SECURITY ANALYSER TOOL 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

2020

By
Vi Lynn Tan (10479810)
Department of Computer Science

Contents

Abstract **6**

Declaration **7**

Copyright **8**

Acknowledgements **9**

1 Introduction **10**

1.1 Problem Description 11

1.2 Objectives 12

1.3 Contribution 12

1.4 Organization of Dissertation 13

2 Background **14**

2.1 Security Vulnerabilities 14

2.2 Software Testing 16

2.2.1 Software testing methods and techniques 18

2.2.2 Software testing as a method to reduce security vulnerabilities within software and systems 22

2.3 Software Model Checking 24

2.4 Witness Validation 27

3 Proposed Methodology **29**

3.1 System Architecture 29

3.2 Algorithms 31

3.2.1 Python script 31

3.2.2 Validation Harness 37

3.2.3 Verifier 37

3.2.4 Complexity, completeness and soundness of algorithms 44
3.3 Illustrative Examples 44
3.3.1 `int` example 45
3.3.2 `Long` example 47
3.3.3 `char` example 50
3.3.4 `Boolean` example 53

4 Experimental Evaluation 56
4.1 Setup 56
4.1.1 Environment setup 56
4.1.2 Environment versions 59
4.1.3 Running the tests 59
4.2 Objectives 60
4.3 Results and Threat to Validity 61

5 Related Work 66

6 Conclusion 68

Bibliography 70

Word Count: 13171

List of Tables

2.1 Vulnerability density results from 2008 study [AM08] 23

4.1 Experimental evaluation results using SV-COMP benchmark files . . 63

List of Figures

1.1 Java software development process [Ora19] 10
1.2 Java technology enabling platform independence through the use of JVM [Ora19]. 11

2.1 JBMC Architecture [CKS19] 26

3.1 Architecture of the proposed extension 30
3.2 Screenshot of JBMC output of an `int` example 46
3.3 Screenshot of Python script output of an `int` example 46
3.4 Screenshot of the validation harness of a `int` example 47
3.5 Screenshot of JBMC output of an `Long` example 48
3.6 Screenshot of Python script output of an `Long` example 49
3.7 Screenshot of the validation harness of a `Long` example 49
3.8 Screenshot of JBMC output of an `char` example 51
3.9 Screenshot of Python script output of an `char` example 52
3.10 Screenshot of the validation harness of a `char` example 52
3.11 Screenshot of JBMC output of an `Boolean` example 54
3.12 Screenshot of Python script output of an `Boolean` example 54
3.13 Screenshot of the validation harness of a `Boolean` example 55

4.1 Project directory 60

Abstract

SECURITY ANALYSER TOOL FOR FINDING VULNERABILITIES IN JAVA PROGRAMS
Vi Lynn Tan
A dissertation submitted to The University of Manchester
for the degree of Master of Science, 2020

The aim of this thesis is to understand the use of software verification as a technique to identify security vulnerabilities and implement an extension to validate the results produced by software verifiers. Software verifiers are developed to be high performing software verification tools that aim to produce an accurate verification of a program based on given specifications. In this research, specifications refer to a correctness property or a violation property. Software verifiers such as the Java Bounded Model Checking (JBMC) tool uses the concept of bounded model checking to perform verification tasks. While software verifiers are highly accurate in verifying a program, they may produce false alarms where a violation identified is not actually a valid bug. Thus, this pushes for the need to validate the verification results.

This thesis describes a proposal for a methodology to perform validation using witnesses. In specific, we explore the use of witnesses in the format of GraphML. Although research on witness validation is not completely novel, most take place for C programs. In this thesis, the focus is on witness validation for Java programs which is implemented using Python and Mockito, a mocking framework in Java. The algorithms for the proposed extension is described along with the results from experimental evaluation. Limitations within implementation is also described in the evaluation chapter. Ideas on future work and improvement on limitations are described in the last chapter on conclusion.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.library.manchester.ac.uk/about/regulations/) and in The University’s policy on presentation of Theses

Acknowledgements

I want to thank Dr Lucas Cordeiro, my thesis supervisor, for providing guidance and support throughout the process of completing my dissertation. Throughout times of uncertainty, I appreciate you helping me to complete my research with my best effort. I would also like to extend my gratitude to Peter Schrammel, one of the main contributors to the JBMC open source project for providing me with guidance and support throughout the project.

I would also like to thank my friends and family for being constant support throughout my MSc. I could not have done it without all this support.

Chapter 1

Introduction

The Java programming language is a general-purpose programming language developed by Sun Microsystems in 1995 [Kri14]. Since its release, it has been widely adopted across industries in producing small to large applications. In 2014, *Infoworld* reported that Java is used by 90% of Fortune 500 companies [Kri14].

Its swift adoption can be attributed to the architecture of the language, which aims to be platform-independent [Bin15]. This is realised by introducing the Java Virtual Machine (JVM), which acts as a middle-man to compile and interpret Java programs on various platforms.

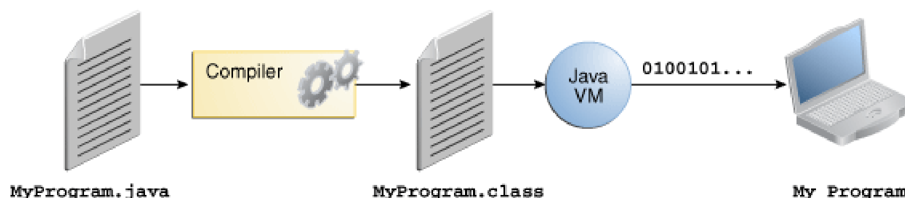


Figure 1.1: Java software development process [Ora19]

As illustrated in Figure 1.1, when a user writes a Java program, this file, known as a source file is saved with a `.java` extension, which enables the compiler to recognise the files to compile. The source file would then need to be compiled using a `javac` compiler, which produces a `.class` file. This `.class` file is in bytecode, an intermediate language [Tec18] consisting of a set of instructions. This is the language used by JVM in order to be able to execute the source code written by the user [Ora19].

Figure 1.2 illustrates the role of JVM and how the introduction of JVM attributes to Java's platform independence ability. JVM is a layer situated between the Java compiler and the operating system. The JVM is tasked to interpret the compiled `.class`

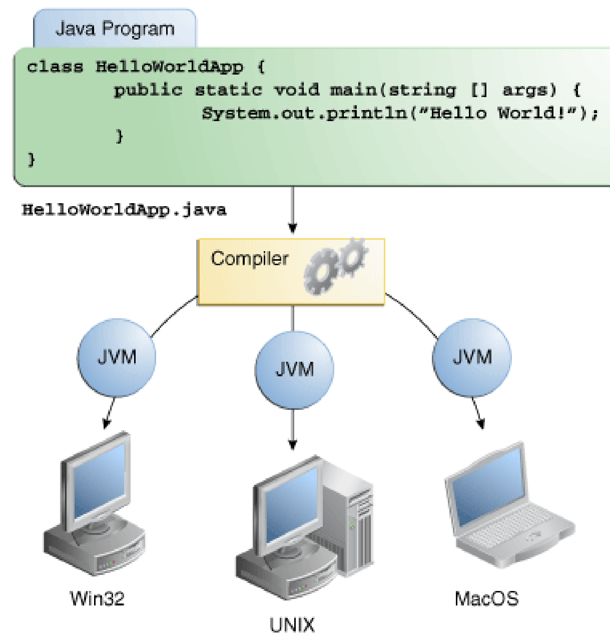


Figure 1.2: Java technology enabling platform independence through the use of JVM [Ora19].

file which contains bytecode into a machine language to be executed at a low-level. Thus, this allows for the same `.class` file to be distributed and used among various platforms with JVM installed [Ora19].

Java Runtime Environment (JRE) employs the use of the just-in-time (JIT) compiler where compilation of Java bytecode into machine readable code is done during runtime. With the JIT compiler, it improves the performance of Java applications as the time needed for compilation before runtime is reduced. JVM and the JIT compiler work hand in hand during runtime. When an application is executed, the JIT compiler will compile the bytecode into machine readable code using some optimisation strategies. These strategies include but are not limited to the use invocation counts whereby most often called methods and classes are compiled first. This leaves JVM the task of merely calling the specific compiled code when that particular method has been invoked by the application [IBM].

1.1 Problem Description

The Java language and platform has various security schemes aimed to thwart security attacks and discover security vulnerabilities at an early stage [Ora17]. However, with

the rapid advance of technology and as noticeable in the news, there is an increasing amount of security issues rising in smart home appliances, or as it is also known as, the internet of things (IoT) [Kel19]. This adds on to the ever increasing amount of security issues that have and still occur within enterprise level servers and our personal computers.

In this dissertation, we focus on using JBMC [CKK+18] as a form of software verification to verify the existence of security vulnerabilities within a Java program. Nonetheless, JBMC may occasionally produce false positives in which the bug may never be achievable during runtime in reality. Thus, there exist a need to verify if the bug found by JBMC is indeed a valid bug in the program.

1.2 Objectives

The overall objective of this dissertation is to understand software model checking as a method to find security vulnerabilities and to implement an extension which performs witness validation based on GraphML witness files produced by Java verifiers such as JBMC.

The specific objectives of this thesis are to:

- Gain a good understanding of the JBMC tool to understand the principles and mechanics used to discover security vulnerabilities.
- Evaluate strategies used in the JBMC in finding security vulnerabilities.
- Research and implement suitable extensions for JBMC in Mockito and Python.
- Evaluate the effectiveness and accuracy of implemented extensions.

1.3 Contribution

This research project produces two main implementations of which is a Python script and a Mockito validation harness. The Python script coordinates the extraction of the counterexample from JBMC, where a counterexample is a concrete value found by JBMC. This concrete value triggers the conditions of a bug and the Python script injects it into the validation harness which is built in Mockito. Mockito is a mocking framework using the Java programming language. In this dissertation, Mockito is used with the aim to simulate the bug found by JBMC by mocking the Java program.

1.4 Organization of Dissertation

The structure of this thesis starts with Chapter 2, a discussion of the relevant background knowledge. That is followed by Chapter 3, the methodology chapter, which goes into detail on the implementation of the extension. Then, in Chapter 4, the evaluation chapter, the methods of testing and key results are displayed and discussed. In Chapter 5, some related research is discussed and compared. Finally, in Chapter 6, the outcome of the thesis is concluded and will be reflected upon.

Chapter 2

Background

The following sections aim to give proper and enough background into the dissertation in order to facilitate understanding to the reader. There are four sections in total. Section 2.1 gives context on security vulnerabilities and how it is a major part of today's world. Section 2.2 describes software testing as a concept and how it is used to reduce the potential bugs and vulnerabilities in various software and systems. Section 2.3 describes the concept of software model checking and dives into JBMC as an instance of a software model checker. Section 2.4 gives an insight into the idea of witness validation as the foundation of the research in this dissertation.

2.1 Security Vulnerabilities

The world we live in today is getting more and more connected. The line between what goes on in the real world and the digital world is getting increasingly blurred. As the number of traditional systems such as banks and services including government services going fully digital, the amount of data in the internet is exponentially growing. In particular, there is a growing concern over the amount of personal data such as credit card numbers being transmitted constantly over the internet. IdentityForce reports that between the period of January and September 2019, there were more than 7.9 billion data records exposed due to data breaches and security attacks [Tur20]. In most cases, the existence of a vulnerability in hardware, software or people within the organisation were the root cause of those data breaches.

A security vulnerability is most commonly described as a weak point within the system, where it could be triggered or taken advantage of or exploited [Han19]. ISO

27000 defines vulnerability as a “weakness or of an asset or control that can be exploited by one or more threats” [27017]. The susceptibility could range from the technical origin such as a software bug or even of human origin such as using weak passwords as they are much easier to recall [Tun20]. Vulnerabilities can be triggered intentionally or unintentionally and can often result in a catastrophe from an organisation’s point of view.

The Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Weaknesses (CWE Top 25) reports a yearly list of frequent issues which resulted in deep consequences over the last two calendar years [CWE20]. The 2020 CWE Top 25 is mainly dominated by specific security weaknesses such as systems able to be taken advantage of using cross-site scripting (XSS). However, there are several class level weaknesses that have persisted throughout the years to still continue affecting various software at the code level. One such common weakness is the CWE-787: Out-of-bounds Write [MiT09] which is defined as the event when the ”software writes data past the end, or before the beginning, of the intended buffer”. The CWE-787 was ranked second on the 2020 CWE Top 25 while its data read counterpart the CWE-125: Out-of-bounds Read is ranked fourth on the same list [CWE20].

The Java language and platform is designed and built with several security features embedded into the Java Development Kit (JDK) and the JVM [Ora17]. For instance, the JDK automatically manages memory usage and garbage collection. This allows potential vulnerabilities to be caught at the development stage by the user or developer. In the JVM bytecode, as described in the previous subsection, is checked for various possible issues that could arise during Java runtime. On top of that, there are features embedded in the Java programming language that can increase the level of security of Java programs. This includes the use of access modifiers, which provide the developer with a means of controlling the level of accessibility [Ora17]. Java programs also employ the use of Application Programming Interfaces (API) and libraries, which aid in providing tools for developers to strengthen the level of security in their Java application. One such API is `java.security` [Ora17]. This library contains various functionalities such as `java.security.MessageDigest.getInstance`, which creates a message digest of the specified choice of an algorithm such as MD5. This message digest can then serve as a cryptographic hash key where it is most commonly used to verify the authenticity of a message [Ora17].

In 2019, MITRE documents through CVE, its security vulnerabilities database, describe that the Java Runtime Environment (JRE) has 17 reported vulnerabilities in

that year alone and 617 in total since 2010. JRE is the most common form of Java technology found on a user's machine since it has JVM bundled into it, which is a vital part of running Java software within a web browser [Ora]. Thus, it is evident that despite the features that Java has by design or through various add-ons, security vulnerabilities are still highly likely to occur.

