The index page is used to display the home page, and is provided to the user interface for operations such as set file selection and verification selection; the result page is used to display the verification result table.

3.3.4 Analysis about the Algorithms

In the app.py, which is related to the front-end implementation, the complexity of the algorithm is O(1). Because the code about the front-end implementation does not contain any loops, the time complexity is always a constant value. In the Witness verification tool implementation code exec.py, we have used a single for loop multiple times, which appears multiple times in Listings 3.9 to 3.11. So the time complexity of the program exec.py is linear, expressed as O(N). Finally, a nested for loop appeared in the program run.py, which is responsible for the overall process of the project, as shown in Listings 3.3 and 3.4, which are two-level loops. So the time complexity of this program is O(N^2). We can get the expected verdicts from the YAML file in the test sets, so it is easy to determine the correctness of the programs. Given the small size and simplicity of the Java programs in the benchmark sets, this can also be manually checked. The completeness of the algorithms is unnecessary to check because the Java programs are instantiated using a mock framework.

3.4 Illustrative Examples

This section is to show some illustrative examples of the verification. We test the Verifier types that may be generated in each situation, and select some representative examples for illustration. The examples contain 7 of the 8 basic data types of Java: short, int, long, boolean, char, float and double. In addition, we also try multiple or no Verifier types in the same benchmark program, which is also shown in this subsection. All the benchmark programs are retrieved from SV-COMP, and we can download them freely from https://github.com/sosy-lab/sv-benchmarks [35].

3.4.1 Short Example

```
1. import org.sosy_lab.sv_benchmarks.Verifier;
2.
3. public class Main {
4.   static int field;
5.   static int field2;
6.
7.   public static void main(String[] args) {
8.     int x = 13000; /* we want to specify in an annotation that this param should be
9.                 symbolic */
10.
11.    Main inst = new Main();
12.    field = Verifier.nondetShort();
13.    if (field < 0) return;
14.    inst.test(x, field, field2);
15.    // test(x,x);
16.  }
17.  /* we want to let the user specify that this method should be symbolic */
18.
19.  /*
20.   * test IF_ICMPGT, IADD & ISUB  bytecodes
21.   */
22.  public void test(int x, int z, int r) {
23.    System.out.println("Testing ExSymExe15");
24.    int y = 3;
25.    r = x + z;
26.    z = x - y - 4;
27.    if (r <= 99) {
28.      System.out.println("branch FOO1");
29.      assert false;
30.    } else System.out.println("branch FOO2");
```

```
31.   if (x <= z) System.out.println("branch BOO1");
32.   else System.out.println("branch BOO2");
33.
34.   // assert false;
35. }
36.}
```

Listing 3.16: Code of short example: ExSymExe15_true.java

Listing 3.16 is a benchmark that calls the Verifier class to generate short type data. Its path in the sv-benchmarks folder is java/jpf-regression/ExSymExe15_true.java. The source code has a very long comment part, we ignore it here, and only intercept the program code part.

In the code, the Main class contains two member variables field and field2, and the value of the variable x in line 8 is 13000. In lines 11-14, first we instantiate an object inst, and then call the method of the Verifier class to assign a random short value to the variable field. The 13th line uses an if statement to ensure that the value of the field is greater than or equal to 0. Line 14 calls the test method, where the parameters are x, field, and field2, and their values are the previously assigned values of 13000, the int type value field of the automatic type conversion, and the initial value of the member variable 0. During the execution of the test function, on line 25, we can see that the parameter r is assigned the value x+z, so here r is 13000 plus a number not less than 0, and the result must be greater than 99. Therefore, in the if statement on lines 27-35, the assert statement should be true.

In the first step, we put this program path into the set file, which is located in sv-benchmarks/java folder, run our extension tool. Here we just use the result, and the process of using the tool will be introduced in detail in Chapter 4. The following Fig. 3.3 is the output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jpf-regression	ExSymExe15_true	short	True	True	True	

Fig. 3.3: Result of the short example

In the result table, we can see that the short type benchmark jpf-regression/ExSymExe15_true.java has been verified by JBMC and the result is True. This shows that no vulnerability in this program has been detected, and the result we have obtained is consistent with the expected result. In addition, we can regard the expected verdict extracted from the YAML file as the correct answer, and the result of our verification is the same, which shows the correctness and accuracy of our verification tool.

Because of the limitation of the test sets, we did not find any short-type benchmarks with vulnerabilities in some test sets, such as jpf-regression and jbmc-regression. In this example, we verified an example without error. Therefore, the result of JBMC execution should be “VERIFICATION SUCCESSFUL”, and there is no extraction of counterexamples and subsequent verification operations of the Witness tool. Part of the log produced by JBMC is shown in Fig. 3.4.

```
32 6198 variables, 8317 clauses
33 SAT checker: instance is UNSATISFIABLE
34 BV-Refinement: got UNSAT, and the proof passes => UNSAT
35 Total iterations: 1
36 Runtime Solver: 0.00513861s
37 Runtime decision procedure: 0.00706913s
38 VERIFICATION SUCCESSFUL
```

Fig. 3.4: JBMC log of the short example

3.4.2 Int Example

```
1. import org.sosy_lab.sv_benchmarks.Verifier;
2.
3. class Main {
4.   public static void main(String[] args) {
5.     int i = Verifier.nondetInt();
6.
7.     if (i >= 10) assert i >= 20 : "my super assertion"; // should hold
```

```
8.  }
9. }
```

Listing 3.17: Code of int example: assert4.java

Listing 3.17 is a benchmark that calls the Verifier class to generate int type data. Its path in the sv-benchmarks folder is java/jbmc-regression/assert4.java. The source code has a very long comment part, we ignore it here, and only intercept the program code part.

In the code, the Verifier class is called in line 5, which provides a random int value to the variable i. Then in line 7, there is an if statement and an assert statement in the branch. It is clear that if the variable i is between 10 and 19, the assert statement will fail as expected.

In the first step, we put this program path into the set file, which is located in sv-benchmarks/java folder, and run our extension tool. The following Fig. 3.5 is the output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jbmc-regression	assert4	int	False	False	False	

Fig. 3.5: Result of the int example

In the result table, we can see that the int type benchmark jbmc-regression/assert4.java has been verified by JBMC and Witness tool, and the results are all False. This shows that one or more vulnerabilities in this program have been detected, and the results we have obtained are consistent with the expected results. Since the result of the JBMC verification tool is False, the subsequent extraction of counterexamples and Witness are all executed normally. A partial screenshot of the JBMC log is shown in the figure below, which shows in line 323 that there is an assertion failure in the program.

```
Violated property:
  file Main.java function Main.main(java.lang.String[]) line 15 thread 0
  assertion at file Main.java line 15 function java::Main.main:([Ljava/lang/String;)V bytecode-index 14
  false


VERIFICATION FAILED
```

Fig. 3.6: JBMC log of the int example

Listing 3.18 shows part of the code in witness file produced by JBMC. The result of JBMC execution is false, so there should be a counterexample whose type is int in the witness file. In line 5 in Listing 3.18, a counterexample of the benchmark is provided, which is int value 10. This means that if the value of the Verifier data in the benchmark program is 10, there may be a weakness in the program.

```
1. <edge source="18.116" target="19.117">
2.   <data key="originfile">Main.java</data>
3.   <data key="startline">13</data>
4.   <data key="threadId">0</data>
5.   <data key="assumption">anonlocal::1i = 10;</data>
6.   <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
7. </edge>
```

Listing 3.18: Witness file of int example

After obtaining the counterexample, we inject it into the Validation Harness for verification. We modify and run the designed Validation Harness template, and then verify the correctness of the program through the JUnit framework. After removing the import statement, the code of modified Validation Harness is shown in Listing 3.19. In line 7 of the code, when the nodetInt() method of the Verifier class is called, the PowerMock framework will return an int value of 10. A screenshot of part of the log after verification by JUnit is shown in Fig. 3.7. In the 9th line of the log, we can see that there is an error in the 16th line of the Validation Harness.java program, which indicates that the program has vulnerabilities and the counterexample we obtained is effective.

```
1. @RunWith(PowerMockRunner.class)
```

```
2. public class ValidationHarness {
3.   @Test
4.   @PrepareForTest(Verifier.class)
5.   public void testCallStaticMethod() {
6.     PowerMockito.mockStatic(Verifier.class);
7.     PowerMockito.when(Verifier.nondetInt()).thenReturn(10);
8.     Main.main(new String[0]);
9.   }
10.}
```

Listing 3.19: Validation Harness of int example

```
4 There was 1 failure:
5 1) testCallStaticMethod(ValidationHarness)
6 java.lang.AssertionError: my super assertion
7         at Main.main(Main.java:15)
8         at ValidationHarness.testCallStaticMethod(ValidationHarness.java:16)
9         at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
10        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
11        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
12        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
13        at org.junit.internal.runners.TestMethod.invoke(TestMethod.java:68)
```

Fig. 3.7: Witness tool log of the int example

3.4.3 Long Example

```
1. import org.sosy_lab.sv_benchmarks.Verifier;
2.
3. public class Main {
4.   public static void main(String[] args) {
5.     long longValue = Verifier.nondetLong();
6.     System.out.printf("long = %s\n", String.valueOf(longValue));
7.     String tmp = String.valueOf(longValue);
8.     assert tmp.equals("100000000000");
9.   }
10.}
```

Listing 3.20: Code of long example: StringValueOf07.java

Listing 3.19 is a benchmark that calls the Verifier class to generate long type data. Its path in the sv-benchmarks folder is java/jbmc-regression/StringValueOf07.java.

In the code, the Verifier class is called in line 5, which provides a random long value

to the variable longValue. In line 7 of the code, the ValueOf() method of the String class is called to convert the variable longValue of type long into a string tmp. Line 8 compares the converted string tmp with the string "100000000000". Obviously, when the generated long type random number is not equal to 100000000000, an assertion error will occur. Thus, the assert statement will fail as expected.

