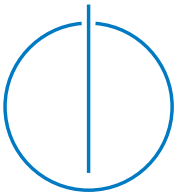
Fakultät für Informatik
Technische Universität München

Dissertation in Informatik

Scalable Greybox Fuzzing for Effective Vulnerability Management

Saahil Ognawala



Fakultät für Informatik
Technische Universität München
Lehrstuhl IV - Software and Systems Engineering

Scalable Greybox Fuzzing for Effective Vulnerability Management

Saahil Ognawala

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Tobias Nipkow, Ph.D.
Prüfer der Dissertation:	
1.	Prof. Alexander Pretschner, Ph.D.
2.	Prof. Cristian Cadar, Ph.D., Imperial College London, United Kingdom

Die Dissertation wurde am 03.07.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.12.2019 angenommen.

Acknowledgements

The struggle itself towards the heights is enough to fill a man's heart. One must imagine Sisyphus happy.

Albert Camus, *The Myth of Sisyphus*

It would not have been possible for me to undertake the tasks entailing this thesis to completion, alone. The following people came, and stayed, like a blessing.

I would like to express my sincere gratitude to Alex for, basically, leading by example. My assurance in my own capabilities was sustained through the knowledge that my advisor believed that our goals were achievable, measurable and significant enough to justify the efforts of a Ph.D. thesis. His reviews and insights were directly responsible for taking this work to an admirable level. My thanks go to my mentor, Martìn, for allowing my naïvité to not annoy him and helping me bring my dissertation ideas to a stage where I could effectively build upon them. All of my past and present colleagues at TU Munich, Traudl, Florian, Dominik, Benni, Enrico, Tobias, Prachi, Alexander, Kristian, Sebastian, Mojdeh, Alei, Severin, Marcus, Patrick, Florian, Ehsan, Mohsen, Amjad, Ana, Thomas and Daniel, deserve special mention for providing an exceptional environment for me to work and play everyday. Thanks to Hafiz, Thomas and Sebastian for reviewing parts of this thesis. I'm indebted, additionally, to the many brilliant master's and bachelor's students, and hiwis who chose to work with me and taught me so much.

I would also like to thank my second supervisor, Cristian, for hosting me at Imperial College London and providing useful and actionable feedback ever since we started our collaboration. My short stay with his research group and interactions with colleagues there provided me with an opportunity to consider my own research and work-ethics from a new and interesting perspective.

Along the way on my Ph.D. journey, I was lucky to have some excellent people as friends who listened to me, and encouraged me to do my best and look at the bigger picture more often. As good fortune would have it, I got to share many days and evenings chatting with Pranjal, and that time was indispensable.

I consider myself entirely a product of my upbringing and, for that, I have my parents to thank. For me, they are the most inspirational couple and parents in the world, and I hope to some day emulate them. My brother, Haaroon, who carries my mother's light in him deserves my thanks for his patience and understanding. I'm proud to be his older brother. Special thanks to my step-mother and sister who agreed to join in the joy with open hearts.

My best friend, Barbara, and I started our Ph.D. journeys together and along the way shared much more than just ideas, frustrations, sympathies, love, invaluable time and wedding rings. No amount of my palak paneer is enough to match her warmth, but I'll keep trying.

Zusammenfassung

Der Stand der Technik bezüglich der Erkennung von Schwachstellen besteht im Wesentlichen aus zwei Arten von Analysen - statischen und dynamischen. Wie die Namen schon andeuten, führen statische Analysemethoden nicht die getestete Programme mit realen Inputs aus, sondern verlassen sich auf Techniken wie beispielsweise abstrakte Interpretation, Daten- oder Informationsflussanalyse und unsichere Codierungsmuster im Programm. Dynamische Analysetechniken dagegen führen das zu testende Programm mit Hilfe von Testfällen aus, die automatisch oder manuell generiert wurden, um eben diese Schwachstellen auszufüren und dabei Fehler wie z.B. Programmabstürze zu beobachten. Während allerdings die statische Analyse dafür berüchtigt ist zu viele "false-positives" zu produzieren (d.h. Schwachstellen, die bei realistischen Eingaben nie ausgelöst werden), ist die dynamische Analyse oft unzureichend in Bezug auf strukturelle Abdeckung und produziert daher viele "false-negatives" (d.h. Schwachstellen, die nicht als solche erkannt wurden). Viele Untersuchungen haben in der Vergangenheit gezeigt, dass dies zumindest für dynamische symbolische Ausführung und auch für Fuzzing zutrifft.

In dieser Dissertation stellen wir eine skalierbare dynamische Analyse zum Auffinden und Berichten von Schwachstellen in mittleren bis großen Softwaresysteme vor. Bei diesem Ansatz haben wir zunächst automatisch Komponenten eines Programms isoliert, das in einer beliebigen Programmiersprache geschrieben wurde, und entfernen damit deren Zugangsbedingungen. Diese isolierten Komponenten werden dann mit drei Modi der dynamischen Analyse untersucht - symbolische Ausführung, Fuzzing und eine neuartiges Greybox-Fuzzing Methode auf Basis einer aktiven Sättigungsüberwachung. Um die Erreichbarkeit der entdeckten Schwachstellen aus den übergeordneten Komponenten zu zeigen, schlagen wir schließlich eine neuartige kompositionelle Analysetechnik vor, die die Zusammenstellung von Analyseergebnisse und gezielte symbolische Ausführung kombiniert.

Der letzte Teil dieser Dissertation behandelt ein Framework zur Bewertung von Schwachstellen, welche auf Heuristiken basiert, die aus dem Bug-Repository- und Code-Mining abgeleitet wurden. Dies beinhaltet einen anpassungsfähigen Mechanismus, der verschiedene semantische und entwicklungsspezifische Einflussfaktoren berücksichtigt um die Priorität von Schwachstellen vorherzusagen und Entwicklern bei der Priorisierung von Bugs zu unterstützen. Mit einer systematischen Auswertung hinsichtlich Effizienz und Effektivität belegen wir, dass unser integriertes Framework für Entwickler effektiver ist in Bezug auf die Verwaltung von Schwachstellen als andere aktuelle Techniken.

Abstract

The state-of-the-art in vulnerability detection consists mainly of two styles of analyses – static and dynamic. As the names suggest, static analysis methods do not execute the program-under-test with real inputs, but instead rely on techniques such as abstract interpretation, data or information flow analysis, and unsafe coding-patterns to predict vulnerabilities in the program. On the other hand, dynamic analysis techniques execute the program-under-test with test-cases, generated automatically or manually, to trigger vulnerabilities and observe failures such as program crash. However, while static analysis is infamous for producing too many "false-positives" (i.e. vulnerabilities that will never trigger with any realistic input to the program), dynamic analysis often falls short in terms of structural coverage and, hence, produces many "false-negatives" (i.e. vulnerabilities which are not flagged as such). Many investigations have shown in the past that this is true for, at least, dynamic symbolic execution and fuzzing.

In this doctoral thesis, we will introduce a scalable dynamic analysis for discovering and reporting low-level vulnerabilities in medium to large-scale software. In this approach, we, first, automatically isolate components of a program, written in an arbitrary programming language, thereby removing their entry conditions. These isolated components are, then, analysed for vulnerabilities using three modes of dynamic analysis – symbolic execution, fuzzing and a novel greybox fuzzing method based on active saturation monitoring. Then, to determine the reachability of the discovered vulnerabilities from higher-level components, we propose a novel compositional analysis technique involving collation of analysis results and targeted symbolic execution.

The final part of this thesis discusses a vulnerability assessment framework, based on heuristics derived from bug-repository- and code-mining. It contains an adaptable mechanism that considers various semantic and development-specific impact factors to predict priority for vulnerabilities to aid developers in the bug triage process. With a systematic evaluation for efficiency and effectiveness, we show that our integrated framework is more effective in managing vulnerabilities for developers than state-of-the-art techniques.

Outline of the Thesis

Chapter 1: Introduction

This chapter presents an introduction to the topic and the fundamental issues addressed in this thesis. It discusses the underlying techniques in the state-of-the-art, problems associated with them, our approach to solving the problems and research questions to evaluate them.

Chapter 2: Symbolic Execution

This chapter presents the essential background on *symbolic execution*, one of the competing, as well as contributing, techniques of dynamic analysis to be discussed in this thesis.

Chapter 3: Guided Fuzzing

This chapter presents the essential background on *guided fuzzing*, one of the competing, as well as contributing, techniques of dynamic analysis to be discussed in this thesis.

Chapter 4: Hybrid Symbolic Execution and Fuzzing

This chapter presents a systematic mapping study to survey other *hybrid* symbolic execution and fuzzing techniques, possibly similar to our own, that have been proposed in the past. Parts of this chapter have previously appeared in [103], where the author of this thesis was the first author.

Chapter 5: Isolating Program Components

This chapter describes the first step of vulnerability discovery – automatically isolating components of the program-under-test to analyse them with dynamic analysis. Parts of this chapter have previously appeared in [102] and [101], where the author of this thesis was the first author.

Chapter 6: Analysing Isolated Components

This chapter describes the second step in the discovery process – parallel analysis of isolated components using symbolic execution, fuzzing and a novel greybox fuzzing approach. Parts of this chapter have previously appeared in [102], [99] and [101], where the author of this thesis was the first author.

Chapter 7: Compositional Analysis

This chapter describes the final step in the discovery process – compositional analysis of the vulnerable components discovered in the previous steps, to determine if it is feasible to exploit them. Parts of this chapter have previously appeared in [102], [104] and [101], where the author of this thesis was the first author.

Chapter 8: Evaluating Vulnerability Discovery

This chapter evaluates the vulnerability discovery part of our thesis by comparing their effectiveness and efficiency to those of various comparable dynamic analysis techniques.

Parts of this chapter have previously appeared in [101], where the author of this thesis was the first author.

Chapter 9: Assessing Discovered Vulnerabilities for Effective Triage

This chapter introduces some ideas combining output generated by the vulnerability discovery steps with various impact factors and heuristics to assess the discovered vulnerabilities. Parts of this chapter have previously appeared in [97], where the author of this thesis was the first author.

Chapter 10: Case Study – Machine Learning Based Score Predictor

This chapter instantiates some of the ideas presented in Chapter 9 in a case study involving vulnerability discovery, a vulnerability scoring scale and a prediction model generated by machine learning. Parts of this chapter have previously appeared in [97], where the author of this thesis was the first author.

Chapter 11: Conclusion

This chapter concludes this thesis by discussion resolutions to our original research questions, contributions of this thesis to the state-of-the-art, limitations of our work and future work.

N.B.: *Multiple chapters of this dissertation are based on different publications authored or co-authored by the author of this dissertation. Such publications are mentioned in the short descriptions above. Due to the obvious content overlapping, quotes from such publications within the respective chapters are not marked explicitly.*

Contents

Part I

Introduction and Background

1 Introduction

This chapter presents an introduction to the topic and the fundamental issues addressed in this thesis. It discusses the underlying techniques in the state-of-the-art, problems associated with them, our approach to solving the problems and research questions to evaluate them.

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*
>
> Donald E. Knuth

The growth in complexity of software today [12] has neatly coincided with the growth in advanced techniques available to malicious hackers willing to cause severe economic [136], social [86] and democratic [111] damage. Tim Berners-Lee, the inventor of the world-wide-web (WWW), reportedly said in 2018 [17] that he was "devastated" by the effects of malpractices based on his proposed open design of the internet, in the form of illegal surveillance, data theft and democratic hijacking. Berners-Lee proposed [18] that the large-scale problems facing, in particular, the web be seen as *bugs* in existing software systems and be treated as such.

Security threats in the web sphere only form the tip of the iceberg when it comes to the present day challenges faced by general-purpose software. Experts have recognised myriad forms of problems associated with embedded-systems, information-systems, business and scientific software that continuously call for improved processes and methods in the software development lifecycle. In this thesis, we tackle the particularly damaging issue of software vulnerabilities, especially ones introduced inadvertently during the development process. By describing the technical shortcomings of existing methods to reduce vulnerabilities and increase trustworthy-ness of general-purpose software, we motivate the need for a more pragmatic mix of a sound, but fast and feasible, program analysis framework to catch implementation flaws early in the development cycle. We present a series of methodologies that are scalable by design to reach deep regions of software which, if not checked, may contain vulnerabilities that can be realistically exploited by malicious actors. We also aim to prove the superiority of these scalable methods in terms of effectiveness and efficiency, as compared to state-of-the-art techniques in the field.

1.1 Software Vulnerabilities

To set the stage for the rest of the thesis, we start this section with a brief introduction and history of software vulnerabilities and how they have been studied over the course of software engineering research.

```
1 class CustomArray {
2     private int [] ar;
3
4     public int get(int i) {
5         return (ar[i]);
6     }
7 }
```

Listing 1.1: *Illustration of bug vs. vulnerability*

1.1.1 Vulnerabilities vs. Bugs

Two important and related concepts that reflect common problems in software are *bugs* and *vulnerabilities*. Radatz et al. in the *IEEE Standard Glossary of Software Engineering Terminology* [117] defines a bug as the same as *error* and *fault*, as follows

Definition 1.1.1. *(Bug) An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program.*

In the context of modern-day software, vulnerabilities can be defined as a specialization of the above definition of bugs. Krsul [77] defines vulnerability as a

Definition 1.1.2. *(Vulnerability) A bug that violates an (implicit or explicit) security policy is called a vulnerability.*

This definition has been built upon by many recent studies [31] to show that bugs and vulnerabilities are, both, often introduced into a software due to human factors, such as under-specification. To illustrate bugs, vulnerabilities and failures, consider the C++ class *CustomArray* in Listing 1.1 and function *get* that returns the element at index i of a private array ar.

The function *get* does not check if the index i lies within the bounds of the array ar. This is a software bug, in the sense that the actual result of calling $get(i)$, where $i \geq len(ar)$ is not the same as the expected result, viz. a graceful error message saying something like "`Array does not currently contain at least i elements`". An error message is also expected for $i < 0$. However, this is also a C++ specific vulnerability, viz. *buffer overflow*. This vulnerability, in the case of C++ programming language, is called as such because it violates an implicit security policy that no instruction in the program shall write outside the bounds of its allocated memory. In this thesis, we will use the terms "vulnerability" and "vulnerable instruction" interchangeably because all discussions of vulnerabilities in this thesis will be direct references to the precise instruction/line-of-code where the vulnerability exists. If this vulnerability were to be exploited, the program would crash as a result. A program crash is a *failure*. We adapt the following definition of failure from [117]

Definition 1.1.3. *(Failure) The inability of a program or its component to perform its required functions within specified performance requirements, due to the presence of a bug, is called a failure.*

Program crash is an instance of a failure that may manifest due to the presence of a buffer overflow vulnerability.

1.1.2 Practical Implications of Software Vulnerabilities

If vulnerabilities exist in software they might affect the confidentiality, integrity or availability of the underlying assets of an organization. The annual OWASP (Open Web Application Security Project) *Top 10* document [55] lists the vulnerabilities that have had the worst effect on web applications in any given year. Of these, SQL injection [64] was the top application security risk in 2017. SQL injection vulnerability allows an attacker to gain unauthorised access to the underlying database and insert, update, delete or view entries in its tables. Most often, this vulnerability occurs because the input accepted by a web-page is not sanitised for SQL keywords before being executed on the database. Similarly, almost all of the top 10 vulnerabilities listed in this document [55] can, in some way, be traced back to improper handling of legal, but malicious, input being accepted by programs or Application Programming Interfaces (APIs). We generalise this observation by stating that, regardless of whether the program is a web application or a command-line application, vulnerabilities manifest as a result of wrongly or incompletely enforcing security principles relevant to the programming context.

Finally, we need to consider the following practical aspect of vulnerabilities – *In what ways can vulnerabilities be exploited during the use of the software?* In addition to failures caused by unintentional usage by benign users, the more severe case to consider here is that of a malicious *Attacker*. Attackers are persons or entities carrying out an attack by following a fixed workflow to exploit vulnerable software. An *attack* or *exploit* [*Noun*] [128] is an automated script, malware or a series of reproducible manual steps that can lead to a failure due to a vulnerability. The act of successfully carrying out an attack is called *exploiting* [*Verb*] the software.

The amount of vulnerability disclosures in user-facing applications has dramatically increased in recent years [128], even though explicit sanity-checks on user-inputs have become more common. This indicates that attackers have also adopted more sophisticated workflows to exploit vulnerabilities, almost like precise lock-picking. In particular, reverse engineering, decompilation and debugging tools allow attackers [68] to craft precise input payloads to navigate complicated sanity-checks in software and, if possible, exploit vulnerabilities.

1.2 Hardening Software

Having discussed the prevalence of vulnerabilities in general-purpose software, let us now discuss some existing ways to proactively ensure that shipped programs do not contain vulnerabilities. Common hardening practices against vulnerabilities include access control, security policy enforcement, cryptography, code obfuscation and others. Access control [120] is a limited protection framework that uses *access control matrices* to describe the rights of users over system resources. Any attempt to access resources that are not explicitly allowed by an access control matrix is deemed an *access control violation*, and corresponding prescriptive remedial measure may be taken. *Security policies* [19] are a set of statements partitioning the states of a program based on whether those states are *authorised* and *secure*. The definitions of these terms may depend on the context of

development, usage, deployment, and other such factors. *Cryptography* [47] is a rigorous way to secure information systems' resources that intends to make it mathematically intractable for potential attackers to get access to them. The goal here is to keep the encrypted information (information on which cryptography has been applied) unknowable to external parties that do not have access to the correct *cryptographic keys* used to encrypt the information. *Integrity protection* [4] is also a popular method that puts checks on a program's control-flow (control-flow integrity) or data-flow (data-flow integrity) at software or hardware (e.g. using trusted platform modules (TPM) [132]) level to ensure that the protected entities are not tampered with by a malicious actor. Finally, software obfuscation [11] is an automated technique to transform the program-under-test to make it harder to understand than the original program. The goal of obfuscation [52] is to mitigate reverse-engineering exploits by concealing the underlying logic or data of the program (including vulnerabilities).

Some other software hardening methodologies that we have not discussed here are employment of secure-coding good practices in organizations, secure third-party implementations of insecure libraries and adoption of security standards in software practices. However, the question of proactively preventing vulnerabilities in software is not the central goal of this thesis. We have only briefly discussed here some existing preventive measures that have been proposed to reduce bugs and vulnerabilities.

1.3 Software Testing

Measures to proactively harden software may effectively prevent vulnerabilities from manifesting. However, in real-world development scenarios, it is often impossible to know apriori the many ways in which an *actual* program (as opposed to a, *designed* but not-implemented program) may lead itself into a vulnerable state during a real usage. The most commonly employed methodology to ensure that implemented software conforms to specifications is *software testing*. According to Pan [107], software testing is ...

> any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.

One of the main goals of software testing is to show the correct behaviour of the software w.r.t. the specification. Typically, this involves writing *test cases* describing a particular, or a class of, inputs to the software and the expected output for that input. The expectation of *running the test* on the software with a listed input is that the software will produce the corresponding expected output. If the expected result is *observed*, the test is said to have *passed* for that input, and if the expected result is not observed, then it is said to have *failed*. The responsible mechanism for indicating whether a test has passed or failed is called a *test-oracle*.

We note from the above brief description that the correctness of software depends solely on the knowledge of what it is *supposed to* do for particular sets of inputs. This knowledge is usually fed to the testing process by an agent that is familiar with the specifications

of the software, as listed *before* development. Most recent advancements in the field of software testing are related to creating or improving automation of such agents, so that

1. reliance on manual intervention in testing processes is reduced, and
2. testing processes can be replicated and automatically rerun when, either the specification or the software changes.

Other goals of software testing may be [107] demonstrating reliability, performance or compliance to security principles.

Software testing can be categorised in several ways, but, for this thesis, we have adapted the *security testing* classification criteria, as defined by Bishop [19]. According to this criterion, testing methods may be classified as follows

1. *Blackbox testing* or, in terms of Bishop [19], *functional testing* is defined as a form of testing where the software or its components are probed to determine whether they produce the desired output for given sets of input, without receiving any feedback from the internal elements or background of the program itself.
2. *Whitebox testing* or, in terms defined by Bishop, *structural testing* usually probes the software-under-test to determine which *structural elements*, such as branching-conditions, source-code, algorithmic basic-blocks, respond to the input and produce the corresponding output.

Whitebox and blackbox testing, therefore, are two opposing viewpoints adopted by software testers to examine a software under test. Most existing testing methodologies in practice lie in between the black and white spectrum of this viewpoint classification, as we describe in the next section.

1.3.1 State-of-the-art in Automated Testing

The central idea behind testing is to determine whether the software complies with a specification. With manual testing, i.e. reliance on manual specification to generate test cases, even though software testers nowadays are trained to approach the software with an "attacker mindset" [114], they often miss input values that may exercise interesting, and often unaccounted-for, behaviour in the software, such as *edge cases* [151]. These rare behaviours are often a result of the application logic combined with peculiarities of the programming environment, which may be ignored in favour of more function requirements by the specification writers. As a result, these peculiarities may also be missed by software testers, who may only be following the explicit functional level specification. We call the scenario described above, where the testing process is not able to cover realistic, but rarely executed, program behaviour as *incomplete testing*.

Automated testing provides an antidote to the problem of incomplete testing by combining deductions based on the implemented software with invariants of the development environment, which ensure there are no vulnerabilities in the program. There exist various automated testing frameworks that let testers write security policies and other invariants in the form of formal specifications [23], state-machines [39] or other fault-modelling methodologies [76]. These techniques, in addition to policy encoding, are also usually tailored to a

programming language and the program's specifications. In the following paragraphs, as in the rest of this thesis, we will delve deeper into two most popular automated testing techniques in the state-of-the-art, one blackbox and one whitebox - fuzzing and symbolic execution, respectively.

Fuzzing

Fuzzing (or *blackbox fuzzing*), as defined by Sutton et al. [134] is an automated method of providing unexpected input to a program and monitoring for exceptional behaviour, such as program crash. Fuzzing does not rely on any information about the internal constraints or branches of the program but, instead, makes use of *mutations* of *seed inputs* to execute the program under test and observe the return values and the external state of the system. By making such observations, a fuzzer may correlate input values to different functional behaviour of the program. In practice, fuzzers employ advanced strategies for input mutation, based on heuristics and inspirations from other fields such as genetic algorithms. The key idea is to select high-value inputs that promise a better reward in terms of new functional behaviour execution or exceptional behaviour. AFL [2], Peach [109] and Honggfuzz [135] are some of the most popular off-the-shelf fuzzers in industry and academia, but there exist many others in the state-of-the-art for various target programs. We will discuss fuzzing in more detail in Chapter 3.

Symbolic Execution

Symbolic execution [71] is a whitebox dynamic analysis technique for software testing and test case generation. It collects constraints based on *symbolic program inputs*, representative of paths in the program. Some of these paths may lead to unhandled exceptional behaviour in the program (such as buffer overflows) leading to a program crash. Solutions to the collected constraints provide concrete values for symbolic inputs that may exercise that path in the program. Symbolic execution has been shown [28, 62] to be capable of extracting complex path conditions to edge-cases and exceptions where other methods, like random testing, have failed to find potential vulnerabilities. KLEE [26], Sage [63] and JPF-SE [7] are some examples of practical symbolic execution engines for programs written in various languages. We will discuss symbolic execution in more detail in Chapter 2.

1.3.2 Problems With the State-of-the-art

The software testing techniques, blackbox and whitebox, that we have discussed so far belong to the so-called, *dynamic analysis* [5, 54] category of program analysis methods, meaning that the program-under-test, or some representation of it, needs to be executed in the process of its analysis. We differentiate this from *static analysis* techniques, such as abstract interpretation [44], where some structural or syntactic elements of the programs are analysed instead of executing the program actually [51].

In this thesis, we will focus on dynamic analysis and, particularly, issues faced by symbolic execution and fuzzing, with a minor focus on the drawbacks of static analysis techniques.

Concretely, the existing problems with symbolic execution and fuzzing may be summarised as follows [103][1].

P1 *Path explosion in symbolic execution*
While symbolic execution is good at covering new branches, it suffers from the well-known path-explosion [29] problem. The number of paths that symbolic execution has to explore increases exponentially with the number of branching-conditions that depend on a symbolic input. As evident by the Halting problem, determining whether an arbitrary program may terminate for a symbolic input is undecidable because there may be input-dependant loops. Due to the path explosion problem, symbolic execution engines can often only analyse programs incompletely, thereby generating an insufficient number of test cases.

P2 *Constraint solving bottleneck in symbolic execution*
Symbolic execution engines suffer from the execution bottlenecks introduced by the constraint solvers. Symbolic execution engines typically use *satisfiability modulo theory* (SMT) [75] solvers as the underlying decision procedure [24] to determine satisfiability of the path conditions (conjunction of branching conditions). However, the time taken for SMT solvers to return depends heavily on the size and complexity of the path conditions. Due to this reason, symbolic execution is severely affected by the scalability problem of the constraint solvers.

P3 *Low path-coverage in fuzzing*
Fuzzing also suffers from insufficient coverage due to its reliance on manual seed inputs and lack of insight into the internals of the program. As a result, fuzzing is repetitive (executes the same paths repeatedly) and incapable of covering branches whose conditions are hard to pass by randomly mutated inputs. As a result, fuzzing tools are unable to generate diverse test cases for many paths in the program. In terms of vulnerability discovery, fuzzing is also unable to find many vulnerabilities in deep parts of the program that are hard to reach for randomly mutated inputs.

P4 *Lack of prioritization*
Vulnerabilities discovered by state-of-the-art dynamic or static analysis techniques are, often, reported without any assessment to aid in bug-triage by development and testing professionals. Without such an assessment framework, it may be challenging to prioritise fixing activities in the presence of a large number of reported vulnerabilities. The main reason for this is because, in the absence of compositional analysis, possible usage of a program's components cannot be replicated.

These problems may exacerbate even more if the components may be reused in an unforeseeable way in the future where the vulnerabilities may have a more considerable impact.

[1] We will discuss these problems in more detail in Chapter 3, and Chapter 2

1.4 Thesis Overview

Having described in brief the existing software testing methods, their classification based on the tester's point-of-view, practical approaches and problems associated with them, we finally turn our focus towards organizing this landscape to enable us to design some solutions to the problems with the state-of-the-art listed above.

1.4.1 Research Questions

We start by listing the main research questions that motivate this thesis. These questions are all related to the state-of-the-art vulnerability detection through dynamic software analysis methods, as described in Section 1.3.1.

RQ1*: What are the concrete shortcomings and gaps in the state-of-the-art in solutions related to symbolic execution and fuzzing?*

In this thesis, we aim to answer this question in Part I and, more specifically, in Chapter 2, Chapter 3 and Chapter 4.

RQ2*: How is structural coverage of components related to vulnerability discovery in them and how may dynamic analysis exploit it?*

In this thesis, we aim to answer this question in Part II and, more specifically, in Chapter 5, Chapter 6 and Chapter 8

RQ3*: How may the exploitability of discovered vulnerabilities be determined using the compositional nature of a program?*

In this thesis, we aim to answer this question also in Part II and, more specifically, in Chapter 7 and Chapter 8.

RQ4*: For all discovered vulnerabilities, how may we prioritise the process of fixing them?*

In this thesis, we aim to answer this question in Part III.

1.4.2 Overview of Solution

Figure 1.1 depicts a high-level overview of the greybox fuzzing framework being proposed in the later chapters of this thesis. To tackle the problems listed in Section 1.3.2, we propose a compositional solution based on symbolic execution and fuzzing in the following steps.