2.2 Software Testing

Testing is a crucial part of the software development process. It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results [Het88]. Software is comparable to other physical processes where input is received and output is produced. However, while most physical processes can fail in a reasonably small set of ways, software can fail in a multitude of different ways. Detecting and reacting to all the different failures, is not an option; especially towards the end of the software development process where the costs of addressing failures is much higher [San19].

Almost any software, of a moderate to high complexity, will present software defects. This is not due to the carelessness or irresponsibility of programmers, but due to the complexity of software. We, as humans, have a limited ability to manage complexity efficiently. Design defects, due to their nature and stage of creation, can never be completely ruled out from high complexity software.

Boris Beizer compares the difficulty in software testing with using pesticide in an analogy known as the Pesticide Paradox [Bei90]:

"Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual."

However, this alone cannot guarantee to make the software better, because the Complexity Barrier [Bei90] principle states:

"Software complexity (and therefore that of bugs) grows to the limits of our ability to manage that complexity."

Through removing the previous defects, we have increased the complexity and now have multiple, more subtle defects to face. We all, as consumers, want extra functionality, extra features to satisfy various known and/or unknown needs. Thus, the complexity barrier is constantly pushed and raised. How close we can reach it, is entirely determined by the strength of our techniques and processes against more complex and subtle defects.

More recent trends in software development emphasise the value of testing throughout the whole development process, especially on issue prevention. Testing activities must start as early as the requirements specification phase with test a strategy, and radiate through test cases and quality practices along the different development phases. Once the development process is "completed", the testing does not stop; it continues after deployment by logging and analysing the data and any potential failure reported by customers [Het88].

Despite the challenges and limitations, testing, in all forms, is an integral part of software development and maybe consume a large part of the effort required for producing software. In fact, it is not uncommon that more than 50% of the development time is spent in testing [Luo01].

Generally speaking, software testing is performed with multiple purposes in mind. Typically those are:

1. To improve the overall quality

   The usage of computers and software has increased dramatically over the years; both in our private daily life, but also in critical applications where the results of a bug [IST20b] can lead to huge losses. Bugs in critical systems have led to air plane tragedies, led space shuttle missions to fail, blocked trading on the stock market and many other incidents. Bugs can cost lives. The quality and reliability of software is literally a matter of life and death. Broadly, quality refers to the software's level of conformance to the specified design requirement. The minimum requirement of quality is therefore, performing as required under specified circumstances and environments. Debugging can be considered a narrowed down form of software testing and is performed intensively by programmers to find and isolate design defects. As mentioned above, our inability to handle increased complexity in an efficient manner, makes it almost impossible to create a complex program which can function flawlessly from the first time. Finding, isolating and fixing problems early is the purpose of debugging during programming [Bei90].

2. To verify and validate

   Testing can often serve as metrics since testers can extract and analyse the data in order to better understand how the product behaves under certain situations and if this is according to the required specifications. This data can also serve to compare different products under the same specifications, based on results

from the same tests. Quality in itself cannot be tested directly, but factors related to it can be tested to increase awareness and visualise the overall perception of quality [Bei90].

The tests performed with a clear purpose of validating product functionality are generally called clean tests, or ”positive tests”. The disadvantages for these types of tests is that they can only validate that the software works for these specific test cases. A defined number of tests will not able to guarantee that a particular software works in all situations. With only one failed test, however, can be enough to prove that the software does not comply with the required specifications [Bei95].

Negative tests, or “dirty tests”, are tests aimed at breaking a software or proving that it does not work. Software must have be capable of handling exceptions in order to survive a significant level of negative tests [Bei95].

3. To estimate reliability

   The probability of a piece of software to operate failure-free for a specified period of time, in a specified environment, is commonly referred to as software reliability. Software reliability is an important aspect in determining a system's robustness. A major contributing factor which leads to software reliability issues is the high complexity of software itself. Despite the fact that software reliability is described as a probabilistic function, and sometimes associated with the notion of time, it must be noted that, compared to hardware reliability, it is not in a direct relationship with time. Mechanical parts wear out with time and usage, while software will not alter or wear out during its life cycle. Unless intentionally upgraded or changed, software does not change as a result of time passing and/or usage [Bei90].

2.2.1 Software testing methods and techniques

There are plenty of methods and techniques by which to perform testing, each serving a different purpose in different phases during the software development process. The reason for using numerous testing methods and techniques is to ensure that software can successfully operate in multiple environments and across various platforms. Typically, those can be grouped in two categories: functional and non-functional testing [Bei90, Luo01].

Functional testing refers to testing the application against the business requirements and it includes different test techniques designed to guarantee that each component of a piece of software can behave as expected using use cases provided by a stakeholder [Luo01]. Generally, these testing methods are performed in order as follows:

- Unit testing

  Unit testing is often considered the first level of testing and is generally performed by the programmers themselves. This process ensures that individual components of a piece of software, at the code level, are functional and work as designed. In a test-driven environment, programmers typically write and run tests before the software is passed to the test team. Unit testing can be performed manually, but in order to speed up the process and expand the test coverage, it can also be automated. Unit testing also helps making debugging easier since finding issues earlier means the cost of fixing is reduced compared to if they were found late during the testing process [Bei95, Luo01].

- Integration testing

  Integration testing is generally performed once each unit is tested and integrated with the other units to create components or modules which are designed to perform various tasks and activities. In order to test these components and ensure that they function as expected as a whole, and that the interactions between units is friction-less, they are tested through what we name as integration testing. These tests are usually designed to follow user scenarios such as creating an account into an application, accessing different functions, opening files, etc. and generally be performed either by programmers or independent testers. They are generally composed of a combination of automated and manual tests [Luo01, Bei95].

- System testing

  System testing is a testing method performed on a completed system, as a whole, and is designed to evaluate and ensure it meets the specified requirements. The functionality of the whole system is verified from end-to-end and most commonly is performed by a separate testing team before a product is pushed into production [Luo01, Bei95].

- Acceptance testing

Acceptance testing is used to assess whether the final software product is ready for delivery or not. In order to pass this phase, the product must be in compliance with all the original business requirements, as well as meeting the end user's needs. This requires the product to be tested both by the internal test team, as well as the end users through beta testing. Beta testing is a critical step of acceptance testing since it allows the business to get real feedback from potential customers and can help highlight any usability concerns [Luo01, Bei95].

For non-functional tests however, the focus is on the operational aspects of a piece of software and include different testing techniques [Bei90]. These typically are:

- Performance testing

  Performance testing is generally described as a non-functional testing practice performed to determine how a system will behave under various circumstances. Performance testing can help investigate and answer questions regarding the maximum number of users a system could handle, how well the system can protect itself in case of an attack, which is the response time under normal and unfavourable load conditions [Bei90, Het88]. The various types of performance testing usually can be broken down as follows:

  Load testing

  Load testing is the process in which the main purpose is to verify a system's ability to handle normal, high, but also low load conditions. This can confirm that the system would work as intended, as well as in exceptional situations [Bei90].

  Stress testing

  Stress testing is the process in which a system is tested by overloading it beyond its design expectations. The main goal is to stress a system until it breaks through both realistic and unrealistic scenarios. It addresses mainly the components that can fail first in order to help design a more solid system [Bei90].

  Endurance testing

  Endurance testing, also known as Soak Testing, is often used to test and investigate a system's behaviour while running at a high level of load for an extended period of time. The main purpose of it is to verify that the system can handle an expected load over a defined period of time. Though

similar to load and stress testing, it differs in the fact that it is designed to be run for a longer period of time compared to a few hours in the case of the above. Maybe the biggest value of endurance testing comes from its ability to uncover and highlight memory leaks [Bei90].

Spike testing

Spike testing is a testing process in which the main aim is to observe a system's behaviour under a sudden increase of load [Bei90]. For example, a large number of transactions, or server requests, etc.

- Usability testing

  Usability testing is mostly focused on measuring an application's usability from an end-user perspective and is most often executed during the acceptance testing phase. Its main goal is to determine if the design of an application meets the workflow intended for different processes, such as creating an account, logging in, logging out, etc. Generally speaking, usability testing is a great way for development teams to test and review how intuitive different aspects of an application are as well as how functional the whole system is [Bei90, Het88].

- Compatibility testing

  Compatibility testing is a testing method which is used to evaluate the ability of an application to work in different environments and circumstances. In other words, it is used to test that an application is compatible with different operating systems, hardware platforms, browsers, display resolutions and aspect ratios, etc. Its main goal is to prevent any potential incompatibility issues which can result from the multitude of setups end users might use [Bei90, Het88].

- Security testing

  Security Testing is a type of software testing that intends to uncover vulnerabilities of the system and determine that its data and resources are protected from possible intruders [IST20a]. It ensures that a system or application is free from any threats or risks that can cause harm through loss of information, revenue, reputation both towards the employees or outsiders of an organisation.

  There are 7 main types of security testing as described in the Open Source Security Testing Methodology Manual [Her10]:

1. Vulnerability scanning refers to the activity of scanning a system against vulnerability signatures.
2. Security scanning can be both an automated or a manual scanning process and generally focuses on identifying a network or a system's weaknesses for which to later provide solutions to reduce the risks.
3. Penetration testing aims at simulating an attack from a malicious actor. This form of testing involves a thorough analysis of a particular system in order to uncover all the potential vulnerabilities in case of an attempted external hacking attack.
4. Risk assessment refers to the activity of analysing the risks observed in the system. Typically the risks are classified through labels such as: Low, Medium and High. This form of testing generally helps create recommendations of measures to reduce risk.
5. Security auditing can be described as an internal investigation of various applications and systems aimed at uncovering security flaws. Audits can be performed on a larger scale, but also on a line by line inspection of code.
6. Ethical hacking refers to hacking an organisation's software systems with the intent of exposing security flaws in the system.
7. Posture assessment is a combination of security scanning, ethical hacking and risk assessment to visualise an overall security status of an organisation.

Only by building a solid testing framework which contains both functional and non-functional testing methods, can we ensure the creation of high-quality software. Which, in return, can be delivered and successfully adopted by our end users.

2.2.2 Software testing as a method to reduce security vulnerabilities within software and systems

There have been multiple publications [AMR07, AM08, HM04, LTW+06, SW11] which support and are consistent with the thesis, that a large percentage of software with quality defects, also presents or is more likely to present security vulnerabilities as well. Low code quality often leads to unpredictable behaviour. It has been observed that, often, vulnerabilities correlate with defects and result from development errors.

In a study from 2004, Jon Heffley and Pascal Meunier [HM04], reported that 64% of the known vulnerabilities in the National Vulnerability Database (NVD) are attributed to programming errors. And up to half of which were: buffer overflow, cross/site scripting and injection flaws [HM04].

Another study from 2006 found that between 9% and 17% of vulnerabilities were memory related and out of those, $72-84\%$ were semantic rather than syntactic [LTW+06]. The average time between the creation and introduction of a vulnerability and when it has been discovered was between two and three years. Following this study, Li [LTW+06] has specifically recommended examining the code before releasing using tools designed to perform this task.

Robert Martin, while summarising findings from the Common Weakness Enumeration (CWE) in 2014, linked vulnerabilities to common development defects. In his study, he included a series of activities that can help finding and addressing issues early in the software development process [Mar14].

A 2011 study of the Firefox browser reported that up to 21.1% of its files contained faults. Out of which 13% were vulnerable to attacks [SW11]. The study concluded that "prediction models based upon traditional metrics and substitute for specialised vulnerability prediction models." However, despite the correlation, further research would be needed to identify vulnerabilities due to the large number of false positives remaining.

During an investigation from 2007 and 2008, on the relationship between defects and vulnerabilities, Omar Alhazmi analysed and compared various versions of Windows and Linux operating systems [AMR07, AM08]. The results can be found below in Table 2.1:

Systems	KSLOC	Known Defects	Known Defects Density (per KSLOC)	Known Vulnerabilities	VKD (per KSLOC)	VKD/DKD Ratio%	Release date
Windows 95	15	5000	0.3333	50	0.0033	1.00	Aug 1995
Windows 98	18	10000	0.5556	84	0.0047	0.84	Jun 1998
Windows XP	40	106500	2.6625	125	0.0031	0.12	Oct 2001
Windows NT	16	10000	0.625	180	0.0113	1.80	Jul 1996
Win 2000	35	63000	1.80	204	0.0058	0.32	Feb 2000
Apache	0.376	1380	3.670212766	132	0.351064	9.565217	Apr 1995
RH Linux 6.2	17	2096	0.123294118	118	0.00694	5.628817	May 2000
RH Linux 7.1	30	3779	0.125966667	164	0.005467	4.339772	Apr 2001
RH Fedora	76			154	0.00203		Nov 2003

Table 2.1: Vulnerability density results from 2008 study [AM08]

The relationship between software security and quality is sometimes analysed individually though more often than not are two sides of the same coin. An issue that manifests as a system failure now, can be a critical vulnerability exploited by an attacker tomorrow. Both QA and Software security aim at removing risks, whether it be a quality or a security risk.

The results of the studies are often incomplete or ambiguous. However, it can provide a starting point in identifying the correlation between coding and design defects and vulnerabilities which can help identify and address them earlier in the development process. The wide range of defects and vulnerabilities, along with the various uncertainties can often lead to a low ratio of vulnerabilities to defects. However, this is expected, and the association is plausible. The amount of vulnerabilities measured is consistent with the vulnerability defect ratio (between 0.3% and 10%). The lower percentage of the ratio represents systems with higher security testing coverage and more refined development practices [AM08].

2.3 Software Model Checking

With the increasing threat of software vulnerabilities, formal software verification and model checking is gaining more prominence [DKW08]. In this section, we will explore concepts on model checking and bounded model checking as methods for formal verification of software.

In 1981, Clarke and Emerson introduced the concept of model checking. The core idea of model checking is ascertain if a correctness property holds by checking exhaustively within all reachable states of a program. In the event, where the property does not hold, the model checking algorithm would return a counterexample. A counterexample is the execution trace which lead to a state whereby the property has been violated [CE81].