In the first step, we put this program path into the set file, which is located in sv-benchmarks/java folder, and run our extension tool. The following Fig. 3.8 is the output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jbmc-regression	StringValueOf07	long	False	False	False	

Fig. 3.8: Result of the long example

In the result table, we can see that the long type benchmark jbmc-regression/StringValueOf07.java has been verified by JBMC and Witness tool, and the results are all False. This shows that one or more vulnerabilities in this program have been detected, and the results we have obtained are consistent with the expected results. Since the result of the JBMC verification tool is False, the subsequent extraction of counterexamples and Witness are all executed normally. A partial screenshot of the JBMC log is shown in the figure below, which shows in line 775 that there is a assertion failure in the program.

```
773 Violated property:
774   file Main.java function Main.main(java.lang.String[]) line 16 thread 0
775   assertion at file Main.java line 16 function java::Main.main:([Ljava/lang/String;)V bytecode-index 25
776   false
777
778
779 VERIFICATION FAILED
```

Fig. 3.9: JBMC log of the long example

Listing 3.20 shows part of the code in witness file produced by JBMC. The result of JBMC execution is false, so there should be a counterexample whose type is long in

the witness file. In line 5 in Listing 3.20, a counterexample of the benchmark is provided, which is long value 0. This means that if the value of the Verifier data in the benchmark program is 0L, there may be a weakness in the program.

```
1. <edge source="61.257" target="65.272">
2.     <data key="originfile">Main.java</data>
3.     <data key="startline">13</data>
4.     <data key="threadId">0</data>
5.     <data key="assumption">anonlocal::1l = 0L;</data>
6.     <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
7.   </edge>
```

Listing 3.21: Witness file of long example

After obtaining the counterexample, we inject it into the Validation Harness for verification. We modify and run the designed Validation Harness template, and then verify the correctness of the program through the JUnit framework. After removing the import statement, the code of modified Validation Harness is shown in Listing 3.22. In line 7 of the code, when the nodetLong() method of the Verifier class is called, the PowerMock framework will return an int value of 0L. A screenshot of part of the log after verification by JUnit is shown in Fig. 3.10. In the 9th line of the log, we can see that there is an error in the 16th line of the Validation Harness.java program, which indicates that the program has vulnerabilities and the counterexample we obtained is effective.

```
1. @RunWith(PowerMockRunner.class)
2. public class ValidationHarness {
3.   @Test
4.   @PrepareForTest(Verifier.class)
5.   public void testCallStaticMethod() {
6.     PowerMockito.mockStatic(Verifier.class);
7.     PowerMockito.when(Verifier.nondetLong()).thenReturn(0L);
8.     Main.main(new String[0]);
9.   }
10.}
```

Listing 3.22: Validation Harness of int example

```
There was 1 failure:
1) testCallStaticMethod(ValidationHarness)
java.lang.AssertionError
        at Main.main(Main.java:16)
        at ValidationHarness.testCallStaticMethod(ValidationHarness.java:16)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
        at org.junit.internal.runners.TestMethod.invoke(TestMethod.java:68)
```

Fig. 3.10: Witness tool log of the long example

3.4.4 Boolean Example

```
1. import org.sosy_lab.sv_benchmarks.Verifier;
2.
3. public class Main {
4.   public static void main(String[] args) {
5.     boolean booleanValue = Verifier.nondetBoolean();
6.     String tmp = String.valueOf(booleanValue);
7.     assert tmp.equals("true");
8.   }
9. }
```

Listing 3.23: Code of boolean example: StringValueOf04.java

Listing 3.21 is a benchmark that calls the Verifier class to generate boolean type data. Its path in the sv-benchmarks folder is java/jbmc-regression/StringValueOf04.java.

In the code, the Verifier class is called in line 5, which provides a random boolean value to the variable booleanValue. In line 6 of the code, the ValueOf() method of the String class is called to convert the variable booleanValue of type boolean into a string tmp. Line 7 compares the converted string tmp with the boolean value "true". Obviously, when the generated boolean type random value is false, an assertion error will occur. Thus, the assert statement will fail as expected.

The first step we put this program path into the set file, which is located in sv-benchmarks/java folder, and run our extension tool. The following Fig. 3.11 is the

output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jbmc-regression	StringValueOf04	boolean	False	False	False	

Fig. 3.11: Result of the boolean example

We can see that the boolean type benchmark jbmc-regression/StringValueOf04.java has been verified by JBMC and Witness tool, and the results are all False. This shows that one or more vulnerabilities in this program have been detected, and the results we have obtained are consistent with the expected results. Since the result of the JBMC verification tool is False, the subsequent extraction of counterexamples and Witness are all executed normally. A partial screenshot of the JBMC log is shown in the figure below, which shows in line 330 that there is a assertion failure in the program.

```
328 Violated property:
329   file Main.java function Main.main(java.lang.String[]) line 15 thread 0
330   assertion at file Main.java line 15 function java::Main.main:([Ljava/lang/String;)V bytecode-index 14
331   false
332
333
334 VERIFICATION FAILED
```

Fig. 3.12: JBMC log of the long example

Listing 3.22 shows part of the code in witness file produced by JBMC. The result of JBMC execution is false, so there should be a counterexample whose type is boolean in the witness file. In line 5 in Listing 3.22, a counterexample of the benchmark is provided, which is boolean value 0. This means that if the value of the Verifier data in the benchmark program is false, there may be a weakness in the program.

```
1. <edge source="48.104" target="49.107">
2.   <data key="originfile">Main.java</data>
3.   <data key="startline">13</data>
4.   <data key="threadId">0</data>
5.   <data key="assumption">anonlocal::1i = 0;</data>
6.   <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
7. </edge>
```

Listing 3.24: Witness file of boolean example

After obtaining the counterexample, we inject it into the Validation Harness for verification. The process and results of this step are similar to the above, and a similar error was reported, which indicates that the program has vulnerabilities and the counterexample we obtained is effective.

3.4.5 Char Example

```
1. import org.sosy_lab.sv_benchmarks.Verifier;
2.
3. public class Main {
4.   public static void main(String[] args) {
5.     char c = Verifier.nondetChar();
6.     assert Character.isLetter(c);
7.   }
8. }
```

Listing 3.25: Code of char example: StaticCharMethods04.java

Listing 3.23 is a benchmark that calls the Verifier class to generate char type data. Its path in the sv-benchmarks folder is java/jbmc-regression/StaticCharMethods04.java.

In the code, the Verifier class is called in line 5, which provides a random char value to the variable c. In line 6 of the code, the isLetter() method of the Character class is called to determine whether char c is a letter. Obviously, if the random char value is not a letter, an assertion false will occur.

We put this program path into the set file and run our extension tool. The following Fig. 3.13 is the output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jbmc-regression	StaticCharMethods04	char	False	False	False	

Fig. 3.13: Result of the char example

We can see that the char type benchmark jbmc-regression/StaticCharMethods04 has been verified by JBMC and Witness tool, and the results are all False. Since the result of the JBMC verification tool is False, the subsequent extraction of counterexamples and Witness are executed. A partial screenshot of the JBMC log is shown in the figure below, which shows in line 227 that there is a assertion failure in the program.

```
225 Violated property:
226   file Main.java function Main.main(java.lang.String[]) line 14 thread 0
227   assertion at file Main.java line 14 function java::Main.main:([Ljava/lang/String;)V bytecode-index 10
228   false
229
230
231 VERIFICATION FAILED
```

Fig. 3.14: JBMC log of the char example

Listing 3.24 shows part of the code in witness file produced by JBMC. The result of JBMC execution is false, so there should be a counterexample whose type is char in the witness file. In line 5 in Listing 3.24, a counterexample of the benchmark is provided, which is char value 60. 60 corresponds to "<" in Java, which is a symbol, not a letter. This means that if the value of the Verifier data in the benchmark program is 60, there may be a weakness in the program.

```
1. <edge source="21.93" target="51.104">
2.   <data key="originfile">Main.java</data>
3.   <data key="startline">13</data>
4.   <data key="threadId">0</data>
5.   <data key="assumption">anonlocal::1i = 60;</data>
6.   <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
7. </edge>
```

Listing 3.26: Witness file of char example

After obtaining the counterexample, we inject it into the Validation Harness for

verification. The process and results of this step are similar to the above, and a similar error was reported, which indicates that the program has vulnerabilities and the counterexample we obtained is effective.

3.4.6 Float Example

```
1. import org.sosy_lab.sv_benchmarks.Verifier;
2.
3. public class Main {
4.
5.   public static void main(String[] args) {
6.     float x = Verifier.nondetFloat();
7.     Main inst = new Main();
8.     inst.test(x);
9.   }
10.
11.  public void test(float x) {
12.    System.out.println("Testing FNEG");
13.    float y = -x;
14.    if (y > 0) System.out.println("branch -x > 0");
15.    else {
16.      assert false;
17.      System.out.println("branch -x <= 0");
18.    }
19.  }
20.}
```

Listing 3.27: Code of float example: ExSymExeFNEG_false.java

Listing 3.25 is a benchmark that calls the Verifier class to generate float type data. Its path in the sv-benchmarks folder is java/jpf-regression/ExSymExeFNEG_false.java.

In the code, the Verifier class is called in line 6, which provides a random float value to the variable x. Line 7 instantiates an object inst, and the 8th line calls the test() method, where the parameter is x. In the test() method, the float variable y is assigned the value -x. There is an if statement on line 14, which asserts false when y is less than or equal to 0. Therefore, if the random float value is not less than 0, an assertion

false will occur.

We put this program path into the set file and run our extension tool. The following Fig. 3.15 is the output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jpf-regression	ExSymExeFNEG_false	float	False	False	False	

Fig. 3.15: Result of the float example

We can see that the float type benchmark jpf-regression/ExSymExeFNEG_false.java has been verified by JBMC and Witness tool, and the results are all False. Since the result of the JBMC verification tool is False, the subsequent extraction of counterexamples and Witness are executed. A partial screenshot of the JBMC log is shown in the figure below, which shows in line 523 that there is a assertion failure in the program.

```
521 Violated property:
522   file Main.java function Main.test(float) line 42 thread 0
523   assertion at file Main.java line 42 function java::Main.test:(F)V bytecode-index 19
524   false
525
526
527 VERIFICATION FAILED
```

Fig. 3.16: JBMC log of the float example

Listing 3.26 shows part of the code in witness file produced by JBMC. The result of JBMC execution is false, so there should be a counterexample whose type is float in the witness file. In line 5 in Listing 3.28, a counterexample of the benchmark is provided, which is float value 0. This means that if the value of the Verifier data in the benchmark program is 0.0f, there may be a weakness in the program.

```
1. <edge source="59.151" target="79.162">
2.     <data key="originfile">Main.java</data>
3.     <data key="startline">32</data>
4.     <data key="threadId">0</data>
```

```
5.      <data key="assumption">anonlocal::1f = 0.0f;</data>
6.      <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
7.    </edge>
```

Listing 3.28: Witness file of float example

After obtaining the counterexample, we inject it into the Validation Harness for verification. The process and results of this step are similar to the above, and a similar error was reported, which indicates that the program has vulnerabilities and the counterexample we obtained is effective.