1. Create an automated procedure to isolate components of a program. The goal of this first step is to allow symbolic execution and fuzzing to analyse the isolated components directly, without first solving the branching conditions in program entry points. This step will be described in detail in Chapter 5.

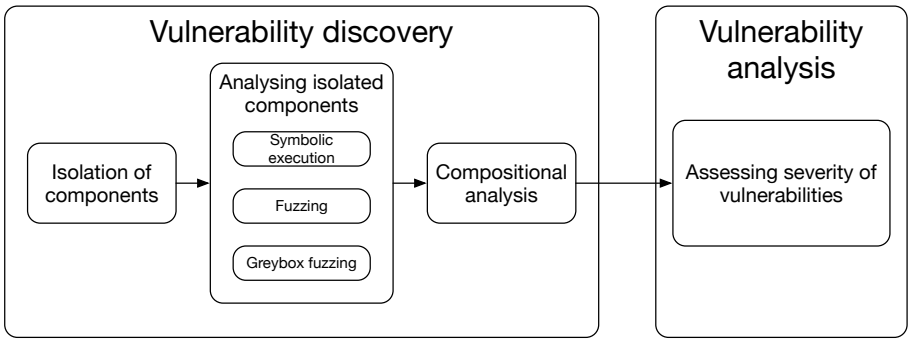


Figure 1.1: *Overview of the scalable greybox fuzzing solution*

2. Dynamically analyse isolated components generated in the first step to finding vulnerabilities in them. The analysis technique can be symbolic execution, fuzzing or a novel hybrid technique that combines them both. The goal of this first step is to increase the structural coverage of a program. This step will be described in detail in Chapter 6.
3. Perform compositional analysis of discovered vulnerabilities to determine whether it is feasible to exploit them from other components interacting with it. The goal of this step is to determine the extent of infection of a discovered vulnerability. This step will be described in detail in Chapter 7.
4. Prioritise the discovered vulnerabilities for effective bug triage. The motivation behind this step is to minimise adverse effects of a discovered vulnerability on the underlying assets of a software. This step will be described in detail in Part III, including a general framework for vulnerability assessment (Chapter 9) and a case study where we instantiated this framework for some real-world open-source programs and libraries (Chapter 10).

Symbolic execution and fuzzing will be combined by monitoring saturation indicators (Chapter 6) and cross-feeding inputs between the two. Compositional analysis will be performed by using static code interpretation to isolate individual components and, later, join them. We propose to prioritise the vulnerabilities using various heuristics learned from the development environment and results of the compositional analysis phase.

1.4.3 Contribution

With the completion of the tasks entailing the goals of this thesis, we claim to have made the following contributions w.r.t. to the gaps in research in state-of-the-art in vulnerability discovery and analysis methods

1. We propose the design and implement a **compositional analysis** framework that applies dynamic analysis methods to isolated components of a program to find vulnerabilities in them and, then, determines the feasibility of the vulnerabilities using targeted path search methods. This contribution closes a gap in the state-of-

the-art, where existing methods can dynamically analyse programs only from their entry points.
2. For analysing isolated components, we describe and implement a novel dynamic analysis method that **improves the coverage of existing fuzzers by combining it with symbolic execution** as follows – performing symbolic execution to generate seed inputs for fuzzing. This contribution closes a gap in the state-of-the-art, where existing fuzzing techniques often do not employ symbolic execution, which can guarantee non-redundant path coverage, to guide their input mutation strategies.
3. For analysing isolated components, we describe and implement a novel dynamic analysis method that **improves the coverage of existing concolic execution engine by combining it with fuzzing** as follows – performing fuzzing to cover most of the easy-to-reach branches in a program and, then, using the generated inputs for concolic execution so as to not call the constraint solver for branches covered already. By combining this contribution with the previous point and repeating concolic execution and fuzzing till they saturate, we close a gap in the state-of-the-art, where existing symbolic execution and fuzzing techniques do not utilize each other's technical aspects to increase effectiveness and performance of their dynamic analyses.
4. For the discovered vulnerabilities, we describe a **generic assessment framework** and instantiate it in a case study with several open-source programs. This contribution closes a gap in the state-of-the-art, where existing techniques do not provide a way to prioritise vulnerabilities to make it easier for the developers and testers to triage them effectively.

In Chapter 4, we will discuss in detail how the above gaps in research were identified with a systematic mapping study. Parts of the contributions of this thesis have previously appeared in the following peer-reviewed publications, co-authored by the author of this thesis:

1. S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer. "MACKE: compositional analysis of low-level vulnerabilities with symbolic execution". In: *International Conference on Automated Software Engineering*. 2016
2. S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner. "Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach". In: *Proceedings of the Symposium on Applied Computing*. ACM. 2018
3. S. Ognawala, R. N. Amato, A. Pretschner, and P. Kulkarni. "Automatically assessing vulnerabilities discovered by compositional analysis". In: *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*. ACM. 2018
4. S. Ognawala, A. Pretschner, T. Hutzelmann, E. Psallida, and R. N. Amato. "Reviewing KLEE's Sonar-Search Strategy in Context of Greybox Fuzzing". In: *1st International KLEE Workshop* (2018)
5. S. Ognawala, F. Kilger, and A. Pretschner. "Compositional Analysis Aided by Targeted Symbolic Execution". In: *arXiv preprint arXiv:1903.02981* (2019)

1.5 Structure

This dissertation is organised in four parts, as follows –

Part I is meant to introduce the readers to the underlying concepts of this thesis, viz. symbolic execution (Chapter 2), fuzzing (Chapter 3) and hybrid techniques that include both (Chapter 4). Then we describe the core contributions of this thesis in Part II and Part III. In Chapter 5, we describe an automated procedure to isolate components of a program, for us to be able to analyse them with dynamic analysis. In Chapter 6, we describe three dynamic analysis techniques for discovering vulnerabilities in isolated components, viz. symbolic execution, fuzzing and a novel hybrid technique based on active monitoring of coverage saturation. In Chapter 7, we describe an automated procedure to compositionally analyse discovered vulnerabilities and determine the extent of their infection. In Chapter 8, we describe the results of applying the above techniques on real-world programs, and answer relevant research questions about the efficiency and effectiveness of the approach.

In Chapter 9, we bring together the results from vulnerability discovery and describe some ideas on how we may automatically prioritise them based on various structural, organisational and asset factors. We instantiate these ideas for a detailed case study in Chapter 10, where we describe a vulnerability assessment technique based on machine learning from some impact factors. We, finally, conclude the thesis in Chapter 11 and list some limitations and future work.

2 Symbolic Execution

This chapter presents the essential background on symbolic execution*, one of the competing, as well as contributing, techniques of dynamic analysis to be discussed in this thesis.*

Symbolic execution was first introduced by King [71] as a testing method to deterministically execute diverse paths in a program by automatically generating corresponding test-cases. In this chapter, we will describe symbolic execution by formalising symbolic inputs (as opposed to *concrete* inputs), program paths as constraints systems over symbolic inputs, some practical approaches for symbolic execution that have been developed over the years, problems associated with them and, finally, concluding notes on the state-of-the-art in symbolic execution.

For illustrating various concepts discussed in this chapter, we will refer to the running example of the C-program, as listed in Listing 2.1.

2.1 Symbolic Program Input

The difference between dynamic and static program analysis techniques is that during dynamic analysis, the program under test is executed, within or without, a controlled environment with test-cases. This description of dynamic analysis is important to understand the idea of inputs in symbolic execution.

For symbolically executing a program, we assume that it may be executed with *any* possible value of input, constrained only by the external constraints of the execution environment, such as the operating system or the syntax rules of the programming language. Concretely, an external constraint may be defined as follows.

Definition 2.1.1. *(External Constraint) Any constraint enforced on a program's input by an agent other than the program itself during or before the invocation of the program is called an* external constraint*.*

To enforce this weakest possible constraint ("*any possible input*"), symbolic execution executes the program with symbolic input, instead of concrete values. We may define symbolic input as follows.

Definition 2.1.2. *(Symbolic Input) An abstraction or a symbol representing the data expected by a program, that has no concrete value assignment and can take any value depending only on their types and external constraints of the program is called* symbolic input*.*

Let us denote a program's input by $\mathcal{I}(X)$, where X is an abstract representation of an interface of the given program[1].

[1] We will describe the concept of interfaces and, in general, program entry points, in Chapter 5.

```
1  int bar1(int c) {
2      if (c<3)
3          return (3/c); /*Maybe divide-by-zero*/
4      else
5          return 0;
6  }
7
8  int bar2(int d) {
9      if (d<50)
10         return 0;
11     else
12         return d;
13 }
14
15 int foo(int b, int c, int d) {
16     if (b==100)
17         return bar1(c);
18     else
19         return bar2(d);
20 }
21
22 int main(int argc, char** argv) {
23     int a, b, c, d;
24     a=atoi(argv[1]); b=atoi(argv[2]);
25     c=atoi(argv[3]); d=atoi(argv[4]);
26
27     if (a<1)
28         return 0;
29     else
30         return foo(b, c, d);
31 }
```

Listing 2.1: *Example C program.*

In the code in Listing 2.1, the data expected by the program is as follows – an integer, `argc`, denoting the number of command-line arguments and the array of command-line arguments, `argv`. Additionally, the variables a, b, c, d are directly assigned from $argv$, which implicitly means that these variables can also be made symbolic. Therefore, for the program in Listing 2.1,

$$\mathcal{I}(X) = \langle argc, argv, a, b, c, d \rangle \tag{2.1}$$

For symbolically executing this program, we need symbolic inputs, instead of concrete inputs, for assigning to each of $argc, argv, a, b, c, d$ in $\mathcal{I}(X)$.

2.2 Path Conditions

Based on logical predicates over symbolic inputs, we will now describe what path conditions are and how they are incrementally composed of branching conditions.

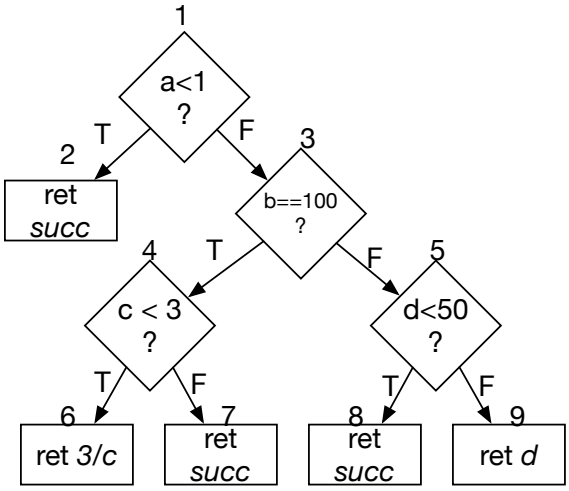


Figure 2.1: *Control-flow graph for C-program in Listing 2.1*

To define path conditions, we require, in addition to the definition of symbolic input, the definition of branching conditions over symbolic inputs.

Definition 2.2.1. *(Branching Condition) Branching condition is a first-order logic formula over symbolic input whose evaluation determines the next instruction set to be executed in the program execution.*

Local variables whose values depend, implicitly or explicitly, on symbolic program input may also be represented in terms of $\mathcal{I}(X)$ if they form a part of the branching condition. For an intuition, please refer to Figure 7.3, depicting the control-flow graph (CFG) lifted from the C-program listed in Listing 2.1. Given that the symbolic input table includes variables a, b, c and d, branching conditions here are $(a < 1)$, $(b == 100)$ and so on.

Let us denote a branching condition by the letter q. Then, formally speaking, q is a Boolean expression over α_i's, where, as discussed in Section 2.1, $\alpha_i \in \mathcal{I}(X)$.

Using the definitions of symbolic input and branching condition, we may now define path condition as follows

Definition 2.2.2. *(Path Condition) Path condition is a logical conjunction of branching conditions that are true for an execution path spanning from a program's entry point to an exit point.*

Typically, a program entry point may be the `main` function or a library's application programming interface (API), and exit point may be a `return` statement, a failure or assertion violations during execution.

Let us denote a path condition by the notation *pc*. Then, for every symbolic execution of a program,

$$pc_{initial} = True \tag{2.2}$$

Here, the subscript *initial* signifies the initial or starting state of the path condition. As discussed earlier, this path condition will be modified by a symbolic execution engine as the execution progresses and encounters branching conditions. Whenever a branching condition, such as `If-else` or `While`, with branching condition, q, is seen by the execution, *pc* is updated as follows

$$pc = pc \wedge r \tag{2.3}$$

where r can be q or $\neg q$.

Let us consider the CFG again in Figure 7.3. In Figure 2.2, we illustrate the step-by-step progression of the *pc* of one of the paths of the program, specifically the one starting from node 1 in Figure 7.3 and ending at node 7. The final path condition (due to execution ending because of `return` statement) for this path will be $(a \geq 1) \wedge (b == 100) \wedge (c \geq 3)$. Similarly, the path condition for another path in the program from node 1 to node 8 is $(a \geq 1) \wedge (b \neq 100) \wedge (d < 50)$, and so on for the other paths.

2.3 Constraint Solving

As explained in the previous section, path conditions are modified by continuously adding branching conditions (or negations thereof) till a program exit point is encountered. At this point, the path condition is sent to a constraint solver to generate test-cases that will execute the corresponding path in the program. Constraint solvers are, more specifically, decision procedures for determining satisfiability of logic formulae [28].

Definition 2.3.1. *(Decision Procedure for Propositional Logic (PL)) An algorithm that, after some finite iterations of computations, determines whether a propositional logic (PL) formula is satisfiable is called a decision procedure for satisfiability of the PL formula.*

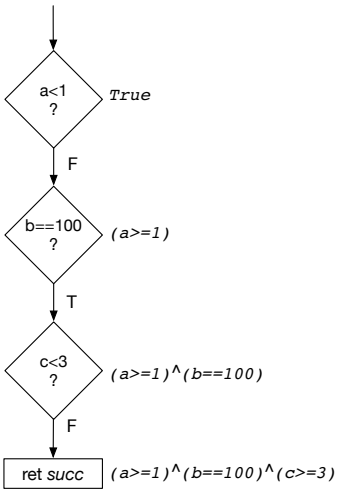


Figure 2.2: *How the path condition is updated during symbolic execution of the program in Listing 2.1*

We have adapted the above definition of decision procedures from [24]. The simplest example of a decision procedure is creating a truth-table[2] consisting of all the variables in the PL formula and checking if there is at least one row where the final column is $True$.

Different decision procedures, such as SAT [49] or SMT solvers [14], have been developed in the past to generate satisfiable models for theories at different expressiveness levels. However, considering the particular challenges and opportunities presented by the constraints generated by symbolic execution, combinations of decision procedures were required to solve for *union of theories* (limited to PL). Examples of individual constraints are – $x < 100$ and $f(x) \leq f(x-1)$. A combined decision procedure will take into account both these constraints and solve for their union (i.e. $(x < 100) \wedge f(x) \leq f(x-1)$)[3].

The underlying constraint solver of a symbolic execution engine, then, generates concrete test-cases that are guaranteed to execute the corresponding paths in the program, thereby providing a concrete model of the symbolic execution, i.e. assignments for symbolic input.

2.4 Symbolic Execution in Practice

So far, in this chapter, we have described the theory of symbolic execution. We will now provide some details on how these steps are implemented in various practical symbolic execution engines.

[2] A table, for a formula with n variables, with 2^n rows, each row assigning a unique combination of $True$ or $False$ assignments to the variables and determining the final result – $True$ or $False$.

[3] The particular example here of combination of the two theories is the simpler, albeit most commonly occuring in symbolic execution, form of constraint that is known as *conjunctive normal form* (CNF).

2.4.1 Concolic Execution

Symbolic execution, in theory, entails executing the program-under-test with symbolic inputs to collect constraints depending on branching conditions in the program. However, in practice, to perform dynamic analysis of a program, most symbolic execution engines [108] initiate the analysis process using concrete inputs to the program. A common strategy in this regard is a combination of concrete and symbolic execution, also known as *concolic execution*. Symbolic execution may be combined with concrete executions in the following ways

1. *Concrete seed inputs*: The analysis process proceeds with an initial set of inputs (*seed inputs*). Based on seed inputs, the concolic execution engine follows the execution by recording the branching decisions taken and, for every branch that is taken by the execution, remembers the negation of the branch to be used for symbolic execution in the subsequent iterations.
   Let us use the example of Listing 2.1 to illustrate this type of concolic execution. Suppose that the seed input for the program is "1 2 3 4". Recall from Equation (2.1) that the concrete form of the symbolic input I^S will be[4]

   $$I = \langle 1, 2, 3, 4 \rangle \tag{2.4}$$

   For the above concrete input, the path that will be followed in Figure 7.3 can be denoted by the path condition, *pc* will be $\neg(a < 1) \wedge \neg(b == 100) \wedge (d < 50)$. The concolic execution engine, then, will add the following path conditions to the list of paths yet to be explored while at nodes 1, 3 and 5 in Figure 7.3 respectively

   $$(a < 1) \tag{2.5}$$
   $$\neg(a < 1) \wedge (b == 100) \tag{2.6}$$
   $$\neg(a < 1) \wedge \neg(b == 100) \wedge \neg(d < 50) \tag{2.7}$$

   These yet-to-be-explored path conditions are, then, queued to be explored next by the concolic execution engine.
2. *Concrete function calls*: In the next variant, the concolic execution engine determines (e.g. using dynamic taint analysis [122]) those particular paths in the program that do not depend on symbolic input and, therefore, are executed concretely at all times. Another scenario where this is applicable is when calls to external functions and libraries need to be concretised if they are not modelled symbolically [26].
   Some symbolic execution engines, such as KLEE [26], ship with symbolic models of common libraries, such as *glibc* [137], and these can be directly utilised for symbolic execution. Other hybrid symbolic and concolic engines, such as Mayhem [33], angr [130] and S2E [37], solve the partial path conditions (up to the point of an external

[4] We have, indeed, simplified the input by converting the actual input to a C-program, `argv` and `argc`, to more meaningful program input, the four integer values.

function call) and use an exemplary input (obtained from constraint solver) to perform a concrete system call.

2.4.2 Path-search Strategies

Progression of symbolic execution in terms of generating unique test-cases cases depends on, in addition to constraint solvers, the path-search strategy employed by the symbolic execution engine. The need for path-search strategy is to pick from a list of candidate instructions one instruction to execute with symbolic inputs. Consider, for instance, line 14 in the program in Listing 2.1. When symbolic execution arrives at this line, a choice needs to be made whether to pick line 15 or line 16 (depicted in nodes 4 and 5 in Figure 7.3, respectively) to symbolically execute next.

We will now briefly look at some typical strategies [115] to pick, during a run of symbolic execution (not yet reached a program exit-point), the next instruction to be symbolically executed.

1. *Breadth-first Search*: As the name suggests, in breadth-first search the list of candidate instructions to be symbolically executed next, is treated in first-in-first-out order. The outcome, intuitively, is that the nodes closest to the explored nodes are explored before the ones that are farther away.
2. *Depth-first Search*: In depth-first search, the intuition is that the node seen most recently by the symbolic execution engine is the one to be explored first. In effect, this entails treating the *seen* list as a last-in-first-out fashion, to explore nodes seen most recently.
3. *Cost-Minimisation Heuristic Search*: Also known as a *greedy approach*, this strategy tries to minimise the effort in terms of exploration of nodes seen, but not visited. The key here is the metric of *cost* itself. While some symbolic execution engines may use the constraint solver to provide a cost metric (time to return satisfiability of the query), others might consider other metrics such as expected distance to an unseen node (predictive heuristics) (*new-paths-first strategy*), distance to a target node (*targeted-search strategy*), or likelihood of finding a vulnerability (*vulnerable-paths-first strategy*).
4. *Random Search*: The final search strategy that may be employed by symbolic execution engines is a random search strategy. As the name suggests again, in this search strategy, the next instruction to be executed from the list of candidates is picked at random. In addition to being inexpensive, various past works have shown [84] that random search is also able to *sometimes* cover deeper parts of a program than depth-first, breadth-first or more expensive heuristics. Moreover, random search strategy can also be made *non-uniform* [26] by assigning higher weights to low-cost paths (as discussed in cost-minimising heuristic search), thereby combining deterministic heuristics with random path exploration.

Finally, as with other technical aspects of symbolic execution, search strategies can also be hugely improved by combining two or more of the above strategies [122] in a weighted manner to reach a pre-defined goal efficiently.

```
1 while (e) inst
```

Listing 2.2: *Sample while loop*

```
1 if (e) inst
2 if (e) inst
3 ⋮
4 if (e) inst
```

Listing 2.3: *Unrolled version of the loop in Listing 2.2*

2.4.3 Bit-vector Constraints

We will now briefly describe a specific optimisation in modern symbolic execution engines that have led to an increase in, both, their popularity as automated test-case systems and their efficiency – constraint solvers. In the past [13] the decision procedures used to solve path conditions to generate satisfiable solutions were based on underlying theories that included arithmetic and logical operations on various data primitives, such as integers, alphanumeric characters, floating point numbers and arrays of the above. As expected, these decision procedures were inefficient in terms of time to decidability.

One significant optimisation in SMT and SAT [49] solvers has been the introduction of *bit-vector types and arrays*. Bit-vectors [15] are fixed-length strings of binary digits (bits), and operations on these bit-vectors can be directly described by their effect on each bit in them. If every primitive datatype and arrays of primitive datatypes can be represented as bit-vectors (by *bit-blasting* them to bit-wise representation), then the programming language constraints, such as in-bounds memory access, can be modelled precisely using only a Boolean system. This is often useful to catch potential bugs such as buffer overflows. Constraint solvers such as STP [56] and CVC [13] have implemented effective bit-vector decision procedures for the specific use-case of symbolic execution engines.

2.4.4 Loop Unrolling and Bounded Models

As described earlier, if there are input-dependant loops in a program then the Halting problem (deciding whether the program will terminate) is undecidable in general. To overcome the path-explosion caused by input-dependent loops, many practical symbolic execution techniques employ *loop unrolling* [43]. Consider the `while` loop in Listing 2.2, and how such a loop may be unrolled in Listing 2.3.

There are several specific ways in which loops may be replaced (called *unrolling*) into a sequence of *if* blocks[5]. The first way [43] is to apply an upper limit to the number of times a loop may be run by, instead, asserting after executing *inst* that the looping condition e is not true for a fixed number (user-defined or pre-coded) of times. The second way [8, 148] is to perform a concrete execution of the loop to find out the number of times a loop

[5] In fact, unrolling may also be done without *if* blocks at all, i.e. simply repeating the instruction `inst` many times.

must be unrolled for *some* concrete and valid inputs for the entry condition, e of the loop and apply this limit just like in the previous option. The last option employed by some symbolic execution engines [78] is to develop special search-strategies that detect a loop in the program's control flow graph and *bias* the execution towards a path condition that will force the looping condition e to be infeasible.

2.4.5 Test-cases Exploiting Vulnerabilities

We will now examine the capability of symbolic execution to discover vulnerabilities in programs. Most symbolic and concolic execution engines [7, 26, 140] include the capacity to catch, so-called, low-level vulnerabilities in general-purpose programs, as well as, the ability to encode more complex vulnerabilities as programmatic assertions. Concretely, we define low-level and complex vulnerabilities as follows.

Definition 2.4.1. *(Low-level Vulnerability) A vulnerability that violates the inherent integrity, availability or confidentiality properties of programs written in a particular programming language and that manifests as a result of incorrect usage of that language's semantic rules, is called low-level vulnerability.*

Definition 2.4.2. *(Complex Vulnerability) A vulnerability that violates a security or safety property of a specified program, non-inherent to the semantics of the programming language and, therefore, unlikely to affect another program written in the same language, and that manifests as a result of incorrect design or implementation of a specification, is called a complex vulnerability.*

For handling and reporting low-level vulnerabilities, symbolic execution engines instrument the program-under-test with assertions before every instance of a potentially vulnerable instruction and generate a report when that assertion fails, e.g. instrumenting all instances of array indexing (`ar[i]`) to check for *IndexOutOfBounds* vulnerability. For handling and reporting complex vulnerabilities, symbolic execution engines allow users (testers) to manually insert assertion statements in the program-under-test to have symbolic execution report all inputs that lead to any violations of the respective assertions. In this way, symbolic execution engines can generate test cases to demonstrate exploitation of vulnerabilities in programs.

2.5 Current Challenges

Having discussed dynamic analysis with symbolic execution, we will now discuss some shortcomings of symbolic execution, more specifically, in terms of insufficient structural coverage and its ability to find vulnerabilities in general-purpose programs.

2.5.1 Path explosion

The first limitation in the state-of-the-art symbolic execution techniques is that the performed analysis suffers from the so-called *path explosion* problem [29]. As discussed in the description of the technique, if there are any input-dependent loops in the program,

then it cannot be determined whether the program will terminate. Additionally, reasoning about programs may also become intractable when inter-procedural calls paths need to be resolved during runtime, e.g. with function callbacks, function pointers or reflection. Over-approximation (assuming that callbacks may be resolved to any matching function) might lead to further deterioration of by introducing unnecessarily many paths for symbolic execution to explore. The outcome of path explosion in symbolic execution is that, if those paths are not explored first that increase the likelihood of achieving a coverage or vulnerability goal, then symbolic execution may progress very slowly.

2.5.2 Bottleneck of Constraint Solving

In Section 2.3 we explained the step of constraint solving, where the path conditions, represented as boolean predicates over symbolic inputs, are issued to decision procedures, such as SMT solvers, to obtain a concrete solution that will lead the execution through the corresponding path in the program. A practical challenge to symbolic execution is the bottleneck of these constraint solvers. This bottleneck affects symbolic execution engines because the decision procedures may be inefficient at returning satisfiability or counter-examples in a reasonable amount of time. While the form of constraints itself may not be directly correlated with the amount of time to return satisfiability, it can, at least, be expected that every additional independent constraint[6] in a path condition (in CNF) adds additional time-to-return for constraint solvers, not accounting for any parallelisation. However, for non-independent constraints too the time-to-return may increase for every additional branching-condition. E.g. , a SAT solver may return satisfiability for $(a < 100)$ faster than for $(a < 100) \wedge (a \geq 50)$. The outcome of inefficient constraint solving is that symbolic execution cannot generate sufficiently many test cases in a reasonable amount for time for a program of arbitrary size.

2.6 State-of-the-art Solutions

Having listed and briefly described the challenges faced by symbolic execution, we now look at some existing solutions, in theory and practice, that have been proposed to deal with the above problems.

2.6.1 Smart Heuristics for Path-search

We already discussed some path-search strategies in Section 2.4.2 that aim to preempt certain paths, such as loop-iterations, to run into path-explosion. There have also been other solution proposals that go beyond the simple strategies of depth-, breadth- or best-first search [29]. For instance, some papers [25, 26] have proposed search strategies guided by inferences drawn by the structure of the program-under-test, such as the static control-flow-graph (CFG). In these CFG guided strategies, the search algorithm may select a state that minimises the distance to a branch that is not yet covered by symbolic execution. Alternatively, those states could be prioritised that contain instructions that

[6] $(a > 10)$ and $(b < 100)$ are independent, while $(a < 10)$ and $(a > 5)$ are dependent constraints.

have been executed less number of times of other instructions. Some proposals for path-search strategies employ heuristics from other fields, such as genetic algorithms [69] or fitness-based metrics [148] to guide the exploration of symbolic execution engines. Finally, a combination of structurally-guided, such as breadth-first or CFG-based, and random strategies [84] have also been proposed and shown to improve performance of the symbolic execution engine in terms of path-search efficiency.