Within the concept of model checking, a model of a software or program is made up of states, transitions and a property or specification. A state is the condition of the program at a specific time, often using the value of a program counter. Transitions describe the event when the program progresses from one state to another state. A property or often known as a safety property within model checking is the specific properties in which the program is checked for to determine correctness. An alternative method to understanding a safety property is such that if the safety property is unreachable then the program is deemed to have been successfully verified. The state

in which a safety property is reachable is also known as a bad state. A model checking algorithm would have to extensively pore through all reachable states of a program. If the state space such that there are a finite number of states possibly reachable, this process of extensive checking is guaranteed to terminate [DKW08].

Building upon the concept of model checking is bounded model checking (BMC). This technique conducts a depth-bounded exhaustive search of the state space. Formally, BMC can be described using the Definition 2.3.1 [CKK+18].

Definition 2.3.1 *Given a transition system M, a property ϕ, and a bound k; BMC unrolls the system k times and translates it into a verification condition (VC) ψ, which is satisfiable if and only if ϕ has a counterexample of depth less than or equal to k.*

JBMC is one of many open-source verifiers based on BMC and is the core of this dissertation. It was developed on top of the CProver framework as a Java version to its predecessor, the C Bounded Model Checker (CBMC). JBMC utilizes three significant concepts in its compositions; Bounded Model Checking (BMC) as indicated in its name, as well as Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT). The development of JBMC desired to address the issue of discovering not obvious bugs in a Java program by verifying its Java bytecode. The process of verification using JBMC can demonstrate program correctness, whereby JBMC can state that the program is doing what it is supposed to do without producing any unintentional results [CKK+18].

The core of JBMC is BMC, where a program is unrolled and checked whether a particular state of the program can be achieved within a fixed number of steps, often known as the upper bound and denoted as k. In JBMC, we are interested in finding out if a bad state, which signifies a vulnerability, is realisable during runtime within the known upper bound, k [CKK+18].

JBMC can be executed on the command line interface (CLI) using the command below.

```
jbmc <filename> <additional properties (optional)>
```

JBMC only accepts class files, which has the *.class* file extension and Java archive (JAR) files, which has the *.jar* file extension. It can also allow additional properties to be further specified, e.g., if a specific upper bound k is to be determined. Below is an example, where k is determined to be ten on a file example.class.

```
jbmc example.class {unwind 10
```

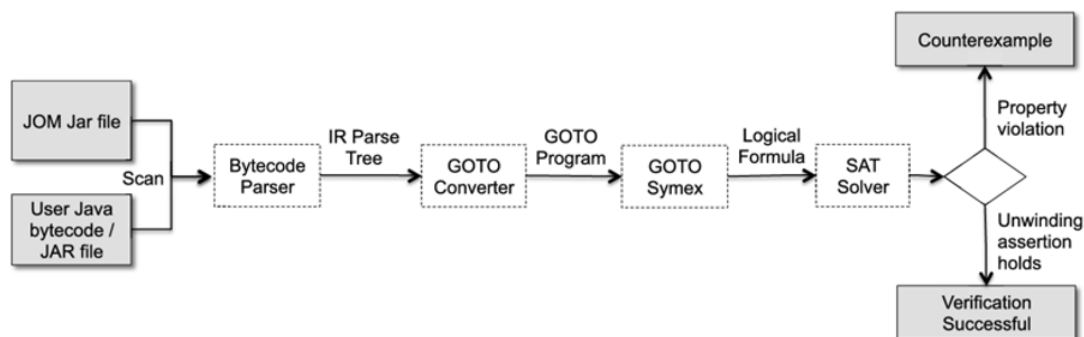


Figure 2.1: JBMC Architecture [CKS19]

Figure 2.1 illustrates the JBMC architecture, which consists of three main sections. The input is represented by the grey rectangles on the left-hand side of the diagram. The white rectangles represent the verification steps in the middle of the diagram. The output is represented by the grey rectangles on the right-hand side of the diagram.

JBMC accepts inputs in the form of a JAR archive file or a Java bytecode class file. Then, JBMC will parse the input file using the Bytecode Parser as shown on the first white box as part of the verification step to produce and feed parse trees into the GOTO converter, which will translate the parse trees into a GOTO program. A GOTO program is a representation of a control flow graph (CFG) for the CProver application, which, as mentioned above, is the underlying framework in which JBMC is built. Now that the file is in the correct format for verification, the GOTO program is passed to the GOTO symex, which is responsible for unwinding loops and unfolding recursive function calls according to the defined upper bound, k. The product of that process is subsequently passed on to the SAT or SMT solver, which will determine if any bugs are found. As a result, it brings the verification lifecycle to an end with an output. If a bug in the program is found, then JBMC will output “VERIFICATION FAILED” with a counterexample, which shows the exact properties in which the bad state is reachable within k steps. However, if JBMC outputs “VERIFICATION SUCCESSFUL”, that signifies that JBMC did not find any vulnerabilities at all, up to the upper bound k [CKK+18].

JBMC has proved to be a valuable tool in the software verification research area. It has participated in the Java track and won the gold medal in the SV-COMP ’19, which is a software verification competition. In SV-COMP, participants send in their submission of software verifiers, which are to be bench marked against many different test cases. The verification results for the test cases as well as the time required to output a result is used as the benchmark in producing the final ranking. JBMC has

done well so far as demonstrated in SV-COMP '19. However, there are still some areas, which can be further enhanced. These include but is not limited to improving ease of verification of programs that require a large upper bound *k*, support of verification of classes or libraries in Java and increasing efficiency for multi-threaded programs [CKK+18].

2.4 Witness Validation

The concept of witness validation is fairly new if compared to software verification and software model checking [BS20]. The first two validators were submitted to SV-COMP in 2015 [BDD+15] and in 2020, there were six validators submitted in total, including the two first submitted in 2015 [oSVSC20]. The motivation behind witness validation lies in the problem that software verifiers sometimes produce false alarms [BS20]. On top of that, the counterexamples produced by software verifiers such as JBMC are often in a format which is not very accessible for further manipulation or verification using different verifiers [BDD+15].

The objectives of witness validation is to improve the trust level of verification results produced by software verifiers and to employ the use of an exchangeable format to represent the witness data [BS20].

A witness file may take one of two forms: a violation witness or a correctness witness. A violation witness contains witness data that describes error paths or violations identified by the software verifier along with the counterexample. On the other hand, a correctness witness contains witness data that describes a correctness proof such that the software verifier has found no possible event in which a violation could take place. With a common witness format in place, this will allow verification results to be validated independently and open up the possibility to take advantage of combining various verification tools. [BS20]

One of the proposed format to be used as the exchangeable format for witness validation is GraphML [Tea]. GraphML is based on XML and was designed initially as a format to represent and store graph structures, hence the name of the format [Tea]. One of the main reasons for the designation of GraphML as the chosen exchangeable file format was the relative ease of use in terms of reading from and writing to GraphML files. This can be attributed to the large number of libraries which support these activities for XML based files. GraphML is extensible by nature which allows custom data to be defined and stored. This allows specific witness information such as error paths

to be represented and stored. Therefore, with the adoption of an exchangeable format such as GraphML by verifiers, witness validation can be done in a considerably more straightforward fashion [BS20].

Chapter 3

Proposed Methodology

This chapter describes the proposed methodology to implement an extension on top of a Java verifier to validate witnesses for Java programs using Python and Mockito. In particular, we describe the system architecture as a whole in the first section, Section 3.1. Then, in Section 3.2, we proceed to dive deeper into the algorithms implemented in the extension, which also covers an evaluation on the complexity, completeness and soundness of the implemented algorithms. In the last section of this chapter, Section 3.3, we provide some illustrative examples to show how implemented extension works.

3.1 System Architecture

Figure 3.1 illustrates the overall architecture and flow of events. The user will provide a Java program in the form of a `.java` file or a `.class` file via a command-line interface (CLI). The Python script will coordinate the whole process and output the results of whether the bug is valid to the user. In the following paragraphs, we describe the steps as illustrated in Figure 3.1.

The first step will be for the user to supply the intended Java program to be tested to the Python script. This step can be either a `.java` file or a `.class` file. However, if the user chooses to supply a `.class` file, the primitive data type of the variable in which the bounded model checking is applied to will need to be provided. This is due to the need to specify the primitive data type of the specific `Verifier` method that will be invoked by the validation harness built using Mockito.

Next, the script will need to ensure that the Java program is in the required `.class` form as this is a requirement of JBMC. If the user has supplied a `.java` file, it will be compiled to produce a `.class` file as needed. The script will then run JBMC using

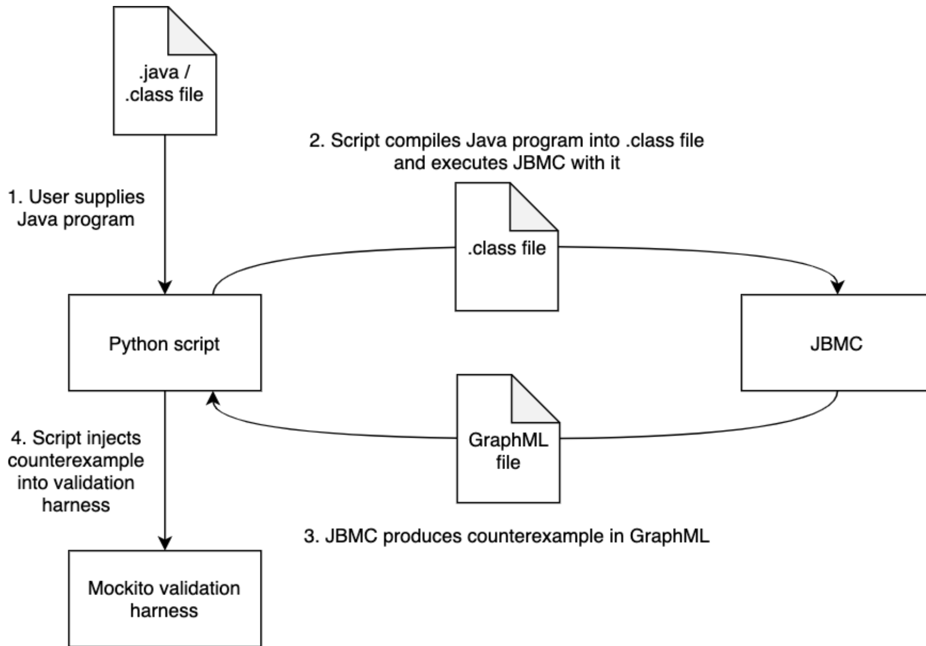


Figure 3.1: Architecture of the proposed extension

system commands, which will run and execute JBMC via the CLI.

Now, JBMC will produce the output of the execution in GraphML [Tea]. GraphML is an XML based file format designed for representation of graphs as the name may indicate. In this dissertation, we take advantage of this file format as it is a feature of JBMC, where it can output the results in this format but also to allow easier manipulation of "application-specific attribute data" [Tea]. The use of GraphML as a format to represent witness data is also commonly used by other Java verifiers in SV-COMP[oSVSC20]. The output will indicate if there exists a presence of an assertion violation within the program and consequently be verified by the Python script [Tan20].

If there exists an assertion violation, then the Python script will obtain the counterexample from the GraphML file and inject it into the Mockito validation harness [Moc]. The validation harness is created from a template whereby actual values and keywords will replace the placeholder values. As the validation harness is built in Mockito, which uses Java, the validation harness file will be saved as a `.java` file. Now in the final step, the script will run and execute the validation harness, which will produce the output on the CLI of the occurrence of an assertion violation. The user is then able to view the details of the assertion violation if it occurs and verify if it is indeed a valid bug.

The extension is focused on verifying whether a bug discovered by JBMC is indeed

a valid bug. Therefore, the emphasis is to perform validation when JBMC produces a "VERIFICATION FAILED" output. The following sections and subsections go more into detail of the algorithms used to perform the steps described above.

3.2 Algorithms

This section describes the different algorithms implemented in this dissertation. Subsection 3.2.1 go into detail of the inner workings of the Python script implemented while subsection 3.2.1.6 describe in detail the template of the validation harness built using Mockito. Subsection 3.2.3 discusses about the modifications done to the `Verifier` class which is used to obtain non-deterministic values and lastly subsection 3.2.4 discusses on the complexity, completeness and soundness of the algorithms implemented as described in subsection 3.2.1 and subsection 3.2.2.

3.2.1 Python script

The Python script can be executed on the CLI with either one of the commands below:

Case One:

```
python3 script.py File.java
```

Case Two:

```
python3 script.py File.class int
```

In both cases, `python3` indicates to the CLI that it is running a Python script and in this case using Python 3. Next to the `python` keyword, separated with a single empty space, `script.py` is indicated as the Python script to be run. For the third argument, the user has the option to provide a `.java` file or provide a `.class` file. If the user has chosen to supply a `.class` file, then the primitive data type of the variable in which the bounded model checking is applied to must be supplied as a fourth argument.

The following subsubsections 3.2.1.1 to 3.2.1.7 detail the content of the Python script, in order of execution, when executed on the CLI.

3.2.1.1 Java class name extraction

```
1 classnameArray = sys.argv[1].split('.')
2 classname = classnameArray[0]
3 if len(classnameArray) == 2 and classnameArray[1] == '
    java':
4         subprocess.Popen(['javac', sys.argv[1]]).wait()
```

Listing 3.1: Excerpt of script to extract class name of the Java program

The code fragment in Listing 3.1 obtains the program's class name as it is a requirement of JBMC. This code section will be able to handle both `.java` files and `.class` files as described previously in 3.2.1.

For instance, the user has supplied a file, `File.java` to the script. As seen in Line 1, the script accesses the passed arguments using `sys.argv[1]`. Then, it splits the string by searching for '.' or the period character, which is done using `.split('.')`. Thus, now in `classnameArray`, which is a string array, would contain two elements: 'File' and 'java'. In Line 2, we are capturing the program class name, which is in the needed format by JBMC, whereby it will be 'File' in this case. In Line 3 and 4, the script checks if it was a `.java` file and will then compile it automatically on the CLI to obtain a `.class` file.