3.4.7 Double Example

```
1. import org.sosy_lab.sv_benchmarks.Verifier;
2.
3. public class Main {
4.
5.   public static void main(String[] args) {
6.     double x = Verifier.nondetDouble();
7.
8.     Main inst = new Main();
9.     inst.test(x);
10.  }
11.
12.  public void test(double x) {
13.
14.    long res = (long) ++x;
15.    if (res > 0) {
16.      assert false;
17.      System.out.println("x >0");
18.    } else System.out.println("x <=0");
19.  }
20.}
```

Listing 3.29: Code of double example: ExSymExeD2L_false.java

Listing 3.27 is a benchmark that calls the Verifier class to generate double type data.

Its path in the sv-benchmarks folder is java/jpf-regression/ExSymExeD2L_false.java.

In the code, the Verifier class is called in line 6, which provides a random double value to the variable x. Line 8 instantiates an object inst, and the 9th line calls the test() method, where the parameter is x. In the test() method, the long variable res is assigned the value x+1. There is an if statement on line 15, which asserts false when res is greater than 0. Therefore, if the random double value is not less than 0, an assertion false will occur.

We put this program path into the set file and run our extension tool. The following Fig. 3.17 is the output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jpf-regression	ExSymExeD2L_false	double	False	False	False	

Fig. 3.17: Result of the double example

We can see that the double type benchmark jpf-regression/ExSymExeD2L_false.java has been verified by JBMC and Witness tool, and the results are all False. Since the result of the JBMC verification tool is False, the subsequent extraction of counterexamples and Witness are executed. A partial screenshot of the JBMC log is shown in the figure below, which shows in line 359 that there is a assertion failure in the program.

```
357 Violated property:
358   file Main.java function Main.test(double) line 42 thread 0
359   assertion at file Main.java line 42 function java::Main.test:(D)V bytecode-index 16
360   false
361
362
363 VERIFICATION FAILED
```

Fig. 3.18: JBMC log of the double example

Listing 3.28 shows part of the code in witness file produced by JBMC. The result of

JBMC execution is false, so there should be a counterexample whose type is double in the witness file. In line 5 in Listing 3.30, a counterexample of the benchmark is provided, which is float value 0.0. This means that if the value of the Verifier data in the benchmark program is 0.0, there may be a weakness in the program.

```
1. <edge source="60.116" target="80.127">
2.   <data key="originfile">Main.java</data>
3.   <data key="startline">32</data>
4.   <data key="threadId">0</data>
5.   <data key="assumption">anonlocal::1d = 0.0;</data>
6.   <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
7.</edge>
```

Listing 3.30: Witness file of double example

After obtaining the counterexample, we inject it into the Validation Harness for verification. The process and results of this step are similar to the above, and a similar error was reported, which indicates that the program has vulnerabilities and the counterexample we obtained is effective.

3.4.8 Multiple Verifiers Example

```
1. import org.sosy_lab.sv_benchmarks.Verifier;
2.
3. class Main {
4.   public static void main(String[] args) {
5.     int v1 = Verifier.nondetInt();
6.     int v2 = Verifier.nondetInt();
7.     assert v1 == v2; // should be able to fail
8.   }
9. }
```

Listing 3.31: Code of multiple verifiers example: return2.java

Listing 3.31 is a benchmark that calls the Verifier class twice to generate int type data. Its path in the sv-benchmarks folder is java/jbmc-regression/return2.java.

In the code, the Verifier class is called in line 5 and line 6, which provides two random int values to the variable v1 and the variable v2. In line 7, there is an assert statement to compare the two variables. Therefore, if the two int variables are not equal, an assertion false will occur.

We put this program path into the set file and run our extension tool. The following Fig. 3.19 is the output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jbmc-regression	return2	int	False	False	False	Multiple verifiers

Fig. 3.19: Result of the multiple verifiers example

We can see that the multiple verifiers benchmark jbmc-regression/return2.java has been verified by JBMC and Witness tool, and the results are all False. Since the result of the JBMC verification tool is False, the subsequent extraction of counterexamples and Witness are executed. A partial screenshot of the JBMC log is shown in the figure below, which shows in line 235 that there is a assertion failure in the program.

```
233 Violated property:
234   file Main.java function Main.main(java.lang.String[]) line 15 thread 0
235   assertion at file Main.java line 15 function java::Main.main:([Ljava/lang/String;)V bytecode-index 12
236   false
237
238
239 VERIFICATION FAILED
```

Fig. 3.20: JBMC log of the multiple verifiers example

Listing 3.32 shows part of the code in witness file produced by JBMC. The result of JBMC execution is false, so there should be two counterexamples in the witness file. In line 6 and line 14 in Listing 3.32, two counterexample of the benchmark are provided, which are int values 1 and 0. This means that if the values of the Verifier data in the benchmark program is 1 and 0, there may be a weakness in the program.

```
<node id="20.95"/>
<edge source="20.95" target="24.112">
  <data key="originfile">Main.java</data>
  <data key="startline">13</data>
  <data key="threadId">0</data>
  <data key="assumption">anonlocal::1i = 1;</data>
  <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
</edge>
<node id="24.112"/>
<edge source="24.112" target="55.123">
  <data key="originfile">Main.java</data>
  <data key="startline">14</data>
  <data key="threadId">0</data>
  <data key="assumption">anonlocal::2i = 0;</data>
  <data key="assumption.scope">java::Main.main:([Ljava/lang/String;)V</data>
</edge>
```

Listing 3.32: Witness file of multiple verifiers example

After obtaining the two counterexamples, we inject them into the Validation Harness for verification. We modify and run the designed Validation Harness template, and then verify the correctness of the program through the JUnit framework. After removing the import statement, the code of modified Validation Harness is shown in Listing 3.33. In line 7 of the code, when the nodetInt() method of the Verifier class is called the first time, the PowerMock framework will return an int value of 1, and when it is called the second time it will get an int value of 0. A screenshot of part of the log after verification by JUnit is shown in Fig. 3.21. In the 8th line of the log, we can see that there is an error in the 16th line of the Validation Harness.java program, which indicates that the program has vulnerabilities and the counterexamples we obtained are effective.

```
@RunWith(PowerMockRunner.class)
public class ValidationHarness {
  @Test
```

```
4.  @PrepareForTest(Verifier.class)
5.  public void testCallStaticMethod() {
6.    PowerMockito.mockStatic(Verifier.class);
7.    PowerMockito.when(Verifier.nondetInt()).thenReturn(1).thenReturn(0);
8.    Main.main(new String[0]);
9.   }
10.}
```

Listing 3.33: Validation Harness of multiple verifiers example

```
 6 java.lang.AssertionError
 7         at Main.main(Main.java:15)
 8         at ValidationHarness.testCallStaticMethod(ValidationHarness.java:16)
 9         at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
10         at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
11         at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
12         at java.base/java.lang.reflect.Method.invoke(Method.java:564)
13         at org.junit.internal.runners.TestMethod.invoke(TestMethod.java:68)
```

Fig. 3.21: Witness tool log of the multiple verifiers example

3.4.9 No Verifier Example

```
1.  public class Main {
2.    public static void main(String[] args) {
3.      try {
4.        Object x = new Integer(0);
5.        String y = (String) x;
6.      } catch (ClassCastException exc) {
7.        assert false;
8.      }
9.    }
10.}
```

Listing 3.34: Code of no verifier example: ClassCastException1.java

Listing 3.34 is a benchmark that does not call the Verifier class. Its path in the sv-benchmarks folder is java/jbmc-regression/ClassCastException1.java.

We can see that the Verifier class is not called in the code. In lines 3 to 8 of the code, there is a try-catch code block. It converts the Integer class to the String class, which

will throw an exception. In line 7, there is an assert statement. Therefore, if a class cast exception is caught, the assertion failure will occur.

We put this program path into the set file and run our extension tool. The following Fig. 3.22 is the output result table of our tool on the web page.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jbmc-regression	ClassCastException1		False	False	False	No verifier type

Fig. 3.22: Result of the no verifier example

We can see that the no verifier benchmark jbmc-regression/ClassCastException1.java has been verified by JBMC and Witness tool, and the results are all False. A partial screenshot of the JBMC log is shown in the figure below, which shows in line 183 that there is a assertion failure in the program.

```
180 Violated property:
181   file Main.java function Main.main(java.lang.String[]) line 13 thread 0
182   Dynamic cast check
183   anonlocal::1a != null && ((struct java.lang.Object *)anonlocal::1a)->@class_identifier == "java::java.lang.String"
184
185
186 VERIFICATION FAILED
```

Fig. 3.23: JBMC log of the no verifier example

In this program, since the methods of the Verifier class are not called, no counterexamples are generated. This method of verifying software vulnerabilities through witness is meaningless. However, our tool can be used normally, because in this case, the Validation Harness code is shown in Listing 3.35. The Mock method in the sixth line of the code does not have any effect, so it is equivalent to verifying the software directly through JUnit, and we can also get the expected results.

```
1. @RunWith(PowerMockRunner.class)
2. public class ValidationHarness {
3.   @Test
4.   @PrepareForTest(Verifier.class)
```

```
5.  public void testCallStaticMethod() {
6.    PowerMockito.mockStatic(Verifier.class);
7.    Main.main(new String[0]);
8.  }
9. }
```