2.6.2 Compositional Symbolic Execution

The observation that real-world programs are constructed as combinations of various interacting components responsible for distinct tasks has been exploited in the past to perform independent symbolic execution of them. Compositional symbolic execution [6, 38, 61] is such a class of dynamic analysis technique where, instead of performing symbolic execution from a single, usually user-facing, interface of a program, symbolic execution can be applied at arbitrary points in the program, such as functions, and the results summarised and "propagated" upwards to eliminate non-interesting paths. Here, symbolic execution is transformed into a problem of demand-driven analysis where targets of analysis are presented to the symbolic execution engine by, first removing input preconditions and then re-introducing them to filter the results in the context of usage of the component [78, 115, 142]. Many papers have also proposed to generate summaries from the symbolic analysis of a component of a program [81, 124, 131]. After the analysed components are reduced to their summaries, typically targeted symbolic execution is used to reason about the *feasibility* of these summaries given the preconditions of the entry-points chosen for the analysis [65, 83]. It has been shown in various studies [41, 102] in the past that compositional symbolic execution can find many more vulnerabilities in medium-to-large real-world programs than plain symbolic execution that may suffer from path-explosion in shallow regions of the execution tree.

2.6.3 Constraint Solving Optimisation

Constraint solvers have also received much attention in research over the past few decades. The main reason behind symbolic execution becoming one of the most popular test case generation technique only so much later than the technique was first introduced by King [71] was that constraint solvers were getting more efficient by focussing on specific domains of application and the rise in feasible parallel computing architectures.

Many symbolic execution techniques try to eliminate individual constraints [26, 123] in path conditions that do not affect the outcome of the constraint solver. For example [29], if the branching condition $\neg(y < 10)$ is to be added to an existing path condition $(z < 5) \wedge (x + y == 100)$ during symbolic execution, then to determine the satisfiability of $(z < 5) \wedge (x + y == 100) \wedge \neg(y < 10)$, we know that the term $(z < 5)$ does not affect the outcome and, hence, can be removed before sending to the constraint solver.

Next, symbolic execution engines may also reuse [28, 123] satisfiability from other constraints that were solved in the past. To achieve this, for a path condition whose satisfiability needs to be checked, the symbolic execution engine may use the test case

generated from a previous path condition to check if it still satisfies the new constraint. Only if this concrete assignment does not satisfy the new path condition, the constraint solver may be called to check satisfiability.

Lastly, as discussed in Section 2.5.2, time-to-return for path conditions may be less if there are more independent constraints in the path conditions than dependent ones. Therefore, an optimisation to be performed [123] *before* constraint solving is to determine if any of the independent constraints have appeared in any constraint previously solved during symbolic execution

2.7 Concluding Notes

In this chapter, we described symbolic execution, its underlying theory, and how the implementations work in practice. We described the technique in terms of symbolic inputs (in place of concrete inputs), path conditions which represent a symbolic execution path in terms of first-order logical formula over symbolic inputs, constraint solving to generate concrete test cases to execute the particular program path. Then, we discussed real-world symbolic execution engines and the higher-level design decisions that are generally made for effective and efficient symbolic execution. The first here is concolic execution, which is a combination of executing the program-under-test with concrete inputs to, either, collect and individually negate branching-conditions, or to interact with the system's environment concretely. In terms of path-search strategies, we discussed some structural strategies, such as breadth-first and depth-first, and other specialised strategies that rely on heuristics and static-analysis, such as cost minimisation and a combination of different strategies. We, then, discussed how path conditions can be represented as bit-vectors to allow for different datatypes, sequences and programmatic structures. Loop unrolling and bounded models were discussed as possible optimisations to go around path-explosion problem efficiently. Finally, we discussed how vulnerabilities in programs are discovered and reported by symbolic execution engines.

After discussing the theory and practice of symbolic execution, we were able to succinctly describe the two main problems that symbolic execution suffers from – path-explosion and the bottleneck of the constraint solver. The solutions proposed in the past to deal with one or both of these issues were listed and categorised as smarter path-search strategies, compositional symbolic execution techniques or optimisation of path conditions before submitting them to constraint solvers for checking satisfiability or generating test cases. Using the results and insights from these existing works, we will, in later chapters, describe our methodology to overcome the issues of path-explosion and constraint solving in symbolic execution.

3 Guided Fuzzing

This chapter presents the essential background on guided fuzzing*, one of the competing, as well as contributing, techniques of dynamic analysis to be discussed in this thesis.*

Guided fuzzing, fuzz testing, or fuzzing [134] for this thesis, is an automated technique for test-case generation where the basic idea is to send random or intentionally malformed input to the program to trigger edge-case behaviours in a program. In this chapter, we will describe the state-of-the-art in fuzzing, the concepts of input mutation, exception handling and other factors considered in implementations of fuzzing for different kinds of programs.

This chapter is structured as follows – In Section 3.1, we introduce the concept of guided fuzzing (only referred to as *fuzzing* in the rest of this thesis) and contrast it with random testing, a well-known automated testing technique in research and practice. We, then, start the step-by-step description of a typical fuzzer with defining and explaining in detail the concept of *seed inputs* in Section 3.2. Based on the definition of fuzzing inputs, we, then, describe input mutation strategies in Section 3.3. In Section 3.4, we look at how fuzzing can monitor programs-under-test during runtime and use the information for guiding input mutation. Then, we provide some details of the implementation of popular real-life fuzzers developed in academia and industry, in Section 3.5, and list some of the challenges faced by fuzzers in Section 3.6. We, finally, conclude the chapter in Section 3.7.

3.1 Random Testing vs. Fuzzing

We will, in this thesis, define fuzzing in terms of being a counterpart or being inspired by *random testing*.

Definition 3.1.1. *(Random Testing) Random testing is a form of automated software testing where inputs are sampled (usually uniformly) at random and independently from a pre-determined domain of inputs for a program and the program is executed to check if the desired output is obtained.*

We have adopted the above definition from Hamlet [66]. The critical difference between random testing and fuzzing is the *independent* aspect of input selection in random testing. The intuition behind random testing with independently sampled input is that this process is not informed by the output or side-effect of executing the program-under-test with those inputs. As a result, random testing is usually able to trigger those edge-cases at a much lower cost [3, 48] than manual testing with input-selection guided by observing the output of the program.

However, fuzzing aims to take random testing a step further by (automatically) monitoring the behaviour and output of the program-under-test for all sampled inputs and guiding the input-selection therefore. Below we define fuzzing, adapting the definition from Sutton et al. [134].

Definition 3.1.2. *(Fuzzing) Fuzzing is a form of automated testing for discovering vulnerabilities in programs by providing unexpected or malformed input and monitoring the process for new coverage, interactions with the environment or exceptions thrown.*

Therefore, standard fuzzing techniques include, in addition to input selection step of random testing, strategies to generate unexpected input based on observation of a program's behaviour (or *process monitoring*, as we will discuss later). We will now describe these steps in more detail in the next sections.

3.2 Seed Input Selection

The first step in fuzzing is to select a set of inputs for the program, usually manually.

Definition 3.2.1. *(Seed Inputs) The initial inputs, chosen from a domain of possible inputs, to automatically test a program with fuzzing, are called seed inputs.*

Let us consider the program in Listing 2.1 (Chapter 2) again. For this program, the inputs may be considered to be of composite type $\mathcal{I}(X)$ (defined the same way as in Equation (2.1)).

Then, let the set of fuzzing inputs be I^F, and

$$I^F = (\mathcal{I}(X))^* \tag{3.1}$$

Fuzzing inputs, I^F, include, both, seed inputs, I^F_{init}, as well as, the inputs generated by the automated process input mutation, to be described in the next section. As with symbolic inputs (Chapter 2), variables `a, b, c, d` are assigned from `argv` and, therefore, may be considered to be implicit fuzzing inputs.

Some examples of seed inputs in this scenario may be the following

$$\langle 4, \text{"0 1 2 100"}, 0, 1, 2, 100\rangle \in I^F_{init}$$
$$\langle 4, \text{"1 50 200 10"}, 1, 50, 200, 10\rangle \in I^F_{init}$$

The process of seed selection may be guided by various objectives of testing, such as coverage maximisation or vulnerability discovery. Fuzzing, then, will utilise the set of seed inputs and perform useful mutations on them to generate new inputs to satisfy the objective mentioned above.

3.3 Input Mutation Strategies

After initialising the set of fuzzing inputs, I^F, the main fuzzing process is ready to mutate the set of inputs to generate, and add to the set, new inputs using the input mutation strategy. Recall that fuzzing is typically a blackbox process, which means that "uniqueness" of paths is deduced by the unique behaviour exhibited by the program when running with

a previously unseen input. Therefore, the aim of this stage of fuzzing, viz. input mutation, is to generate new inputs from the existing set of inputs that *may* exercise a previously unseen path.

Let us start by defining what is meant by input mutation.

Definition 3.3.1. *(Input mutation) The process of modifying an existing input in the set of inputs for fuzzing, by replacing, adding or deleting some bits, to generate a new input is called input mutation.*

The choice operation (addition, deletion, or replacement) on existing input may be guided by a so-called, *input mutation strategy*. This strategy may differ from fuzzer to fuzzer in practice, but typically it includes, first, choosing which existing input(s) to mutate and, then, which operation(s) to apply on it (them). We can extend the formalisation to include input mutation as follows. For every iteration of input mutation,

$$I^F \Leftarrow I^F \cup mutate(select(I^F)) \tag{3.2}$$

In the above equation, I^F is as defined in Equation (3.1) and *mutate* and *select* functions have the following signatures.

$$\begin{aligned} \textbf{select} &: Seq\ i \rightarrow Seq\ i \\ \textbf{mutate} &: Seq\ i \rightarrow i \end{aligned}$$

, where i is of type $\mathcal{I}(X)$. The *select* function returns one or more inputs from the set of all inputs in I^F that should be mutated. The *mutate* function uses the input list returned by *select*, performs a mutation based on the mutation strategy and returns an input to be added to I^F. The *mutate* function can take many forms, e.g. $bitflip\_i$ (to flip the i^{th} bit from 0 to 1, or vice-versa), *mask* (to mask a portion of the input bit-sequence with another sequence), *add_bytes* (to add random bytes in between or at the ends of an existing input), *mix_and_match* (to combine certain bit-sequences from two or more inputs returned by *select*) or a combination of these mutation strategies. Many modern-day fuzzers such as AFL [2] and AFLFast [21] advanced heuristics and genetic algorithms for the *mutate* function too.

The operations in Equation (3.2) are repeated until a fuzzing goal is reached, e.g. a target instruction is reached, or a failure is triggered (such as program crash). In some fuzzers, there may even be an additional step to Equation (3.2), where certain inputs from I^F may be removed based on a heuristic to decide whether they are valuable enough to be mutated further.

3.4 Process Monitoring

The iterations of fuzzing must be monitored to determine what progress, if any, is being made by the fuzzer. The goal of this step in fuzzing is to, ideally, provide feedback to the input mutation step to perform the mutations effectively. This goal is achieved by profiling [16] the program-under-test. We have adopted the definition of profiling from [134].

Definition 3.4.1. *(Runtime Profiling) The process of, for an (symbolic or concrete) execution of a program, recording the program or structural elements that were covered is known as runtime profiling or, simply, "profiling".*

Various structural elements may be profiled during runtime of fuzzing, such as instructions in source-code, intermediate code (e.g. LLVM [79]), basic-blocks in CFG or branches in CFG. Profiling for instructions at various levels can be achieved by inserting instrumentation in the code before starting the process of fuzzing. At other abstraction levels, profiling can be achieved by, first, lifting the program-under-test to an abstract representation and then mapping executed statements to basic-blocks or branches.

In addition to profiling the program-under-test, the monitoring process in fuzzing also includes externally monitoring the interaction of the program with the environment.

Definition 3.4.2. *(Interaction with Environment) The transition in state that a program-under-test triggers on the parent process, such as the spawning shell, is called its interaction with the environment.*

The interaction with the environment is useful to determine if an iteration of input mutation led to a previously unknown state-transition in the environment, such as a new return value, an exception thrown or a segmentation fault resulting in a program crash. In UNIX-based systems, interaction with the environment can be monitored for a running process by processing the return value or catching special signals such as `SIGSEGV`, `SIGSYS` or similar. Depending on the signal value caught or return value received by the monitoring process, the fuzzer can inform the *select* and *mutate* functions to prioritise inputs in I^F.

The tuple, runtime-profile and interaction with the environment, can be used to determine the uniqueness of test-cases in I^F, which may be used as a proxy for *unique functional behaviour* of the program-under-test.

3.5 Fuzzing in Practice

We will now move our discussion to some practical aspects of fuzzing that most fuzzers have to tackle in their implementations.

3.5.1 Types of Fuzzers

Sutton et al. [134] categorise fuzzers into types depending on the way inputs are generated – *mutation-based* fuzzers and *generation-based* fuzzers. Mutation-based fuzzers are ones where existing inputs are selected and mutated, as described in Section 3.3, to generate new inputs that might demonstrate functional behaviour previously unseen in the program. Unlike

mutation-based fuzzers, generation-based fuzzers, in this terminology, implies techniques that generate inputs by methods depending *only* on the program behaviour or structure and rarely on the already-existing set of inputs. The most compelling example of a generational fuzzer is symbolic execution, which we described and discussed at length in Chapter 2. Many authors [60] describe specific adaptations or implementations of symbolic and concolic execution as *whitebox fuzzing*. Examples of such adaptations are described in Section 2.4, where we discussed topics such as concolic execution where we begin the analysis with an initial set of inputs, *seed inputs*, to gather path conditions on symbolic execution and iteratively negate the branching conditions in them to generate new inputs. Such practical adaptations of symbolic execution also fall under the category of whitebox fuzzing, where the inputs are generated using the path conditions gathered by the analysis.

For this thesis, we will *only* consider mutation-based fuzzers when talking about fuzzers, in general. If the generation-based fuzzers are of symbolic execution variety as described above, we will classify them as such.

3.5.2 Instrumentation

Instrumentation of the tested object is an important practical aspect to consider for fuzzers. In theory, as described above, the progress of the fuzzer in terms of input generation through mutation strategies is guided by process monitoring. Process monitoring, as described in Section 3.4, includes runtime monitoring of the program and interaction with the environment. Of these two monitoring steps, only runtime monitoring should require any code instrumentation (to record the program elements executed by the inputs). However, in practice, there is also a need for, so-called, *sanitisation* of various source-code locations to signal whenever an interaction with the environment leads to a failure, such as a program crash. We will briefly describe now how instrumentation, including sanitisation, is achieved by various fuzzers in practice.

For measuring coverage during the input mutation iterations, AFL [2] and Libfuzzer [125] insert special instrumentation in the source-code before compilation to monitor branch-coverage. Before starting fuzzing, these fuzzers allocate a large shared-memory space that is utilised across all iterations of input mutation. In this shared memory [2], the fuzzer represents unique branches in the program using random indices generated during runtime, and the value at these indices are incremented every time the corresponding branch is taken by an input. Path coverage can also be determined implicitly from branch coverage by representing paths as the concatenation of branches taken in sequence.

For inputs resulting in failures in the program-under-test, most fuzzers employ sanitisers to detect and report the failures gracefully. The most common form of sanitisation is called *address sanitisation*, of which ASAN [127] is an implementation. The goal of address sanitisation is to detect if the program can, for valid inputs, access (read or write) memory locations that are outside the bounds of the allocated application memory space thereby resulting into a program crash (due to a segmentation fault). This is achieved by, first, replacing memory allocation (e.g. `malloc`), access (e.g. pointer-dereference or array-indexing) and deletion (e.g. `free`) instructions by custom instructions and, then, actively monitoring and reporting when any of these instructions spill over the allocatted

memory space[1].

Other sanitisers include UBSAN [143], that detects undefined behaviour such as integer underflows and overflows, and TSAN [139], which detects race conditions and other issues in multithreaded programs. By using such sanitisers, a fuzzer can detect when a mutation of input causes a program to crash and report the vulnerable instruction.

3.5.3 Test Minimisation

Mutation-based fuzzers, as described above, apply random mutations to generate new inputs for the fuzzer expecting to uncover new functional behaviour in the program. While many of the mutated inputs may lead to a failure, they might not always be realistic inputs to the program (e.g. containing NULL characters in the middle of streams). A key requirement of many fuzzers, then, is to collect these *special* inputs (triggering failures) and *minimise* them to generate inputs that exercise the same behaviour as the original mutated input.

The goal in this step is to discover realistic test inputs that may allow a user to crash the program. A typical test minimisation algorithm [2] takes an input generated by the fuzzer and *trims* it by – 1. replacing long sequences containing the same bytes, with a single occurrence, or 2. removing long sequences of NULL characters in between or the end of the byte-stream, or 3. performing normalisation on sub-strings containing repeating characters, based on the lengths of the substrings. Of course, there is no guarantee in any of the above possible test minimisation technique that the resultant input will lead to the same functional behaviour as the original input. To ensure that the "failure-triggering" property is preserved, the minimised inputs are used to execute the instrumented program (Section 3.5.2) to determine if the vulnerability can still be exploited.

3.6 Current Challenges and Solutions

Having seen an instance of program analysis with fuzzing, we will now critically analyse the reasons behind fuzzing not achieving sufficient structural coverage. One of the research questions stated in this thesis asks how structural coverage is related to vulnerabilities. Therefore, here, we will also analyse vulnerability discovery with fuzzing. The main drawbacks associated with fuzzing are the reliance of the mutation-based fuzzers on seed inputs and redundant path coverage, which we will now elaborate.

3.6.1 Reliance on Seed Inputs

According to our definitions and methodology descriptions, fuzzers can only generate new inputs based on two feedback items during fuzzing – input used to execute and the functional behaviour exercised by the input. As such, a mutation-based fuzzer is not able to *infer* the semantics or intended behaviour of program-under-test in the same way as symbolic execution techniques can (by instrumenting the program and collecting branching

[1] The details of the implemented algorithm can be found at `https://github.com/google/sanitisers/wiki/AddressSanitizerAlgorithm`.

conditions, as PL formulae over program inputs). Therefore the ability of the mutation strategies to generate inputs to uncover previously-unseen functional behaviour depends on the so-called, quality of seed inputs [32, 88, 119].

In [119], Rebert et al. asked questions to empirically correlate seed input selection and the ability of mutation-based fuzzers to find vulnerabilities and cover interesting (rarely executed) parts of the program. The conclusion of this study on various open-source programs and different sizes of randomly sampled sets of seed inputs was that, in general, a focussed set of a *minimally reduced* set of seed inputs helps the fuzzer find more vulnerabilities and cover more program elements than a larger, but possibly redundant, set of seed inputs. A minimally reduced set is a reduced set of seed inputs that contain the most important inputs that are likely to execute diverse functional behaviour in the program-under-test. The problem is that of reducing the initial seed inputs to a minimal set. This topic has received much attention in research, as we will discuss in Section 3.6.3, and continues to be a key focal point of improving mutation-based fuzzers.

3.6.2 Redundant Path Coverage

Another outcome of the relative "blindness" of the process of fuzzing is that many inputs and their mutations end up covering the same subset of paths in the program repeatedly. Unlike symbolic execution, fuzzing does not consume and iteratively negate branching conditions in programs to guide its progress. As a result, it is often very hard for random mutations of inputs to generate valid inputs that can satisfy branching conditions that are satisfiable for a very small subset of the input domain [103].

Redundant, and low, overall path coverage is not only a problem in itself with fuzzing. The capability of fuzzers to find vulnerabilities is also reduced because of low-coverage, because [80] if certain portions of a program are not executed by a dynamic analysis technique such as fuzzing then potential vulnerabilities in these portions cannot be discovered. Therefore, as we will discuss in Section 3.6.3, many recent research outputs have focussed on increasing the coverage of fuzzing tools to find more vulnerabilities.

3.6.3 State-of-the-art Solutions

Having listed and briefly described the challenges currently being faced by fuzzing, we now turn our attention to existing solutions in fuzzing theory and practice that have been proposed to deal with the above problems.

Input Selection and Prioritisation

The first class of solutions to improve the state-of-the-art in fuzzing suggest improvement and modifications to the way seed inputs are selected for fuzzers and, especially, ways to mutate them to increase structural coverage in the program. Rebert et al. [119] and Cha et al. [32] have proposed to adapt the seed-input selection to suit the program-under-test, by including variants of popular structured text-file and image formats, such as PDF, PNG or JPEG. By using expert intervention to infer the minimum requirements to bypass sanity-checks on input, the above authors have shown that guided seed input selection can

increase the effectiveness of blackbox fuzzing significantly. The same consequence can be extended to automatically inferring the grammar and structure of valid input for a program. Such solutions were proposed by Ganesh et al. [57], Wang et al. [146] and others in the past. In these automated solutions, input bytes may be tainted and those bytes determined that are, either, the most often accessed by branching conditions in the program, or the most interesting in terms of triggering a component of interest. Other solutions, such as by Rawat et al. [118], have focussed on input prioritisation which dictates to the mutation strategies which inputs (seed or generated by the fuzzer) to pick next for mutating. They include fuzzers that makes use of the control-flow and data-flow characteristics of the binary target to inform the input mutation strategy.

Targeted Fuzzing

The second class of optimisations in fuzzing are those that do not strictly treat the program-under-test as blackbox but have *some* control or view of what structures are present and triggered inside them. Due to partial visibility into the internals of the system, these fuzzers can more effectively dictate the mutation strategies to increase structural coverage or find more vulnerabilities in the program. Böhme et al. [20] introduced targeted fuzzing by modelling the fuzzing process as a Markov-chain and using *power schedules* to control which seeds should be mutated next to increase the likelihood that a target location in the program-under-test may be executed. The resultant fuzzing tool was called AFLGo. The same team of researchers, then, improved this technique [21] in AFLFast, by extending the power schedules also to predict how many *unique* inputs (executing unique program paths) might be generated by mutating a given input. In other related work, Lemieux et al. [80] proposed a technique to effectively mask those parts of the input bytes that do not lead to coverage of new branches to focus on those bytes that are likely to trigger edge cases in *rare* branches (only taken for a few inputs). Many of these optimised fuzzers have demonstrated their superiority over naive fuzzers by reported several previously unreported vulnerabilities in widely-used programs.

3.7 Concluding Notes

In this chapter, we described the theory and practice of guided fuzzing or, more succinctly, simply fuzzing. We discussed how fuzzing differs from random testing by actively monitoring the program-under-test and the external environment to guide the dynamic analysis procedure. Fuzzing starts by executing the program on seed inputs that are, usually, provided manually. The fuzzer, then, generates new inputs by selecting and mutating existing inputs to, hopefully, trigger new functional behaviour in the program such as a unique interaction with the environment or a program crash. These environment interactions can be tracked by monitoring the process in a controlled manner and feeding the information back to the fuzzer to instruct the input mutation stage. The fuzzer repeats these steps until the desired testing goal is reached, such as instruction coverage, branch coverage, path coverage, or finding vulnerabilities.

In practice, as we saw in Section 3.5, mutation-based fuzzers, like the ones we discussed in this chapter, depend on and employ various implementation and design optimisations for effective dynamic analysis. To monitor the progress of the fuzzer and to spot a vulnerable instruction whenever a program's execution leads to failure, most fuzzers employ light-weight instrumentation and address sanitisation. When an input exploits a vulnerability, the next step for the fuzzer is to determine whether this is a realistic input that can be supplied to the program and whether it can be modified and shortened while still exposing the same vulnerability in the program. This is accomplished by test minimisation where various aggregation methods can help to reduce the size of a crashing test input.

Lastly, we described the two most detrimental facts that affect the performance of fuzzer in practice – its reliance on the quality of seed inputs and redundant path coverage. In Section 3.6.3 we described various existing solutions in research that try to deal with these two problems by, either focussing on the process of input selection and mutation or targeting the process of fuzzing towards rare-branches or vulnerabilities by optimising the mutation process. Using these insights into current state-of-the-art in fuzzing, we will, in the coming chapters, describe novel compositional techniques to mitigate the issues associated with it.

4 Hybrid Symbolic Execution and Fuzzing

This chapter presents a systematic mapping study to survey other hybrid *symbolic execution and fuzzing techniques, possibly similar to our own, that have been proposed in the past. Parts of this chapter have previously appeared in [103], where the author of this thesis was the first author.*

Due to the similarity in their goals, symbolic execution and fuzzing have the potential to be used as mutually beneficial methods for discovering vulnerabilities. Fuzzing is a useful technique for exploring *some* paths in a program in full depth. Symbolic execution, on the other hand, is a useful technique for exploring most branches in a program at low-depths (closer to an entry point). There are *technical aspects* of fuzzing and symbolic execution, such as path search and input mutation, that may be used or modified to increase the effectiveness of either of these two techniques.

In this chapter, we will survey the field of hybrid techniques of fuzzing and symbolic execution (henceforth referred to as, only, *hybrid techniques*) as viewed from the perspective of these, and other, technical aspects. For this study, we will roughly follow the established methodology of systematic mapping study described by Kitchenham et al. [72], combined with standard systematic literature review prescribed by Keele [70]. An overview of our complete methodology in this chapter is depicted in Figure 4.1. This chapter includes

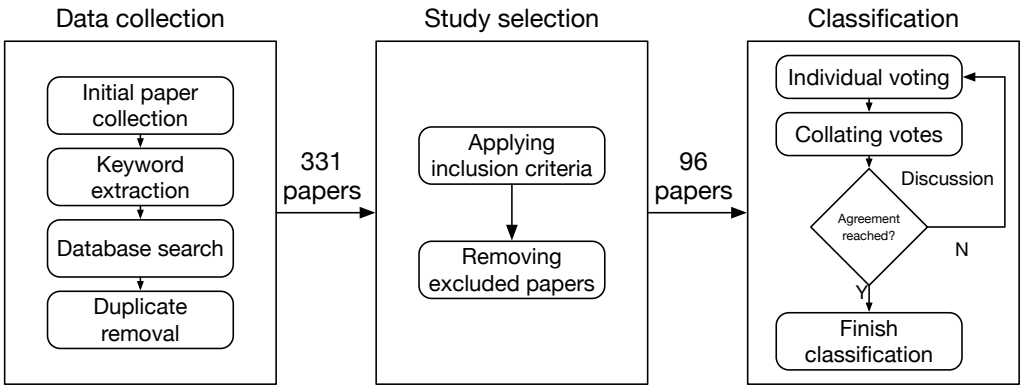


Figure 4.1: *Overview of the methodology*

a structured data collection step, classification based on relevant categories, voting for classification of the collected papers, and drawing results from the classification to answer relevant research questions. In relevant sections of this chapter, we will expand upon and clarify the items in Figure 4.1.

This chapter is organised as follows – In Section 4.1, we describe our data-collection procedure on various databases of publications, and how we selected publications from them. In Section 4.2, we list the classification criteria to categorise the included papers for further

analysis. Then, in Section 4.3, we discuss the results of our manual classification task and our observations and synthesis about the field of hybrid solutions involving symbolic execution and fuzzing. We expand concretely upon the gaps in research in this field and our contributions in Section 4.4 and, finally, conclude the chapter in Section 4.5.

4.1 Collecting Data about Past Work

We, first, describe the systematic procedure for data collection and study selection. Please note that we have put a limit of 2018 for all our searches. We did not set a limit on the starting year, but the earliest papers that our searches obtained were from 2005.

We started the data collection[1] procedure with a search on popular bibliographic databases. Search keywords were primarily based on an *initial set* (size 20) of papers. We included some of these papers based on our domain knowledge, knowing them as key contributions in the field of fuzzing and symbolic execution. The rest of the initial papers were obtained by *snowballing* through the related work of the initial set, as described by Wohlin [147]. Primary search keywords were derived from the initial set by forming a word-cloud on the combined texts of their abstracts and choosing the most commonly occurring (stemmed) words. The main keywords derived, therefore, were *{"test", "symbol", "execute", "fuzz"}*. These search keywords were modified, as shown in [105], to fit the *advanced search* syntax for all chosen databases.