3.2.1.2 JBMC execution

```
1 cmd = 'jbmc '+ classname +' --stop-on-fail --graphml-
    witness  witness'
2 try:
3     result = subprocess.check_output(cmd, shell=True)
4 except subprocess.CalledProcessError as e:
5     result = e.output
```

Listing 3.2: Excerpt of script to execute JBMC

The code fragment in Listing 3.2 executes JBMC with the `classname` obtained in subsubsection 3.2.1.1. It saves the output in a GraphML file by supplying additional arguments of `--graphml-witness` and `witness` whereby the latter is the filename to save the output to. Without supplying the additional arguments, JBMC will output the results on the CLI by default. This will make data access and manipulation easy in the upcoming parts of the script.

3.2.1.3 Violation check

```
1 witnessFile = nx.read_graphml("witness")
2 violation = False
3 for violationKey in witnessFile.nodes(data=True):
4     if 'isViolationNode' in violationKey[1]:
5         violation = True
6
7 if (violation):
8     .
9     .  ## Refer to subsubsections
10    .  ## 3.2.1.4 to 3.2.1.7
11    .
12 else:
13     print ('No violation found')
14     exit(1)
```

Listing 3.3: Excerpt of script to check for violation

The code fragment in Listing 3.3 will open the GraphML witness file obtained from subsection 3.2.1.2. As top-level data are represented as nodes in GraphML, the script will go through each node in search for the violation key as seen in Lines 3 to 5. If a violation key exists, it signifies that a violation has indeed occurred. However, if no violation occurred, the violation key will not be found in the witness file. Thus, we have to assign a default false value for the violation flag as seen in Line 2 of the code fragment in Listing 3.3. This violation flag will be reassigned to the Boolean value, `True` if the violation key is found as seen in Line 5.

In Line 7, we check the Boolean value of violation as obtained in Lines 1 to 5. If there exists a violation such that it carries a Boolean value of `True`, the script excerpts in subsections 3.2.1.4 to 3.2.1.7 will be executed. The details of these subsections will be described in its respective subsection. However, if there exists no violation found such that it carries a Boolean value of `False`, the script will inform the user of the findings and terminate successfully as seen in Lines 13 and 14.

3.2.1.4 Verifier data type determination

```
1 if (len(sys.argv) == 3):
2     type = sys.argv[2].lower()
```

```
3 else:
4     with open(sys.argv[1], "rt") as fin:
5         for line in fin:
6             index = line.find('verifier.')
7             if(index != -1):
8                 type = line[index + 15 : -4].lower().
    replace(')','').replace('(','')
```

Listing 3.4: Excerpt of script to determine the verifier data type

The code fragment in Listing 3.4 determines the verifier data type. This is to facilitate an accurate creation of the validation harness from its template. This is described further in subsection 3.2.1.6. In this part of the script, Lines 1 and 2 obtain the verifier data type from the command arguments if the user has supplied a `.class` file as mentioned in 3.2.1. With the `.lower()` function, we ensure uniformity. This will prevent errors in the code execution due to differences in alphabetical capitalisation. If the user has supplied a `.java` file, the script will go into the `else` statement block as illustrated in Lines 4 to 8. Here, it will pore through the `.java` file in search for the `"verifier."` phrase as the text that comes after this phrase is the desired result. In Line 8, some text manipulation is performed to ensure only the required portion of the text is obtained as it may contain other characters such as `;`, `(` or `)`.

3.2.1.5 Counterexample extraction

```
1 for data in witnessFile.edges(data=True):
2     if 'assumption' in data[2]:
3         str = data[2]['assumption']
4         if (str.startswith('anonlocal')):
5             counterexample = str.split(' = ')[1][:-1]
```

Listing 3.5: Excerpt of script to extract the counterexample produced by JBMC

The code fragment in Listing 3.5 will access the same GraphML witness file as in subsection 3.2.1.3. This portion of the script will obtain the counterexample from the witness file. The counterexample is usually in the form of `anonlocal::1i = 60;` or similar. In this example, `60` is the counterexample value and therefore the script will perform some string manipulation to only obtain the desired part of the whole data string.

3.2.1.6 Validation harness creation from template

```
1 with open("ValidationHarnessTemplate.txt", "rt") as fin:
2     with open("ValidationHarness.java", "wt") as fout:
3         for line in fin:
4             line = line.replace('ClassName', classname)
5             if(type == 'int'):
6                 line = line.replace('Type', 'nondetInt').
   replace('Counterexample', counterexample)
7             if(type == 'short'):
8                 line = line.replace('Type', 'nondetShort'
   ).replace('Counterexample', counterexample)
9             if(type == 'long'):
10                line = line.replace('Type', 'nondetLong')
   .replace('Counterexample', counterexample)
11            if(type == 'float'):
12                line = line.replace('Type', 'nondetFloat'
   ).replace('Counterexample', counterexample)
13            if(type == 'double'):
14                line = line.replace('Type', 'nondetDouble
   ').replace('Counterexample', counterexample)
15            if(type == 'string'):
16                try:
17                    counterexample = int(counterexample)
18                    line = line.replace('Type', '
   nondetString').replace('Counterexample', 'null')
19                except ValueError:
20                    line = line.replace('Type', '
   nondetString').replace('Counterexample', '"' +
   counterexample + '"')
21            if(type == 'char'):
22                line = line.replace('Type', 'nondetChar')
   .replace('Counterexample', '\'' + chr(int(
   counterexample)) + '\'')
23            if(type == 'boolean'):
24                if(counterexample == '1'):
```

```
25                     line = line.replace('Type', '
    nondetBoolean').replace('Counterexample', 'true')
26                 if(counterexample == '0'):
27                     line = line.replace('Type', '
    nondetBoolean').replace('Counterexample', 'false')
28             fout.write(line)
```

Listing 3.6: Excerpt of script to create the validation harness from the template

The code fragment in Listing 3.6 will create the validation harness from the template based on the verifier data type obtained in 3.2.1.4. There exist two placeholders to be replaced when creating the validation harness from the template: `Classname` and `Type`. `Classname` refers to the `classname` of the Java program which was the first to be obtained as described in subsection 3.2.1.1. However, `Type` refers to the verifier data type, which was described in subsection 3.2.1.4. In the majority of the cases, the replacement of the placeholders with the actual counterexample value is straightforward. Some primitive data types such as `string`, `char` and `Boolean` required a bit of manipulation. This is to ensure that the counterexample is represented in its proper form to be run and executed in Mockito.

3.2.1.7 Validation harness compilation and execution

```
1 subprocess.Popen(['javac', 'ValidationHarness.java']).
    wait()
2
3 subprocess.Popen(['java', '-ea', 'ValidationHarness']).
    wait()
```

Listing 3.7: Excerpt of script to compile and execute the validation harness

The code fragment in Listing 3.7 compiles and executes the validation harness after its creation from the template is completed in subsection 3.2.1.6. In the code fragment in Listing 3.7, Line 1 will compile the validation harness with a Java compiler and Line 2 will run the compiled validation harness. Once the program has completed executing, it will output the result on the CLI as a standard text output.

3.2.2 Validation Harness

```
1 import static org.mockito.Mockito.*;
2 import org.sosy_lab.sv_benchmarks.Verifier;
3
4 public class ValidationHarness {
5   public static void main(String[] args) {
6     Verifier verifier = mock(Verifier.class);
7     ClassName.verifier = verifier;
8     when(ClassName.verifier.Type()).thenReturn(
      Counterexample);
9
10    ClassName.main(new String[0]);
11   }
12 }
```

Listing 3.8: Validation harness template built with Mockito

The code fragment in Listing 3.8 is the template of the validation harness built using the Mockito framework. This is written in Java as it is the language used by Mockito. Here, the main goal is to intercept `Verifier` from providing a non-deterministic value and inject it with the counterexample found by JBMC. This will simulate the program with the counterexample, which is essentially a static value to verify whether the bug found by JBMC is valid.

As seen in Listing 3.8, Line 6 mocks the Java class. This will allow the counterexample value to be injected each time the `Verifier` class is invoked, which is done in Line 8. Then finally, in Line 10, the validation harness will call the Java program, which triggers the counterexample value injection and thus, a result will be outputted.

3.2.3 Verifier

```
1 /*
2  * Contributed by Peter Schrammel
3  *
4  * Licensed under the Apache License, Version 2.0 (the "
     License");
5  * you may not use this file except in compliance with
     the License.
```

```
6  * You may obtain a copy of the License at
7  *
8  *          http://www.apache.org/licenses/LICENSE-2.0
9  *
10 * Unless required by applicable law or agreed to in
     writing, software
11 * distributed under the License is distributed on an "AS
      IS" BASIS,
12 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
     express or implied.
13 * See the License for the specific language governing
     permissions and
14 * limitations under the License.
15 */
16
17 package org.sosy_lab.sv_benchmarks;
18
19 import java.util.Random;
20
21 public final class Verifier
22 {
23   public static void assume(boolean condition)
24   {
25     if(!condition) {
26       Runtime.getRuntime().halt(1);
27     }
28   }
29
30   public static boolean nondetBoolean()
31   {
32     return new Random().nextBoolean();
33   }
34
35   public static byte nondetByte()
36   {
```

```
37     return (byte)(new Random().nextInt());
38   }
39
40   public static char nondetChar()
41   {
42     return (char)(new Random().nextInt());
43   }
44
45   public static short nondetShort()
46   {
47     return (short)(new Random().nextInt());
48   }
49
50   public static int nondetInt()
51   {
52     return new Random().nextInt();
53   }
54
55   public static long nondetLong()
56   {
57     return new Random().nextLong();
58   }
59
60   public static float nondetFloat()
61   {
62     return new Random().nextFloat();
63   }
64
65   public static double nondetDouble()
66   {
67     return new Random().nextDouble();
68   }
69
70   public static String nondetString()
71   {
```

```
72     Random random = new Random();
73     int size = random.nextInt();
74     assume(size >= 0);
75     byte[] bytes = new byte[size];
76     random.nextBytes(bytes);
77     return new String(bytes);
78   }
79 }
```

Listing 3.9: Modified `Verifier.java`

```
1 /*
2  * Contributed by Peter Schrammel
3  *
4  * Licensed under the Apache License, Version 2.0 (the "
     License");
5  * you may not use this file except in compliance with
     the License.
6  * You may obtain a copy of the License at
7  *
8  *         http://www.apache.org/licenses/LICENSE-2.0
9  *
10 * Unless required by applicable law or agreed to in
     writing, software
11 * distributed under the License is distributed on an "AS
      IS" BASIS,
12 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
     express or implied.
13 * See the License for the specific language governing
     permissions and
14 * limitations under the License.
15 */
16
17 package org.sosy_lab.sv_benchmarks;
18
19 import java.util.Random;
20
```

```
21 public class Verifier
22 {
23   public void assume(boolean condition)
24   {
25     if(!condition) {
26       Runtime.getRuntime().halt(1);
27     }
28   }
29
30   public boolean nondetBoolean()
31   {
32     return new Random().nextBoolean();
33   }
34
35   public byte nondetByte()
36   {
37     return (byte)(new Random().nextInt());
38   }
39
40   public char nondetChar()
41   {
42     return (char)(new Random().nextInt());
43   }
44
45   public short nondetShort()
46   {
47     return (short)(new Random().nextInt());
48   }
49
50   public int nondetInt()
51   {
52     return new Random().nextInt();
53   }
54
55   public long nondetLong()
```

```
56    {
57      return new Random().nextLong();
58    }
59
60    public float nondetFloat()
61    {
62      return new Random().nextFloat();
63    }
64
65    public double nondetDouble()
66    {
67      return new Random().nextDouble();
68    }
69
70    public String nondetString()
71    {
72      Random random = new Random();
73      int size = random.nextInt();
74      assume(size >= 0);
75      byte[] bytes = new byte[size];
76      random.nextBytes(bytes);
77      return new String(bytes);
78    }
79  }
```

Listing 3.10: Modified `Verifier.java`

The code fragment in Listing 3.10 shows the modified `Verifier.java` file. The original file as shown in Listing 3.9 obtained as part of the SV-COMP benchmarks repository on Github. The purpose of this file is to provide a non-deterministic value upon request to the caller. The proposed extension involves the use of Mockito as a mocking framework. However, Mockito does not support the mocking of final or static classes and methods. Thus, there is a need for changes to be applied as shown in 3.10.

This change will need to be propagated to the test cases or benchmark files as well. Listing 3.11 shows the `assert2.java` benchmark which have been obtained directly from SV-COMP's benchmarks repository on Github. Listing 3.12 shows the `assert2.java` benchmark file after the required changes have been applied and is

ready for testing.

```
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/assert2
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 >= 1000)
16       assert i > 1000 : "i is greater 1000"; // should
   fail
17   }
18 }
```

Listing 3.11: `assert2.java` before applied changes

```
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/assert2
7  * The benchmark was taken from the repo: 24 January 2018
8  */
9 import org.sosy_lab.sv_benchmarks.Verifier;
10
11 public class assert2 {
```

```
12   public static Verifier verifier = new Verifier();
13
14   public static void main(String[] args) {
15       int i = verifier.nondetInt();
16
17       if (i >= 1000)
18         assert i > 1000 : "i is greater 1000"; // should
     fail
19   }
20 }
```

Listing 3.12: `assert2.java` after applied changes

3.2.4 Complexity, completeness and soundness of algorithms

The complexity of the algorithms is linear, represented as $O(N)$. Referring to subsection 3.2.1, it can be observed that there are multiple single `for` loops such as in subsection 3.2.1.3 and subsection 3.2.1.4. However, there exists no occurrence of any nested `for` loops or recursive functions, which may affect the complexity differently. Thus, this makes the algorithms linear in terms of complexity.