Listing 3.35: Validation Harness of no verifier example

3.4.10 Summary

In this subsection, we give a lot of examples to verify the correctness of the algorithm in this chapter and test whether it achieves the expected effect. The examples we use contain 7 basic types in the Java language, including short, int, long, boolean, char, float and double. We failed to use the short type benchmark whose verification result was False, and did not verify the byte type. These shortcomings are all caused by the test set. In addition, our tool supports multi-file verification. The unified verification results of the examples in this subsection are shown in Figure 3.24.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jpf-regression	ExSymExe15_true	short	True	True	True	
2	jbmc-regression	assert4	int	False	False	False	
3	jbmc-regression	StringValueOf07	long	False	False	False	
4	jbmc-regression	StringValueOf04	boolean	False	False	False	
5	jbmc-regression	StaticCharMethods04	char	False	False	False	
6	jpf-regression	ExSymExeFNEG_false	float	False	False	False	
7	jpf-regression	ExSymExeD2L_false	double	False	False	False	
8	jbmc-regression	return2	int	False	False	False	Multiple verifiers
9	jbmc-regression	ClassCastException1		False	False	False	No verifier type

Fig. 3.24: Result of all the examples mentioned in this subsection

Chapter 4 Experimental Evaluation

In this chapter, we focus on describing the experimental evaluation of the implemented algorithms in Chapter 3. By verifying a large number of verification sets, we can have a detailed evaluation of the effect of our algorithm. There are 5 sections in this chapter. Section 4.1 introduces the benchmarks used in the program. Section 4.2 introduces required environment and configurations for the experimental evaluation, including programming language, necessary libraries and so on. We describe our objectives of the evaluation in section 4.3. We show all the verification results in Section 4.4 to illustrate the effectiveness of our algorithm, and analyze the results to reveal the advantages and disadvantages of our algorithm. Finally, we explained the reasons for the limitations of the algorithm. We discuss about the threats to validity in Section 4.5.

4.1 Description of the benchmarks

The benchmark programs used in our experimental evaluation are all derived from SV-COMP, and we have many ways to obtain a complete verification set. One way is to get all the benchmarks on the official website of the competition: https://sv-comp.sosy-lab.org/2021, or in the relevant GitHub repository: https://github.com/sosy-lab/sv-benchmarks. In this project, we only focus on benchmarks about the Java language, so the programs we need are in the /java folder. The files in the Java folder are shown in Figure 4.1.

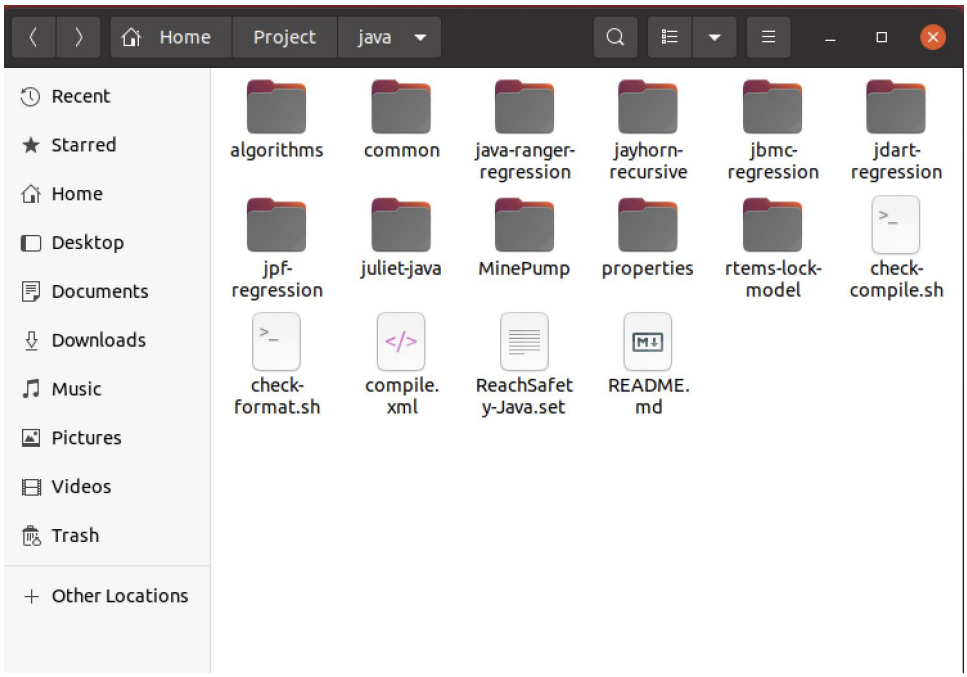

Fig. 4.1: sv–benchmark/java

In this folder, the part we need to pay attention to in this project includes folders containing several benchmark programs, such as jbmc-gression, the folder common containing Verifier, and the set file ReachSafety.set. Among them, the folder where the benchmark is located contains the program's .java file and related YAML files. They are mentioned in Figure 3.2 in subsection 3.3.1 and explained. The following list 4.1 is a complete benchmark code:

```
1. /*
2.  * Origin of the benchmark:
3.  *     license: 4-clause BSD (see /java/jbmc-regression/LICENSE)
4.  *     repo: https://github.com/diffblue/cbmc.git
5.  *     branch: develop
6.  *     directory: regression/cbmc-java/assert1
7.  * The benchmark was taken from the repo: 24 January 2018
8.  */
9. import org.sosy_lab.sv_benchmarks.Verifier;
10.
11. class Main {
12.   public static void main(String[] args) {
13.     int i = Verifier.nondetInt();
```

```
14.
15.    if (i >= 10) assert i >= 10 : "my super assertion"; // should hold
16.
17.    if (i >= 20) assert i >= 10 : "my super assertion"; // should hold
18.  }
19.}
```

Listing 4.1: assert1.java

In the benchmark program assert1.java, the first to eighth lines are the comments of the program, indicating the version, location, and time of the program. The import statement on line 9 is used to import the Verifier class, which is used to generate Verifier type variables by calling static methods. The code below is the program that needs to be verified. There is often an assert statement in the program to detect software vulnerabilities.

4.2 Setup

This section is to describe in detail the environment in which we design and implement the algorithm program, including the programming language, Java and Python. In addition, we also explained some libraries and dependencies required by the program in this section, as well as their versions.

4.2.1 Programming language

The verification object of our project is a Java program, so we must install the Java Development Kit (JDK) before the project runs. JDK is available to download and install at: https://www.oracle.com/java/technologies/javase-downloads.html. JDK version used in the program is 14.0.2.

We designed and implemented several scripts by Python, and they control the process of the verification execution. Python is available to install at the official website:

https://www.python.org/downloads/. Python version used in the program is 3.8.10.

Then for this project, the following necessary Python libraries should also be installed:

- subprocess
- sys
- networkx
- flask
- yaml

4.2.2 Libraries and tools

JBMC verification tool is the core of the program, which is based on CBMC. CBMC is freely download on the GitHub repository: https://github.com/diffblue/cbmc. In the CBMC project, JBMC tool is in the jbmc folder. For instructions on using JBMC, such as compiling and running, and verifying samples, you can view the Readme file on GitHub, or visit the website: http://www.cprover.org/jbmc/. The JBMC version in our program is 5.27.0.

Powermock is the mock framework we used, which is introduced on the GitHub website: https://github.com/powermock/powermock. In the implementation of the project, we just use the relative .jar files, which we can download from https://mvnrepository.com. The following dependencies are necessary:

- mockito-core-2.23.0.jar
- powermock-api-mockito2-2.0.2.jar
- powermock-mockito2-2.0.2-full.jar
- objenesis-3.0.1.jar
- javassist-3.24.0-GA.jar
- hamcrest-core-1.3.jar

- cglib-nodep-3.2.9.jar
- junit-4.12.jar

The version we used is shown in the list. Finally, the environment variables need to be set using some commands. Here we added them to the file /etc/bash.bashrc (Linux OS).

```
POWERMOCK_HOME=/home/stewartwang/Lib/Powermock
export POWERMOCK_HOME
export CLASSPATH=$CLASSPATH:$POWERMOCK_HOME/powermock-mockito2-2.0.2-full.jar:$POWER
MOCK_HOME/hamcrest-core-1.3.jar:$POWERMOCK_HOME/junit-4.12.jar:$POWERMOCK_HOME/mocki
to-core-2.23.0.jar:$POWERMOCK_HOME/cglib-nodep-3.2.9.jar:$POWERMOCK_HOME/objenesis-3
.0.1.jar:$POWERMOCK_HOME/javassist-3.24.0-GA.jar:$POWERMOCK_HOME/powermock-api-mocki
to2-2.0.2.jar:.
```

Fig. 4.2: Configure the environment variables of Powermock

4.2.3 Running process

The following list is the specification of the hardware used in this project:

- Laptop model: MacBook Pro (16-inch, 2019)
- Processor: 2.6 GHz Intel Core i7
- RAM: 16GB 2667 MHz DDR4
- Operating System: Linux Ubuntu on VMware Fusion (Version 20.04.2.0)

The whole project of the dissertation has been push to my GitHub repository. It is available to clone the project by command line or download the zip file from: https://github.com/SongtaoWang-98/JBMC_extension.git. There are mainly 3 Python scripts, a Validation Harness template file and other 4 folders in the project. The functions of these folders include saving logs, providing verification benchmarks, and providing required Verifier class.

The complete sv-benchmarks folder can be obtained from the following website: https://github.com/sosy-lab/sv-benchmarks. And in the sv-benchmarks folder, we mainly focus on the benchmark programs in the Java area, that is, the Java program in

/java folder. In addition, my project file also contains the entire sv-benchmarks folder, which can be used directly.

After the installations and configurations, our verification tool is ready to run. First, we should run the tool by the command line:

python3 app.py

Now we can use the web application on http://127.0.0.1:5000/index to execute the verification for the benchmarks. The simple interface is shown in Fig. 4.3.

Fig. 4.3: Front–end interface

Then we should make sure which program to verify, and modify the Java.set file in sv-benchmarks/java folder. The following Fig 4.4 shows the content of the set file in subsection 3.4.10:

Fig. 4.4: test.set file

Finally it is easy to select the options and the path of the set file, and click the execute

button. After several seconds we can get a result table on the web page, as is shown in Fig. 3.24.

4.3 Objectives of the evaluation

The overall goal of the experimental evaluation in this chapter is to show the results of using the verification tool we designed and implemented, and to illustrate its effectiveness in Java software verification. Through actual running tests, we can better explain the function and value of our software.