The databases chosen to perform the search were

1. ACM Digital Library (*15 papers*),
2. IEEE Xplore (*31 papers*),
3. Scopus (*49 papers*),
4. SpringerLink (*227 papers*), and
5. CiteSeerX (*9 papers*).

The next step in the collection process was to remove the duplicate results. These could be 1. same paper appearing in more than one databases, or 2. different versions of the same paper, e.g. extended journal version of a conference proceedings paper.

After removing duplicates from results from all five search engines, we were left with *331 unique publications* matching the search keywords.

4.1.1 Study Selection

To ensure that the results used for classification were from software engineering field and, at least, contributed to symbolic execution *or* fuzzing, we needed to apply the following *inclusion criteria* [70] to each of them –

1. Is the result a research paper (as opposed to a poster, keynote or presentation)?
2. Is the paper related to software testing or engineering?
3. Does the paper contribute to symbolic execution or fuzzing?

[1] We have made all the data available online at [105]

We manually applied these inclusion criteria, and only those papers were selected for classification for which the answers to both of the above questions were *"yes"*.

After applying the inclusion criteria, there remained *96 selected papers* that we used for the classification stage and for answering all relevant research questions.

4.2 Classification of Solution Proposals

We classified all included papers according to *six criteria*. The final classification was decided by a full majority in a three-way voting by three experts (including the author of this thesis) [103] in the field. This classification will be useful to us later in answering relevant questions regarding state of the art in hybrid techniques. Particularly, regarding *greybox fuzzing*, a combination involving fuzzing and symbolic execution, criteria 4 and 5 will lead us to categorising these solutions and decide the degree of "novelty" in the proposed solutions. The categories for all classification criteria were obtained by analysing the key terms and concepts addressed (not only explicit keywords) in our initial set.

Criterion 1: General Field of Contribution:

The choices for this classification criterion were

a. Symbolic execution,
b. Fuzzing, and
c. Both

The first classification criterion, if applied correctly, also served as a validation for the inclusion criteria because if a paper can not, effectively, be classified as one of the above three choices, then it should have been excluded from the study space in the first place.

Criterion 2: Introduction of a Novel Technique:

This criterion was to identify the field where a new solution has been proposed. The choices for this criterion were

a. Symbolic execution,
b. Fuzzing,
c. Hybrid technique, and
d. None

The hybrid category was chosen only if, 1. there are modifications proposed for, both fuzzing and symbolic execution, or 2. solution involves modification of both fuzzing and symbolic execution, but it is not apparent which one of these two techniques have been primarily subjected to the modification. We selected "None" when the paper in question does not propose a new solution, but, as we describe in the following criteria, contributed differently.

Criterion 3: Description of State-of-the-art

This criterion was to find if the paper presents a systematic state-of-the-art study or any meta-study, like our own. The choices for answering this question were, also,

a. Symbolic execution,
b. Fuzzing,
c. Hybrid technique, and
d. None.

Criterion 4: Novelty in the Solution for Fuzzing:

This criterion enumerates the technical aspects of fuzzing. The choices for this criterion were

a. Input mutation – includes modification in mutation strategy of random fuzzing,
b. Static analysis – uses a *code analysis* technique such as static analysis, information flow analysis or symbolic execution to optimise fuzzing,
c. Expert guidance – uses any technique, other than code analysis, to optimise fuzzing, and
d. No modification – does not propose optimising any technical aspects of fuzzing.

This list of fuzzing aspects was determined by the authors using state-of-the-art descriptions in [134, 141].

Criterion 5: Novelty in the Solution for Symbolic Execution:

This criterion enumerates the technical aspects of symbolic execution. The choices for this criterion were

a. Path search – proposes a new path search strategy, such as directed search, or modification of an old path search strategy,
b. Compositional analysis – proposes to treat the system under test compositionally. This means disintegrating the modular system and analysing interactions of individual components,
c. Constraint solving – proposes an optimisation in the constraint solver of the symbolic execution engine, and
d. No modification – does not propose optimising any technical aspects of symbolic execution.

The list of symbolic execution aspects was determined by the authors using state-of-the-art description in [29] and recent solution proposals such as [40, 83].

Criterion 6: Description of New Implementation in the Paper:

In this criterion, we tried to find if the paper provides a new implementation or evaluation of an existing symbolic execution or fuzzing solution. The choices for answering this question were –

a. Yes, and
b. No.

4.3 Results of Classification

Based on the classification criteria, let us now list the results of the classification.

4.3.1 Summarising Solution Proposals

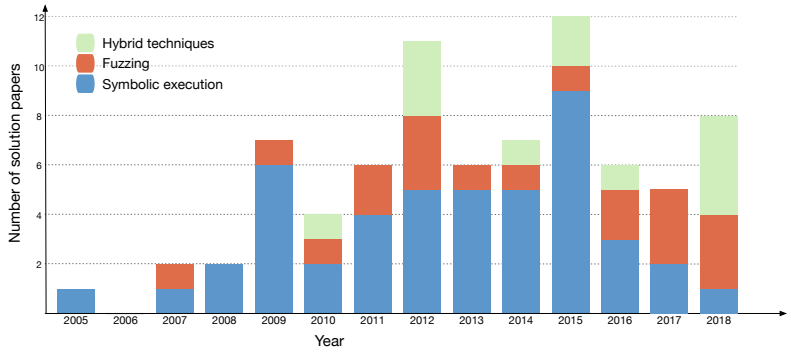


Figure 4.2: *Number of solution proposals by year – Vertically stacked values*

The first result we obtained from our survey methodology is the set of solution proposals (not state-of-the-art studies or implementations of existing solutions) in research of symbolic execution and fuzzing. Figure 4.2 depicts the trend in solution proposals over several years. We can see from this trend that the earliest papers retrieved by our search methodology were from 2005. We can see from Figure 4.2 that most of the solutions proposals deal with symbolic execution, with only a few solutions proposed based on fuzzing. Possibly, due to an increase in the efficiency of constraint solvers during this period, symbolic execution became more viable as a testing technique.

Of all the solution proposals, our analysis found only *9*, listed in Table 4.1, that proposed modifications in, both symbolic execution and fuzzing. We analysed all hybrid studies in detail by looking at their texts because we recognise that our search keywords may also return publications that compare evaluation studies of symbolic execution compared with fuzzing or vice-versa, with one or both of them not contributing anything to the proposed solution design.

4.3.2 Solutions In-depth

For all the obtained solutions listed in Section 4.3.1, we wanted to further classify them based on the technical aspects of symbolic execution and fuzzing addressed in them.

The pair-wise distribution of the technical aspects of fuzzing and symbolic execution in *all solution papers* is shown in Figure 4.3. The largest group of solutions (28) propose modifications to neither symbolic execution or fuzzing. Most of these contributions, such as [27, 116], showed up in our search results because all terms in our search string occur in

them, but the proposed solutions therein use symbolic execution or fuzzing to solve domain-specific problems (such as malware detection) without adding any modifications to basic symbolic execution technique. In many cases, they may also include new implementations, such as [34], of existing techniques. Other than these groups of papers, we can see that most solution proposals directly improve either symbolic execution or fuzzing technique, but not both. E.g. in [53], the authors propose symbolic execution in binary programs when fuzzing cannot increase the coverage anymore, by focusing on uncovered paths during symbolic execution – which means that effectively only symbolic execution technique is modified, while fuzzing is used in its original form. Similar combinations exist in techniques such as [32, 74].

We note *twelve* works over the years that propose hybrid dynamic analysis involving symbolic execution and fuzzing. While some of them do modify both symbolic execution and fuzzing others modify only one of the two. Fuzzbuster, by [91] Musliner et al., generalises the constraints to reach a vulnerable instruction using a modification in symbolic execution. These vulnerabilities are discovered, however, only using an off-the-shelf fuzzer. This means that the tool suffers from the same drawbacks as a naïve fuzzer, i.e. not enough path diversity. In [36], Chen et al. propose a *directed fuzzing* strategy that uses symbolic execution to formulate the program behaviour in the form of a complex control-flow-graph.

Table 4.1: *List of all hybrid solution proposals*

Year	Title	Authors
2012	*Using concolic testing to refine vulnerability profiles in Fuzzbuster* [91]	Musliner et al.
2012	*A directed fuzzing based on the dynamic symbolic execution and extended program behaviour model* [36]	Chen et al.
2012	*Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution* [106]	Pak
2014	*Automatic software vulnerability detection based on guided deep fuzzing* [30]	Cai et al.
2015	*Binary-oriented hybrid fuzz testing* [53]	Fangquan et al.
2015	*Program-adaptive mutational fuzzing* [32]	Cha et al.
2016	*Deepfuzz: Triggering vulnerabilities deeply hidden in binaries* [22]	Böttinger et al.
2016	*Driller: Augmenting Fuzzing Through Selective Symbolic Execution* [133]	Stephens et al.
2018	*Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach* [99]	Ognawala et al.
2018	*Badger: Complexity Analysis with Fuzzing and Symbolic Execution* [96]	Noller et al.
2018	*SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing* [145]	Wang et al.
2018	*Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing* [150]	Zhang et al.

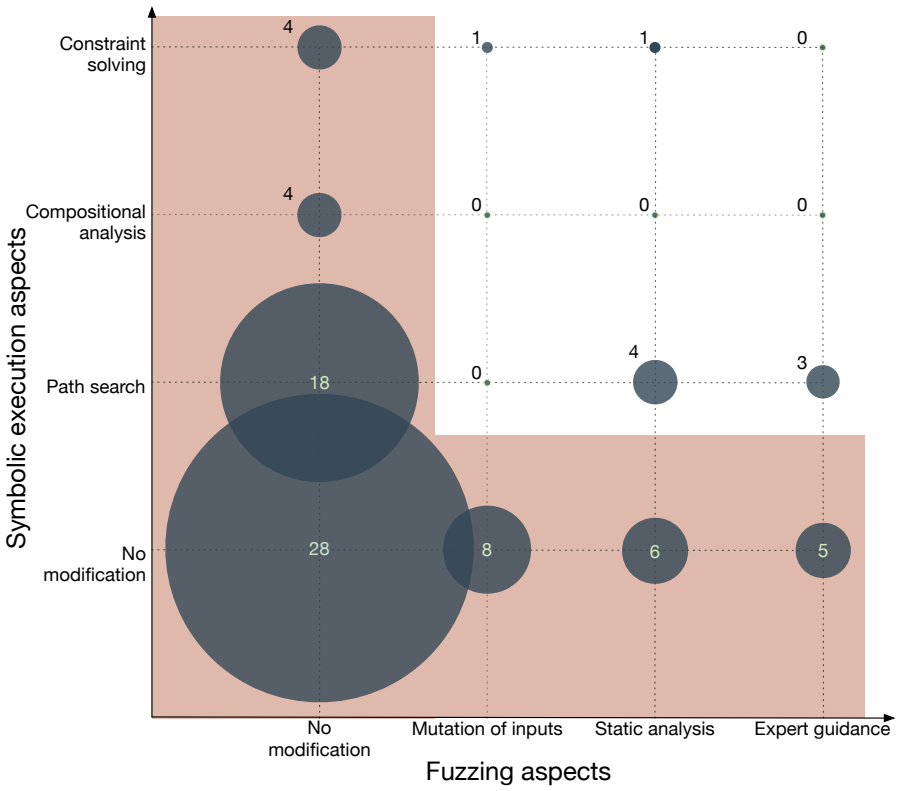


Figure 4.3: *Technical aspects of symbolic execution and fuzzing in solution proposals*

In this way, they claim that the fuzzer has an inside view of the program. Another hybrid technique, proposed by Pak [106], uses symbolic execution to gather as many unique constraints within the user-defined resource limit as possible and uses solutions to these constraints as the "random" input seeds for the fuzzer. Even though this technique introduces enough diversity in the seed inputs than manual entry, it relies much too heavily on the fuzzer to completely analyse all paths beyond the user-set limits. Cai et al. [30] introduce a tool, called *Sword*, that checks the software for vulnerabilities with symbolic execution and only fuzzes those paths that are, therefore, deemed dangerous. The authors do not aim to improve the efficiency of either symbolic execution or fuzzing. Fangquan et al. propose a hybrid approach for testing binaries [53] that starts with fuzzing and switches to symbolic execution when fuzzing does not make progress any more. However, this paper does not propose an instrumentation or input-sharing mechanism to ensure that the same paths are not covered by both fuzzing and symbolic execution. Cha et al. propose a novel algorithm [32] that uses symbolic execution for extracting path conditions for the paths executed by fuzzing and uses these path conditions to determine dependence between individual bits of the input, to guide fuzzing's input mutation strategy. In [22], Böttinger et al. present a probabilistic approach to treating the path-explosion problem in symbolic execution by targeting those branches in a program that are least likely to be triggered by fuzzing. Driller [133] is a hybrid solution proposal that combines fuzzing with selective concolic execution. In this paper, Stephens et al. propose to use "usual"

fuzzing to cover parts of the program that are easy to reach and, then, use targeted concolic execution to *unlock* parts that are guarded by a branching condition that could not be solved by fuzzing. Ognawala et al. presented Munch [99], which is an open-source hybrid tool with two modes of operation - fuzzing followed by targeted symbolic execution for parts uncovered by fuzzing, and symbolic execution followed by fuzzing with the inputs generated by symbolic execution as seeds. A similar approach to the latter variant of Munch was also proposed by Wang et al. in SAFL [145] with symbolic execution used to generate seeds for fuzzing. Badger [96] is a hybrid tool that sequentially employs fuzzing and symbolic execution to maximise a resource-related cost instead of merely increasing structural coverage or targeting vulnerabilities. In [150] Zhang et al. have proposed a two-step improvement over state of the art fuzzing and symbolic execution – first, symbolic pointers are only concretised when switching over to fuzzing (in case of saturation) and, then, the inputs to the fuzzers are mutated in a priority queue prioritised by their distance to new structural coverage in the program.

There are *five* intersections of symbolic execution and fuzzing's technical aspects that have not yet been addressed in the available literature. We will discuss this gap later in this chapter.

4.3.3 Summarising the State-of-the-art

The first inference we may draw from the results is that most of the hybrid solution proposals we obtained with our systematic database search were ideas that involve improving either symbolic execution or fuzzing using fuzzing or symbolic execution, respectively, in their original form (i.e. without modification in their technical aspects). Many of the obtained papers from our search and selection strategy did not propose *any* improvement in symbolic execution or fuzzing, at all. However, these papers were not merely included as a side-effect. For example, the paper by Cha et al. [32] proposes a novel solution for optimising seed input generation in fuzzing by tainting bits in input vectors that correspond to certain branching conditions in the program. This is an exciting use of whitebox program information to empower an input mutation strategy. However, since this technique, and others such as [45, 113], do not propose improvements in both symbolic execution and fuzzing, they cannot be considered as genuinely hybrid solutions.

To properly categorise the hybrid solutions, it was essential to classify the technical aspects of their contribution to symbolic execution and fuzzing (if any), thereby creating *9* slots where those techniques could fit that propose improvements to both symbolic execution and fuzzing. As we showed in Section 4.3.2, only 4 of these 9 open avenues for improving both symbolic execution and fuzzing, were seen amongst the solution proposals. The most popular [22, 30, 91] avenue among these has been *static analysis+path search*. As described in Section 4.2, these hybrid solutions, generally, propose to alleviate path-explosion problem in symbolic execution by bypassing easy branching-condition checks with fuzzing. At the same time, the whitebox view obtained from symbolic execution could be used to guide the fuzzer towards more unseen paths than before. The second most popular avenue [36, 106] for hybrid techniques is to use fuzzing to take some load off the inefficient *constraint solving* issues associated with symbolic execution. In the same

papers, static analysis and mutation strategies of fuzzing are also optimised using symbolic execution aspects. The least popular intersection of symbolic execution and fuzzing aspects is *expert guidance+path search*, in which Pham et al. [112] have proposed to use symbolic execution for representing the input structure of programs as models, and using these models to generate seed inputs to increase path diversity with fuzzing.

However, some hybrid papers obtained by us have reported more vulnerabilities in open-source benchmarks than state of the art symbolic execution, fuzzing and hybrid solutions, without proposing a modification in *both*, symbolic execution and fuzzing. *Driller* [133] is such a hybrid study that proposes to improve state of the art by using fuzzing, as usual, to generate inputs for paths executed most often, while modifying symbolic execution using compositional analysis to generate inputs that satisfy branches that are rarely executed (with inputs generated by fuzzing). Another example of a hybrid study that employs, but does not modify, both symbolic execution and fuzzing, is QSym [149]. QSym improves the speed of concolic execution by selectively executing only those instructions symbolically that affect the constraints involved in branching conditions. This improves the performance of many off-the-shelf hybrid fuzzers by decreasing the overhead of concolic execution.

Therefore, we have seen from the existence of hybrid techniques at various intersections of symbolic execution and fuzzing suggests that there are many opportunities to explore w.r.t. more efficient techniques of automatically generating test cases and, hence, finding vulnerabilities in systems. Additionally, none of the solutions included in this chapter include any strategy to prioritise the discovered bugs to make it easier for the developers to triage them.

4.4 Identifying Gaps and Our Contributions

Based on the classification of papers in the field of hybrid symbolic execution and fuzzing research, we can now point to the lack of research in certain unexplored intersections of technical aspects of symbolic execution and fuzzing. The following avenues have not been addressed in existing literature and, hence, constitute the gap in the research in hybrid symbolic execution and fuzzing.

1. A hybrid solution to find vulnerabilities by exploiting a program's *compositional* nature. We have seen from our results that none of the hybrid technique proposals has exploited the compositional structure of the programs-under-test, even though there have been plenty of advances in targeted vulnerability detection with symbolic execution [83, 102] and fuzzing [129], both. With such targeted approaches, *compositional analysis* can alleviate the path-explosion problem in symbolic execution and low path coverage in fuzzing, at the same time.

   Contribution 1: In the dynamic analysis technique described in Part II, we propose to analyse isolated components with symbolic execution, fuzzing and the greybox fuzzing approach mentioned above, instead of only analysing from a program entry point. Using compositional analysis we, then, determine the feasibility of discovered vulnerabilities in isolated components. This compositional approach allows dynamic

analysis to cover more paths and find more vulnerabilities than state-of-the-art techniques.

2. A more targeted symbolic execution approach that spends more analysis time on hard-to-reach branches. In the frame of the listed technical aspects, this can be achieved by exploiting the intersections that, especially, improve *constraint solving*, viz. *constraint solving*+(input mutation, static analysis or expert guidance).

   Contribution 2: In the greybox fuzzing strategy described in Chapter 6, we propose to use inputs generated by fuzzing as seed inputs for concolic execution, thereby taking load off the constraint solver for easy-to-reach branches.

3. A fuzzer with a more efficient input mutation strategy, using targeted search capabilities provided by symbolic execution. In the frame of the listed technical aspects, this can be achieved by exploiting the intersections that improve upon *input mutation* or *expert guidance*, viz. *(input mutation, expert guidance)*+(path search, compositional analysis or constraint solving).

   Contribution 3: In the dynamic analysis technique described in Part II, we propose to a) use the inputs generated by symbolic execution as seed inputs for fuzzing, thereby providing mutation strategy of our fuzzer with diverse inputs, and b) fuzz isolated components of a program, thus allowing the input mutation strategy to generate byte streams for smaller compartments of the program.

4. None of the analysed solution proposals in the field of symbolic execution and fuzzing has discussed a bug triage process that lets testers and developers prioritise the fixing for the vulnerabilities discovered by the proposed solutions. This is an important challenge to be addressed by research because many of the discovered vulnerabilities, especially by fuzzing, can often not be exploited without, first, minimising the generated inputs to generate a valid input that can be realistically used to execute the program-under-test. Also, not all discovered vulnerabilities may be fixed within a feasible amount of time and resources, nor would they all cause similar amount of damage to the underlying asset.

   Contribution 4: We describe a generic framework in Part III to prioritise the vulnerabilities discovered by dynamic analysis, using several features of the program-under-test, vulnerable components, context of their development and usage, and the sensitivity of the assets underlying the vulnerable system.

4.5 Concluding Notes

In this chapter, we have presented an exploration of hybrid techniques of symbolic execution and fuzzing, in terms of their technical aspects. Using a systematic approach to a mapping study, we have shown that only some possible intersections of the technical aspects of symbolic execution and fuzzing have been addressed in designs of hybrid techniques. Most

of the hybrid technique proposals in academia have not utilised the flexibility of individual technical aspects enough, as discussed in Section 4.3.2.

With this map of the state-of-the-art in hybrid techniques, we have provided ample evidence of the gaps that exist therein. We have, with examples of hybrid techniques, argued that an ideal hybrid method will alleviate the drawbacks of *both*, symbolic execution (path explosion and constraint solving) and fuzzing (low coverage). In the following parts of this thesis, we will discuss such a technique, with implementation, that aims to close many of these gaps in research of symbolic execution and fuzzing.

Part II

Vulnerability Discovery

5 Isolating Program Components

This chapter describes the first step of vulnerability discovery – automatically isolating components of the program-under-test to analyse them with dynamic analysis. Parts of this chapter have previously appeared in [102] and [101], where the author of this thesis was the first author.

In the previous chapters, we discussed at length the problems associated with the state of the art in whitebox and blackbox testing. The methods that rely on dynamic analysis (actually running the program-under-test) were found to be insufficient in terms of, both, coverage and vulnerabilities because many paths in the program were not executed by these methods. As a result, the particular vulnerabilities that lie on these paths were, also, not found by these methods.

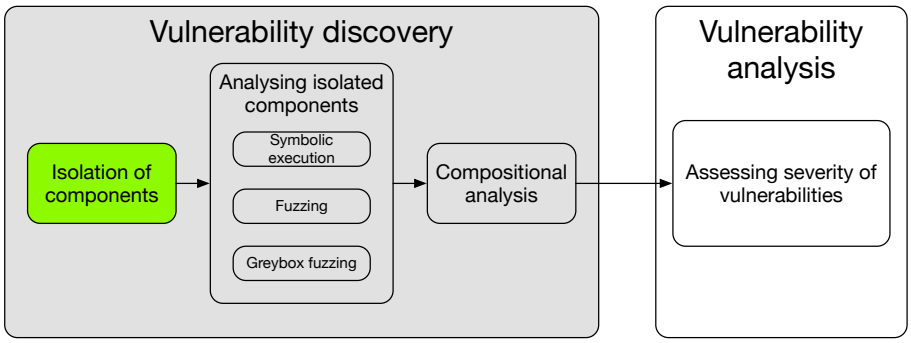


Figure 5.1: *Isolation of components in the solution framework*

The approach described in this thesis can find those vulnerabilities that were missed by symbolic execution and fuzzing by removing branching conditions that need to be *true* before a failure (due to the vulnearbility) can be triggered. In this chapter, we will describe the process of isolating components and making them *executable* by dynamic analysis methods. In Figure 5.1, we have depicted the sequence of steps involved in our framework. The first step of isolating components in the program is highlighted in this figure. We will start in Section 5.1 by describing and formalizing how conventional dynamic analysis methods analyse program entry points. Then, we will introduce in Section 5.2 the concept of *components* and formalise them. Using these two concepts, in Section 5.3, we will describe how our framework can analyse the isolated components as if they were program entry points. We will instantiate this description for C-language programs in Section 5.4 and lay the foundations of test drivers to be adapted by the dynamic analysis techniques of symbolic execution or fuzzing. Finally, we will conclude the chapter in Section 5.5.

5.1 Program Entry Points

We start this chapter by looking at the current state-of-the-practice in dynamic analysis in terms of *how the analysis of a program is initiated.* We begin this discussion by, first,

defining a *program entry point*.

Definition 5.1.1. *(Program Entry Point) A component of a program that is first to be invoked when a program-under-test is run by an external entity (user or another program), and whose inputs are known to this entity is called a program entry point.*

We will define *component* in Section 5.2. For programs on a command-line interface (CLI), such as for a UNIX-like system, the program entry point (or *entry point*) is the `main` function. This is because the inputs supplied on the CLI are used to call the `main` function directly and this input is known to the end-user of the program. Techniques such as symbolic execution and fuzzing typically perform testing by invoking the program with inputs, I (described as I^S in Chapter 2 and I^F in Chapter 3), that are expected by the `main` function. There may also be programs that are not intended to be invoked from CLI but whose interfaces are, nonetheless, known and utilised in other contexts such as implementing third-party programs. A common example of an interface other than CLI may be a *graphical user interface* (GUI). Another example is an *application programming interface* (API) where the programmer of the third-party programs is aware of the interface functions, classes or packages through an API documentation. Symbolic execution and fuzzing, then, perform testing of these APIs by writing *test drivers* [125, 126] that invoke the API by supplying inputs, as described in the documentation.

Formally, we will use the function *exe* (short for "executable") to denote whether a component is a program entry point or not.

$$\mathbf{exe} : M \rightarrow Bool$$

Function *exe* returns $True$ is a component is a program entry point (is first to be invoked when the program is run by an external entity), and $False$ otherwise.

It is clear from the definition of program entry point that only a component, m, that accepts at least one input (Chapter 2) can be a program entry point.

$$\mathcal{I}(m) \neq \emptyset$$

Here, we have used the definition of $\mathcal{I}$ from Chapter 2, i.e. the list of inputs to a component. Intuitively, the above is true because if a program entry point does not accept any input from the user, then any dynamic analysis technique will be able to cover only a single path in the program for any test case. In this thesis, we will ignore such cases where the program-under-test only has one realistic path that can be executed.

5.2 Granularity of Analysis (or Definition of Components)

In the previous section, we described how state-of-the-art whitebox and blackbox testing techniques supply inputs to a program. The goal of this thesis is to design a *scalable* methodology of testing based on these dynamic analysis techniques. Therefore our solution will exploit the *compositionality* of a program-under-test to allow taking lateral views of the program and perform dynamic analysis at multiple entry points. We define compositionality as follows

Definition 5.2.1. *(Composition of Program) The composition of a program described the construction of a program in the form of the following two items: 1. list of components in the program, and 2. for every component in the program, the list of components it may interact with directly.*

Components may be defined specifically for programming languages. Formally, C_L returns the set of all components, m, as defined for a programming language, L. Specifically, let us first consider the case of C programming language. Components in C language programs (as returned by C_C) can be functions or basic-blocks. Similarly, in Java, which is a purely object-oriented programming language, components (as returned by C_{Java}) can be Java classes.

However, as defined above, we must also explicitly describe composition of the program in terms of direct interaction between components. To do this, we define a generic relation $parent_L$.

$$\mathbf{parent_L} : M_L \rightarrow M_L^* \tag{5.1}$$

where M_L is a component, as defined for the programming language, L.

The $parent_L$ relationship has the semantics of a so-called, "container" that lists all the components of a program P that contain the given component in them. For example, in case of C language programs, $parent_C(m_1)$ relationship can be a *caller* relationship that returns all the functions, m_2, that *call* m_1, according to the static call-graph of the program. Then, we have

$$parent_C \equiv caller \tag{5.2}$$

Similarly, for a Java program, $parent_{Java}$ may be a class-based relationship (either through *container* classes, *inherited* classes, or packages containing classes).

A careful extension of the definition of components will show that this definition is *recursive*, i.e. a component itself can be comprised of other components. As a corollary, a program itself can also be treated as a component. Intuitively, this is true because a program can be part of a larger system (of which the program is a component) where it may be called on request, using its program entry points.