Based on the results obtained from the set of benchmarks by SV-COMP, the algorithms are sound as correct results are consistently produced. This can be manually verified given that the Java programs within the benchmark sets are small in size and not too complicated. Since the Java programs are instantiated using Mockito, there is little interest in the completeness of the algorithms.

3.3 Illustrative Examples

The following subsections outline several illustrative examples. These examples are distinguished by the different `Verifier` data type to illustrate the workings of the script on various primitive data types. All the following Java program examples are retrieved from SV-COMP's benchmarks, which are open source and can be freely obtained from `https://github.com/sosy-lab/sv-benchmarks`.

3.3.1 `int` example

```
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/assert2
7  * The benchmark was taken from the repo: 24 January 2018
8  */
9 import org.sosy_lab.sv_benchmarks.Verifier;
10
11 public class assert2 {
12   public static Verifier verifier = new Verifier();
13
14   public static void main(String[] args) {
15     int i = verifier.nondetInt();
16
17     if (i >= 1000)
18       assert i > 1000 : "i is greater 1000"; // should
     fail
19   }
20 }
```

Listing 3.13: Illustrative example of `int` Java program

Listing 3.13 is a Java program, which contains a bug. In Line 18, the program asserts that the `int` value of `i` must be more than 1000. However, it is immediately noticeable that in Line 17, the *if-statement* checks for values of `i` of more than or equal to 1000. Thus, the `assert` statement will fail as expected.

First, we obtain the initial results from JBMC by running the following command on the CLI.

```
jbmc assert2 --stop-on-fail
```

In Figure 3.2, it can be observed that the `assert2` program contains a bug as it has output “VERIFICATION FAILED”. It should also be pointed out that the counterexample is noticeable in this screenshot. At the top of the figure, it reads the following:

```
common — -bash — 101×35
State 245 file assert2.java function assert2.main(java.lang.String[]) line 15 thread 0
----------------------------------------------------
  anonlocal::1i=1000 (00000000 00000000 00000011 11101000)

State 254 file assert2.java function assert2.main(java.lang.String[]) line 18 thread 0
----------------------------------------------------
  dynamic_object15={ .@class_identifier="java::java.lang.String" } ({ ? })

State 255 file assert2.java function assert2.main(java.lang.String[]) line 18 thread 0
----------------------------------------------------
  dynamic_object15.@class_identifier="java::java.lang.String" (?)

State 257 file assert2.java function assert2.main(java.lang.String[]) line 18 thread 0
----------------------------------------------------
  dynamic_object15.@class_identifier="java::java.lang.AssertionError" (?)

State 260 file assert2.java function assert2.main(java.lang.String[]) line 18 thread 0
----------------------------------------------------
  this=&dynamic_object15 (00000000 00010010 00000000 00000000 00000000 00000000 00000000 00000000)

State 261 file assert2.java function assert2.main(java.lang.String[]) line 18 thread 0
----------------------------------------------------
  stub_ignored_arg1=&i_20is_20greater_201000.@java.lang.Object (00000000 00010011 00000000 00000000 0
0000000 00000000 00000000 00000000)

Violated property:
  file assert2.java function assert2.main(java.lang.String[]) line 18 thread 0
  assertion at file assert2.java line 18 function java::assert2.main:([Ljava/lang/String;)V bytecode-
index 15
  false


VERIFICATION FAILED
Vis-MacBook-Pro:common vilynn$
```

Figure 3.2: Screenshot of JBMC output of an `int` example

```
State 245 file assert2.java function assert2.main(java.lang.String[])
line 15 thread 0
----------------------------------------------------
  anonlocal::1i=1000 (00000000 00000000 00000011 11101000)
```

This is, in fact, the counterexample, or rather the value of 1000 is the counterexample, which will be extracted by the Python script. Now we will verify this with the Python script and validation harness built into Mockito.

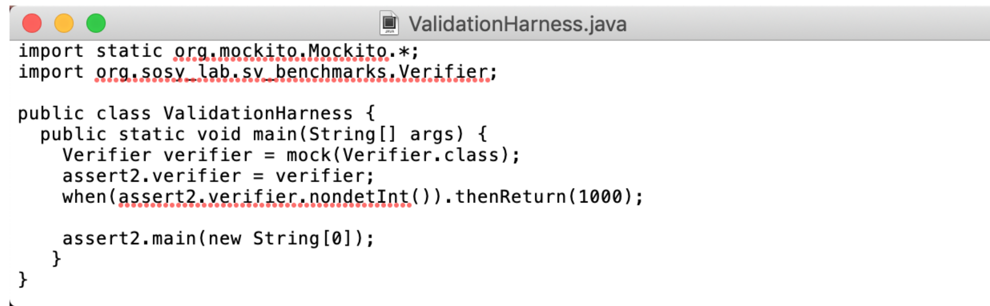
```
common — -bash — 101×15
Last login: Mon Aug 24 11:19:46 on console
Vis-MacBook-Pro:~ vilynn$ cd Desktop/sv-benchmarks/java/common/
Vis-MacBook-Pro:common vilynn$ python script.py assert2.java
Warning: failed to access directory `'
stub class symbol java::java.lang.Object already exists
Exception in thread "main" java.lang.AssertionError: i is greater 1000
        at assert2.main(assert2.java:18)
        at ValidationHarness.main(ValidationHarness.java:10)
Vis-MacBook-Pro:common vilynn$
```

Figure 3.3: Screenshot of Python script output of an `int` example

In Figure 3.3, the Python script is executed on the CLI. This is done by with the following command:

```
python script.py assert2.java
```

The output result of the script can be seen in the screenshot whereby the result matches what had been obtained from JBMC directly. In both cases, the `assert` statements are violated. Therefore, this means that this is indeed a valid bug, which has been successfully verified.



```
ValidationHarness.java
import static org.mockito.Mockito.*;
import org.sosy_lab.sv_benchmarks.Verifier;

public class ValidationHarness {
  public static void main(String[] args) {
    Verifier verifier = mock(Verifier.class);
    assert2.verifier = verifier;
    when(assert2.verifier.nondetInt()).thenReturn(1000);

    assert2.main(new String[0]);
   }
}
```

Figure 3.4: Screenshot of the validation harness of a `int` example

Figure 3.4 shows a screenshot of the validation harness generated from the script based on the template. Here, it can be witnessed that the creation of the validation harness is done correctly. The Java class name replaced the placeholder values, `assert2`, the verifier data type replaced by `nondetInt()` and the counterexample value of 1000 was injected accurately.

3.3.2 Long example

```
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/jbmc-strings/StringValueOf07
7  * The benchmark was taken from the repo: 24 January 2018
8  */
9 import org.sosy_lab.sv_benchmarks.Verifier;
10
11 public class StringValueOf07 {
12   public static Verifier verifier = new Verifier();
```

```
13   public static void main(String[] args) {
14     long longValue = verifier.nondetLong();
15     System.out.printf("long = %s\n", String.valueOf(
     longValue));
16     String tmp = String.valueOf(longValue);
17     assert tmp.equals("100000000000");
18   }
19 }
```

Listing 3.14: Illustrative example of `Long` Java program

Listing 3.14 is a Java program, which contains a bug. In Line 17, the program asserts that the `long` value of `tmp` must be equal to than 100000000000. However, it is immediately noticeable that the range of`long` values start at 0, since it is a primitive number type. Thus, since the value of `tmp` can be anything other than 100000000000, the `assert` statement will fail as expected.

First, we obtain the initial results from JBMC by running the following command on the CLI.

```
jbmc StringValueOf07 --stop-on-fail
```

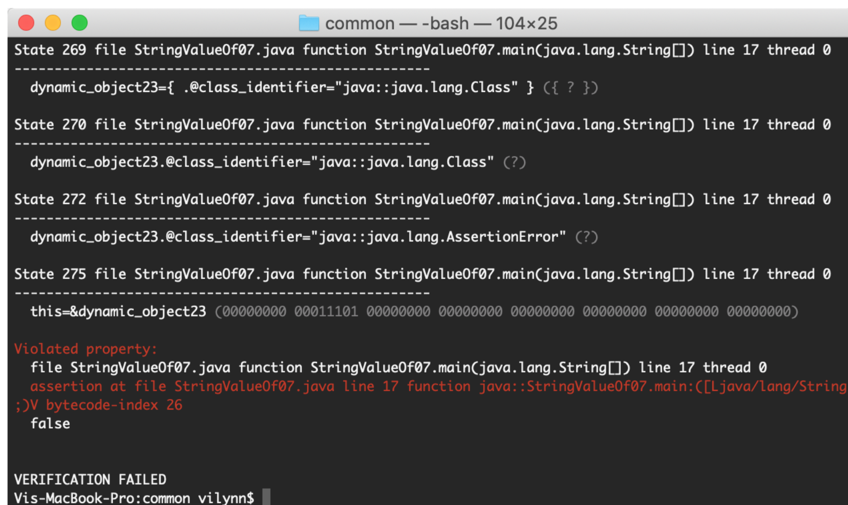


Figure 3.5: Screenshot of JBMC output of an `Long` example

In Figure 3.5, it can be observed that the `StringValueOf07` program contains a bug as it has output "VERIFICATION FAILED". The counterexample is not noticeable in this screenshot but should read as the following:

```
State 162 file StringValueOf07.java function StringValueOf07.main
(java.lang.String[]) line 14 thread 0
----------------------------------------------------
  anonlocal::1l=0L (00000000 00000000 00000000 00000000 00000000
  00000000 00000000 00000000)
```

The counterexample, or rather the value of $0L$ is the counterexample, which will be extracted by the Python script. Now we will verify this with the Python script and validation harness built into Mockito.

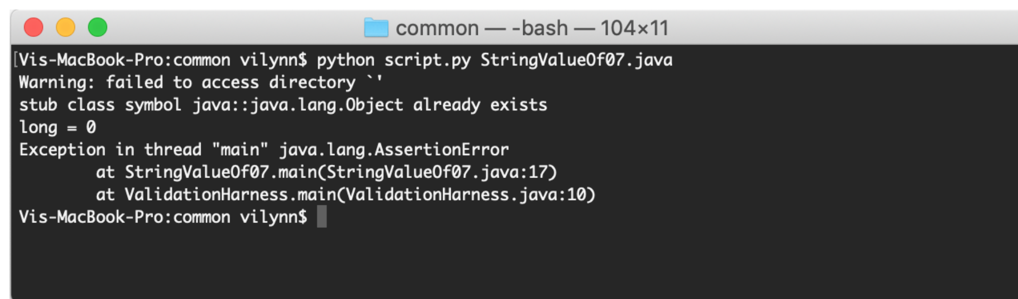


Figure 3.6: Screenshot of Python script output of an `Long` example

In Figure 3.6, the Python script is executed on the CLI. This is done by with the following command:

```
python script.py StringValueOf07.java
```

The output result of the script can be seen in the screenshot whereby the result matches what had been obtained from JBMC directly. In both cases, the `assert` statements are violated. Therefore, this means that this is indeed a valid bug, which has been successfully verified.

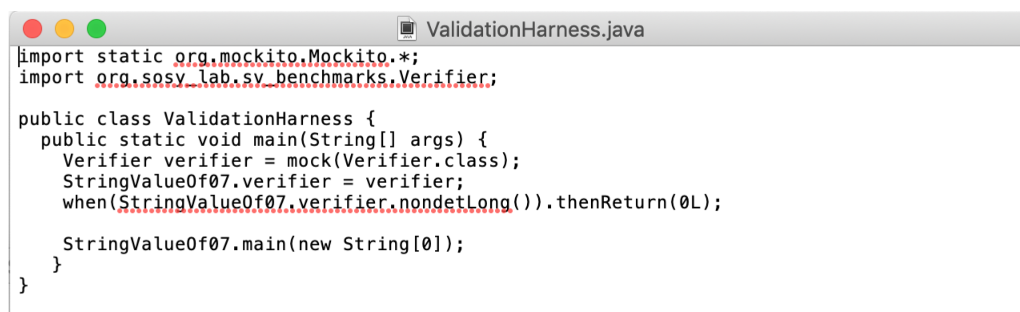


Figure 3.7: Screenshot of the validation harness of a `Long` example

Figure 3.7 shows a screenshot of the validation harness generated from the script based on the template. Here, it can be witnessed that the creation of the validation harness is done correctly. The placeholder values were replaced by the Java class name, `StringValueOf07`, the verifier data type replaced by `nondetLong()` and the counterexample value of 0*L* was injected accurately.

3.3.3 `char` example

```
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/jbmc-strings/
     StaticCharMethods04
7  * The benchmark was taken from the repo: 24 January 2018
8  */
9 import org.sosy_lab.sv_benchmarks.Verifier;
10
11 public class StaticCharMethods04 {
12   public static Verifier verifier = new Verifier();
13
14   public static void main(String[] args) {
15     char c = verifier.nondetChar();
16     assert Character.isLetter(c);
17   }
18 }
```

Listing 3.15: Illustrative example of `char` Java program

Listing 3.15 is a Java program, which contains a bug. In Line 16, the program asserts that the `char` variable `c` must be of a letter. However, symbols such as exclamation marks (!) and question marks (?) are also part of the `char` family. Thus, the `assert` statement will fail as expected.