The specific objectives of this chapter are as follows:

- Test my verification tool by SV-COMP benchmarks, and organize the results of execution.
- Analyze the output results, especially compare the results between JBMC and my tool.
- Compare with the previous related extension and illustrate the innovations and improvements in my extension.
- Find out the deficiencies that still exist in my extension tool, and explain the reasons.

4.4 Results

In section 4.3, we show some of the results table after testing our extension tool. Here are three tables of the verification sets: jbmc-regression, jpf-regression and jdart-regression. The results are shown in Table 4.1, Table 4.2 and Table 4.3.

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jbmc-regression	ArithmeticException1	int	False	False	False	
2	jbmc-regression	ArithmeticException5		True	True	True	No verifier type

3	jbmc-regression	ArithmeticException6	int	False	False	False	
4	jbmc-regression	ArrayIndexOutOfBoundsException1	int	False	False	False	
5	jbmc-regression	ArrayIndexOutOfBoundsException2	int	False	False	False	
6	jbmc-regression	ArrayIndexOutOfBoundsException3	int	False	False	Unknown	Array index should be < length
7	jbmc-regression	BufferedReaderReadLine	string	False	Unknown	Unknown	Execution time out
8	jbmc-regression	CharSequenceBug	string	False	Unknown	Unknown	
9	jbmc-regression	CharSequenceToString	string	True	False	True	
10	jbmc-regression	ClassCastException1		False	False	False	No verifier type
11	jbmc-regression	ClassCastException2		True	True	True	No verifier type
12	jbmc-regression	ClassCastException3		False	False	False	No verifier type
13	jbmc-regression	Class_method1		True	True	True	No verifier type
14	jbmc-regression	Inheritance1		True	True	True	No verifier type
15	jbmc-regression	NegativeArraySizeException1		False	False	False	No verifier type
16	jbmc-regression	NegativeArraySizeException2		False	False	False	No verifier type
17	jbmc-regression	NullPointerException1		True	False	True	No verifier type
18	jbmc-regression	NullPointerException2		False	False	False	No verifier type
19	jbmc-regression	NullPointerException3		False	False	False	No verifier type
20	jbmc-regression	NullPointerException4		False	False	False	No verifier type
21	jbmc-regression	RegexMatches01		True	Unknown	Unknown	Execution time out
22	jbmc-regression	RegexMatches02	string	False	Unknown	Unknown	Execution time out
23	jbmc-regression	RegexSubstitution01		True	False	True	No verifier type
24	jbmc-regression	RegexSubstitution02	string	False	False	False	Multiple verifiers
25	jbmc-regression	RegexSubstitution03		True	False	True	No verifier type
26	jbmc-regression	StaticCharMethods01		True	True	True	No verifier type
27	jbmc-regression	StaticCharMethods02	string	False	False	False	
28	jbmc-regression	StaticCharMethods03	string	False	False	False	
29	jbmc-regression	StaticCharMethods04	char	False	False	False	
30	jbmc-regression	StaticCharMethods05	string	False	False	False	Null pointer exception
31	jbmc-regression	StaticCharMethods06	string	True	False	True	Null pointer exception
32	jbmc-regression	StringBuilderAppend01		True	Unknown	True	No verifier type
33	jbmc-regression	StringBuilderAppend02	string	False	False	False	Multiple verifiers
34	jbmc-regression	StringBuilderCapLen01		True	False	True	No verifier type
35	jbmc-regression	StringBuilderCapLen02	string	False	False	Unknown	
36	jbmc-regression	StringBuilderCapLen03	string	False	False	Unknown	
37	jbmc-regression	StringBuilderCapLen04	string	False	False	Unknown	
38	jbmc-regression	StringBuilderChars01		True	False	True	No verifier type
39	jbmc-regression	StringBuilderChars02	string	False	False	Unknown	
40	jbmc-regression	StringBuilderChars03	string	False	False	Unknown	
41	jbmc-regression	StringBuilderChars04	string	False	Unknown	Unknown	Execution time out
42	jbmc-regression	StringBuilderChars05	string	False	False	Unknown	
43	jbmc-regression	StringBuilderChars06	string	False	False	Unknown	
44	jbmc-regression	StringBuilderConstructors01	string	True	True	False	

45	jbmc-regression	StringBuilderConstructors02	string	False	False	Unknown	
46	jbmc-regression	StringBuilderInsertDelete01		True	False	True	No verifier type
47	jbmc-regression	StringBuilderInsertDelete02	string	False	False	False	Multiple verifiers
48	jbmc-regression	StringBuilderInsertDelete03	string	False	False	False	Multiple verifiers
49	jbmc-regression	StringCompare01		True	False	True	No verifier type
50	jbmc-regression	StringCompare02	string	False	False	Unknown	Multiple verifiers
51	jbmc-regression	StringCompare03	string	False	False	Unknown	Multiple verifiers
52	jbmc-regression	StringCompare04	string	False	Unknown	Unknown	Execution time out
53	jbmc-regression	StringCompare05	string	False	False	Unknown	
54	jbmc-regression	StringConcatenation01	string	True	False	Unknown	Multiple verifiers
55	jbmc-regression	StringConcatenation02	string	False	False	Unknown	Multiple verifiers
56	jbmc-regression	StringConcatenation03	string	False	False	Unknown	Multiple verifiers
57	jbmc-regression	StringConcatenation04	string	False	False	Unknown	
58	jbmc-regression	StringConstructors01		True	False	True	No verifier type
59	jbmc-regression	StringConstructors02	string	False	False	Unknown	
60	jbmc-regression	StringConstructors03	string	False	False	Unknown	Multiple verifiers
61	jbmc-regression	StringConstructors04	string	False	False	Unknown	
62	jbmc-regression	StringConstructors05	string	False	False	Unknown	
63	jbmc-regression	StringContains01	string	False	Unknown	Unknown	Execution time out
64	jbmc-regression	StringContains02	string	False	False	Unknown	
65	jbmc-regression	StringIndexMethods01		True	True	Unknown	No verifier type
66	jbmc-regression	StringIndexMethods02	string	False	Unknown	Unknown	Execution time out
67	jbmc-regression	StringIndexMethods03	string	False	Unknown	Unknown	Execution time out
68	jbmc-regression	StringIndexMethods04	string	False	False	Unknown	
69	jbmc-regression	StringIndexMethods05	string	False	Unknown	Unknown	
70	jbmc-regression	StringMiscellaneous01		True	False	True	No verifier type
71	jbmc-regression	StringMiscellaneous02	string	False	False	Unknown	
72	jbmc-regression	StringMiscellaneous03	string	False	Unknown	Unknown	Execution time out
73	jbmc-regression	StringMiscellaneous04		True	False	True	No verifier type
74	jbmc-regression	StringStartEnd01		True	True	True	No verifier type
75	jbmc-regression	StringStartEnd02	string	False	False	False	Multiple verifiers
76	jbmc-regression	StringStartEnd03	string	False	False	False	Multiple verifiers
77	jbmc-regression	StringValueOf01		True	False	True	No verifier type
78	jbmc-regression	StringValueOf02	string	False	False	Unknown	
79	jbmc-regression	StringValueOf03	string	False	False	Unknown	
80	jbmc-regression	StringValueOf04	boolean	False	False	False	
81	jbmc-regression	StringValueOf05	string	False	False	False	
82	jbmc-regression	StringValueOf06	int	False	False	False	
83	jbmc-regression	StringValueOf07	long	False	False	False	
84	jbmc-regression	StringValueOf08	string	False	False	False	
85	jbmc-regression	StringValueOf09	string	False	False	False	
86	jbmc-regression	StringValueOf10	string	False	False	Unknown	
87	jbmc-regression	SubString01		True	False	True	No verifier type

88	jbmc-regression	SubString02	string	False	False	Unknown	
89	jbmc-regression	SubString03	string	False	False	Unknown	
90	jbmc-regression	TokenTest01		True	Unknown	Unknown	Execution time out
91	jbmc-regression	TokenTest02	string	False	Unknown	Unknown	Execution time out
92	jbmc-regression	Validate01		True	False	True	No verifier type
93	jbmc-regression	Validate02	string	False	False	Unknown	Multiple verifiers
94	jbmc-regression	aastore_aaload1	int	True	Unknown	Unknown	Execution time out
95	jbmc-regression	array1	int	True	Unknown	Unknown	Execution time out
96	jbmc-regression	array2	int	True	True	True	
97	jbmc-regression	arraylength1	int	True	True	True	
98	jbmc-regression	arrayread1	int	True	True	True	
99	jbmc-regression	assert1	int	True	True	True	
100	jbmc-regression	assert2	int	False	False	False	
101	jbmc-regression	assert3	int	False	False	False	
102	jbmc-regression	assert4	int	False	False	False	
103	jbmc-regression	assert5	int	True	True	True	
104	jbmc-regression	assert6	int	True	True	True	
105	jbmc-regression	astore_aload1		True	True	True	No verifier type
106	jbmc-regression	athrow1		False	False	False	No verifier type
107	jbmc-regression	basic1		True	True	True	No verifier type
108	jbmc-regression	bitwise1	int	True	True	True	
109	jbmc-regression	boolean1	boolean	True	True	True	
110	jbmc-regression	boolean2	boolean	True	True	True	
111	jbmc-regression	bug-test-gen-095	string	False	False	Unknown	
112	jbmc-regression	bug-test-gen-119-2		True	False	True	No verifier type
113	jbmc-regression	bug-test-gen-119	boolean	True	False	True	Null pointer exception
114	jbmc-regression	calc	int	True	False	True	Multiple verifiers
115	jbmc-regression	cast1	int	True	True	True	
116	jbmc-regression	catch1		True	True	True	No verifier type
117	jbmc-regression	char1	string	True	True	False	
118	jbmc-regression	charArray	string	True	False	True	Null pointer exception
119	jbmc-regression	classtest1		True	True	True	No verifier type
120	jbmc-regression	const1		True	True	True	No verifier type
121	jbmc-regression	constructor1		True	True	True	No verifier type
122	jbmc-regression	enum1		True	True	True	No verifier type
123	jbmc-regression	exceptions1		False	False	False	No verifier type
124	jbmc-regression	exceptions10		False	False	False	No verifier type
125	jbmc-regression	exceptions11		False	False	False	No verifier type
126	jbmc-regression	exceptions12		False	False	False	No verifier type
127	jbmc-regression	exceptions13		False	False	False	No verifier type
128	jbmc-regression	exceptions14		True	True	True	No verifier type
129	jbmc-regression	exceptions15		True	True	True	No verifier type
130	jbmc-regression	exceptions16	int	False	False	False	