5.3 Making Components Executable

After deciding upon the granularity of analysis based on the programming language, the next and final step in isolating components is to make *some or all of the components executable*. The goal in this step is to add more program entry points, so that a dynamic analysis technique (such as symbolic execution or fuzzing) may be able to analyse these components independently of each other, thereby increasing the likelihood of a vulnerability to be discovered.

Throughout this section and, in fact, this thesis, we assume that isolation of components and their analysis will not result in other *side-effects that are outside the scope* of the component. For example, if execution of a component may write to a file on the file-system, that is not read again by the same component, our analysis may not be able to capture effects resulting from a change on the file-system. Similarly, our analysis will also not be able to determine *implicit input* to the isolated components. For example, if a component may be reading the current timestamp from the system clock, our analysis will not be able to determine that the system clock is an implicit input for the component. This also applies to global variables in the program-under-test.

As described in Section 5.1, the requirements for a component, m, to be a program entry point are that 1. m is a component of P, 2. m accepts non-empty input, and 3. m is part of a program's API or executable through a CLI. The first two requirements are trivial and can be checked by, first, checking if $m \in C_L(P)$ and, then, list of formal parameters of m.

To convert any arbitrary component of a program into a potential program entry points, we need a procedure to make all components that satisfy the following condition executable.

$$m \in C_L(P) \wedge \mathcal{I}(m) \neq \emptyset$$

Let us call such a procedure "*isolate*".

$$\textbf{isolate} : M_L \rightarrow M_L,$$

where M_L is a component, as defined for a programming language, L. The function, *isolate*, returns a component ("functionally" equivalent to the input component) that is executable. We can illustrate the working of *isolate* function as follows – let m be a component that is *not* a program entry point, i.e. $exe(m) = False$. Then, $isolate(m)$ returns m', where $exe(m') = True$. The returned component, m', will have the same inputs, outputs and interactions with the environment as the component, m, but with the additional property that it can be executed through a CLI or an API.

5.3.1 Notes on Path Explosion

The motivation behind isolating components is to deal with the path explosion problem described in Chapter 1. Both, symbolic execution and fuzzing achieve insufficient path coverage in real-world programs. The reason for low coverage for symbolic execution is the *path explosion* problem while the random mutations in blackbox fuzzing may fail to generate inputs that execute branches with branching conditions that are hard to satisfy. When a program P's components are isolated, as described earlier, the number of paths to be executed by any dynamic analysis technique get reduced, purely due to the "artificial" removal of the preconditions that need to be true for a component to be executed when starting analysis from the program entry points.

By forcing dynamic analysis of isolated components (with fewer paths than the entire program, P), we now have test cases (for components) that may execute instructions, and find possible vulnerabilities, that could not have been found otherwise.

```
1  int bar1(int c) {
2      if (c<3)
3          return (3/c); /*Maybe divide-by-zero*/
4      else
5          return 0;
6  }
7  int bar2(int d) {
8      if (d<50)
9          return 0;
10     else
11         return d;
12 }
13 int foo(int b, int c, int d) {
14     if (b==100)
15         return bar1(c);
16     else
17         return bar2(d);
18 }
19 int main(int argc, char** argv) {
20     int a, b, c, d;
21     a=atoi(argv[1]); b=atoi(argv[2]);
22     c=atoi(argv[3]); d=atoi(argv[4]);
23
24     if (a<1)
25         return 0;
26     else
27         return foo(b, c, d);
28 }
```

Listing 5.1: *Example C program.*

5.4 Generating Test Drivers: Description of Practice

We will now demonstrate how we apply the theoretical formulation of isolation of a program's components to practice. For this description, let us consider again the example C-program in Listing 5.1. Because this program is in C-language, we consider functions as components. We know from the descriptions in Section 5.2 that $C_C(P)$ returns the set of functions in a given C-program. For the program in Listing 5.1, this means

$$C_C(P) = \{\texttt{bar1}, \texttt{bar2}, \texttt{foo}, \texttt{main}\}$$

We know that for C-programs, the `main` function is executable from, e.g. CLI. Therefore,

$$\begin{aligned} exe(\texttt{bar1}) &= False \\ exe(\texttt{bar2}) &= False \\ exe(\texttt{foo}) &= False \end{aligned}$$

The above equation states that the set of components (functions) that need to be made executable is $\{\texttt{bar1}, \texttt{bar2}, \texttt{foo}\}$. For these components, we need an automated procedure to make them executable. Let us call such a procedure a *test driver*. The goal of test drivers, as we will describe in Chapter 6, is to allow our framework to directly analyse isolated functions with any dynamic analysis method of our choice. In this section, we will

describe a high-level design of the `isolate` function used to generate the test drivers.

Let us consider the function `foo` from Listing 5.1. As such for C programs, the only function executable function from the Unix CLI is the `main` function. Hence, our proposal is to *reformulate* the `main` function such that it directly calls function m *unconditionally*. This is, clearly, not possible by simply passing the same arguments to m that were passed by the CLI to `main` because the list of formal parameters is different. Therefore, before calling m from `main`, we need to *at least* generate the arguments $\mathcal{I}(m)$ from `argv`.

Putting together the above ideas, we can now summarise the function isolation of an arbitrary function, m, as in Algorithm 1.

Algorithm 1 Making function m executable

```
1: function MAIN'(&m, argv)
2:     I ← GETPARAMETERLIST(m)
3:     extracted_I ← EXTRACTARGUMENTS(argv, &I)
4:     return m(extracted_I)
5: end function
```

On line 2 of Algorithm 1, it runs an automated procedure to obtain the list of formal parameters of m in their respective datatypes. On line 3, the algorithm runs an automated procedure to convert `argv`, which is passed from Unix CLI, to the form of `I`. The function `getParameterList` obtains the list of formal parameters (and their types) given the function's definition, and the function `extractArguments` extracts the concrete or symbolic values for actual arguments to be passed to the function m from command line arguments $argv$. We will discuss the algorithms and implementations of `getParameterList` and `extractArguments` in Chapter 6, since these functions depend on the type of dynamic analysis technique used. Finally, on line 4, the algorithm calls the isolated component m with the extracted arguments $extracted\_I$.

From Algorithm 1, we can see that the function m can be called unconditionally whenever the program P is invoked from the CLI. As such, we can say for this program that $main'$ is the required executable function m'.

Technical perspective: In Algorithm 1, m is the function that needs to be made executable and "&m" is the address of the code segment of the particular function that needs to be isolated.

Concretely, replacing `main` with `main'` is not technically possible because C programs can only have a single `main` function. Therefore, our $main'$ must accommodate all isolated functions, $m \in C_C(P)$, in a single function. In our framework, we solve this problem by using a *switching* mechanism in $main'$ that selects the test driver m' based on the choice provided on the CLI.

5.4.1 Implementation Details

The sequence of steps in the implemented framework to isolate components of a C program and generate test drivers is illustrated in Figure 5.2.

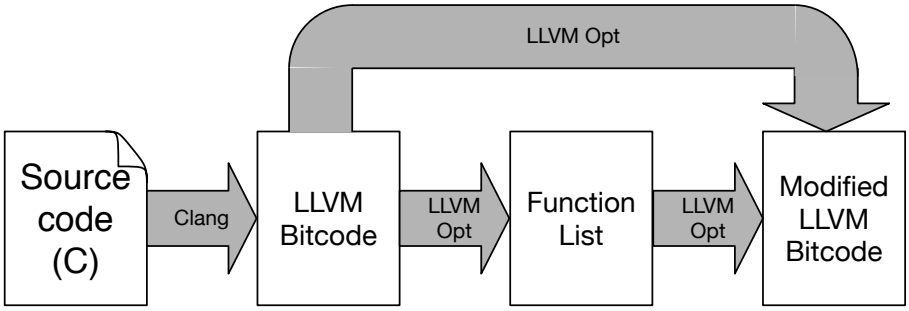


Figure 5.2: *Technical implmentation of test driver creation*

We have used LLVM Opt [82] for implementing most of the steps in Figure 5.2. First of all, using Clang [42], we compile the program-under-test to LLVM intermediate representation, which is a cross-platform bitcode. Then, we *rename* the existing `main` function to `main_old`. Then, using LLVM Opt, we list all functions that take at least one argument of a non-pointer or single-pointer datatype[1]. For all such functions, we add them to the `switch-case` block in a new `main` function, also using LLVM Opt. This `main` function is then inserted in the bitcode.

LLVM [79] is a cross-platform intermediate representation of programs written in several programming languages. The reason for using an LLVM-Opt pass for generating test drivers is that, in practice, manipulating the original source-code of the program (after adding test drivers) will require recompiling it for running dynamic analysis for every test driver. With LLVM-Opt, the manipulated intermediate bitcode can be directly used by our framework to run the analysis, as described in Chapter 6.

5.5 Concluding Notes

In this chapter, we first described the theory of isolated components by, first, considering the program as a collection of interacting components and, then, formally describing program entry points. We, then, proposed to make individual components (accepting inputs from outside) executable by isolating them so that an end-user may be able to invoke them directly, without going through all the inter-component interactions intended by the original program. Isolation of components allows a dynamic analysis technique, such as symbolic execution, to directly analyse paths inside a component without applying the pre-conditions on the input that are imposed by its *parent* components. In this way, our framework alleviates the path-explosion problem by forcing the execution of paths in isolated components where potential vulnerabilities may lie. We also described the automatic generation of test drivers, as implemented in the framework described in this thesis, which is the practical implementation of the theory of isolation of components.

Once the components of a program-under-test are isolated, they are now ready to be analysed by dynamic analysis techniques, which will be described in the next chapters.

[1] We will discuss in Chapter 6 why arguments of double pointers or more cannot be supported in our framework

6 Analysing Isolated Components

This chapter describes the second step in the discovery process – parallel analysis of isolated components using symbolic execution, fuzzing and a novel greybox fuzzing approach. Parts of this chapter have previously appeared in [102], [99] and [101], where the author of this thesis was the first author.

In the previous chapter, we described in detail the idea of isolation of components of programs and instantiated this idea for C programs where the components considered were *functions*. Now we will move one step further and describe in detail analysis of isolated components generated by the first step described in Chapter 5.

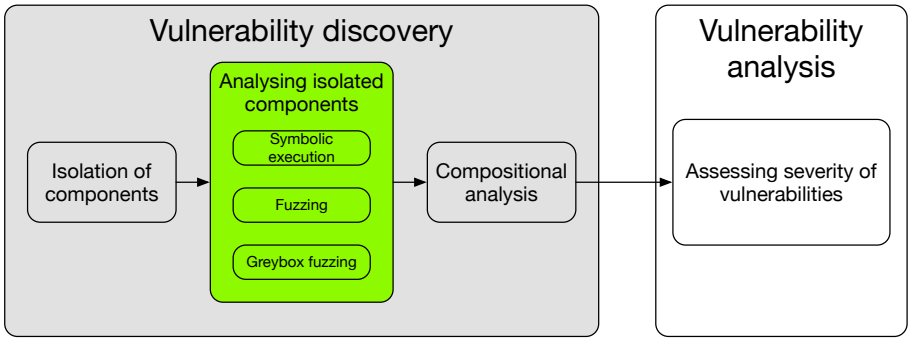


Figure 6.1: *Analysing isolated components in the solution framework*

Analysis of isolated components is illustrated in Figure 6.1 as the step following isolation of components. When dynamic analysis techniques such as symbolic execution and fuzzing are used to analyse isolated components instead of the full program at the program entry points, the instructions in these isolated can be directly executed without first having to pass the branching conditions guarding these components. This is because, as described in Chapter 5, these isolated components can now be directly executed by the end-user, as can they be by these dynamic analysis techniques.

In this chapter, we will, first, formalise some common concepts related to paths and failures in isolated components, in Section 6.1. Then, using these concepts and the ones developed in the previous chapters as our base, we will describe the design and implementation of our analysis framework with three modes of analysis – 1. symbolic execution (Section 6.2), 2. fuzzing (Section 6.3), and 3. greybox fuzzing (Section 6.4). The first two of these will be the usual state-of-the-art symbolic execution and fuzzing techniques applied to isolated components. For both these techniques, we will discuss their limitations in terms of when they saturate for components that are complex in terms of their path conditions. The third technique, greybox fuzzing, is a novel adaptive technique developed by us based on our insights from the saturation of symbolic execution and fuzzing. After describing the three techniques of analysing isolated components, we will

list the output of the analysis (including vulnerabilities) in Section 6.5. Finally, we will conclude this chapter in Section 6.6.

6.1 Formalising Paths and Failures

We will start this chapter by listing and defining some common terminologies to be used in the rest of this chapter to describe the progress of dynamic analyses.

First of all, let us revisit the concept of *paths*, as applied to programs-under-test. In Chapter 2, we described *path conditions*, which are the conjunction of branching conditions from a program entry point to any program exit points. We can also apply the same idea to an isolated component, m. To further this discussion, we first define in-component paths as follows.

Definition 6.1.1. *(In-component Path) For isolated components, paths are a conjunction of the branches that must be taken for the program execution to start at the invocation of the component (e.g. by another component) and end at one of the following – 1. an* `exit` *statement or an equivalent instruction to end execution of the component and return to a parent component, or 2. caught or uncaught exceptional behaviour (e.g. failure), or 3. an instruction executed before exiting due to (externally imposed) timeout.*

Extending the definition of paths, a *path condition* is a first-order logic formula over the inputs, $\mathcal{I}(m)$[1] of the component m and any external interactions with the environment, such as reading or writing to files.

For describing the techniques in this thesis, we will always consider unique inputs, which are defined as follows.

Definition 6.1.2. *(Unique Inputs) The set of unique inputs contain those inputs for a component which execute the unique in-component paths.*

For illustrating *unique* in-component paths let us consider two path conditions (for two paths executed by dynamic analysis), pc_1 and pc_2. If,

$$pc_1 = \bigwedge_i b_i$$
$$pc_2 = \bigwedge_j b_j$$

where b_i and b_j are branching conditions that are true for the respective paths executed, then pc_1 and pc_2 are identical if the number of branching conditions is the same for pc_1 and pc_2 and all b_i are the identical to b_j. If there exists no pc_n, $n \neq 1$ that is identical to pc_1, then pc_1 is a unique in-component path.

Now, let I be the set of unique inputs (provided manually or generated automatically) and PC be the set of corresponding path conditions executed by those inputs for a component. I.e.

[1] Please note that while I lists of actual/passed parameters (or inputs) generated by dynamic analysis, $\mathcal{I}(m)$ denotes the list of formal parameters of component m.

$$PC(m) = \{pc_1, pc_2 \dots\} \tag{6.1}$$
$$I(m) = \{i_1, i_2 \dots\} \tag{6.2}$$

In Equation (6.1), an input $i_k \in I(m)$ is said to be unique because it executes a unique path with path condition $pc_k \in PC(m)$

Failures can, then, be represented as in-component paths too. As shown in Equation (6.1), all unique paths discovered by dynamic analysis and their corresponding inputs are included in PC and I, respectively. If we denote failures as in-component paths, with set of path conditions PC_{fail}, then

$$PC_{fail}(m) \subseteq PC(m) \tag{6.3}$$
$$I_{fail}(m) \subseteq I(m) \tag{6.4}$$

With the above definitions of in-component paths, inputs and failures, we can now describe the design of our techniques for dynamically analysing isolated components.

6.2 Analysing Components with Symbolic Execution

The first technique that we will describe for analysing isolated components is *symbolic execution*. We recall from Chapter 2 that symbolic execution utilises the concepts of *symbolic inputs*, *branching* and *path conditions* to generate inputs that execute different paths in the program-under-test. In this section, we will adapt these concepts to isolated components that we created in the previous chapter.

We know that a component is made executable by the process of isolation. Let us consider such a component, m, with $exe(m) = False$. Then, we have $exe(m') = True$, where $m' = isolate(m)$ (see Chapter 5).

For analysing components with symbolic execution the inputs to the components must be, first of all, made symbolic, i.e. if $\mathcal{I}(m)$ is the list of formal parameters for m (and, consequently, for m'), then let **makeSymbolic** be a function that returns the symbolic arguments from formal parameters.

$$\textbf{makeSymbolic} : i \rightarrow i^S,$$

where $i \in \mathcal{I}$ and $i^S \in \mathcal{I}^S$. Here $\mathcal{I}^S$ is the symbolic representation of an input $\mathcal{I}$ to a component. Adapting the definitions for in-components paths, symbolic inputs are the abstraction of the actual data expected by the component that does not *yet* have a concrete value assignment but can take any value depending on the external constraints of the component ($True$ in case of isolated components). The in-component path conditions can be sent to a constraint solver, such as SMT solver, to solve for symbolic inputs and generate a concrete value assignment. Finally, as described in Section 6.1, failures (such as program crash) can be represented as a subset of paths discovered by symbolic execution

of the isolated component and the corresponding subset of inputs can be saved as *exploits* for the components.

6.2.1 Adaptation of Test Drivers

Let us now consider the formulation of symbolic component analysis and apply that to the practical case of C programs with functions as components.

Our goal in the analysis step is to generate an automated process to allow symbolic execution of isolated C functions, $m \in C_C(P)$. In Chapter 5, we described some algorithms to make m executable but left the functions `getParameterList` and `extractArguments` (Algorithm 1) abstract to be expanded upon by the technique of analysis, such as symbolic execution. We recall the function isolation algorithm and adapt it for m and symbolic execution in Algorithm 2.

Algorithm 2 Making a function $m \in C_C(P)$ executable for symbolic execution

```
1: function MAIN'(&m, argv)
2:     I ← GETPARAMETERLIST(m)
3:     sym_I ← EXTRACTARGUMENTSSYMBOLIC(argv, &I)
4:     return m(sym_I)
5: end function
```

For analysing the isolated component, m, with symbolic execution, we will now expand the algorithm for `extractArgumentsSymbolic`. This algorithm is shown in Algorithm 3.

Algorithm 3 Generating symbolic arguments from given list of arguments for an isolated function

```
1: function EXTRACTARGUMENTSSYMBOLIC(mainArgv, I)
2:     for i ∈ I do
3:         type ← TYPEOF(i)
4:         if type ≠ pointer then
5:             i ← EXTRACTNONPOINTERTYPESYMBOLIC(type, &i)
6:         else
7:             if type == pointer then
8:                 basicType ← TYPEOF(*i)
9:                 i ← EXTRACTPOINTERTYPESYMBOLIC(basicType, &i)
10:            end if
11:        end if
12:    end for
13: return I
14: end function
```

In the algorithm listed in Algorithm 3, for every formal parameter i in list $\mathcal{I}$, we generate an equivalent symbolic argument to be passed to the invocation of the function. The exact function to extract symbolic arguments depends on whether the argument is of a basic non-pointer C-datatype, such as `int`, `char` or `double`, or a dynamically or statically allocated single-pointer C-datatype, such as `(int *)` or `(char *)`.

Extraction of a non-pointer datatype is shown in Algorithm 4. In this case, the algorithm uses the (implementation-dependant) type size of the basic C-datatype of the parameter i and passes it to `makeSymbolic` to get a symbolic equivalent of the parameter.

Algorithm 4 Generating symbolic arguments for non-pointer datatypes

```
1: function EXTRACTNONPOINTERTYPESYMBOLIC(basicType, &i)
2:     size = SIZEOF(basicType)
3:     sym_i ← makeSymbolic(size)
4: return sym_i
5: end function
```

Extraction of arguments of single pointer C-datatype is shown in Algorithm 5. For pointer arguments, this algorithm "unravels" [102] the structure pointed to by the pointer (such as a C-structure or a statically or dynamically allocated array) and assigns a symbolic equivalent to each element of the structure.

Algorithm 5 Generating symbolic arguments for pointer datatypes

```
1: function EXTRACTPOINTERTYPESYMBOLIC(basicType, &i)
2:     for ind ∈ ENUMERATE(i) do
3:         elementSize ← SIZEOF(i[ind])
4:         sym_i[ind] ← makeSymbolic(elementSize)
5:     end for
6: return sym_i
7: end function
```

We see from the above algorithms that we have only considered two types of function parameters for symbolic execution – non-pointer types and single-pointer types. We exclude any parameters of double- or more pointers because of the difficulty of expanding them element-by-element, e.g. for 2-dimensional arrays, and the exponential number of possibilities of allowed sizes in them. Furthermore, for parameters of pointers to C-structure type (`struct`), we handle them by unravelling the structure and applying the procedures listed in Algorithm 4 or Algorithm 5 on individual fields of the structure, depending on their datatype. This technique works too if the structure parameter contains a field that itself is a pointer to another structure (the pointed structure is also unravelled, and the same methodology applied recursively). If the structure itself contains double-pointers, we exclude that function from the analysis.

The test driver can now be used by an off-the-shelf symbolic execution engine to generate test cases (value assignments for the symbolic arguments) executing different paths, including failures, in C functions.

Implementation

Algorithms for generating test drivers for analysis with symbolic execution and extracting symbolic arguments are all implemented in our framework using LLVM Opt [82]. We refer the reader to Figure 5.2. In this figure, the modified bitcode contains drivers for testing

functions in the program-under-test with symbolic execution. The bitcode is modified by applying a pass over the LLVM bitcode of the program-under-test generated using Clang [42] to generate a driver for every isolated function (with at least one parameter and no parameters with double- or more pointer type).

Finally, we utilise KLEE [26] for symbolically executing the function drivers, as they can now be invoked directly from the CLI.

6.2.2 Notes on Saturation

We described above the methodology of analysing isolated components with symbolic execution by extracting symbolic arguments using the formal parameter list and executing the isolated function with an off-the-shelf symbolic execution engine. However, as described in Chapter 2, symbolic execution faces many challenges due to which it may only be able to achieve incomplete analysis in many real-world programs. These limitations of practical symbolic execution are also true for analysis of isolated components. In this section, we discuss some *saturation indicators* signalling that symbolic execution is found to no longer be progressing by generating new test cases to achieve a given analysis goal, i.e. vulnerability discovery.

1. **Path-level saturation:** One of the main saturating factors affecting symbolic execution currently [29] is the well-known path-explosion problem, that we discussed at length in Chapter 2. Due to path-explosion in isolated functions (e.g. due to input-dependent loops), our symbolic execution engine KLEE [26] might not be able to generate new test cases within an externally imposed timeout. Therefore, test case generation is the first indicator of whether or not symbolic execution is saturated.
2. **Constraint-level saturation:** The second problem associated with symbolic execution is the bottleneck of constraint solver. As discussed in Chapter 2, the constraint solver being used by our symbolic execution may also not return a satisfiability solution or generate a test case within an externally imposed timeout. In this case, the factor responsible for saturation of symbolic execution may be the size or the complexity of the path condition [73]. The indicator of constraint-level saturation is the amount of time spent by the symbolic execution engine in waiting for a solution from the underlying decision procedure, such as SAT or SMT.
3. **Instruction-level saturation:** Saturation in terms of new instructions (source-code level or LLVM bitcode level) covered by symbolic execution may often be a result of the above two modes of saturation. Path-explosion may mean that, even though new paths may be continuously uncovered, those paths may not necessarily include previously unseen instructions. Similarly, with constraint-level saturation, symbolic execution may not be able to cover instructions that can only be executed if a "hard" branching condition is first solved by the constraint solver. The advantage of using instruction-level monitoring as opposed to paths or constraints is that it is relatively more straightforward to detect.

Depending on the goal of analysis, any of the above types of saturation can be monitored by an external process to determine whether symbolic execution has made any progress in

a reasonably large amount of time, and decide whether the analysis should be stopped. Please note that the above saturation indicators are not mutually exclusive, i.e. two or more of them could be true at the same time.

6.3 Analysing Components with Fuzzing

We will now describe analysis of isolated components with fuzzing. We recall from Chapter 3 that fuzzing is a way to automatically generate test cases for a program-under-test by continuously selecting from the set of existing inputs and mutating them, based on some mutation strategies, to uncover previously unseen functional behaviour from the program. With fuzzing, the generated test cases can be considered proxies for different paths executed in the program. In this way, the reported paths include, just as with symbolic execution, failures discovered by fuzzing. In this section, we will adapt the formalisations from Section 6.1 to analyse isolated components with fuzzing.

We recall from Chapter 5 the function exe that describes executable component m' equivalent to the isolated component m. For fuzzing, the key idea is to be able to invoke the fuzzer with seed inputs, I^F_{init}, which are some starting values assignments for the arguments of a function. Generating seed inputs is an automated procedure in our framework that generates default value assignments for formal parameters, $\mathcal{I}(m)$. Let us call this procedure **generateSeedInput**.

$$\textbf{generateSeedInput} : M \rightarrow \mathcal{I}^F,$$

where $\mathcal{I}^F$ is the fuzzing input for a component. Please note that the step of generating and using seed inputs was not present in symbolic execution, but is an additional step required for fuzzing.

After generating seed inputs, the next step is an automated procedure to supply seed inputs to the execution of the component, m'. With these inputs, the fuzzer is able to generate new inputs based on the monitoring results of the execution of the isolated component (Section 3.4). These new inputs are added to the list of existing inputs and the mutation process continues until a coverage goal or a time-limit is reaches.

6.3.1 Adaptation of Test Drivers

Let us now describe the adaptation of the test driver described in Section 5.4. Our goal in this analysis step is to generate an automated process to allow fuzzing of an isolated C function, $m \in C_C(P)$. As with symbolic execution, we first describe the high-level algorithm for making m executable, in Algorithm 6.

The reader will note that the design of a fuzzing-based driver is almost the same as one for symbolic execution. The reason for this is so that the same drivers can be re-used based on whether the choice of analysis method is symbolic execution or fuzzing.

We will now expand the algorithm `extractArgumentsFuzzing` in Algorithm 7.

Algorithm 6 Making a function m executable for fuzzing

```
1: function main'(&m, argv)
2:     I ← getParameterList(m)
3:     fuzzed_I ← extractArgumentsFuzzing(argv, &I)
4:     return m(fuzzed_I)
5: end function
```

Algorithm 7 Generating fuzzed arguments from given list of arguments for an isolated function

```
1: function extractArgumentsFuzzing(mainArgv, I)
2:     remData ← mainArgv
3:     remSize ← sizeof(remData)
4:     for i ∈ I do
5:         type ← typeOf(i)
6:         if type ≠ pointer then
7:             i ← extractNonPointerTypeFuzzing(type, &i, &remData, &remSize)
8:         else
9:             if type == pointer then
10:                basicType ← typeOf(*i)
11:                i ← extractPointerTypeFuzzing(basicType, &i, &remData, &remSize)
12:            end if
13:        end if
14:    end for
15: return I
16: end function
```

The algorithm `extractArgumentsFuzzing` is significantly different than `extractArgumentsSymbolicExecution` (Algorithm 3) and the difference lies in how the values for the function arguments are extracted. In symbolic execution, our framework *generates* these value assignments using the underlying constraint solvers of the symbolic execution engine. However, in fuzzing-based analysis, the value assignment must be *derived* from the raw input generated by the fuzzer. The reason for this difference is because of how the fuzzers are designed. Fuzzers, in practice, operate on raw byte-streams that are saved, selected and mutated to generate new byte-streams that, first of all, need to be type-cast to the input expected by the program-under-test. For programs meant to be executed on CLI, byte-stream inputs are easy to be cast directly to `argv`. However, for isolated functions, a byte-stream needs to be first type-cast into the datatypes of the parameter list, $\mathcal{I}(m)$. To automate the process of extracting the exact number of bytes from the input byte-stream and casting to the expected datatype, we keep track of the number of bytes (`remSize`) in the byte-stream (`remData`) and call the relevant extraction algorithm based on whether a parameter is of a non-pointer or pointer type.