First, we obtain the initial results from JBMC by running the following command on the CLI.

```
jbmc StaticCharMethods04 --stop-on-fail
```

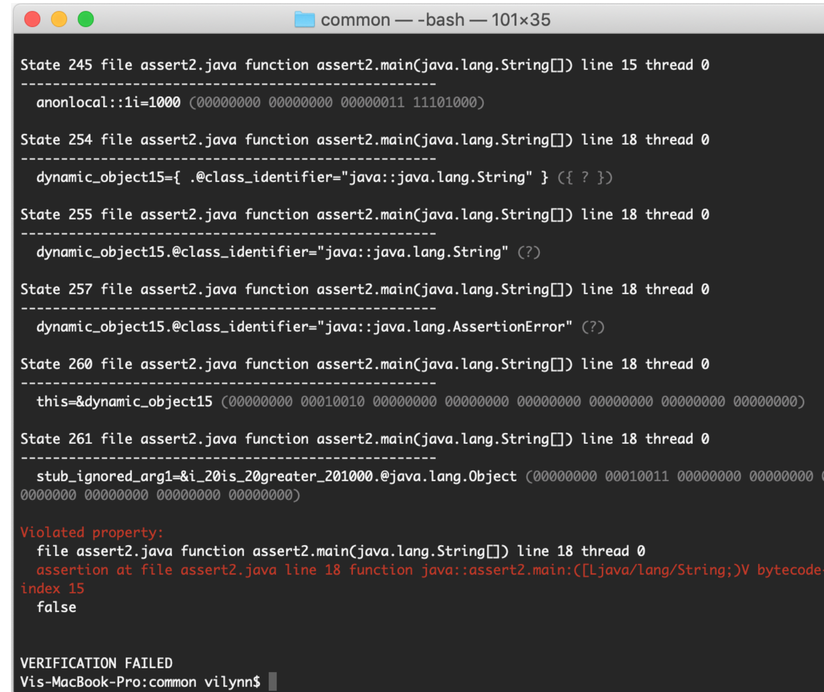


Figure 3.8: Screenshot of JBMC output of an `char` example

In Figure 3.8, it can be observed that the `StaticCharMethods04` program contains a bug as it has output “VERIFICATION FAILED”. It should also be pointed out that the counterexample is noticeable in this screenshot. At the top of the figure, it reads the following:

```
State 123 file StaticCharMethods04.java function
StaticCharMethods04.main(java.lang.String[]) line 15 thread 0
----------------------------------------------------
  anonlocal::1i=60 (00000000 00000000 00000000 00111100)
```

This is, in fact, the counterexample, or rather the value of 60, which is the `int` value of a `char`, is the counterexample that will be extracted by the Python script. Now we will verify this with the Python script and validation harness built into Mockito.

In Figure 3.9, the Python script is executed on the CLI. This is done by with the following command:

```
python script.py StaticCharMethods04.java
```

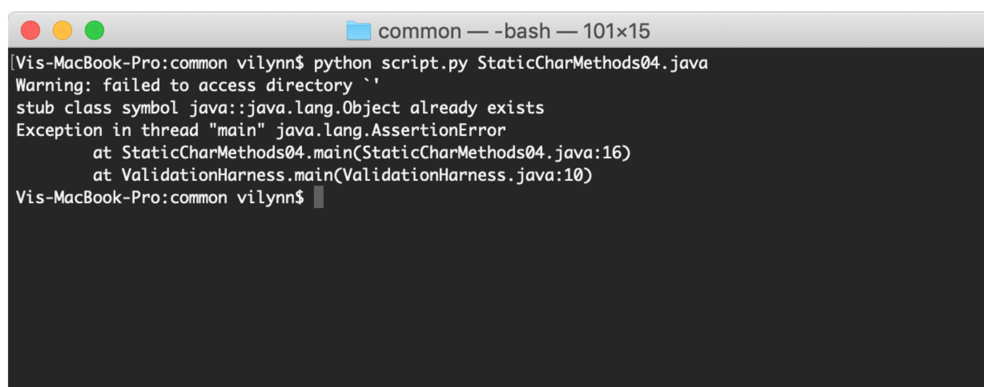


Figure 3.9: Screenshot of Python script output of an `char` example

The output result of the script can be seen in the screenshot whereby the result matches what had been obtained from JBMC directly. In both cases, the `assert` statements are violated. Therefore, this means that this is indeed a valid bug, which has been successfully verified.

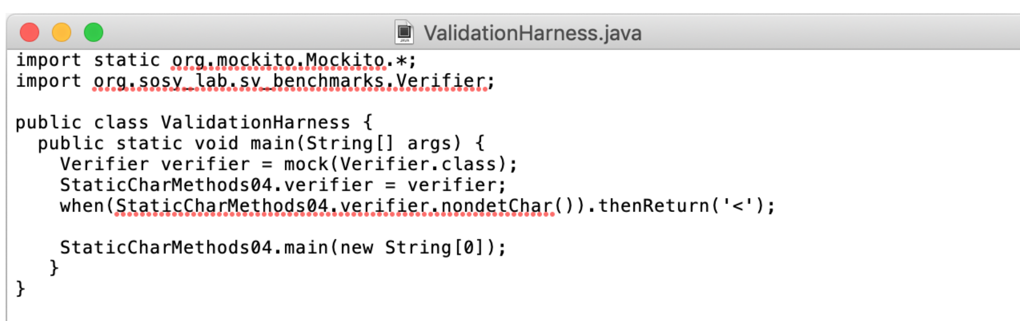


Figure 3.10: Screenshot of the validation harness of a `char` example

Figure 3.10 shows a screenshot of the validation harness generated from the script based on the template. Here, it can be witnessed that the creation of the validation harness is done correctly. The Java class name replaced the placeholder values, `StaticCharMethods04`, the verifier data type replaced by `nondetChar()` and the counterexample value of 60, which was then converted in character ’¡’ was injected accurately.

3.3.4 Boolean example

```
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/jbmc-strings/StringValueOf04
7  * The benchmark was taken from the repo: 24 January 2018
8  */
9 import org.sosy_lab.sv_benchmarks.Verifier;
10
11 public class StringValueOf04 {
12   public static Verifier verifier = new Verifier();
13
14   public static void main(String[] args) {
15     boolean booleanValue = verifier.nondetBoolean();
16     String tmp = String.valueOf(booleanValue);
17     assert tmp.equals("true");
18   }
19 }
```

Listing 3.16: Illustrative example of `Boolean` Java program

Listing 3.16 is a Java program, which contains a bug. In Line 17, the program asserts that the `Boolean` value of `tmp` must always be equal to `true`. However, a `Boolean` variable will not always carry a `true` value. All `Boolean` values can be either `true` or `false`. Thus, the `assert` statement will fail as expected.

First, we obtain the initial results from JBMC by running the following command on the CLI.

```
jbmc StringValueOf04 --stop-on-fail
```

In Figure 3.11, it can be observed that the `StringValueOf04` program contains a bug as it has output “VERIFICATION FAILED”. The counterexample is not noticeable in this screenshot but should read as the following:

```
State 134 file StringValueOf04.java function
```

```
common — -bash — 101×35
State 176 file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 17 thread
0
----------------------------------------------------
  stub_ignored_arg1=&true.@java.lang.Object (00000000 00010100 00000000 00000000 00000000 00000000 00
000000 00000000)

State 183 file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 17 thread
0
----------------------------------------------------
  dynamic_object17={ .@class_identifier="java::java.lang.String" } ({ ? })

State 184 file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 17 thread
0
----------------------------------------------------
  dynamic_object17.@class_identifier="java::java.lang.String" (?)

State 186 file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 17 thread
0
----------------------------------------------------
  dynamic_object17.@class_identifier="java::java.lang.AssertionError" (?)

State 189 file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 17 thread
0
----------------------------------------------------
  this=&dynamic_object17 (00000000 00010101 00000000 00000000 00000000 00000000 00000000 00000000)

Violated property:
  file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 17 thread 0
  assertion at file StringValueOf04.java line 17 function java::StringValueOf04.main:([Ljava/lang/Str
ing;)V bytecode-index 15
  false


VERIFICATION FAILED
Vis-MacBook-Pro:common vilynn$
```

Figure 3.11: Screenshot of JBMC output of an `Boolean` example

```
StringValueOf04.main(java.lang.String[]) line 15 thread 0
----------------------------------------------------
  anonlocal::1i=0 (00000000 00000000 00000000 00000000)
```

The counterexample in essence itself is the value of 0, which is in fact equivalent to the `Boolean` value of `false`. This counterexample value will be extracted by the Python script and represented as `false`. Now we will verify this with the Python script and validation harness built into Mockito.

```
common — -bash — 101×15
Vis-MacBook-Pro:common vilynn$ python script.py StringValueOf04.java
Warning: failed to access directory `'
stub class symbol java::java.lang.Object already exists
Exception in thread "main" java.lang.AssertionError
        at StringValueOf04.main(StringValueOf04.java:17)
        at ValidationHarness.main(ValidationHarness.java:10)
Vis-MacBook-Pro:common vilynn$
```

Figure 3.12: Screenshot of Python script output of an `Boolean` example

In Figure 3.12, the Python script is executed on the CLI. This is done by with the following command:

```
python script.py StringValueOf04.java
```

The output result of the script can be seen in the screenshot whereby the result matches what had been obtained from JBMC directly. In both cases, the `assert` statements are violated. Therefore, this means that this is indeed a valid bug, which has been successfully verified.

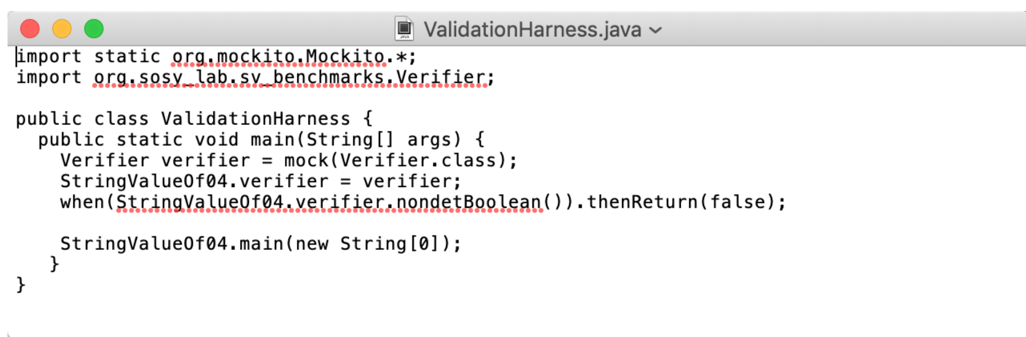


Figure 3.13: Screenshot of the validation harness of a `Boolean` example

Figure 3.13 shows a screenshot of the validation harness generated from the script based on the template. Here, it can be witnessed that the creation of the validation harness is done correctly. The placeholder values were replaced by the Java class name, `StringValueOf04`, the verifier data type replaced by `nondetBoolean()` and the counterexample value of 'False' was injected accurately.

Chapter 4

Experimental Evaluation

This chapter describes the experimental evaluation of the algorithms implemented, which is divided into three main sections. Section 4.1 details the requirements needed to set up the environment for the experimental evaluation. Section 4.2 describes the objectives of this experimental evaluation, While Section 4.3 outlines the results and describes threats to the validity of the results.

4.1 Setup

4.1.1 Environment setup

The following subsections list the required steps to be performed before starting the experimental evaluation. The order is not strict except for Java as it is required to be pre-installed before JBMC [CKS19] and Mockito [Moc].

4.1.1.1 Java environment installation

Java is available for download at the following official site:

`https://www.java.com/en/download/manual.jsp`

The Java Development Kit (JDK) is available for download at the following site:

`https://www.oracle.com/java/technologies/javase-downloads.html`

A comprehensive installation guide for installing the JDK on various platforms are available here:

`https://www3.ntu.edu.sg/home/ehchua/programming/howto/JDK_Howto.html`

4.1.1.2 Python environment installation

Python is available for download at the following official site:

`https://www.python.org/downloads/`

For this project, it is mandatory to install Python 3. A comprehensive installation guide for installing Python on various platforms is available here:

`https://realpython.com/installing-python/`

Ensure that the following Python packages are installed as well:

- `subprocess`
- `sys`
- `networkx`

A comprehensive guide on how to install Python packages is available here:

`https://packaging.python.org/tutorials/installing-packages/`

4.1.1.3 Java verifier installation (JBMC)

As JBMC is built based on C Bounded Model Checker (CBMC), it is useful to have CBMC installed first. CBMC is available for download here:

`https://github.com/diffblue/cbmc`

JBMC is available as part of the CBMC repository. The standalone JBMC tool is available for download from its GitHub repository here:

`https://github.com/diffblue/cbmc/tree/develop/jbmc`

The installation instructions for both CBMC and JBMC are available in the README files located in the Github repository.

Then, finally add the JBMC `PATH` variable to point to the directory containing the JBMC executable as below:

```
export PATH="path-to-directory/cbmc/jbmc/src/jbmc:$PATH"
```

4.1.1.4 Mockito installation

Mockito is available to download from the following site:

`https://mvnrepository.com/artifact/org.mockito/mockito-core`

There are three dependencies of Mockito required to be downloaded separately:

- `byte-buddy`

- byte-buddy-agent
- objenesis

These dependencies are available on the same site and can be seen under the ”Compile Dependencies” section of the page. It lists the compatible version with the mockito-core version selected as well as the latest version of each dependency.

Then, the environment variables need to be set using the following commands (below shown for MacOS):

```
Mockito_HOME=/Library/Mockito
export Mockito_HOME

export CLASSPATH=$CLASSPATH:$Mockito_HOME/mockito-core-3.3.3.jar:
$Mockito_HOME/byte-buddy-1.10.5.jar:$Mockito_HOME/byte-buddy-agent
-1.10.5.jar:$Mockito_HOME/objenesis-2.6.jar:.
```

This can alternatively be added to the ./bash_profile file.

4.1.1.5 Proposed extension

The Python script and validation harness template implemented in this project is available for download at the Github repository below:

https://github.com/vilynntan/msc-project

It includes the Python script, the validation harness template, the modified Verifier.java file from SV-COMP benchmarks repository and some sample benchmarks.

4.1.1.6 Benchmarks

The complete list of benchmarks are available for download at the Github repository below:

https://github.com/sosy-lab/sv-benchmarks

The benchmark files used can be found at sv-benchmarks/java/jbmc-regression.