131	jbmc-regression	exceptions18		True	True	True	No verifier type
132	jbmc-regression	exceptions2		False	False	False	No verifier type
133	jbmc-regression	exceptions3		False	False	False	No verifier type
134	jbmc-regression	exceptions4		True	True	True	No verifier type
135	jbmc-regression	exceptions5		True	True	True	No verifier type
136	jbmc-regression	exceptions6		False	False	False	No verifier type
137	jbmc-regression	exceptions7		False	False	False	No verifier type
138	jbmc-regression	exceptions8		False	False	False	No verifier type
139	jbmc-regression	exceptions9		True	True	True	No verifier type
140	jbmc-regression	fcmpx_dcmpx1		True	True	True	No verifier type
141	jbmc-regression	iarith1		True	True	True	No verifier type
142	jbmc-regression	iarith2		True	True	True	No verifier type
143	jbmc-regression	if_acmp1		True	True	True	No verifier type
144	jbmc-regression	if_expr1	int	True	True	True	
145	jbmc-regression	if_icmp1	int	True	True	True	
146	jbmc-regression	ifxx1		True	True	True	No verifier type
147	jbmc-regression	instanceof1		True	True	True	No verifier type
148	jbmc-regression	instanceof2		True	False	True	No verifier type
149	jbmc-regression	instanceof3		True	True	True	No verifier type
150	jbmc-regression	instanceof4		True	True	True	No verifier type
151	jbmc-regression	instanceof5		True	True	True	No verifier type
152	jbmc-regression	instanceof6		True	True	True	No verifier type
153	jbmc-regression	instanceof7		True	True	True	No verifier type
154	jbmc-regression	instanceof8		True	True	True	No verifier type
155	jbmc-regression	interface1		False	False	False	No verifier type
156	jbmc-regression	java_append_char	boolean	False	False	True	
157	jbmc-regression	lazyloading4		True	True	True	No verifier type
158	jbmc-regression	list1	int	True	True	Unknown	
159	jbmc-regression	long1		True	True	True	No verifier type
160	jbmc-regression	lookupswitch1	int	True	True	True	
161	jbmc-regression	multinewarray		True	True	True	No verifier type
162	jbmc-regression	overloading1		True	True	True	No verifier type
163	jbmc-regression	package1		True	True	True	No verifier type
164	jbmc-regression	putfield_getfield1		True	True	True	No verifier type
165	jbmc-regression	putstatic_getstatic1		True	True	True	No verifier type
166	jbmc-regression	recursion2		True	True	True	No verifier type
167	jbmc-regression	return1		False	False	False	No verifier type
168	jbmc-regression	return2	int	False	False	False	Multiple verifiers
169	jbmc-regression	store_load1		True	True	True	No verifier type
170	jbmc-regression	swap1		True	True	True	No verifier type
171	jbmc-regression	synchronized		True	False	True	No verifier type
172	jbmc-regression	tableswitch1	int	True	True	True	
173	jbmc-regression	uninitialised1		True	True	True	No verifier type

174	jbmc-regression	virtual1		True	True	True	No verifier type
175	jbmc-regression	virtual2		False	False	False	No verifier type
176	jbmc-regression	virtual4		True	True	True	No verifier type
177	jbmc-regression	virtual_function_unwinding		True	True	True	No verifier type

Table 4.1: Result of jbmc-regression

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jpf-regression	ExDarko_false	int	False	False	Unknown	Multiple verifiers
2	jpf-regression	ExDarko_true	int	True	True	True	Multiple verifiers
3	jpf-regression	ExException_false	int	False	False	False	Null pointer exception
4	jpf-regression	ExException_true	boolean	True	True	True	
5	jpf-regression	ExGenSymExe_false	int	False	False	Unknown	
6	jpf-regression	ExGenSymExe_true	int	True	True	True	
7	jpf-regression	ExLazy_false	int	False	False	Unknown	
8	jpf-regression	ExLazy_true	int	True	True	True	Multiple verifiers
9	jpf-regression	ExMIT_false	int	False	False	Unknown	
10	jpf-regression	ExMIT_true	int	True	True	True	
11	jpf-regression	ExSymExe10_false	int	False	False	True	
12	jpf-regression	ExSymExe10_true	int	True	True	True	
13	jpf-regression	ExSymExe11_false	int	False	False	False	
14	jpf-regression	ExSymExe11_true	int	True	True	True	
15	jpf-regression	ExSymExe12_false	int	False	False	False	
16	jpf-regression	ExSymExe12_true	short	True	True	True	
17	jpf-regression	ExSymExe13_false	int	False	False	False	Multiple verifiers
18	jpf-regression	ExSymExe13_true	int	True	True	True	
19	jpf-regression	ExSymExe14_false	int	False	False	True	
20	jpf-regression	ExSymExe14_true	short	True	True	True	
21	jpf-regression	ExSymExe15_false	int	False	False	False	
22	jpf-regression	ExSymExe15_true	short	True	True	True	
23	jpf-regression	ExSymExe16_false		False	False	False	No verifier type
24	jpf-regression	ExSymExe16_true		True	True	True	No verifier type
25	jpf-regression	ExSymExe17_false		False	False	False	No verifier type
26	jpf-regression	ExSymExe17_true		True	True	True	No verifier type
27	jpf-regression	ExSymExe18_false	int	False	False	False	
28	jpf-regression	ExSymExe18_true		True	True	True	No verifier type
29	jpf-regression	ExSymExe19_false	int	False	False	False	Multiple verifiers
30	jpf-regression	ExSymExe19_true	int	True	True	True	
31	jpf-regression	ExSymExe1_false	int	False	False	False	Multiple verifiers
32	jpf-regression	ExSymExe1_true		True	True	True	No verifier type
33	jpf-regression	ExSymExe20_false	int	False	False	False	Multiple verifiers

34	jpf-regression	ExSymExe20_true		True	True	True	No verifier type
35	jpf-regression	ExSymExe21_false	int	False	False	False	Multiple verifiers
36	jpf-regression	ExSymExe21_true		True	True	True	No verifier type
37	jpf-regression	ExSymExe25_false	int	False	False	False	Multiple verifiers
38	jpf-regression	ExSymExe25_true		True	True	True	No verifier type
39	jpf-regression	ExSymExe26_false	int	False	False	False	
40	jpf-regression	ExSymExe26_true		True	True	True	No verifier type
41	jpf-regression	ExSymExe27_false	int	False	False	False	Multiple verifiers
42	jpf-regression	ExSymExe27_true		True	True	True	No verifier type
43	jpf-regression	ExSymExe28_false	int	False	False	False	Multiple verifiers
44	jpf-regression	ExSymExe28_true		True	True	True	No verifier type
45	jpf-regression	ExSymExe29_false	int	False	False	False	Multiple verifiers
46	jpf-regression	ExSymExe29_true		True	True	True	No verifier type
47	jpf-regression	ExSymExe2_false	int	False	False	True	Multiple verifiers
48	jpf-regression	ExSymExe2_true		True	True	True	No verifier type
49	jpf-regression	ExSymExe3_false	int	False	False	False	Multiple verifiers
50	jpf-regression	ExSymExe3_true		True	True	True	No verifier type
51	jpf-regression	ExSymExe4_false	int	False	False	True	Multiple verifiers
52	jpf-regression	ExSymExe4_true		True	True	True	No verifier type
53	jpf-regression	ExSymExe5_false	int	False	False	False	Multiple verifiers
54	jpf-regression	ExSymExe5_true		True	True	True	No verifier type
55	jpf-regression	ExSymExe6_false	int	False	False	False	Multiple verifiers
56	jpf-regression	ExSymExe6_true		True	True	True	No verifier type
57	jpf-regression	ExSymExe7_false	int	False	False	False	
58	jpf-regression	ExSymExe7_true	int	True	True	True	
59	jpf-regression	ExSymExe8_false	int	False	False	False	Multiple verifiers
60	jpf-regression	ExSymExe8_true		True	True	True	No verifier type
61	jpf-regression	ExSymExe9_false		False	False	False	No verifier type
62	jpf-regression	ExSymExe9_true		True	True	True	No verifier type
63	jpf-regression	ExSymExeArrays_false		False	False	False	No verifier type
64	jpf-regression	ExSymExeArrays_true		True	True	True	No verifier type
65	jpf-regression	ExSymExeBool_false		False	False	False	No verifier type
66	jpf-regression	ExSymExeBool_true		True	True	True	No verifier type
67	jpf-regression	ExSymExeComplexMath_false		False	False	False	No verifier type
68	jpf-regression	ExSymExeComplexMath_true		True	False	True	No verifier type
69	jpf-regression	ExSymExeD2I_false		False	False	False	No verifier type
70	jpf-regression	ExSymExeD2I_true	double	True	True	True	
71	jpf-regression	ExSymExeD2L_false	double	False	False	False	
72	jpf-regression	ExSymExeD2L_true	double	True	True	True	
73	jpf-regression	ExSymExeF2I_false		False	False	False	No verifier type
74	jpf-regression	ExSymExeF2I_true	float	True	True	True	
75	jpf-regression	ExSymExeF2L_false	float	False	False	False	
76	jpf-regression	ExSymExeF2L_true	float	True	True	True	