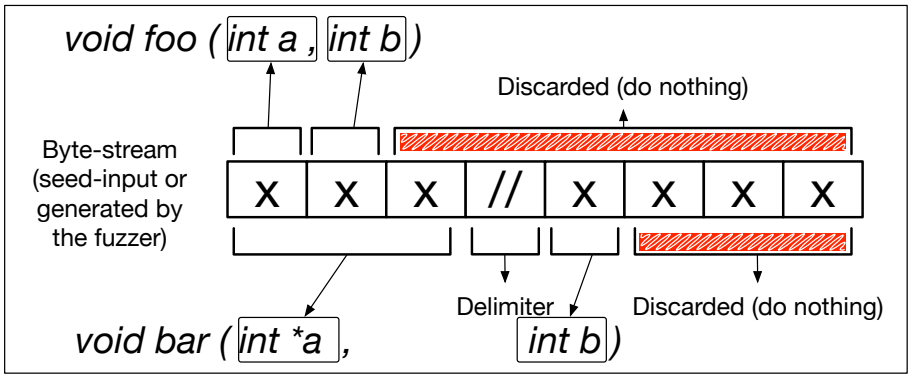


Figure 6.2: *Splitting a byte-stream (generated by the fuzzer) and extracting function arguments*

The automated process of extracting arguments for functions is illustrated in Figure 6.2, where we describe the intuition of `extractArgumentsFuzzing` procedure visually. The algorithm for extracting non-pointer datatypes is shown in Algorithm 8. This algorithm copies `typeSize` number of bytes from the byte-stream to the argument and casts it to the required `type`. If there are not at least `typeSize` number of bytes left in the stream to continue assigning, the stream is padded with null character bytes.

For parameters of pointer types, our framework utilises a special *delimiter* character in the byte-stream that splits it for assigning to dynamically-sized parameter types. As shown in Algorithm 9, `extractPointerTypeFuzzing` copies the generated byte-stream to `i` till the first delimiter character is encountered, or the end of the byte-stream has been reached. The helper function `lookForDelimiter` returns the location of the first delimiter character in the byte-stream or the end of the stream, whichever appears first. The helper function `roundDown` rounds `givenSize` down to a multiple of `typeSize`. We leave it up to the mutation strategies of the underlying fuzzer to insert the delimiter character at appropriate positions in the input byte-stream. Pointers, structures and structures with pointer fields are all handled the same way as they are with symbolic execution (Section 6.2).

Algorithm 8 Generating fuzzed arguments for non-pointer datatypes

```
1: function extractNonPointerTypeFuzzing(basicType, &i, remData, remSize)
2:     i ← basicType(i)
3:     typeSize ← sizeof(basicType)
4:     if typeSize ≤ remSize then
5:         i ← basicType(remData[: typeSize])
6:         remData ← remData[typeSize :]
7:         remSize ← remSize − typeSize
8:     else
9:         i[0 : typeSize] ← copy('\0', typeSize)
10:        i[0 : remSize] ← remData
11:        remData ← [ ]
12:        remSize ← 0
13:    end if
14: return i
15: end function
```

Algorithm 9 Generating fuzzed arguments for pointer datatypes

```
1: function extractPointerTypeFuzzing(basicType, &i, remData, remSize)
2:     i ← basicType(i)
3:     typeSize ← sizeof(basicType)
4:     int bufSize, int givenSize
5:     givenSize ← lookForDelimiter(remData, remSize)
6:     bufSize ← roundDown(givenSize, typeSize)
7:     i ← remData[: bufSize]
8:     if (bufSize ≤ remSize) then
9:         remSize ← remSize − givenSize
10:        remData ← remData[givenSize :]
11:    else
12:        remainingData ← []
13:        remainingSize ← 0
14:    end if
15: return i
16: end function
```

Seed argument generation

We will now briefly discuss the generation of seed inputs for isolated function in C programs. The goal of this step is to generate some default and initial value assignments (*seed arguments*) for function arguments that can be used by the fuzzer for starting its process. Our methodology to generate seed arguments is a *static* methodology, i.e. without executing the function first, e.g. with symbolic execution. We propose two different ways of generating seed arguments for isolated functions.

1. The framework generates two byte-streams – an *empty* stream and a stream consisting of a random ASCII character.
2. The framework generated a byte-stream with as many delimiter characters in the seed argument as there are formal parameters of the isolated function. With this method, it is easier (possible with fewer cycles of input mutation) for the fuzzer to generate non-empty byte-streams for *all* function arguments.

With the design of test driver and seed argument generation, the test driver can now be used by an off-the-shelf fuzzer to generate test cases executing different paths, including failures, in C function.

Implementation

Just like with symbolic execution, the above algorithms for generating test drivers, extracting arguments from the byte-streams generated by the fuzzer for pointer and non-pointer type arguments of isolated functions are all implemented as LLVM Opt passes in our automated framework. We used ASAN [1] for instrumenting the Clang-compiled binary and replaying the test cases generated by the fuzzer with this instrumented version. The goal of this step was to find memory-related vulnerabilities and stack-traces of failures (crashing executions) to be used for further analysis. For fuzzing, we use AFL [2], which is a popular off-the-shelf fuzzer for programs on CLI and has helped in finding various vulnerabilities in many open-source and proprietary software in the past.

6.3.2 Notes on Saturation

As with symbolic execution, with fuzzing too, the analysis of programs-under-test may be incomplete, even over a considerably large amount of time, mainly due to the current practical challenges facing guided fuzzing that we detailed in Chapter 3. These challenges also affect the analysis of isolated components, because of which the dynamic analysis may get *saturated* after a certain amount of time. We will now list and briefly discuss two indicators that analysis of isolated components with fuzzing may be saturated.

1. **Mutation-level saturation:** The first saturation indicator may be the number of inputs that our off-the-shelf fuzzer, AFL [2], has generated so far and the rate of increase thereof. In many scenarios when the input first needs to be parsed according to a strict protocol, the fuzzer may get stuck in the "shallow" (in the function's CFG) parts of the function trying to generate valid inputs to satisfy certain

branching conditions [99]. In such cases, the fuzzer might have run through all possible mutations from the mutation strategies, but still, be unable to generate new test cases to execute previously unseen paths in the function. One possible reason for this type of saturation may be that the set of seed inputs is small or not diverse enough to allow our fuzzer to exercise new behaviour in a function.

2. **Instruction-level saturation:** An instruction-level saturation, just like with symbolic execution, may be an indication that the fuzzer has not covered any new instructions, and, therefore, new paths, in the function. As always, unlike mutation-level saturation where the mutation strategy may not be transparent to the tester, instruction-level saturation is easier to monitor by keeping track of all instructions (in the source-code or binary level) that have been executed by at least one test case generated by the fuzzer.

The above reasons and indicators may be useful in signalling to an automated testing strategy that the current strategy of fuzzing has reached a plateau in terms of its effectiveness in achieving the desired coverage goal.

6.4 Analysing Components with Greybox Fuzzing

In the previous sections, we saw that, for analysing isolated components, symbolic execution and fuzzing may only be effective up to a limit in terms of paths executed, branches and instructions covered or test cases generated. We propose in this section that after any of these saturation limits are reached for symbolic execution, we should adapt our framework to switch to fuzzing, and vice-versa. The reason for making this design decision is based on our claims and findings in Chapter 4, where we discussed at length how symbolic execution and fuzzing may be able to complement each other when one of them saturates.

In this section, we will describe *greybox fuzzing*, which is a hybrid of symbolic execution and fuzzing that can detect saturation of a technique during analysis and switch to a different technique when saturation occurs. The key idea here is to start analysing an isolated component with either symbolic execution or fuzzing. During the entire analysis, we let a *saturation monitor* run in the background which will continuously watch whether the analysis has reached any of the saturation points listed in Section 6.2.2 or Section 6.3.2. If the saturation monitor detects saturation, our framework saves the results (test cases and stack-traces) of the analysis and supply the test cases to the other technique (fuzzing or symbolic execution) for continuing the analysis. By *sharing inputs* between them, our framework can preserve the information about coverage achieved by either of the techniques, to avoid redundant analysis in the isolated component as much as possible. The framework will keep switching between fuzzing and symbolic execution in this way till, either, a coverage goal is achieved (e.g. LLVM instruction coverage) or a reasonable time-limit is reached.

Sharing inputs: For fuzzing, the procedure of using test cases supplied by symbolic execution is trivial – a generated test case (as a solution to a path condition generated by symbolic execution) can be converted to a byte-stream ready to be accepted by the fuzzer.

However, the equivalent procedure for symbolic execution to reuse test cases generated by a fuzzer is less straightforward. As described in Chapter 2, *concolic execution* is a specialised practical implementation that combines concrete execution with purely symbolic execution. We employ concolic execution, to share inputs from fuzzing in the following way – the test cases generated by the fuzzer are consumed by the concolic execution engine to run the isolated component and collect the paths that are executed by them. For each branch, b, taken by an input in an executed path's condition, the concolic execution engine adds a path condition with the negation of the branch $\neg b$ as a possible path condition that *was not* executed by any of the test cases generated by the fuzzer. In this way, by leveraging concolic execution and fuzzing, we relieve the constraint solver of the symbolic execution engine of the load of generating and solving those path conditions that were already found by the fuzzer, thereby allowing it to focus on previously unseen paths.

6.4.1 Adaptation of Test Drivers

Let us now describe the adaptation of the test driver described in Section 5.4. For this, we will bring together the designs of test drivers for analysing isolated components with symbolic execution (Section 6.2.1) and fuzzing (Section 6.3.1). Our goal in this step, as always, is to analyse an isolated function, $m \in C_C(P)$, with greybox fuzzing, i.e. monitoring and switching between symbolic execution and fuzzing whenever one technique saturates. The high-level algorithm of a test driver for m with greybox fuzzing is listed in Algorithm 10.

Algorithm 10 Making a function m executable for greybox fuzzing

```
1:  function main'(&m, argv)
2:      I ← getParameterList(m)
3:      sym_I ← extractArgumentsSymbolic(argv, &I)
4:      fuzzed_I ← extractArgumentsFuzzing(argv, &I)
5:      while True do
6:          fuzzing_progress ← ∅
7:          symbolic_execution_progress ← ∅
8:          fuzzed_I ← extractArgumentsFuzzing(sym_I, &fuzzed_I)
9:          while fuzzingNotSaturated(fuzzed_I) do
10:             fuzzing_progress ← m(fuzzed_I)
11:         end while
12:         sym_I ← extractArgumentsSymbolicExecution(fuzzed_I, &sym_I)
13:         while symbolicExecutionNotSaturated(sym_I) do
14:             symbolic_execution_progress ← m(sym_I)
15:         end while
16:         if fuzzing_progress == ∅ ∧ symbolic_execution_progress == ∅ then
17:             break
18:         end if
19:     end while
20: end function
```

In this algorithm, first of all, the symbolic and fuzzing inputs are extracted from the

same, and unchanged, formal parameter list of the function m. Then, we enter a continuous `while`-loop that, first of all, converts the symbolic input, sym_I to $fuzzed\_I$ by reusing the `extractArgumentsFuzzing` procedure described in Section 6.3.1. Reusing `extractArgumentsFuzzing` is equivalent to populating seed inputs for fuzzing, by converting symbolic input to byte-stream input format accepted by most fuzzers. Next, we run the fuzzing process, by calling the isolated function m with $fuzzed\_I$, till the fuzzer reaches saturation. In Section 6.4.2, we will describe in some detail the `fuzzingNotSaturated` algorithm based on the ideas listed in Section 6.3.2.

When fuzzing saturates based on some criteria, we convert the test cases/inputs generated by the fuzzer to symbolic input, sym_I, by reusing the `extractArgumentsSymbolic` procedure described in Section 6.2.1. Just as with fuzzing, the test cases generated by fuzzing are converted to concrete inputs for concolic execution of isolated function, m. With these inputs as seed inputs, we symbolically execute m till symbolic execution reaches saturation. Based on the ideas listed in Section 6.2.2, we will describe the `symbolicExecutionNotSaturated` algorithm in Section 6.4.2.

Finally, when both, fuzzing and symbolic execution, have been saturated, the outer `while`-loop will end (line 16-17) and the analysis terminated.

6.4.2 Monitoring Saturation

Let us now discuss the details of how our framework actively monitors saturation of fuzzing and symbolic execution, to determine when to switch over to the other technique. We have described the indicators and factors leading to saturation of symbolic execution (Section 6.2.2) and fuzzing (Section 6.3.2) earlier in this chapter and we will use these concepts for designing the monitoring mechanism too.

Saturation of Fuzzing

Let us start with the algorithm for determining whether fuzzing is saturated during the analysis of an isolated component. The function `fuzzingNotSaturated`, which was called on line 7 in Algorithm 10 is listed in Algorithm 11.

In Algorithm 11, the basic idea is to use the following (externally set) thresholds

1. *Test-case* threshold is the maximum number of test cases generated by the fuzzer for which the framework is willing to wait before signalling that no new instructions have been covered for a sufficiently long time. If there is at least one test case in the `test_case_threshold` most recently generated test cases that cover at-least one new instruction, then the framework signals that fuzzing is not saturated yet.
2. *Time* threshold is the maximum amount of time that our framework will wait for new instruction coverage, after which it will signal that fuzzing is saturated. We need this threshold based on time in cases when even though no new test cases were generated for a very long time, there was at least one new instruction covered by the previous `test_case_threshold` test cases. In such a case, without a time-based threshold, the framework might wait forever without realising that none of the input mutations performed by the fuzzer have resulted in any progress of the analysis.

Algorithm 11 Monitoring saturation of fuzzing

```
1: function FUZZINGNOTSATURATED(fuzzed_I)
2:     not_mutated ← []
3:     for i ∈ fuzzed_I do
4:         if MUTATED(i) then
5:             not_mutated.APPEND(i)
6:         end if
7:     end for
8:     if LEN(not_mutated)> 0 then return True
9:     else
10:        for i ∈ fuzzed_I[−TEST_CASE_THRESHOLD :] do
11:            if NEWCOVERED(i)> 0 then return True
12:            end if
13:        end for
14:    end if
15:    if (LASTNEWINSTRUCTIONCOVERAGE(fuzzed_I)< TIME_THRESHOLD)
   then return True
16:    else return False
17:    end if
18: end function
```

Saturation of Symbolic Execution

Now, we describe the algorithm for determining whether symbolic execution is saturated during the analysis of an isolated component. The function `symbolicExecutionNotSaturated`, which was called on line 11 of Algorithm 10, is listed in Algorithm 12.

Algorithm 12 Monitoring saturation of symbolic execution

```
1: function SYMBOLICEXECUTIONNOTSATURATED(sym_I)
2:     if TIMECONSTRAINTSOLVERENGAGED( )< CONSTRAINT_SIZE_THRESHOLD
   then
3:         if (LASTNEWINSTRUCTIONCOVERAGE(sym_I)< TIME_THRESHOLD)
   then return True
4:         else return False
5:         end if
6:     elsereturn False
7:     end if
8: end function
```

In Algorithm 12 the basic idea that if currently, the underlying constraint solver is engaged by the symbolic execution engine (e.g. for generating a test case), then the framework must check if the time since it has been engaged is less than the (externally set) threshold on how long must the constraint solver take to return either a UNSAT solution or a concrete test case for the current path condition. If the constraint solver returns within the defined time limits, then we proceed to check, similar to Algorithm 11, how long it

has been since symbolic execution covered a new instruction. If in the last *time threshold*, there has been at least one newly covered instruction, then the framework signals that symbolic execution is not yet saturated. Otherwise, it signals that symbolic execution is saturated and Algorithm 10 must switch to fuzzing or end the analysis.

Implementation

Since, the algorithm for greybox fuzzing described in Algorithm 10 reuses the algorithms from symbolic execution and fuzzing, our implementation is also able to reuse the components developed for analysing isolated components with symbolic execution and fuzzing, i.e. Clang and Opt for bitcode level manipulation, KLEE for symbolic and concolic execution and AFL for fuzzing. However, we modified KLEE [100] to include the ability to consume seed inputs directly from the AFL fuzzer [2].

6.5 Output of the Analysis

The output of the analysis of isolated components with symbolic execution is the following

1. **Test-cases for functions:** The first output of the analysis are the test cases, I, generated by symbolic execution, ($I = I^S$), fuzzing ($I = I^F$), or greybox fuzzing, ($I = I^S \bigcup I^F$). For our study, we assume that we only generate *one test case per path*. This means, of course, that the number of test cases is also the number of unique paths discovered by the dynamic analysis method.

   $$I = \{i_1, i_2, \ldots i_n\}$$

   where n is the number of generated test cases. Each of i_k contains values of arguments expected by the function, m. I.e.

   $$\begin{aligned} \textbf{typeof}(i_k) &= \mathcal{I}(m) \\ k &\in [1, n] \end{aligned}$$

2. **List of exploits:** The next output is the list of exploits for the vulnerabilities discovered by the analysis of isolated functions. As discussed earlier, we will be considering only buffer-overflows, array index-out-of-bounds and null-pointer dereference vulnerabilities for our analyses. The discovered vulnerabilities are reported as the set of test cases (exploits) that lead to a failure (program crash) resulting due to one of the above vulnerabilities[2]. Let this set of test cases be I_{fail}. Then,

   $$I_{fail} \subseteq I^S.$$

3. **Stack-traces of crashes:** The final output of analysing isolated functions is a set of stack-traces for failures resulting due to the vulnerabilities discovered by the analysis.

[2]Since test cases are proxies for paths in the isolated functions, executing them with a test case is the same as executing the corresponding path.

A *stack-trace*, S_{m_0}, of a vulnerable function m_0, when running with arguments $i_f \in I_{fail}$, is an ordered set of instructions

$$S_{m_0}(i_f) = \langle L(m_0), L(m_1), \ldots L(m_n) \rangle \tag{6.5}$$

where m_i calls m_{i+1}. Here, L returns the vulnerability (line number in source-code) and the name of the function in which the vulnerable instruction lies.

6.6 Concluding Notes

In this chapter, we discussed our methodology to analyse isolated components (generated automatically with the methodology discussed in Chapter 5) with the goal of covering as many paths and, as a result, discovering as many potential vulnerabilities in them as possible. We first laid the groundwork of analysing isolated components by formalising the concepts of *in-component* paths and failures in terms of inputs to the isolated component.

Then, we described three different ways of analysing isolated components. With symbolic execution, we adapted the above formalisation for inputs to components and described how it could be instantiated for C-language programs by adapting the procedure for creating test-drivers, symbolising function parameters and symbolically executing the isolated function. The result of this analysis is a list of test cases, each executing a unique in-component path, a list of potential vulnerabilities in the isolated function and a stack-trace for every discovered vulnerability. Similar to symbolic execution, fuzzing can also be used to analyse the isolated components by adapting the design of automatically generated test-drivers, extracting a C-function's arguments from a byte-stream generated by a fuzzer and fuzzing the isolated function. The result of the analysis is the same as that for symbolic execution

Finally, we described a greybox fuzzing technique for analysing isolated components by using a technique-switching based on actively monitoring saturation indicators from symbolic execution and fuzzing. Practically speaking, we analyse an isolated C-function with symbolic execution until the analysis saturates at which point we switch to fuzzing till it saturates too. All through the process of greybox fuzzing, a common list of generated test cases (and paths) is maintained and cross-fed between our symbolic execution engine and fuzzer to facilitate reuse of past iterations of analysis. With a greybox fuzzing approach such as our's, we can effectively tackle the problems associated with state-of-the-art symbolic execution and fuzzing techniques. We will instantiate our approaches and evaluate their performance w.r.t. state-of-the-art in Chapter 8.

In the beginning, we called the discovered vulnerabilities in this step as *potential* vulnerabilities because, due to their isolated nature, one cannot guarantee that in any *real usage* of the program-under-test, with inter-component interactions as intended, the discovered vulnerabilities may be exploited. In the next chapter, we will describe a compositional analysis technique to determine whether the discovered vulnerabilities may be exploited in real-world usage.

7 Compositional Analysis

This chapter describes the final step in the discovery process – compositional analysis of the vulnerable components discovered in the previous steps, to determine if it is feasible to exploit them. Parts of this chapter have previously appeared in [102], [104] and [101], where the author of this thesis was the first author.

In the previous chapter, we described our methodology to analyse isolated components with three analysis techniques – symbolic execution, fuzzing and greybox fuzzing. The goal of this analysis step was to find as many potential vulnerabilities in components that make up a program by increasing the structural coverage as much as possible. By analysing isolated components, we may be able to find many vulnerabilities with the assumption that the components are isolated, i.e. be executable directly from a user-facing interface, such as CLI. We will now, in this chapter, describe a compositional analysis step, as depicted in the overview diagram Figure 7.1, which follows analysis of isolated components, to determine which components may be affected negatively if a vulnerability in an isolated component is not fixed.

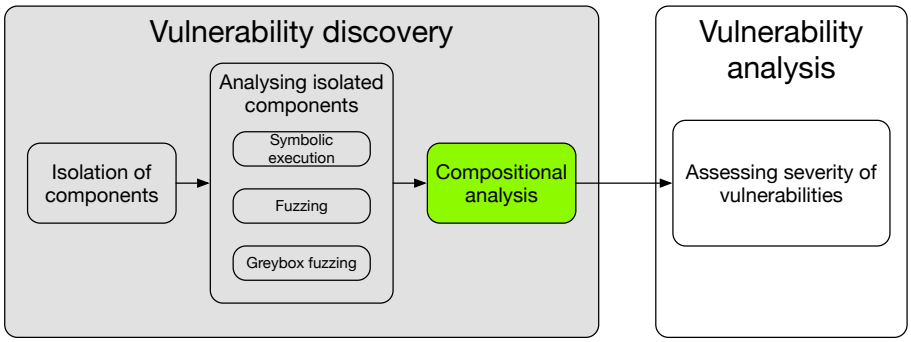


Figure 7.1: *Compositional analysis in the solution framework*

This chapter is organised as follows – we start in Section 7.1 by describing the need for compositional analysis of vulnerabilities discovered during analysis of isolated components. Then, we describe in Section 7.2 our compositional analysis and list some common terminology. In Section 7.3, we describe in detail the first step of statically combining error reports generated by the analysis step. For the remaining reports, we describe in Section 7.4 a targeted search-based strategy to determine their feasibility. For both phases of feasibility determination, we list the output of the compositional analysis of vulnerabilities. Finally, we conclude the chapter in Section 7.6.

7.1 Un-isolating Components – Motivation

In Chapter 6, the common thread between the three methods of analysing isolated components was the output of the analysis. To recap for the readers, the output of the analysing

```
1  int bar1(int c) {
2      if (c<3)
3          return (3/c); /*Maybe divide-by-zero*/
4      else
5          return 0;
6  }
7  int bar2(int d) {
8      if (d<50)
9          return 0;
10     else
11         return d;
12 }
13 int foo(int b, int c, int d) {
14     if (b==100)
15         return bar1(c);
16     else
17         return bar2(d);
18 }
19 int main(int argc, char** argv) {
20     int a, b, c, d;
21     a=atoi(argv[1]); b=atoi(argv[2]);
22     c=atoi(argv[3]); d=atoi(argv[4]);
23
24     if (a<1)
25         return 0;
26     else
27         return foo(b, c, d);
28 }
```

Listing 7.1: *Example C program (Copy of Listing 5.1).*

an isolated component is as follows –

1. List of test cases,
2. List of exploits for discovered vulnerabilities, and
3. Stack-traces of crashing executions.

In this list, we have generalised the output list from analyses performed by the three techniques on C-language programs, which is what we have instantiated our framework with.

The goal of this thesis, as stated earlier, is to increase the effectiveness of vulnerability discovery in programs, w.r.t. state-of-the-art dynamic analysis software testing techniques. When we isolate the components of a program and analyse these components in isolation, as shown in Chapter 6, we are more likely to find vulnerabilities in these isolated components than state-of-the-art techniques that may get stuck in "shallow" parts of the program and never even execute the components where these vulnerabilities lie. However, for dynamically analysing isolated components we had first removed, by the process of making components executable (Section 5.3), all the context under which this component may be used by the end-users. We illustrate this scenario by using our running example of the C code listed originally in Chapter 5 and repeated here in Listing 7.1. The call-graph of this program (which functions call which other, and in what order) is shown in Figure 7.2 and the corresponding control-flow graph of the entire program is shown in Figure 7.3.

In Figure 7.2, we can see that the function `bar1` can only be reached in the original program by, first, calling `foo` by `main` and then calling `bar1` by `foo`. Concretely, an input

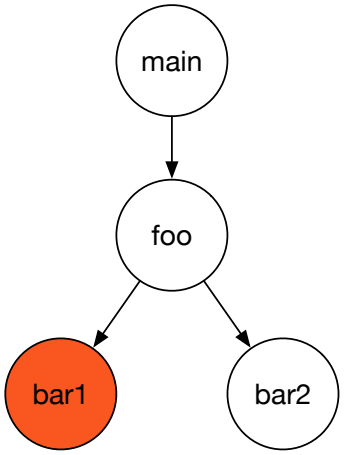


Figure 7.2: *Call graph of program listed in Listing 7.1*

has to satisfy the following sequence of branching conditions to reach node 6 in Figure 7.3 – $a \geq 1$, $b == 100$ and $c < 3$. Once all these conditions are $True$, only then is there a possibility that the vulnerability on node 6 (division-by-zero) may be found.

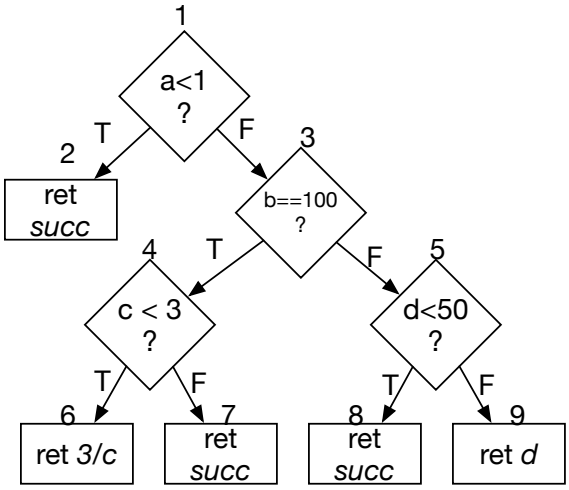


Figure 7.3: *Control-flow graph for the program listed in Listing 7.1*

On the other hand, if the components, i.e. functions in case of C-language programs, are isolated and analysed just like we described in the previous chapters, then the vulnerability on node 6 can be reached by generating an input that satisfies just *one* branching condition – $c < 3$. Symbolic execution and fuzzing are more likely, therefore, to find the potential vulnerability on line 3 in Listing 7.1. The reason for this increased likelihood is, as mentioned before that these dynamic analysis techniques will not run into one of the saturation conditions described earlier, such as path-explosion or mutation-level saturation. This demonstrates that our strategy of isolating components by making it possible for us to dynamically analyse them may help us in finding more potential vulnerabilities than state-of-the-art techniques that analyse a program only from a program entry point.

However, the term *"potential"* itself is key in our framework. Suppose again that the vulnerability on node 6 was discovered by our framework, as described in Chapter 6. The results, then, of the analysis of the function `bar1` will be test cases I, list of exploits I_{fail} and a list of stack-traces for crashing executions of `bar1`. We are interested here in I_{fail}. Then,

$$I_{fail} = \{0\}, \tag{7.1}$$

which is a value assignment for the input to `bar1`, $\mathcal{I}(bar1) = \{c\}$. To determine whether this vulnerability can be exploited in the program through an intended program entry point, it is not sufficient to generate an exploit in terms of $\mathcal{I}(bar1)$ only. An exploit needs to also consider the branching conditions that need to be $True$ *before* the branch at node 4 (Figure 7.3) can even be reached.