The benchmarks for the Java category of SV-COMP mainly involves verification of reachability. This category known as ReachSafety in SV-COMP is one of the categories tested for C programs among five other categories; ConcurrencySafety, MemSafety, NoOverflows, SoftwareSystems and Termination [oSVSC20].

4.1.2 Environment versions

In this project, the software and its respective versions used are as below:

- JDK/JRE (Version 1.8.0_77)
- `python3` (Version 3.8.3)
- `pip3` (Version 20.0.2)
- Homebrew (Version 2.4.7)
- CBMC/JBMC (Retrieved from GitHub repository on 12th March 2020)
- `mockito-core` (Version 3.3.3)
- `byte-buddy` (Version 1.10.5)
- `byte-buddy-agent` (Version 1.10.5)
- `objenesis` (Version 2.6)
- MacOS Terminal (Version 2.9.5)

The specification of the hardware used in this project are as below:

- Model: Macbook Pro (Late 2013)
- Processor: 2.6 GHz Intel Core i5
- RAM: 8GB
- Operating System: MacOS Mojave (Version 10.14.6)

4.1.3 Running the tests

The benchmark file desired to be tested needs to be on the same directory level as the Python script and the validation harness template. Some samples of the benchmark files have been uploaded as part of the project repository as guidance. These include `assert2.java`, `StaticCharMethods04.java` and `StringValueOf04.java`. The directory style and sample benchmarks is illustrated in Figure 4.1.

Name	Date Modified	Size	Kind
assert2.class	Yesterday at 12:16 am	763 bytes	Java class file
assert2.java	30 Aug 2020 at 2:51 pm	566 bytes	Java
▼ org	31 Aug 2020 at 3:33 pm	--	Folder
▼ sosy_lab	29 Apr 2020 at 7:45 pm	--	Folder
▼ sv_benchmarks	5 Aug 2020 at 10:27 pm	--	Folder
Verifier.class	5 Aug 2020 at 10:27 pm	1 KB	Java class file
Verifier.java	4 Aug 2020 at 2:20 pm	2 KB	Java
script.py	Yesterday at 12:15 am	4 KB	Python Source
StaticCharMethods04.class	30 Aug 2020 at 6:26 pm	776 bytes	Java class file
StaticCharMethods04.java	30 Aug 2020 at 6:23 pm	545 bytes	Java
StringValueOf04.class	30 Aug 2020 at 6:43 pm	848 bytes	Java class file
StringValueOf04.java	30 Aug 2020 at 6:43 pm	598 bytes	Java
ValidationHarness.class	Yesterday at 12:16 am	807 bytes	Java class file
ValidationHarness.java	Yesterday at 12:16 am	334 bytes	Java
ValidationHarnessTemplate.txt	4 Aug 2020 at 2:33 pm	345 bytes	Plain Text
witness	Yesterday at 12:16 am	7 KB	TextEdit

Figure 4.1: Project directory

Before running the script on the CLI, a crucial step is to ensure that the necessary changes have been implemented as described in Section 3.2.3. Now, the tests are ready to be carried out.

First, run the benchmark tests with JBMC. As the benchmark files are `.java` files, it would need to be compiled beforehand. This can be done with the command below:

```
javac assert2.java
```

Then, run the compiled file with JBMC using the command below:

```
jbmc assert2 --stop-on-fail
```

The output will be printed on the CLI along with the stack trace if the verification has failed.

Next, run the script with the benchmark using the command below:

```
python3 script.py assert2.java
```

The output will be printed on the CLI.

All results are recorded and displayed in Table 4.1 in Section 4.3.

4.2 Objectives

The overall objective of this experimental evaluation is to be able to evaluate the performance of the algorithms implemented for the witness validator. This will allow us to understand the effectiveness and value that the proposed extension may bring forth.

The specific objectives of the experimental evaluation are:

- Test the algorithms against benchmarks used in SV-COMP.
- Obtain and analyse results.
- Compare results produced by the algorithms against those produced by JBMC.
- Identify any potential discrepancies between results.

4.3 Results and Threat to Validity

No	FileName	Testable?	Type	JBMC output	Script output	Comment
1	aastore_aaload1	Y	int	N/A	N/A	Execution time out
2	array1	Y	int	N/A	N/A	Execution time out
3	array2	N	int	S	N/A	
4	arraylength1	N	int	S	N/A	
5	arrayread1	N	int	S	N/A	
6	assert1	N	int	S	N/A	
7	assert2	Y	int	F	F	
8	assert3	Y	int	F	F	
9	assert4	Y	int	F	F	
10	assert5	N	int	S	N/A	
11	assert6	N	int	S	N/A	
12	astore_aload1	N		N/A	N/A	No verifier type
13	athrow1	N		N/A	N/A	No verifier type
14	basic1	N		N/A	N/A	No verifier type
15	bitwise1	N	int	S	N/A	
16	boolean1	N	bool	S	N/A	
17	boolean2	N	bool	S	N/A	
18	bug-test-gen-095	Y	string	F	F	
19	bug-test-gen-119	Y	bool	F	F	Null pointer exception
20	bug-test-gen-119-2	N		N/A	N/A	No verifier type
21	calc	N		N/A	N/A	Multiple verifiers
22	cast1	N	int	S	N/A	
23	catch1	N		N/A	N/A	No verifier type
24	char1	Y	string	F	F	Null pointer exception
25	charArray	Y	string	F	F	Null pointer exception
26	classtest1	N		N/A	N/A	No verifier type
27	const1	N		N/A	N/A	No verifier type
28	constructor1	N		N/A	N/A	No verifier type
29	enum1	N		N/A	N/A	No verifier type
30	exceptions1	N		N/A	N/A	No verifier type
31	exceptions2	N		N/A	N/A	No verifier type
32	exceptions3	N		N/A	N/A	No verifier type
33	exceptions4	N		N/A	N/A	No verifier type
34	exceptions5	N		N/A	N/A	No verifier type
35	exceptions6	N		N/A	N/A	No verifier type
36	exceptions7	N		N/A	N/A	No verifier type
37	exceptions8	N		N/A	N/A	No verifier type
38	exceptions9	N		N/A	N/A	No verifier type
39	exceptions10	N		N/A	N/A	No verifier type
40	exceptions11	N		N/A	N/A	No verifier type
41	exceptions12	N		N/A	N/A	No verifier type
42	exceptions13	N		N/A	N/A	No verifier type
43	exceptions14	N		N/A	N/A	No verifier type
44	exceptions15	N		N/A	N/A	No verifier type
45	exceptions16	Y	int	F	F	
46	exceptions18	N		N/A	N/A	No verifier type
47	fcmpx_dcmpx1	N		N/A	N/A	No verifier type

No	FileName	Testable?	Type	JBMC output	Script output	Comment
48	iarith1	N		N/A	N/A	No verifier type
49	iarith2	N		N/A	N/A	No verifier type
50	if_acmp1	N		N/A	N/A	No verifier type
51	if_expr1	N	int	S	N/A	
52	if_icmp1	N	int	S	N/A	
53	ifxx1	N		N/A	N/A	No verifier type
54	instanceof1	N		N/A	N/A	No verifier type
55	instanceof2	N		N/A	N/A	No verifier type
56	instanceof3	N		N/A	N/A	No verifier type
57	instanceof4	N		N/A	N/A	No verifier type
58	instanceof5	N		N/A	N/A	No verifier type
59	instanceof6	N		N/A	N/A	No verifier type
60	instanceof7	N		N/A	N/A	No verifier type
61	instanceof8	N		N/A	N/A	No verifier type
62	interface1	N		N/A	N/A	No verifier type
63	java_append_char	Y	bool	F	F	
64	lazyloading4	N		N/A	N/A	No verifier type
65	list1	N	int	S	N/A	
66	long1	N		N/A	N/A	No verifier type
67	lookupswitch1	N	int	S	N/A	
68	multinewarray	N		N/A	N/A	No verifier type
69	overloading1	N		N/A	N/A	No verifier type
70	package1	N		N/A	N/A	No verifier type
71	putfield_getfield1	N		N/A	N/A	No verifier type
72	putstatic_getstatic1	N		N/A	N/A	No verifier type
73	recursion2	N		N/A	N/A	No verifier type
74	return1	N		N/A	N/A	No verifier type
75	return2	N		N/A	N/A	Multiple verifiers
76	store_load1	N		N/A	N/A	No verifier type
77	swap1	N		N/A	N/A	No verifier type
78	synchronized	N		N/A	N/A	No verifier type
79	tableswitch1	N	int	S	N/A	
80	TokenTest01	N		N/A	N/A	No verifier type
81	TokenTest02	Y	string	N/A	N/A	Execution time out
82	uninitialised1	N		N/A	N/A	No verifier type
83	Validate01	N		N/A	N/A	No verifier type
84	Validate02	N		N/A	N/A	Multiple verifiers
85	virtual_function_unwinding	N		N/A	N/A	No verifier type
86	virtual1	N		N/A	N/A	No verifier type
87	virtual2	N		N/A	N/A	No verifier type
88	virtual4	N		N/A	N/A	No verifier type
89	ArithmeticException1	Y	int	F	F	
90	ArithmeticException5	N		N/A	N/A	No verifier type
91	ArithmeticException6	Y	int	F	F	
92	ArrayIndexOutOfBoundsException1	Y	int	F	F	
93	ArrayIndexOutOfBoundsException2	Y	int	F	F	
94	ArrayIndexOutOfBoundsException3	Y	int	F	F	Counterexample (array index) $<$array length
95	BufferedReaderReadLine	Y	string	N/A	N/A	Execution time out
96	CharSequenceBug	Y	string	F	F	Null pointer exception
97	CharSequenceToString	Y	string	F	F	Null pointer exception
98	Class_method1	N		N/A	N/A	No verifier type
99	ClassCastException1	N		N/A	N/A	No verifier type
100	ClassCastException2	N		N/A	N/A	No verifier type
101	ClassCastException3	N		N/A	N/A	No verifier type
102	Inheritance1	N		N/A	N/A	No verifier type
103	NegativeArraySizeException1	N		N/A	N/A	No verifier type
104	NegativeArraySizeException2	N		N/A	N/A	No verifier type
105	NullPointerException1	N		N/A	N/A	No verifier type
106	NullPointerException2	N		N/A	N/A	No verifier type
107	NullPointerException3	N		N/A	N/A	No verifier type
108	NullPointerException4	N		N/A	N/A	No verifier type
109	RegexMatches01	N		N/A	N/A	No verifier type
110	RegexMatches02	Y	string	N/A	N/A	Execution time out
111	RegexSubstitution01	N		N/A	N/A	No verifier type
112	RegexSubstitution02	N		N/A	N/A	Multiple verifiers
113	RegexSubstitution03	N		N/A	N/A	No verifier type
114	StaticCharMethods01	N		N/A	N/A	No verifier type
115	StaticCharMethods02	Y	string	F	F	Null pointer exception

No	FileName	Testable?	Type	JBMC output	Script output	Comment
116	StaticCharMethods03	Y	string	F	F	Null pointer exception
117	StaticCharMethods04	Y	char	F	F	
118	StaticCharMethods05	N		N/A	N/A	Multiple verifiers
119	StaticCharMethods06	Y	string	F	F	Null pointer exception
120	StringBuilderAppend01	N		N/A	N/A	No verifier type
121	StringBuilderAppend02	N		N/A	N/A	Multiple verifiers
122	StringBuilderCapLen01	N		N/A	N/A	No verifier type
123	StringBuilderCapLen02	Y	string	F	F	Null pointer exception
124	StringBuilderCapLen03	Y	string	F	F	Null pointer exception
125	StringBuilderCapLen04	Y	string	F	F	Null pointer exception
126	StringBuilderChars01	N		N/A	N/A	No verifier type
127	StringBuilderChars02	Y	string	F	F	Null pointer exception
128	StringBuilderChars03	Y	string	F	F	Null pointer exception
129	StringBuilderChars04	Y	string	N/A	N/A	Execution time out
130	StringBuilderChars05	Y	string	F	F	Null pointer exception
131	StringBuilderChars06	Y	string	F	F	Null pointer exception
132	StringBuilderConstructors01	Y	string	F	F	Null pointer exception
133	StringBuilderConstructors02	Y	string	F	F	Null pointer exception
134	StringBuilderInsertDelete01	N		N/A	N/A	No verifier type
135	StringBuilderInsertDelete02	N		N/A	N/A	Multiple verifiers
136	StringBuilderInsertDelete03	N		N/A	N/A	Multiple verifiers
137	StringCompare01	N		N/A	N/A	No verifier type
138	StringCompare02	N		N/A	N/A	Multiple verifiers
139	StringCompare03	N		N/A	N/A	Multiple verifiers
140	StringCompare04	N		N/A	N/A	Multiple verifiers
141	StringCompare05	Y	string	F	F	Null pointer exception
142	StringConcatenation01	N		N/A	N/A	Multiple verifiers
143	StringConcatenation02	N		N/A	N/A	Multiple verifiers
144	StringConcatenation03	N		N/A	N/A	Multiple verifiers
145	StringConcatenation04	Y	string	F	F	Null pointer exception
146	StringConstructors01	N		N/A	N/A	No verifier type
147	StringConstructors02	Y	string	F	F	Array size should be $>=0$
148	StringConstructors03	N		N/A	N/A	Multiple verifiers
149	StringConstructors04	Y	string	F	F	Null pointer exception
150	StringConstructors05	Y	string	F	F	Null pointer exception
151	StringContains01	N		N/A	N/A	Multiple verifiers
152	StringContains02	Y	string	F	F	Null pointer exception
153	StringIndexMethods01	N		N/A	N/A	No verifier type
154	StringIndexMethods02	Y	string	F	F	Null pointer exception
155	StringIndexMethods03	Y	string	F	F	Null pointer exception
156	StringIndexMethods04	Y	string	F	F	Null pointer exception
157	StringIndexMethods05	Y	string	F	F	Null pointer exception
158	StringMiscellaneous01	N		N/A	N/A	No verifier type
159	StringMiscellaneous02	Y	string	F	F	Null pointer exception
160	StringMiscellaneous03	N		N/A	N/A	Multiple verifiers
161	StringMiscellaneous04	N		N/A	N/A	No verifier type
162	StringStartEnd01	N		N/A	N/A	No verifier type
163	StringStartEnd02	N		N/A	N/A	Multiple verifiers
164	StringStartEnd03	N		N/A	N/A	Multiple verifiers
165	StringValueOf01	N		N/A	N/A	No verifier type
166	StringValueOf02	Y	string	F	F	Null pointer exception
167	StringValueOf03	Y	string	F	F	Null pointer exception
168	StringValueOf04	Y	bool	F	F	
169	StringValueOf05	Y	string	F	F	Null pointer exception
170	StringValueOf06	Y	int	F	F	
171	StringValueOf07	Y	long	F	F	
172	StringValueOf08	Y	string	F	F	Null pointer exception
173	StringValueOf09	Y	string	F	F	Null pointer exception
174	StringValueOf10	Y	string	F	F	Null pointer exception
175	SubString01	N		N/A	N/A	No verifier type
176	SubString02	Y	string	F	F	Null pointer exception
177	SubString03	Y	string	F	F	Null pointer exception