77	jpf-regression	ExSymExeFNEG_false	float	False	False	False	
78	jpf-regression	ExSymExeFNEG_true	float	True	True	True	
79	jpf-regression	ExSymExeGetStatic_false		False	False	False	No verifier type
80	jpf-regression	ExSymExeGetStatic_true		True	True	True	No verifier type
81	jpf-regression	ExSymExeI2D_false	int	False	False	False	
82	jpf-regression	ExSymExeI2D_true	int	True	True	True	
83	jpf-regression	ExSymExeI2F_false		False	False	False	No verifier type
84	jpf-regression	ExSymExeI2F_true	boolean	True	True	True	
85	jpf-regression	ExSymExeLCMP_false	long	False	Unknown	Unknown	
86	jpf-regression	ExSymExeLCMP_true	int	True	True	True	
87	jpf-regression	ExSymExeLongBytecodes_false	long	False	False	False	
88	jpf-regression	ExSymExeLongBytecodes_true	int	True	True	True	
89	jpf-regression	ExSymExeResearch_false	int	False	False	False	
90	jpf-regression	ExSymExeResearch_true	int	True	True	True	
91	jpf-regression	ExSymExeSimple_false	int	False	False	False	
92	jpf-regression	ExSymExeSimple_true	int	True	True	True	
93	jpf-regression	ExSymExeSuzette_false	int	False	False	False	
94	jpf-regression	ExSymExeSuzette_true	int	True	True	True	
95	jpf-regression	ExSymExeSwitch_false	int	False	False	False	
96	jpf-regression	ExSymExeSwitch_true	int	True	True	True	
97	jpf-regression	ExSymExeTestAssignments_false	int	False	False	False	
98	jpf-regression	ExSymExeTestAssignments_true	int	True	True	True	
99	jpf-regression	ExSymExeTestClassFields_false	int	False	False	Unknown	
100	jpf-regression	ExSymExeTestClassFields_true	int	True	True	True	
101	jpf-regression	ExSymExe_false		False	False	False	No verifier type
102	jpf-regression	ExSymExe_true		True	True	True	No verifier type
103	jpf-regression	TestLazy_false	int	False	False	False	Null pointer exception
104	jpf-regression	TestLazy_true	int	True	False	True	Null pointer exception

Table 4.2: Result of jpf−regression

NO	Test suite	Title name	Type	Correct output	JBMC output	Witness output	Comment
1	jdart-regression	OverapproximationString01	string	False	False	False	Null pointer exception
2	jdart-regression	URLDecoder01	string	True	False	True	Null pointer exception
3	jdart-regression	URLDecoder02	string	False	False	False	Null pointer exception
4	jdart-regression	addition01	int	False	Unknown	Unknown	Execution time out
5	jdart-regression	array-iteration01	int	False	False	False	Multiple verifiers
6	jdart-regression	boundcheck100	int	False	Unknown	Unknown	Execution time out
7	jdart-regression	boundcheck200	int	False	Unknown	Unknown	Execution time out
8	jdart-regression	boundcheck30	int	False	Unknown	Unknown	Execution time out
9	jdart-regression	double2long	double	False	False	False	

10	jdart-regression	float	float	False	False	False	Multiple verifiers
11	jdart-regression	list2	int	True	True	True	
12	jdart-regression	radians	double	False	False	True	
13	jdart-regression	shifting	int	False	False	False	
14	jdart-regression	shifting2	int	False	False	False	
15	jdart-regression	shifting3	int	False	False	False	
16	jdart-regression	startswith	string	True	False	Unknown	Multiple verifiers

Table 4.3: Result of jdart-regression

In the three result tables, there are 8 columns in each table. They are “NO.”, “Test suite”, “Title name”, “Type”, “Correct result”, “JBMC result”, “Witness result” and “Comment”. The first column "NO." is used to record the number, and each table starts counting from 1. It can be seen from the number that the three tables contain 177, 104 and 16 benchmarks respectively. The "Test suite" column shows the name of the test set, indicating which test set the benchmark comes from. The names of the three test sets we verified are "jbmc-regression", "jpf-regression" and "jdart-regression". The "Title name" column is used to display the name of each benchmark program. The "Type" column is used to indicate the type of data generated by the Verifier class in the program. If the Verifier class is not called, the corresponding result should be empty; if the Verifier class is called multiple times, then the corresponding result of the program is the data type of the first call. Among them, due to the selected benchmarks, the result of this column is one of the following seven types: short, int, long, boolean, char, float, double. The following three columns are the recorded results. "Correct result" records the expected correct result extracted from the yml file; "JBMC result" records the result verified by the JBMC tool; "Witness result" column is the result verified by the Witness extension tool. Among them, the result of "True" program verification is successful, "False" indicates that the verification result in the program is a failure, and "Unknown" indicates that the result cannot be obtained due to reasons such as verification timeout.

The last column "Command" is a supplementary description of the verification result,

used to illustrate the special circumstances of the verification. In this experimental evaluation, there are several results that often appear as follows. "No verifier type" means that the Verifier class is not called to generate variables in this program, so Witness does not play a role in this case. We will not do much research on the examples in this case. "Multiple verifiers" indicates that this program calls the Verifier class multiple times to generate variables, so that the program runs normally, and multiple counterexample values are generated in the algorithm to verify. In this case, the "Type" column shows the type of the variable generated when the method is called for the first time. "Execution time out" means that the program execution has timed out. This is because the execution took too long or used too much memory, and hence no decision on whether the verification was successful or not could be made. "Null pointer exception" means that JBMC has detected a null pointer exception. In this case, there is no counterexample. The occurrence of the null pointer exception is mostly related to the String type. In addition, the rest of the results are manually recorded, used to annotate some specific situations.

4.5 Threats to validity

Judging from the results produced, the function and efficiency of our tools are in line with expectations. From a functional point of view, our verification tool can integrate JBMC and Witness counter-example verification two verification methods in a very convenient way, and verify and output a large number of benchmark programs. For most of the benchmark programs, the output of the JBMC verification results and the results of the Witness extension tool are the same as the expected results. Only a small part of the results have problems, which are basically caused by the String type, and this is also the defect of this tool. In addition, there is a shortcoming in the experimental evaluation stage is the lack of verification of a kind of data type, that is, the byte type. This problem is caused by the test set we use, because the benchmarks we use do not include programs that generate byte types in the Verifier class.

Compared with the previous related extension tools, our program has solved many existing problems and made great improvements on this basis. First of all, our tool is much easier to use, because we can directly verify the verification benchmarks provided by SV-COMP without any changes. And we support multi-file verification and front-end operations. Only need to operate in the web browser, you can verify a large number of benchmarks at a time and get the result table, instead of verifying one by one through the command line and manually sorting the results. For verified software types, we have implemented verification of No verifier and multiple verifier type software, and basically verified all data types, including float and double types. In addition, we also tried to verify other test sets such as jpf-regression and jdart-regression, and achieved good results.

Chapter 5 Related work

Java Bounded Model Checking (JBMC) has only gradually become a mature field since 2018, so this relatively novel direction needs to be continuously supplemented and expanded. Compared with JBMC, C Bounded Model Checking (CBMC) is more mature. As the basis of the JBMC algorithm, CBMC has attracted people's attention since 2014 [49]. Therefore, witness validators for C programs have been studied and implemented by many people, and they have shown good results. MetaVal is a witness validator for the C program developed by the team of the University of Munich in Germany. It was submitted to SV-COMP 2020 and showed good results [39]. During the verification process, it confirmed 3,653 violation witnesses and 16,376 correct witnesses. Such excellent results confirmed the effectiveness of MetaVal. In addition, NITWIT, developed by the RWTH Aachen University team in Germany, is another witness verification tool submitted to SV-COMP 2020. Due to its smaller memory occupation, its verification speed is faster than other competitors [50].

For the original JBMC tool, users usually use the command line to run it. In the process of software verification, there are often hundreds of benchmarks that need to be verified, so it is necessary to integrate other tools into JBMC. BenchExec is such a tool, which is used in SV-COMP for reliable benchmarking and resource measurement, and can be easily installed to run experimental comparisons and produce results [51]. For Java programs, we extend the framework by introducing new assertions for specifying attributes. And we also need to implement two necessary files to complete the integration: tool information module and benchmark definition [10].

Last year, someone proposed an idea to realize the witness verification of Java programs, and realized the basic extension [11]. Her thinking and main goal are

similar to the realization of MetaVal and NITWIT: that is, to verify counterexamples in the form of witnesses generated by software verification tools. The extension tool she has implemented verifies the validity of the violations identified by the software verifier for Java programs, and serves as a proof of the feasibility of verifying the witness of Java programs [11]. However, she only implemented a fairly simple algorithm, and there are still many problems with the verification of Java programs. In this thsis, we made some adjustments and optimizations based on the above-mentioned program witness verification method, and verified with SV-COMP benchmarks. The methods and algorithms we have implemented are still relatively rudimentary, but they have solved the shortcomings of the previous algorithms and have effectively verified the software. After our research, the feasibility of the Java program witness verification method has been further proved.

Chapter 6 Conclusion

The main work of this paper is to develop and evaluate an extension tool based on JBMC, which is used to find Java software vulnerabilities. This idea is about the implementation of witness verification, and it can prove the validity of witness verification based on GraphML for the violation witness. The related extended algorithm has been basically implemented. Although it is relatively rough and simple, it has been able to show the possibility of witness verification in Java software verification. In the extension tool, there are still some shortcomings due to the imperfection of the algorithm and the defects of the framework or the benchmark program used. In this dissertation, our main goal is to understand and master the implemented algorithm, and to carry out further work and improvement.

Software verification has become very important today, because software security is related to personal property and information security. The field of software verification has gradually become a hot topic. As the implementation of Bounded Model Checking algorithm, CBMC can effectively verify C programs. On top of this, JBMC develops based on CBMC, which can verify vulnerabilities in Java programs. As a more mature field, CBMC already has many extensions and developments, including witness verification extensions, which can perform very well in SV-COMP. As for JBMC, the extension based on witness verification is still in its infancy, only with the idea and basic implementation.

In the previously completed projects based on witness verification, there are shortcomings caused by various reasons. Our main task is to discover and improve them. The following are the improvements we have completed in this thesis. First, we are able to validate the counterexample with the original program, instead of modifying the benchmarks and Verifier program. Second, we support verification for benchmarks with multiple verifiers and no verifier, and also support almost all kinds

of data types. Furthermore, we verified more verification sets including jpf-regression and jdart-regression, which further proves the effectiveness of our verification tool. Besides, we designed and implemented a simple web application for the extension tool, which supports multiple files inputting and integration of the produced results.