Due to the above reason, we need a final element in our framework's pipeline that can effectively determine which parent components (Equation (5.1)) are affected by vulnerabilities discovered by analysis of an isolated component. We will, in the next sections, describe the design and implementation of such an element.

7.2 Two-step Feasibility Determination

In this chapter, we will describe a two-step process to determine feasibility of a discovered vulnerability in an isolated component, in other components of the program that interact with it. We define vulnerability feasibility as below.

Definition 7.2.1. *(Vulnerability Feasibility) If a vulnerability discovered in an isolated component m_1 can be exploited by* at least *one test case (function arguments) for component m_2 where $m_2 \in parent_L(m_1)$, then the vulnerability is said to be feasible in m_2.*

In the above definition, function $\mathbf{parent_L}$ is the parent-child relationship for programming language L, as defined in Chapter 5. By extending the above analysis recursively, feasibility of a vulnerability may be determined till a top-level component (e.g. program entry point). Therefore, we can extend the definition of feasibility to introduce the notion of *chain of vulnerable components*, defined as follows

Definition 7.2.2. *(Chain of Vulnerable Components) An ordered set of components, $< m_0, m_1 \ldots m_n >$ is a chain of vulnerable components if, for $k \in [0, n]$*

1. $PC_{m_k}^{fail} \neq \phi$,
2. $m_{k+1} \in parent_L(m_k)$, *and*
3. *the vulnerability reported in all m_k's is the same as the vulnerability reported in m_0.*

We note from the definition of vulnerability feasibility that, in order to determine whether a vulnerabillity is feasible in a parent component of a vulnerable component, our framework needs to prove that there exists a test case for m_2 that exploits the discovered vulnerability in m_2 ($m_1 \in m_2$). Such proof may be in the form of an in-component path for component m_2 that leads to the vulnerability in m_1. We may formalise this proof process as follows.

Let $pc_f \in PC_{fail}(m_1)$, where $\mathbf{PC_{fail}}$ returns the set of path conditions leading to failure in m_1 (Section 6.1) where $m_1 \in C_L(P)$. Then, the failure represented by pc_f is feasible in $m_2 \in parent_L(m_1)$ if

$$\exists pc \in PC(m_2) : pc \wedge pc_f \tag{7.2}$$

Intuitively, if a path discovered by dynamic analysis in m_2 *ends* in a vulnerability discovered in m_1, then that vulnerability in m_1 is feasible in m_2.

Automatically generating the above proof may be done in two *phases* by our framework, which are listed below 1. going through all existing test cases and corresponding path conditions in m_2 to check which ones satisfy Equation (7.2), or 2. when the set of all generated test cases do not cover all *possible* path conditions for m_2, trying to generate more test cases for m_2 such that Equation (7.2) is satisfied. In the following sections, we will describe both these steps, sequentially.

7.3 Phase One – Collating Analysis Results

The first step for showing that a vulnerability discovered in a component is feasible in any of its parent components is to iterate through the analysis results of the component and all its parents to determine if there are any *matching paths*. We concretely define a matching path as follows

Definition 7.3.1. *(Matching path) A failure path, with path condition pc, in a parent component, is said to be a matching path to a failure path in its child component if the last instruction executed with an input satisfying pc is the vulnerable instruction discovered in the child component.*

Collating results this way by matching paths is the most straightforward way to determine the feasibility of vulnerabilities in connected (through $\mathbf{parent_L}$ function) components of a program.

7.3.1 Stack-trace Matching

We will now describe the process of collating analysis results as applied to programs written in C-language. As described in Chapter 6, the output of the first step of analysing isolated component by any method includes the generated test cases, exploits, and stack-traces of crashing executions of the isolated components. For the first step of feasibility determination, we will make use of the stack-traces that are output from the previous step, to determine whether we have a matching path in a parent component. We call two stack-traces, S_{m_1} and S_{m_2}, of the crashing executions of functions m_1 and m_2, respectively, *matching stack-traces* if $S_{m_1} \subset_O S_{m_2}$. Here "$\subset_O$" denotes an *ordered subset* meaning that the elements in the smaller ordered set appear in the same order in the larger ordered set.

Let us illustrate stack-trace matching by reconsidering the C-program in Listing 7.1. For stack-trace matching, let us assume that a divide-by-zero vulnerability was discovered by our framework during the analysis of the isolated components `bar1`, `foo` and `main`. We can

```
1 Divide-by-zero error
2 ... in bar1 on line 3 (return (3/c)): c=0
```

Listing 7.2: *Stack-trace of a crashing execution of* `bar1` *(Listing 7.1)*

```
1 Divide-by-zero error
2 ... in bar1 on line 3 (return (3/c)): c=0
3 ... in foo on line 15 (return bar1(c)): c=0
```

Listing 7.3: *Stack-trace of a crashing execution of* `foo` *(Listing 7.1)*

see by manually inspecting this program that the divide-by-zero vulnerability discovered in all these three functions are, indeed, due to the *same vulnerable instruction* on line 3 in the `bar1` function. This information is also reflected in the stack-traces of the respective crashing executions of `bar1` (Listing 7.2), `foo` (Listing 7.3) and `main` (Listing 7.4).

The goal of stack-trace matching is to determine the *chain of functions* (adapted from the chain of components, as defined in Section 7.2) in which a given vulnerability is feasible. The detailed algorithm for achieving this goal in the first phase, by stack-trace matching, is listed in Algorithm 13.

This function takes as input an isolated function, m, and the stack-trace of its crashing execution, S_m, due to a vulnerability discovered in it by analysis with, e.g. symbolic execution or greybox fuzzing. This algorithm checks for all its callers (functions that potentially call m) whether they and their respective callers are also "affected" by the same vulnerability that is represented by S_m. We define being *affected* as having reported an identical vulnerability to S_m. The function `getAffectedAncestors`, which we will describe in more detail in Algorithm 14, returns a list of chain of functions through which the vulnerability represented by S_m is feasible. Each element (chain) in this list is, first, concatenated with m before being added to the list, `chains`. This represents that the vulnerability or vulnerable instruction, responsible for all the failures reported in the chain of functions is inside the function, m. This list of chains is the final output of the function `getChainsOfVulnerableComponents`.

Note that a vulnerability in a given function may be feasible in more than one chain of functions. An example of such a case is illustrated in Figure 7.4. Suppose that the vulnerabilities in a and b were both found to be feasible in c. This leads to the following scenarios – Both, none or only one of the vulnerabilities in c are feasible in, both, d and c.

Now, let us look at the algorithm for the function `getAffectedAncestors`. This algorithm is listed in Algorithm 14

For getting a list of all ancestor functions (callers, callers of callers, and so on) `getAffectedAncestors`, first and foremost, checks if there is a matching stack-trace for m and the parent function, p, i.e. $\neg(S_m \subset_O S_p)$. If the stack-traces do not match, then the

```
1 Divide-by-zero error
2 ... in bar1 on line 3 (return (3/c)): c=0
3 ... in foo on line 15 (return bar1(c)): c=0
4 ... in main on line 27 (return foo(b, c, d)): b=100, c=0, d=1
```

Listing 7.4: *Stack-trace of a crashing execution of* `main` *(Listing 7.1)*

Algorithm 13 Determining the chain of functions for which a vulnerability in function m is feasible

```
1: function getChainsOfVulnerableComponents(m, S_m)
2:     chains ← {}
3:     for p ∈ caller(m) do
4:         affected_ancestors ← getAffectedAncestors(p, S_m)
5:         for a ∈ affected_ancestors do
6:             chains.add(m + a)
7:         end for
8:     end for
9:     return chains
10: end function
```

Algorithm 14 Recursively generating the list of affected functions for vulnerability represented by S_m, as determined by stack-trace matching

```
1: function getAffectedAncestors(p, S_m)
2:     affected ← {}
3:     if ¬ stackTraceMatch(p, S_m) then
4:         return []
5:     end if
6:     callers ← caller(p)
7:     for c ∈ callers do
8:         a ← getAffectedAncestors(c, S_m)
9:         if a ≠ φ then
10:            affected.add(p + a)
11:        end if
12:    end for
13:    return affected
14: end function
```

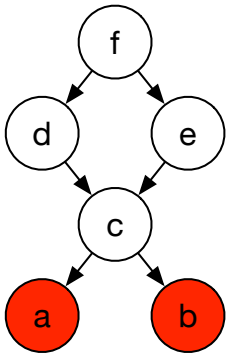


Figure 7.4: *Scenario illustrating a case where multiple chains for the same vulnerability exist*

algorithm returns an empty list of chains, signalling that the vulnerability is not feasible in this caller function. If the stack-traces do match, i.e. $S_m \subset_O S_p$, then the algorithm recursively checks for matching stack-traces in the caller functions of p. The recursion ends when either, as mentioned earlier, there is no stack-trace match or no more calling functions can be found for p. This case may occur when p is already one of the program entry points. The helper function `stackTraceMatch` follows a simple algorithm that checks for all stack-traces from the analysis of isolated function p, whether any of them match S_m.

Output: Hence, the output of the first phase of determining the feasibility of vulnerabilities is a list of chains of vulnerable components for vulnerabilities discovered by analysing isolated components using symbolic execution, fuzzing or greybox fuzzing. We, naturally, remove all redundant elements in this list of chains by removing those chains of vulnerable components that were also reported for the same vulnerability in another component.

7.4 Phase Two – Targeting Vulnerable Components

In the first phase of determining the feasibility of vulnerabilities, our framework can only report for those components where a vulnerability was already discovered by the initial analysis step for isolated components. If a vulnerability in m_1 is found to be *not feasible* in m_2, where $m_2 \in parent_L(m_1)$, by the first phase of feasibility determination, that does not necessarily mean that the vulnerability may not be exploited through m_2, but only that there were no matching paths from the analysis of isolated components m_1 and m_2.

We know from our extensive discussion on the limitations of symbolic execution (Chapter 2) and fuzzing (Chapter 3) that both these techniques are unable to achieve sufficient coverage in the programs (or components) under test. Therefore, it is possible that the in-component path leading to the vulnerability in m_1 was not covered by the dynamic analysis method employed for analysing the isolated component m_2. In such a case, phase one will, naturally, be unable to find a matching path and will, therefore, report that the vulnerability is not feasible in m_2.

Therefore, to extend our feasibility determination to include cases of incomplete analysis in parent components, we propose the second phase of feasibility determination. In this

phase, we, first of all, aim to *reduce the in-component paths* to be analysed in a parent component by summarising the vulnerable component. Secondly, we employ a targeted strategy to determine feasibility of the vulnerability in the parent component. We now describe both these steps in more detail.

7.4.1 Summarising Vulnerable Components

As discussed earlier, the main reason why a dynamic analysis technique may not achieve sufficient structural coverage is because of in-component path explosion or branching conditions that are hard to pass for a random technique. Therefore, we present in this section a summarisation strategy to decrease the number of paths for dynamic analysis techniques to cover, instead of the entire component. These component summaries are comprised of the *failing paths only* because our goal is to only find and report vulnerabilities in the program.

The process of summarisation is as follows – for every isolated component that has been analysed by symbolic execution, fuzzing or greybox fuzzing, we rewrite the component as an exclusive disjunction of the failure paths. Recall from Equation (6.3) that the set of failure paths in a component is a subset of the set of all paths discovered by dynamic analysis, i.e. $PC_{fail}(m) \subset PC(m)$. Originally, a component can be represented as an exclusive disjunction of the path conditions, i.e.

$$\bigoplus_{pc_k \in PC(m)} pc_k \tag{7.3}$$

where "$\oplus$" denotes mutual exclusion (XOR). However, if a dynamic analysis technique finds it difficult to achieve sufficiently high path coverage in $PC(m)$, then we reduce the number of paths to only the failures, i.e.

$$\bigoplus_{pc_k \in PC_{fail}(m)} pc_k \tag{7.4}$$

Note that removing all paths with $(PC - PC_{fail})$ from the component (or a representation thereof) means that we might be removing important side-effects of these paths. However, as described before, we already analysed all isolated components (including possible side-effects) and, in this step, only care about determining whether a vulnerability in a component is feasible in a parent component.

Generating Function Summaries from Test Cases

Let us now adapt the concepts described above to C-language programs, where the components are functions. After analysing isolated functions, our framework outputs a list of test cases, including those that exploit vulnerabilities in an isolated function, m_1. In this step, our framework replaces the isolated function, m_1 with a summary of the failure path conditions, using test cases. Recall that $\mathcal{I}(m_1)$ returns the inputs (formal parameters) for an isolated function m_1. Also, recall that if I is the set of all inputs (test

cases) generated by a dynamic analysis technique for m_1, then $I_{fail} \subset I$ is the set of all inputs that correspond to the failures triggered in m_1. For the function m_1, let $i_k \in I_{fail}$ denote the k^{th} input (test case) generated by a dynamic analysis technique that exploits a vulnerability. Then, the summarised version of the isolated component m_1 is as shown in Algorithm 15.

Algorithm 15 Summarised version of the isolated function m_1

```
1: function m1Summarised(i_actual)
2:     ASSERT(i_actual ≠ i_1)
3:     ASSERT(i_actual ≠ i_2)
4:     ⋮
5:     ASSERT(i_actual ≠ i_m)
6:     return m1(i_actual)
7: end function
```

In Algorithm 15, $i_k : k \in [1, m]$ are, as described above, the test cases generated by dynamic analysis that exploit a vulnerability. The equality $(i_{actual} == i_k)$ is defined as follows

$$i_{actual} == i_k \iff (\bigwedge_{f^i \in \mathcal{I}(m)} f^i_{actual} == f^i_k). \tag{7.5}$$

The negation of the equality in Equation (7.5), then, when at least one of the arguments is not equal in its value.

Intuitively, the summarised function compares each of the actual parameter value to the concrete argument values that were found to exploit a vulnerability in m_1. We assert a *negation* of the equality because, then, our framework reports an assertion error (and stops further processing) when there is a match found for i_{actual}. If no match is found in $m_1Summarised$, then we proceed by calling the original function m_1, as intended, to ensure that we include its side effects.

Suitability of Test Cases for Summarization

Please note that, unlike some past works such as [85, 115], where formal parameters are compared to the input preconditions (in the form of boolean constraints), our framework only matches the concrete values of the arguments, because we are limited by our output from the previous stage, which only includes a list of exploits for the vulnerabilities, and not all path conditions. However, we argue that matching only test cases is sufficient for feasibility determination because

1. If our symbolic execution engine can at least generate test cases that violate the assertions in a summarised function, it means that there were no constraints in the calling functions that stopped the vulnerability from being exploited – popularly known as the *sanitisation* of the inputs.
2. By matching concrete test cases instead of path conditions (as input preconditions), we may be saving constraint solving resources.

7.4.2 Determining Feasibility Through Targeted Symbolic Execution

After replacing the vulnerable components with their summaries, as described in Section 7.4.1, our framework is now ready to *re-analyse* the parent components but, hopefully, without running into the problem of path-explosion. This hope is based on the fact that in the summarised version of vulnerable components, the number of paths has been reduced to only the ones that may possibly trigger the same failures. Next, in this section, we employ a targeted strategy to determine, based on the component summaries, if the discovered vulnerabilities are feasible in the parent components. The goal of this strategy is the same as that of phase one of feasibility determination, but the techniques described here will only be applied for vulnerabilities for which feasibility could not be determined by collating results from analysis of isolated components. The basic idea is as follows.

Let $m_2 \in parents_L(m_1)$ and $m_1Summarised$ be the summarised version of the component m_1, as described in Section 7.4.1. Then, our framework analyses the isolated component m_2 with targeted symbolic execution, with the target set as $m_1Summarised$. Targeted symbolic execution, instead of "normal" symbolic execution or fuzzing, ensures that only those in-component paths that lead to $m_1Summarised$ are ever added to the queue of paths to be explored by the symbolic execution engine. In addition to component summarisation, targeted symbolic execution further helps in reducing path-explosion by reducing the overall number of candidate paths to be explored in the isolated components. In the following subsections, we will describe some technical and adaptation details of our framework related to determining feasibility through targeted symbolic execution.

Targeted Symbolic Execution for Functions

We start the technical description [104] by describing the design of a targeted search-strategy for symbolic execution for arbitrary functions in C-language programs. Whenever a new instruction is executed by symbolic execution, it needs to make a decision about which instruction to execute next. In the case of non-branching statements (such as assignment or function call), there is only one choice for which instruction is to be executed next. However, if the current instruction is a branching-condition, with two possibilities for the next instruction, the symbolic execution engine calls the function *selectState* to select which instruction to execute. The algorithm for *selectState* for a targeted search strategy in symbolic execution, as implemented in KLEE22 [100], is shown in Algorithm 16.

The *selectState* algorithm takes as input the current `position` of the instruction pointer in the program and the `target` set by our framework (the summarised function, $m_1Summarised$). The list of all possible instructions that can be executed next by symbolic execution are returned by `getNextExecutable` function. For all returned candidate instructions that can be executed next, the algorithm first checks if 1. the instruction has been explored already, and 2. the instruction is *valid*, based on whether the associated branching condition is satisfiable by the current path condition on the symbolic execution. Then, the `getMinFutureDistance` function returns the minimum number of instructions that *will* need to be executed to reach `target` if the candidate instruction was executed next. We will describe `getMinimumFutureDistance` later. The candidate

Algorithm 16 Select the next instruction to be executed by targeted path-search strategy in symbolic execution

```
1: function selectState(position, target)
2:     candidate_states ← getNextExecutable(position)
3:     lowest_distance ← ∞, selected ← ϕ
4:     for c ∈ candidate_states do
5:         if c ∉ explored ∧ isValid(c) then
6:             min_future_distance ← getMinFutureDistance(c, target)
7:             if min_future_distance < lowest_distance then
8:                 lowest_distance ← min_future_distance
9:                 selected ← c
10:            end if
11:        end if
12:    end for
13:    return selected
14: end function
```

instruction with the lowest value for `min_future_distance` is, then, picked and executed next to take the shortest feasible path to the target from any instruction.

We will now describe the algorithm for calculating the minimum number of steps to be taken from any given instruction to the target instruction. This algorithm is listed in Algorithm 17.

Algorithm 17 Calculating minimum possible distance to the target function

```
1: function getMinFutureDistance(s, target)
2:     if isTarget(s, target) then
3:         return 0
4:     else if unreachable(s, target) then
5:         return ∞
6:     else
7:         direct_dist ← shortestDistance(s, target)
8:         ancestor_dist ← getMinFutureDistance(stack_pop, target)
9:         indirect_dist ← shortestDistance(s, return_statement) +ancestor_dist
10:        return min(direct_dist, indirect_dist)
11:    end if
12: end function
```

For an instruction s, there may be three possible values for the minimum distance to a target -

1. If the target cannot be reached (according to the control-flow graph), then the distance is ∞,
2. if the target can be reached, then the value is the *minimum* of
   a) the number of direct steps (without tracking back in the program's call-stack) to the target entry point, and

b) the minimum distance (recursive calculated with `getMinFutureDistance`) of its direct ancestor in the program's call-stack plus the minimum steps till the next `return` statement, at which point the call-stack will be popped once.

With the above algorithm, targeted search for symbolic execution can effectively remove those instructions from further exploration from which the target function cannot be reached.

7.5 Output of Compositional Analysis

In Section 7.3 and Section 7.4, we described our methodology for compositionally analysing the results of the analysis of isolated components described in Chapter 6. Recall that the phase two of compositional analysis, i.e. targeting vulnerable components, is only applied to the components where the feasibility of certain vulnerabilities in their child components could not be determined by phase one. However, the output of both phases of analyses is the same. For every vulnerability found in isolated components, the output of compositional analysis is as follows

1. The location (file and line number) of the vulnerable instruction,
2. name of the component (or function, in case of C-language programs) where the vulnerable instruction lies, and
3. chain of vulnerable components.

The first two items in the list of output are self-explanatory. The last item is the chain of components, which we have defined in Section 7.2, is, in the case of the C-language programs, the sequence of function-calls in which a vulnerability was found to feasible.

7.6 Concluding Notes

In this chapter, we described the final step of the vulnerability discovery methodology described in this thesis. This step takes as input the output of the analysis of isolated components described in Chapter 6. Using the results of the analysis of isolated components, we undertake a two-phase process (Section 7.2) to determine the feasibility of the discovered vulnerabilities in components that interact with the vulnerable components in a relationship described by the $parent_L$ function.

In the first phase (Section 7.3), compositional analysis compares the paths covered by analysis of isolated components and generates a report whenever a path covered in $parents_L(m)$ *ends* in a path in $PC_{fail}(m)$, meaning simply that the same failure was also triggered while analysing the parent component. However, due to insufficient path coverage in parent components, it can be the case that such a path might not have been found while analysing the said parent component in isolation.

To handle this case, we proposed the second phase (Section 7.4) of compositional analysis where the aim is to deal with the problem of insufficient path coverage by removing the paths in the parent and child components that *do not* trigger the failure resulting due to the vulnerability whose feasibility needs to be determined. This is achieved by,

first, summarising the vulnerable component in terms of the triggered failures and, then, employing symbolic execution with a targeted search strategy to reach the summarised component from its parent components. Please note that in our compositional analysis technique, we detect recursion by detecting loops in a program's call-graph and performing the feasibility determination step (Section 7.4.1) only once in a loop.

The result of the above two-phase process is a list of *chains of vulnerable components*, each listing a vulnerability and the components affected by it in the order of their hierarchy defined by the $parent_L$ function.

In this thesis, we report *all discovered vulnerabilities*, regardless of whether they were found to be feasible in a program entry point or not. In Part III, we will discuss some strategies to assess and prioritise vulnerabilities when they might be *false-positives*. However, before that, in the next chapter, we will evaluate the vulnerability discovery framework in terms of its effectiveness and efficiency.

8 Evaluating Vulnerability Discovery

This chapter evaluates the vulnerability discovery part of our thesis by comparing their effectiveness and efficiency to those of various comparable dynamic analysis techniques. Parts of this chapter have previously appeared in [101], where the author of this thesis was the first author.

In this part of the thesis, we have discussed a compositional analysis framework that discovers vulnerabilities in programs using symbolic execution, fuzzing and a novel greybox fuzzing technique. In this chapter, we will now try to evaluate the efficiency and effectiveness of our framework by stating and answering relevant research questions related to the promised effects of it.

This chapter is organised as follows – We start in Section 8.1 by giving an overview of *Macke*, our compositional greybox fuzzing framework putting in practice the techniques that we have described so far. Then, in Section 8.2, we list the baseline for comparing Macke with, i.e. state-of-the-art symbolic execution and fuzzing tools with similar goals and claims as ours. In Section 8.3, we list and elaborate on the research questions that our set of experiments will aim to answer. We describe our experimental setup in Section 8.4. In Section 8.5 and Section 8.6, we discuss the observations of our experiments w.r.t. coverage and vulnerabilities discovered in the benchmarked programs. In Section 8.7, we will discuss observations w.r.t. vulnerabilities discovered in open-source libraries. To analyse the observations and get a consistent big picture, we synthesise our results w.r.t. the research questions in Section 8.8. We finally conclude the chapter in Section 8.9.

8.1 Operationalisation of Framework in *Macke*

We have implemented the three-step technique of vulnerability discovery discussed in Chapter 5, Chapter 6 and Chapter 7 in *Macke*[1] (**M**odular **a**nd **C**ompositional analysis using **K**lee (and AFL) **E**ngine), our open-source vulnerability discovery tool. Macke requires, in practice, that the C-program to be analysed be compiled to LLVM intermediate representation using *Clang* compiler [42]. An overview of the steps involved in dynamically analysing C-programs, after compilation, using Macke, as illustrated in Figure 8.1, is as follows.

1. Generate the list of functions that can be analysed in isolation, i.e. accepting at least one parameter and no parameter of double-pointer type.
2. Generate test-drivers for each function to be analysed based on the mode of analysis chosen, viz. symbolic execution, fuzzing or greybox fuzzing. For details on adaptation of test-drivers for the mode of analysis, please refer to the discussion in Chapter 6.
3. Create a pool of threads (using Python's *multiprocessing* module [89]) from all the test-drivers generated in the previous step.

[1] Macke can be downloaded for free at `https://github.com/tum-i22/macke`

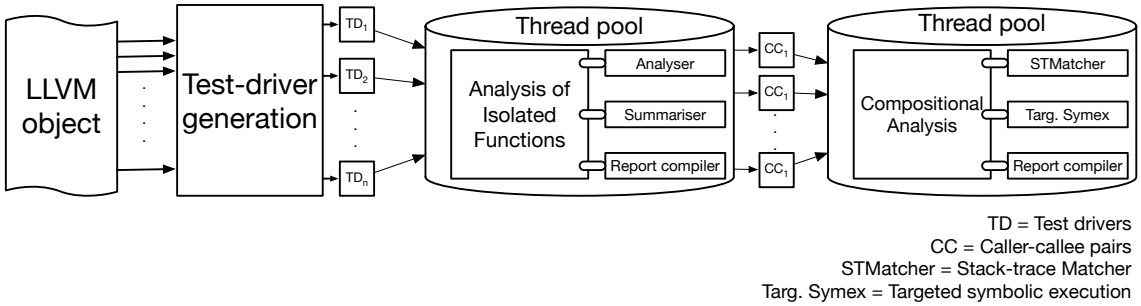


Figure 8.1: *Step-by-step functioning of Macke*

4. For each thread of analysis (test-drivers for isolated functions)
   a) Run the analysis for the required amount of time (or till all paths are explored).
   b) After the analysis is finished, compile analysis results. Analysis results include the lines covered, the lines not covered, vulnerabilities found and the stack-trace of the crashes resulting from the vulnerabilities.
5. Once all isolated functions are analysed, summarise the vulnerable functions, as described in Chapter 7.
6. Create another pool of threads for every *caller-callee* pair that needs to be compositionally analysed, as described in Chapter 7
7. For each thread of compositional analysis
   a) First, check if the same vulnerability was reported in the *caller* function, by stack-trace matching.
   b) If the vulnerability was not reported, run targeted symbolic execution (using Klee22 [100]) from the caller function to the summarised version of the callee.
   c) After compositional analysis is finished, compile its results. Analysis results include the new lines covered and chains-of-vulnerable components (if any).
   d) In every case that a vulnerability was found to be feasible from the caller function, add $parents(caller)$ to the pool of threads for compositional analysis.

With the above practical design, Macke can discover vulnerabilities in C-programs, by isolating functions, analysing them using symbolic execution, fuzzing or greybox fuzzing and compositionally analysing vulnerable functions to determine the feasibility of vulnerabilities. We will now, in the rest of this chapter, use Macke to demonstrate the benefits of a scalable greybox fuzzing approach over state-of-the-art dynamic analysis techniques.

8.2 Comparison Baseline

We start the description of our experiments by listing the tools that we will compare with our framework. We picked our baseline tools mainly based on whether a tool or technique

1. was available as an open-source tool,
2. had reasonably complex user-guide or documentation and could be used "out-of-the-box" for our analysed programs,

3. provided actionable items for discovered vulnerabilities, i.e. source-code instruction containing vulnerabilities, and
4. promised a higher coverage or higher rate of vulnerability discovery for *general-purpose UNIX-based* programs, compared to other techniques.

With the above inclusion criteria in sight, we concretely divide the state-of-the-art, and comparative, tools in the following two categories.