Table 4.1: Experimental evaluation results using SV-COMP benchmark files

Table 4.1 shows the results from the experimental evaluation using the SV-COMP benchmarks [oSVSC19]. There are six columns; “Filename”, “Testable?”, “Type”, “JBMC output”, “Script output” and “Comment”. The “Filename” column indicates the file name of the benchmark. The “Testable?” column indicates if the file is eligible for testing using “Y” for yes and “N” for no. Files with no or multiple calls to the `Verifier` class are not eligible for evaluation as they are not supported. This will be described in the “Comment” section. The “Type” column indicates the `Verifier`’s data types such as integer or string. The “JBMC output” column indicates the result of the benchmark when executed using JBMC. Similarly, the “Script output” column indicates the result of the benchmark when executed using the Python script developed for this dissertation. For both the “JBMC output” and “Script output” column, “S” indicates that the verification has been successful, “F” indicates that the verification has failed and “N/A” indicates that a result has not been able to be obtained. In all cases of “N/A”, a description as to why the result had not been able to be obtained is described in the “Comment” section.

The “Comment” section provides more insight into the results. If this column is left empty, it indicates that the benchmark has achieved the same outcome when executed using both JBMC and the script. There are three types of comments which indicate that the evaluation could not be performed using JBMC and/or the Python script. The first is ”Execution time out” which indicates that the CLI has terminated the execution. This is due to the execution taking too much time or memory and thus no outcome of whether the verification was successful or otherwise could be reached. The second is ”No verifier type” which indicates that the particular benchmark did not involve the use of the `Verifier` method. As the proposed extension involves the use of injecting a static counterexample value into the mocked `Verifier` method, no evaluation can be performed if the benchmark does not involve the use of the `Verifier` method. The third is ”Multiple verifiers” which indicates that the particular benchmark involves the use of multiple various invocation to the `Verifier` method. As the proposed extension is limited to only one `Verifier` method invocation call, no evaluation can be performed on that particular benchmark. Other comments such as ”Null pointer exception”, ”Counterexample (array index) ¡ array length” and ”Array size should be ¿=0” are specific errors which arised due to assertion violation within the `Verifier` method. This limitation is exclusive to the `String` data type as there are some issues when the `Verifier` method in invoked to return a non-deterministic string.

The results have indicated that the performance of this script has been consistent

with JBMC's output. There have not been any cases in which the script produces a different output to JBMC. This is an expected outcome since JBMC has been performing well on these benchmarks, such that it has successfully produced the correct outcome of all of the test cases [Bey20]. As such, there is no concrete example of any benchmark in which the script will be able to prove that JBMC has produced a false positive in identifying bugs in the program. On top of that, the performance of the proposed extension has not been tested against several primitive data types such as byte, short, float and double. This is because of limitations due to the set of benchmarks available.

Chapter 5

Related Work

Research within the realm of software verification for Java programs and witness validation specifically for Java verifiers are relatively new, if to be compared with within the C language verifiers. Implementations for witness validators for C programs have shown good results and promise for this research area [BS20].

MetaVal [BS20] is a witness validator for C Programs developed by a team from LMU Munich, Germany. In its submission to SV-COMP 2020, it has confirmed 3,653 violation witnesses and 16,376 correctness witnesses. The overall results based on a large benchmark set indicate that MetaVal improves the effectiveness of the validation process as a whole. It uses what it calls a transformer which receives the original input program and applies selected transformations. For instance, if the transformer receives a violation witness, it will output a new program which strictly contains necessary information related to the counterexample. This pruning process is similar to the condition reducer found in reducer-based conditional model checking. MetaVal has been implemented to enable the use of any verifier such as CPAChecker or Ultimate Automizer, both are submissions to SV-COMP 2020. This will allow the witness validator to choose the best verifier in an ”off-the-shelf” manner.

NITWIT [ŠBK20] is another tool for witness validation first submitted to SV-COMP in 2020. Developed by a team from RWTH Aachen University, Germany, NITWIT is described as ”interpretation-based violation witness validation” for C programs. NITWIT describes an interpreter as ”a program that takes as input a program, parses it and executes commands as part of its own runtime instead of producing machine code like a compiler”. The interpreter will perform direct translation of programs into their exact behaviour representation in which values of all variables in the program are tracked and statements are executed on the basis of results from expressions and

control flow [ŠBK20]. This tool, which has a small memory footprint and is proven to be notably faster than other competitors within the category [ŠBK20], has successfully validated 8,526 witnesses out of 11,533 witnesses [oSVSC20]. Additionally, it was the only tool in SV-COMP 2020 which managed to validate 399 witnesses which were not validated by other validators within the competition [oSVSC20, ŠBK20].

This dissertation proposed an extension to perform witness validation on Java programs. The main goal is similar to what MetaVal and NITWIT desire and had achieved; which is to verify the counterexample in the form of a witness produced by a software verification tool. In this dissertation, the proposed extension implementation verifies the validity of the violations identified by a software verifier for Java programs such as JBMC. The research has managed to demonstrate using some instances from SV-COMP benchmarks using a unique methodology of using Mockito, a mocking framework. The proposed methodology is still in the early stages with implementations of rather straightforward and purely functional algorithms without any optimisation. However, this research should serve as a demonstration of feasibility in validating witnesses for Java programs.

Chapter 6

Conclusion

This dissertation has managed to produce an implementation of a witness validation extension for software verifiers of Java programs. In specific it has demonstrated the ability to perform witness validation of violation witnesses based on GraphML. The implementation was built using Python and Mockito. Limitations of the implementation have been identified and described in experimental evaluation chapter, Chapter 4. While the implementation is still rough around the edges, the initial results shows some potential especially in the field on interest, for finding security vulnerabilities.

Now, more than ever, it is a constant race against finding and verifying security vulnerabilities preventively to avoid catastrophic security breaches and attacks. As we have discussed in Section 2.2, the cost of a single security incident can cause a huge blow. Software verification currently is and will continue growing to be of high importance. Thus, this implementation extension for Java verifiers which aims to perform witness validation will hopefully support the research area around this topic.

Further work and improvements can be done on the proposed extension such as implementation of a test suite which will increase the level of automation within this implementation. There could also be more done to improve support for `string` variables as described in Chapter 4. Due to the limitations with the benchmark files, not all primitive data types could be tested. A collaboration with the committee of SV-COMP could perhaps result in a better benchmark data set. Furthermore, there should also be more testing to be conducted to support witness validation of various other Java verifiers such as JayHorn [KRS19] and Java Ranger [SHW+20]. This will promote the use of witness validation and allow flexibility in choosing the best state-of-the-art verifier. And lastly, the algorithms should be optimized for performance since the software and

programs that are being built are only getting bigger in size and more complex. Ultimately, performance as noted by the team that developed NITWIT [ŠBK20] will be the focal point which sets you apart.

On a side note, throughout this dissertation, we have found ourselves in unprecedented times. There were various aspects in the process of completing this dissertation that had to be adapted in order to ensure that progress was still being made. We are fortunate enough to live in a connected world, where non-physical communication is easy. Times like this only reinforces the need for software security and software verification is one of many ways we could go about it.

Bibliography

[27017] ISO/IEC 27000:2017. Information technology - security techniques - information security management systems - overview and vocabulary (iso/iec 27000:2016). Technical report, BSI, 2017.

[AM08] O. H. Alhazmi and Y. K. Malaiya. Application of vulnerability discovery models to major operating systems. *IEEE Transactions on Reliability*, 57(1):14–22, 2008.

[AMR07] O.H. Alhazmi, Y.K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.

[BDD+15] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 721–733, New York, NY, USA, 2015. Association for Computing Machinery.

[Bei90] Boris Beizer. *Software testing techniques*. International Thomson computer Press, 2 edition, 1990.

[Bei95] Boris Beizer. *Black box testing: techniques for functional testing of software and systems*. Wiley, 1995.

[Bey20] Dirk Beyer. Advances in automatic software verification: Sv-comp 2020. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 347–367, Cham, 2020. Springer International Publishing.

[Bin15] Andrew Binstock. Java's 20 years of innovation. `https://www.forbes.`

`com/sites/oracle/2015/05/20/javas-20-years-of-innovation/`, May 2015.

[BS20] Dirk Beyer and Martin Spiessl. Metaval: Witness validation via verification. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 165–177. Springer, 2020.

[CE81] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching-time temporal logic. In Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1981.

[CKK+18] Lucas C. Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtík. JBMC: A bounded model checking tool for verifying java bytecode. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 183–190. Springer, 2018.

[CKS19] Lucas C. Cordeiro, Daniel Kroening, and Peter Schrammel. JBMC: bounded model checking for java bytecode - (competition contribution). In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 219–223. Springer, 2019.

[CWE20] MiTRE CWE. 2020 cwe top 25 most dangerous software weaknesses. `https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html`, 2020.

[DKW08] Vijay D'silva, 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):1165–1178, 2008.

[Han19] Derek Handova. What are the different types of security vulnerabilities? https://www.synopsys.com/blogs/software-security/types-of-security-vulnerabilities/, August 2019.

[Her10] Pete Herzog. *Open Source Security Testing Methodology Manual (OSSTMM)*. Institute for Security and Open Methodologies, 3 edition, 2010.

[Het88] Bill Hetzel. *The complete guide to software testing*. QED Information Sciences, 2 edition, 1988.

[HM04] J. Heffley and P. Meunier. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? In *37th Annual Hawaii International Conference on System Sciences*, 2004.

[IBM] IBM. The jit compiler. https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jit_overview.html.

[IST20a] ISTQB. Advanced level security tester - istqb® international software testing qualifications board. https://www.istqb.org/certification-path-root/advanced-security-tester.html, 2020.

[IST20b] ISTQB. Istqb glossary. https://glossary.istqb.org/en/term/defect-3, 2020.

[Kel19] Tom Kellermann. If your home is getting smarter, don't leave it vulnerable to hackers: Cyber strategist. https://www.cnbc.com/2019/11/30/how-to-defend-your-smart-home-from-hackers-after-black-friday-buys.html, 2019.

[Kri14] Paul Krill. 4 reasons to stick with java – and 4 reasons to dump it. https://www.infoworld.com/article/2687995/4-reasons-to-stick-with-java.html, 2014.

[KRS19] Temesghen Kahsai, Philipp Rümmer, and Martin Schäf. Jayhorn: A java model checker. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 214–218, Cham, 2019. Springer International Publishing.

[LTW+06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? *Proceedings of the 1st workshop on Architectural and system support for improving software dependability - ASID '06*, pages 25–33, 2006.

[Luo01] Lu Luo. *Software testing techniques: Technology Maturation and Research Strategy*. Institute for Software Research International, Carnegie Mellon University, 2001.

[Mar14] Robert A. Martin. *Non-Malicious Taint: Bad Hygiene is as Dangerous to the Mission as Malicious Intent.* CrossTalk. MiTRE, 2014.

[MiT09] MiTRE. Cwe-787: Out-of-bounds write. `https://cwe.mitre.org/data/definitions/787.html`, 2009.

[Moc] Mockito. Mockito framework site. `https://site.mockito.org/`.

[Ora] Oracle. What is java and why do i need it? `https://java.com/en/download/faq/whatis_java.xml`.

[Ora17] Oracle. Java security overview. `https://docs.oracle.com/javase/9/security/java-security-overview1.htm`, 2017.

[Ora19] Oracle. About the java technology. `https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html`, 2019.

[oSVSC19] International Competition on Software Verification (SV-COMP). Collection of verification tasks. `https://github.com/sosy-lab/sv-benchmarks`, 2019.

[oSVSC20] International Competition on Software Verification (SV-COMP). 9th intl. competition on software verification. `https://sv-comp.sosy-lab.org/2020/index.php`, 2020.

[San19] Sanket. Exponential cost of fixing bugs. `https://deepsource.io/blog/exponential-cost-of-fixing-bugs/`, 2019.

[ŠBK20] Jan Švejda, 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 40–57, Cham, 2020. Springer International Publishing.

[SHW$^+$20] Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. Java ranger at sv-comp 2020 (competition contribution). In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–397, Cham, 2020. Springer International Publishing.

[SW11] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2011.

[Tan20] Vi Lynn Tan. msc-project — github. `https://github.com/vilynntan/msc-project`, 2020.

[Tea] GraphML Team. The graphml file format. `http://graphml.graphdrawing.org/`.

[Tec18] TechTerms. Bytecode definition. `https://techterms.com/definition/bytecode`, 2018.

[Tun20] Abi Tyas Tunggal. What is a vulnerability? `https://www.upguard.com/blog/vulnerability#causes`, May 2020.

[Tur20] Steve Turner. 2020 data breaches — the worst so far. `https://www.identityforce.com/blog/2020-data-breaches`, January 2020.