However, the project I implemented is still rough, and there are still some unsolved problems in it. One very important point is that the String type is still not supported by our witness verification tool, which is a big flaw in the implementation process. Secondly, the algorithm should be optimized to deal with larger-scale programs, because for the time being, our algorithm can only deal with benchmarks with a small amount of code. In addition, our tool can be integrated with other automated testing tools, such as BenchExec. Using mature automation tools can complete our verification tasks more efficiently.

Finally, in the software design and thesis writing stage, I often find myself inadequate in various aspects. Thanks to the Internet, I can learn about many fields that have never been involved before, and gain a lot of new knowledge from them. Regarding the field of software verification, it is likely to get faster development in the future, and I will continue to pay attention to it and continue to learn and understand relevant knowledge.

Bibliography

[1] Schildt, H., 2006. *Java The Complete Reference, Seventh Edition*. McGraw-Hill Publishing, pp.6-10.

[2] Krill, P., 2021. *4 reasons to stick with Java -- and 4 reasons to dump it*. [online] InfoWorld. Available at: <https://www.infoworld.com/article/2687995/4-reasons-to-stick-with-java.html> .

[3] Pypl.github.io. PYPL PopularitY of Programming Language index. [online] Available at: <https://pypl.github.io/PYPL.html> .

[4] Mercer, J., Why is Java so popular for developers and programmers?. [online] FRG Consulting. Available at: <https://www.frgconsulting.com/blog/why-is-java-so-popular-developers/> .

[5] Docs.oracle.com. About the Java Technology (The Java™ Tutorials > Getting Started > The Java Technology Phenomenon). [online] Available at: <https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html> .

[6] Education, I., What is the JRE (Java Runtime Environment)?. [online] Ibm.com. Available at: <https://www.ibm.com/cloud/learn/jre> .

[7] Livshits, V. and Lam, M., 2005. Finding security vulnerabilities in java applications with static analysis. *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, 14, p.18.

[8] Docs.oracle.com. 2021. *Java Security Overview*. [online] Available at: <https://docs.oracle.com/javase/9/security/java-security-overview1.htm> .

[9] Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P. and Trtik, M., 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. *Computer Aided Verification - 30th International Conference*, volume 10981 of Lecture Notes in Computer Science, pp.183-190.

[10] Cordeiro, L., Kroening, D. and Schrammel, P., 2018. *Benchmarking of Java Verification Tools at the Software Verification Competition (SV-COMP)*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1809.03739> .

[11] Tan, V., 2020. *Security analyser tool for finding vulnerabilities in Java programs*. University of Manchester.

[12] Bekker, E., 2021. *2020 Data Breaches - The Most Significant Breaches of the Year | IdentityForce®*. [online] We Aren't Just Protecting You From Identity Theft. We Protect Who You Are. Available at: <https://www.identityforce.com/blog/2020-data-breaches> .

[13] ISO. 2021. *ISO/IEC 27005:2018*. [online] Available at: <https://www.iso.org/standard/75281.html> .

[14] Upguard.com. 2021. *What is a Vulnerability? | UpGuard*. [online] Available at: <https://www.upguard.com/blog/vulnerability#causes> .

[15] Handova, D., *What are the different types of security vulnerabilities? | Synopsys*. [online] Software Integrity Blog. Available at: <https://www.synopsys.com/blogs/software-security/types-of-security-vulnerabilities/ > .

[16] En.wikipedia.org. OWASP - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/OWASP> .

[17] Owasp.org. OWASP Top Ten Web Application Security Risks | OWASP. [online] Available at: <https://owasp.org/www-project-top-ten/> .

[18] Cwe.mitre.org. 2021. CWE - CWE-1350: Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses (4.5). [online] Available at: <https://cwe.mitre.org/data/definitions/1350.html> .

[19] Cvedetails.com. n.d. Vulnerability distribution of cve security vulnerabilities by types. [online] Available at: <https://www.cvedetails.com/vulnerabilities-by-types.php> .

[20] Meng, N., Nagy, S., Yao, D., Zhuang, W. and Argoty, G., 2018. Secure coding practices in Java. *Proceedings of the 40th International Conference on Software Engineering*,.

[21] Professionalqa.com. n.d. Software Verification |Professionalqa.com. [online] Available at: <https://www.professionalqa.com/software-verification> .

[22] "IEEE Guide for Software Verification and Validation Plans," in IEEE Std 1059-1993 , vol., no., pp.1-87, 28 April 1994, doi: 10.1109/IEEESTD.1994.121430.

[23] Gibson, Robin. Managing computer projects: avoiding the pitfalls. Prentice-Hall, Inc., 1992.

[24] Software Testing Techniques, B.Beizer, Van Nostrand Reinhold, 1983.

[25] Lu Luo. Software testing techniques: Technology Maturation and Research Strategy. Institute for Software Research International, Carnegie Mellon University, 2001.

[26] Sawant, Abhijit A., Pranit H. Bari, and P. M. Chawan. "Software testing techniques and strategies." International Journal of Engineering Research and Applications (IJERA) 2.3 (2012): 980-986.

[27] Ajay Jangra, Gurbaj Singh, Jasbir Singh and Rajesh Verma,"EXPLORING TESTING STRATEGIES,"International Journal of Information Technology and Knowledge Management, Volume 4, NO.1,January-June 2011.

[28] En.wikipedia.org. n.d. Software verification - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Software_verification> .

[29] Techopedia.com. n.d. What is Static Verification? - Definition from Techopedia. [online] Available at: <https://www.techopedia.com/definition/13696/static-verification> .

[30] D'silva, Vijay, Daniel Kroening, and Georg Weissenbacher. "A survey of automated techniques for formal software verification." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27.7 (2008): 1165-1178.

[31] Clarke, E. and Emerson, E., 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, volume 131 of Lecture Notes in Computer Science, pp.52-71.

[32] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 1579. Berlin, Germany: SpringerVerlag, 1999, pp. 193–207.

[33] Cprover.org. n.d. JBMC – A Bounded Model Checking Tool forVerifying Java Bytecode. [online] Available at: <http://www.cprover.org/jbmc/> .

[34] Cordeiro, L., Kroening, D. and Schrammel, P., 2019. JBMC: Bounded Model Checking for Java Bytecode - (competition contribution). *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics*, volume 11429 of Lecture Notes in Computer Science, pp.219-223.

[35] Sv-comp.sosy-lab.org. 2021. *SV-COMP 2021 - 10th International Competition on Software Verification*. [online] Available at: <https://sv-comp.sosy-lab.org/2021> .

[36] Beyer D. (2019) Automatic Verification of C and Java Programs: SV-COMP 2019. In: Beyer D., Huisman M., Kordon F., Steffen B. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2019. Lecture Notes in Computer Science, vol 11429. Springer, Cham. pp. 133-155.

[37] Beyer D. (2021) Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In: Groote J.F., Larsen K.G. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2021. Lecture Notes in Computer Science, vol 12652. Springer, Cham. Pp.401-422

[38] Svejda J., Berger P., Katoen JP. (2020) Interpretation-Based Violation Witness Validation for C: NITWIT. In: Biere A., Parker D. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2020. Lecture Notes in Computer Science, vol 12078. Springer, Cham. Pp40-57

[39] Beyer, D. and Spiessl, M., 2020. MetaVal: Witness Validation via Verification. *Computer Aided Verification - 32nd International Conference*, volume 12225 of Lecture Notes in Computer Science, pp.165-177.

[40] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. 2015. Witness validation and stepwise testification across software

verifiers. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). Association for Computing Machinery, New York, NY, USA, 721–733.

[41] Graphml.graphdrawing.org. n.d. The GraphML File Format. [online] Available at: <http://graphml.graphdrawing.org/> .

[42] GitHub. n.d. GitHub - powermock/powermock: PowerMock is a Java framework that allows you to unit test code normally regarded as untestable.. [online] Available at: <https://github.com/powermock/powermock> .

[43] GitHub. n.d. Features And Motivations · mockito/mockito Wiki. [online] Available at: <https://github.com/mockito/mockito/wiki/Features-And-Motivations> .

[44] Introduction to PowerMockito | Baeldung. n.d. Introduction to PowerMock. [online] Available at: <https://www.baeldung.com/intro-to-powermock> .

[45] Stefan Bechtold, C., n.d. JUnit 5 User Guide. [online] Junit.org. Available at: <https://junit.org/junit5/docs/current/user-guide/> .

[46] En.wikipedia.org. n.d. HTML - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/HTML> .

[47] Web.archive.org. n.d. Foreword — Flask Documentation (0.10). [online] Available at: <https://web.archive.org/web/20171117015927/http://flask.pocoo.org/docs/0.10/foreword> .

[48] GitHub. n.d. GitHub - pallets/flask: The Python micro framework for building web applications.. [online] Available at: <https://github.com/pallets/flask> .

[49] Kroening D., Tautschnig M. (2014) CBMC – C Bounded Model Checker. In: Ábrahám E., Havelund K. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2014. Lecture Notes in Computer Science, vol 8413. Springer, Berlin, Heidelberg.

[50] Jan Svejda, Philipp Berger, and Joost-Pieter Katoen. Interpretation-based violation witness validation for c: Nitwit. In Armin Biere and David Parker, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 40 to 57, Cham, 2020. Springer International Publishing.

[51] Beyer, D., Löwe, S. & Wendler, P. Reliable benchmarking: requirements and solutions. Int J Softw Tools Technol Transfer 21, 1–29 (2019).

[52] GitHub. n.d. GitHub - vaibhavbsharma/java-ranger: Java Ranger is a path-merging extension of Symbolic PathFinder. [online] Available at: <https://github.com/vaibhavbsharma/java-ranger> .

[53] Sharma, V., Hussein, S., Whalen, M., McCamant, S. and Visser, W., 2020. Java Ranger: statically summarizing regions for efficient symbolic execution of Java. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,.

[54] Ti.arc.nasa.gov. n.d. Java PathFinder. [online] Available at: <https://ti.arc.nasa.gov/tech/rse/vandv/jpf/> .

[55] Luckow K. et al. (2016) JDart: A Dynamic Symbolic Analysis Framework. In: Chechik M., Raskin JF. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2016. Lecture Notes in Computer Science, vol 9636. Springer, Berlin, Heidelberg.