1. **Basic tools:** The first set of tools that we will compare with are basic symbolic execution and fuzzing tools. For symbolic execution, we will pick *KLEE* [26], and for fuzzing, we will pick *AFL* [2]. Both of these tools are considered state-of-the-art in the fundamental techniques used in this paper.
2. **Coverage-guided tools:** Next, we include more sophisticated tools involving symbolic execution and fuzzing, that improve upon these fundamental techniques by actively monitoring structural coverage of the program-under-test. We will pick *AFLFast* [21] and *Munch* [99] in this category. More details on these tools can be found in Chapter 4.

There are other frameworks and tools, such as Driller [133], AFLGo [20], FairFuzz [80] and Angora [35], that have proposed many improvements over state-of-the-art mutation-based fuzzing. However, these tools did not meet the inclusion criteria specified above and, therefore, are not included in our comparison.

8.3 Research Questions

To evaluate Macke, we will try to answer the following research questions

RQ1 How does the in-depth coverage of analysed programs compare amongst the three techniques of analysing isolated components?

RQ2 How does the in-depth coverage of analysed programs by our framework compare to those of the basic and coverage-guided tools?

RQ3 How does the vulnerability finding capability compare amongst the three techniques of analysing isolated components and compositional analysis?

RQ4 How does the vulnerability finding capability of our framework compare to those of the basic and coverage-guided tools?

RQ5 Can our framework be effectively used to test libraries without manual intervention, such as writing drivers?

8.4 Experimental Setup

For answering the research questions, we selected *eight open-source C programs* and *12 GNU Binutils*, listed in Table 8.1, for RQ1 to RQ4. In terms of LOC and functions, this set contains a wide range from basic utilities to much larger programs. For answering RQ3, we selected three case studies (Table 8.1) of open-source libraries.

Table 8.1: *Open-source programs analysed*

Prog.#	Program	LOC	Functions	Analysis time (minutes)	
				Macke	Other tools
1	bc 1.06	3.5k	129	22.5	283.2
2	bzip2 1.0.6	3.3k	108	36.4	383.3
3	diff 3.4	7.8k	391	103.0	849.9
4	flex 2.6.0	6.5k	260	58.0	716.6
5	grep 2.25	8.0k	461	130.5	933.3
6	less 481	7.9k	459	66.7	800.1
7	lz4 r131	4.7k	205	68.6	999.9
8	sed 4.2.2	3.2k	213	57.9	349.9
	Binutils 2.31.1				
9	addr2line	49.0k	1485	607.0	1440.2
10	ar	50.2k	1547	605.0	1584.0
11	as	65.8k	2088	603.2	1584.1
12	cxxfilt	48.9k	1484	274.3	1584.0
13	gprof	51.2k	1540	404.4	1439.9
14	ld	64.3k	1953	510.2	1440.3
15	nm	49.5k	1509	383.4	1440.3
16	objcopy	56.2k	1656	187.1	1584.1
17	objdump	64.5k	1876	234.2	1584.0
18	ranlib	50.3k	1547	394.0	1440.3
19	readelf	17.5k	249	42.9	514.8
20	size	49.1k	1491	380.5	1439.9
	Libraries				
21	Libtiff 4.0.9	82.7k	639	368.0	–
22	Libpng 1.6.35	43.8k	516	360.0	–
23	Libcurl 7.59.0	209.2k	692	680.0	–

We repeated all experiments *five times* on an Intel Xeon CPU E5-1650 v2 with *12 cores*, 3.50GHz per core, 126 GB of RAM, and running 64-bit Ubuntu 16.04.4 LTS.

In each repetition, we allowed Macke to fuzz each isolated function for *60 seconds* each (maximum) and we gave each run of targeted-search a total of *60 seconds* to reach the vulnerable function and trigger the crash. Generation of test-cases for functions could be carried out in parallel (Chapter 6) for isolated functions in Macke, i.e. all 12 cores could be utilised at the same time. Therefore, for a fair comparison, we allowed the other baseline tools, KLEE [26], AFL [2], AFLFast [21] and Munch [99], to run for approximately 12 times the total time taken by Macke or 24 hours, whichever was smaller. Times taken in each repetition for every benchmarked program are listed in Table 8.1.

8.5 Coverage

For evaluating coverage, we will first compare the three modes of analysis in Macke with each other, viz. symbolic execution, fuzzing and greybox fuzzing. Then, we will compare the best of these three methods of Macke to the basic and coverage-guided tools.

In Figure 8.2, we have depicted the percentage line-coverage in all analysed programs for the three modes of analysis of isolated functions by Macke, symbolic execution, fuzzing and greybox fuzzing. In Figure 8.3, we depict line coverage grouped by the respective

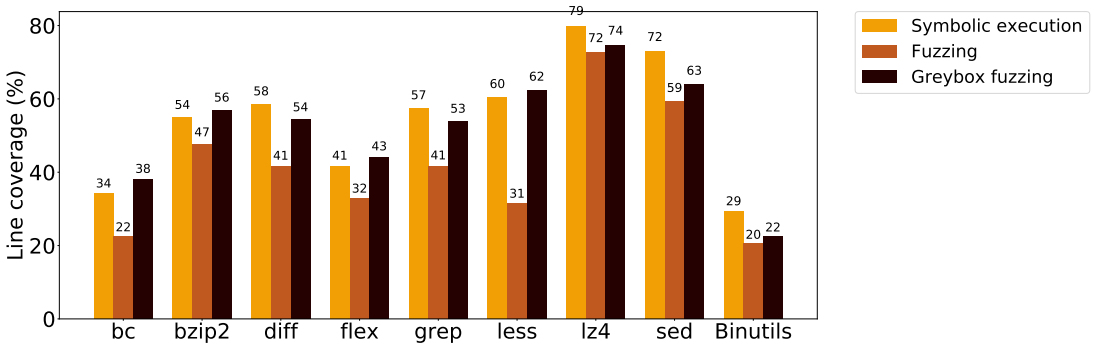


Figure 8.2: *Comparison of average line coverage with different modes of analysis for isolated functions*

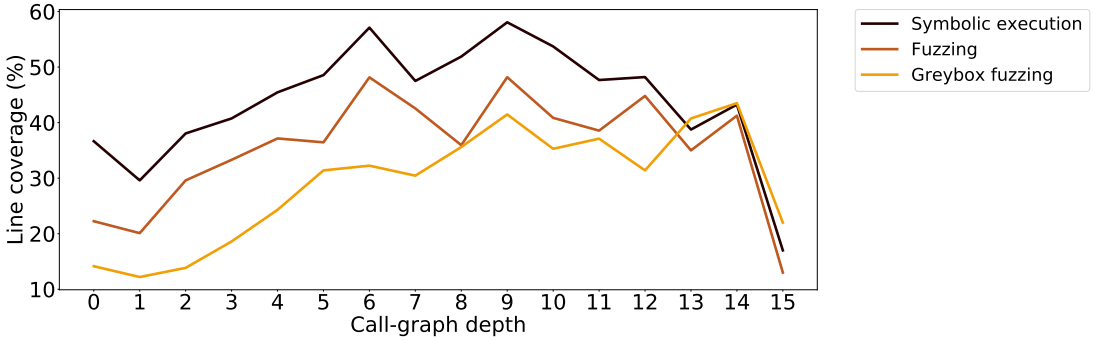


Figure 8.3: *Comparison of average line coverage grouped by call-graph depth, with different modes of analysis for isolated functions*

lines' depth in the call-graph of all analysed programs. Please note that at every call-graph depth, we only averaged over those programs that contained at least one function at that depth.

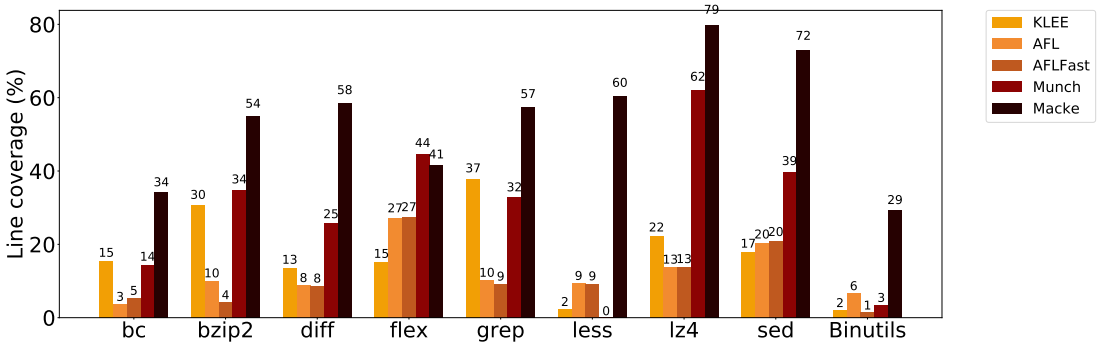


Figure 8.4: *Comparison of average line coverage*

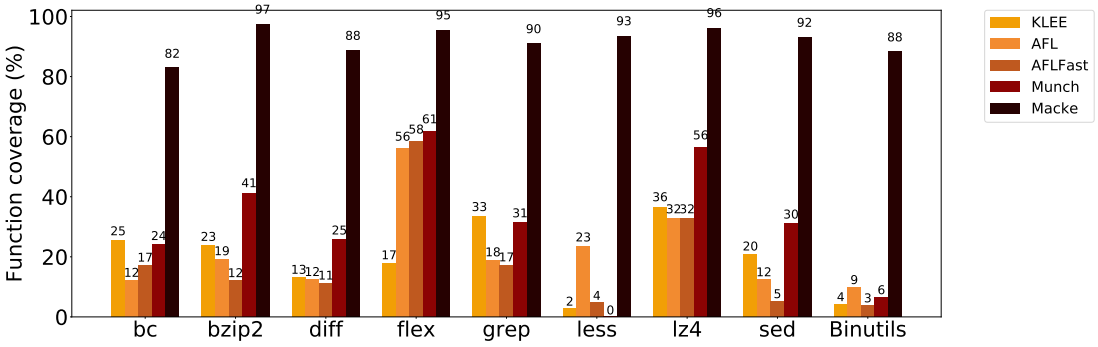


Figure 8.5: *Comparison of average function coverage*

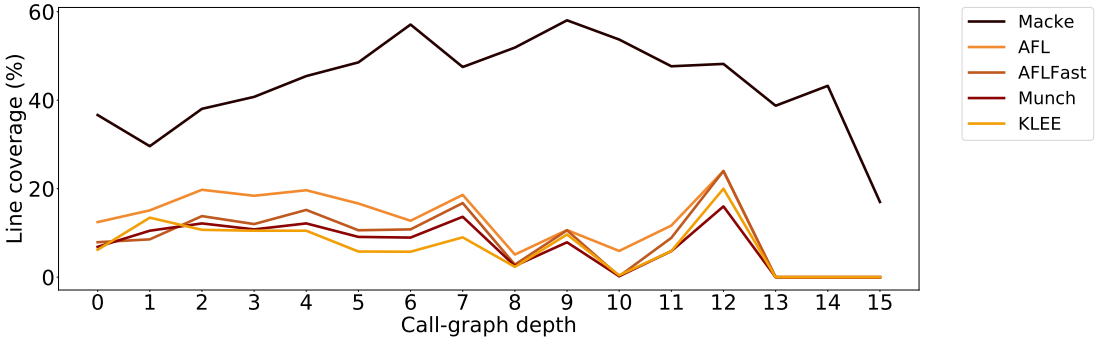


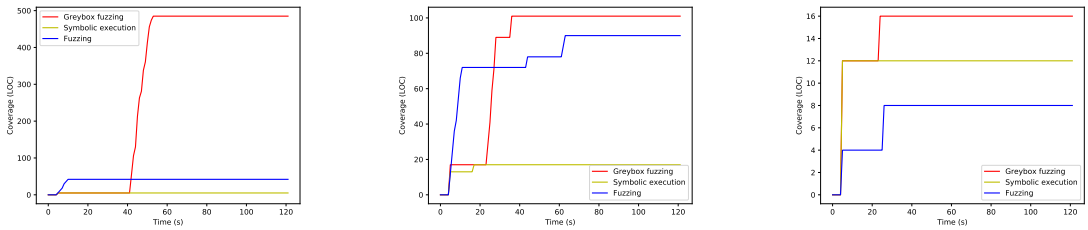
Figure 8.6: *Comparison of average line coverage grouped by call-graph depth*

How Does Greybox Fuzzing Improve Coverage? In Figure 8.7, we illustrate time-wise line coverage for three arbitrary functions in the *bc* program for which greybox fuzzing achieved higher (or equal) coverage than fuzzing or symbolic execution. For all the three functions, we see that greybox fuzzing was able to detect a "plateau" in coverage (which, expectedly, overlaps with the point where fuzzing or symbolic execution saturates overall) and progress by switching to either fuzzing or symbolic execution, resulting in higher line coverage.

We will now pick the analysis method from Macke that achieves the higher coverage over *most* of the analysed benchmarks, and use it to compare with the basic and coverage-guided tools. As seen above, this analysis method in our evaluation was symbolic execution.

Figure 8.4 shows the average (over 5 repetitions) line-coverage(%) achieved by Macke, KLEE, AFL, AFLFast and Munch in the given time-limits over all functions. Figure 8.5 shows the average (over 5 repetitions) function-coverage(%) achieved by the same techniques. Figure 8.6 shows the average (over all functions and 5 repetitions) line-coverage at every depth of the call-graph of all programs, e.g. the lines of code in the `main` function are counted at $x = 0$ in Figure 8.6, and so on.

Figures 8.4 to 8.6 show that the in-depth line coverage and function coverage for all programs is larger with Macke than plain symbolic execution and fuzzing, as implemented in



(a) `yyerror` **(b)** `make_arg_str` **(c)** `bc_new_num`

Figure 8.7: *Time-wise line coverage for some functions in* bc *by greybox fuzzing, symbolic execution and fuzzing.*

KLEE and AFL, respectively, as well as advanced tools that improve upon these techniques based on coverage, viz. AFLFast and Munch. Macke even achieves higher coverage at $depth = 0$ due to the following reason – more lines in the `main` function are covered when targeted symbolic execution is used for determining the feasibility of vulnerabilities, than when only fuzzing or symbolic execution is applied without a set target. The reason that function coverage was not 100% for Macke was, as explained in Section 6.3 that Macke does not analyse functions that contain any parameter of double- or more pointer type. In this study, we take no measures to handle this case and, instead, rely on targeted symbolic execution in the compositional analysis step (Chapter 7) to generate test-cases for those functions that could not be analysed by Macke.

8.6 Vulnerabilities

An increased coverage can be easily explained because we directly fuzz isolated functions. Let us now see how it affects vulnerability discovery. For evaluating vulnerabilities, we compare Macke to basic and coverage-guided tools. Table 8.2 lists the results related to vulnerability discovery for Macke's three modes of analysing isolated components in C programs. For the three methods, we have listed in this table the following three measures – The column "Vulnerabilities" lists the total number of discovered vulnerabilities, determined by a uniquely vulnerable instruction (Chapter 6). To determine whether any calling function could exploit the vulnerabilities discovered by Macke, we list the "$|chain| > 1$" criterion in the next column that counts only those vulnerabilities that can be, according to compositional analysis, exploited by *at least one* calling function. Some of these chains were reported by simple stack-trace matching, as described in phase one of feasibility determination, while others were reported from targeted symbolic execution towards summarised functions in phase two. The number of such chains whose ends (highest-level vulnerable functions) were reported by phase two, and not by simple stack-trace matching, are listed in the next column in Table 8.2 as "$chain \prec P2$"[2].

We can see from Table 8.2 that fuzzing was almost always (except *diff* and less) able to find the most amount of vulnerabilities. Moreover, the number of chains discovered by

[2] "$chain \prec P2$" should be read as "chains ending with a function found by phase-2 of feasibility analysis."

Table 8.2: *Vulnerability-related metrics for Macke*

Prog.	Vulnerabilities			$\|chain\| > 1$			$chain \prec P2$		
	Fuzz	Symex	GBFuzz	Fuzz	Symex	GBFuzz	Fuzz	Symex	GBFuzz
bc	72	57	60	42	30	40	7	16	16
bzip2	96	71	72	33	22	31	0	0	0
diff	219	256	256	205	166	227	32	7	9
flex	124	106	110	46	40	53	3	12	11
grep	319	261	246	186	132	176	24	7	6
less	167	166	173	151	124	168	15	14	13
lz4	102	92	94	119	100	137	43	3	5
sed	124	93	95	109	86	111	37	8	3
addr2line	979	804	407	404	226	280	10	9	4
ar	983	794	459	386	234	323	10	8	2
as	1230	1051	532	586	355	480	24	10	5
cxxfilt	895	811	411	294	234	284	4	6	3
gprof	982	837	490	398	238	336	9	6	11
ld	1218	1005	515	522	326	437	15	13	9
nm	948	847	358	362	247	245	10	7	5
objcopy	862	0	0	221	0	0	0	0	0
objdump	1008	0	0	258	0	0	0	0	0
ranlib	944	850	440	360	242	304	9	7	2
readelf	159	106	100	62	50	91	11	0	0
size	948	791	477	375	216	313	9	9	2

Macke with $|chain| > 1$ was also more for this mode for most of the programs. However, there were some programs where the number of such chains was found to be higher with greybox fuzzing. Lastly, for most programs, Macke's fuzzing mode discovered more chains of vulnerabilities with $chain \prec P2$. From Table 8.2, we can see that fuzzing performs better than Macke's other modes of analysis (symbolic execution and greybox fuzzing) for most programs. The similarity between symbolic execution and greybox fuzzing can be attributed to the fact that both these modes start with symbolic execution and greybox fuzzing only switches to fuzzing if saturation is detected *within the given timeout limit of 1 minute*. Additionally, greybox fuzzing mode also spends a few CPU cycles calculating incremental coverage, which may only pay off if the timeout was longer than 1 minute.

The overall distribution of the lengths of chains of vulnerable functions is shown in Figure 8.8, Figure 8.10 and Figure 8.12. This distribution shows that all three modes of analysis in Macke can generate chains of various lengths and, in fact, finds that about half of all discovered vulnerabilities can be exploited by at least one other function. In some cases, chains of 7 or more functions in the call-graph are also found.

On an average, for 68% (66% for symbolic execution, 76% for fuzzing and 62% for greybox fuzzing) of all chains ending at the `main` function, targeted symbolic execution was necessary to exploit the vulnerabilities. Additionally, we can see from Figure 8.9, Figure 8.11 and Figure 8.13 that the average lengths of chains of vulnerable functions with

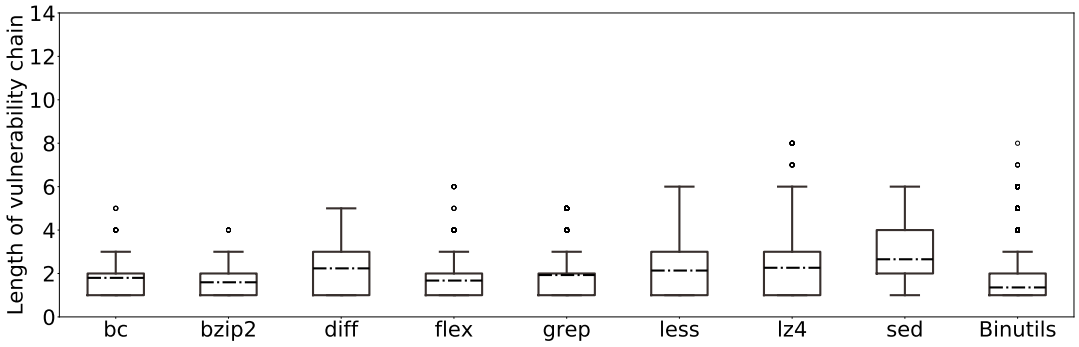


Figure 8.8: *All chains found by Macke (symbolic execution mode)*

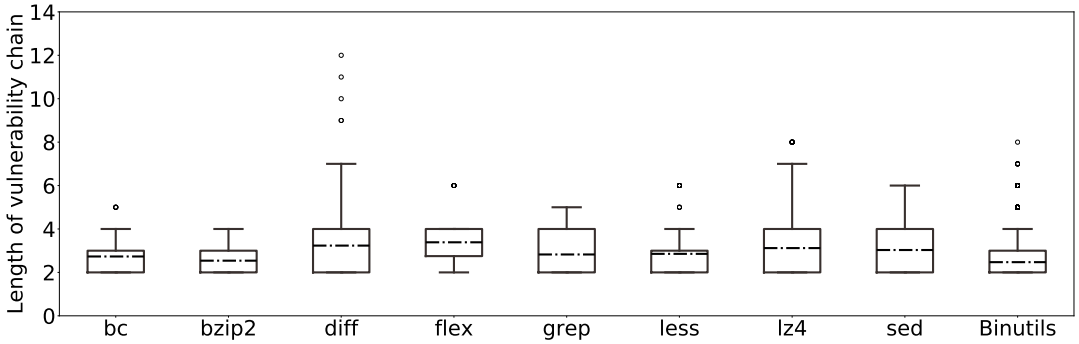


Figure 8.9: *chain* $\prec P2$ *(Symbolic execution mode)*

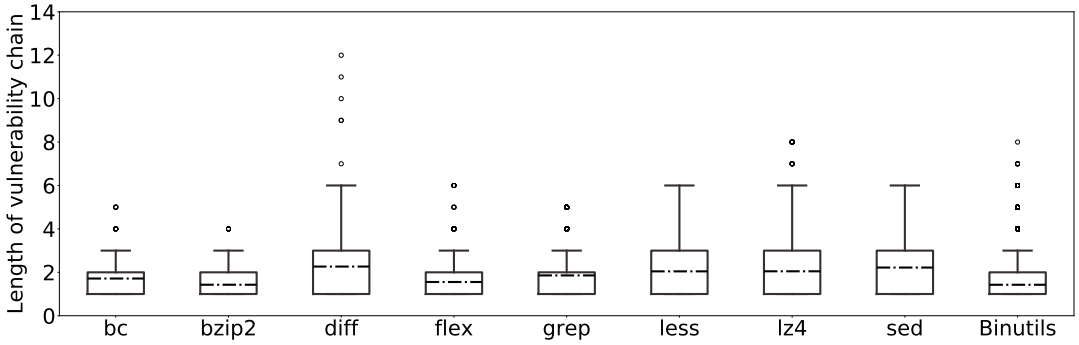


Figure 8.10: *All chains found by Macke (fuzzing mode)*

chain $\prec P2$ is *higher* than all chains combined. The above demonstrates the usefulness of combining targeted symbolic execution with isolated functions' analysis for discovering *high-impact vulnerabilities*.

However, some of the vulnerabilities discovered by Macke may never be exploited because their calling functions might sanitise the inputs before calling the vulnerable functions. Therefore, as a final comparison with state-of-the-art tools, we present in Table 8.3 the

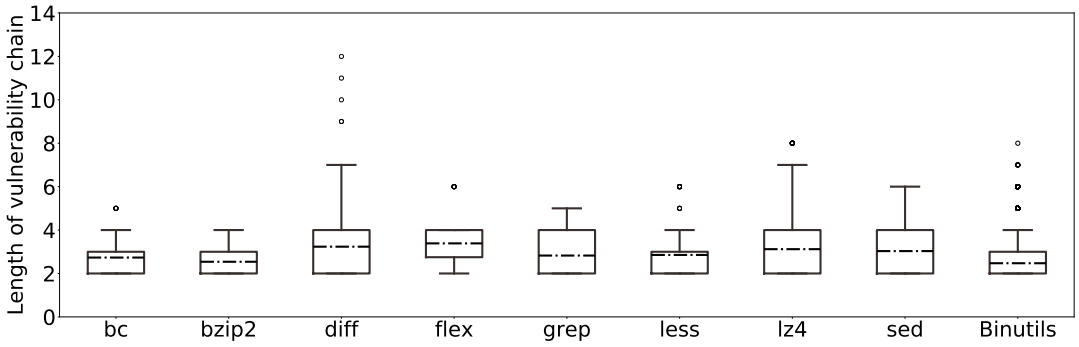


Figure 8.11: *chain ≺ P2 (Fuzzing mode)*

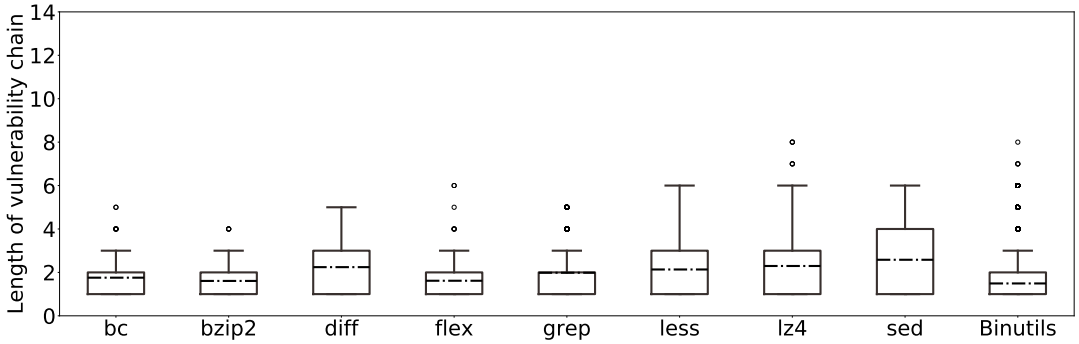


Figure 8.12: *All chains found by Macke (Greybox fuzzing mode)*

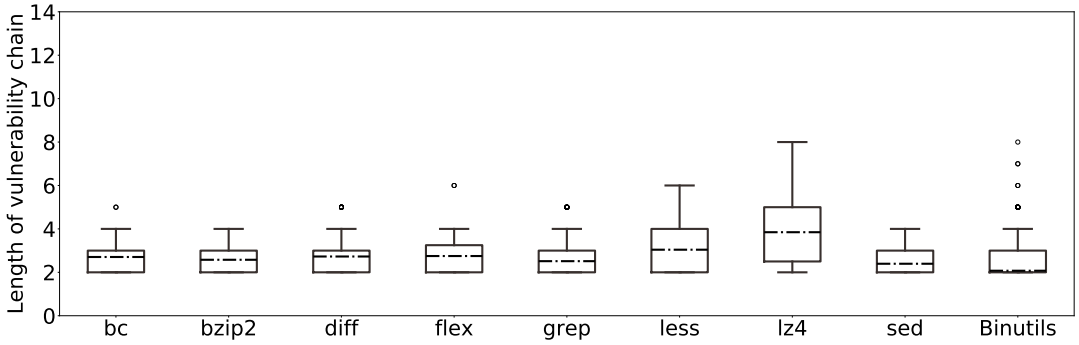


Figure 8.13: *chain ≺ P2 (Greybox fuzzing mode)*

count of vulnerabilities that could be exploited by the *main* function of a program. Since this factor can be measured for any baseline tool of our study, we have included basic tools (KLEE and AFL) and coverage-guided tools (AFLFast and Munch) for comparison.

We can see from Table 8.3 that for all but one (*Flex* will be explained later) programs, the number of such vulnerabilities found by Macke was higher than or equal to other baseline tools, in almost 90% less time (considering parallelism).

Table 8.3: *Vulnerability-related metrics for all tools*

Prog.	`main` vulnerabilities						
	Macke Symex	Macke Fuzz.	Macke GBFuzz	KLEE [26]	AFL [2]	AFLFast [21]	Munch [99]
bc	5	3	5	0	1	1	0
bzip2	0	0	0	0	0	0	0
diff	2	2	2	0	0	0	0
flex	0	0	0	0	1	1	1
grep	0	0	1	0	0	0	0
less	1	1	1	1	0	0	0
lz4	2	1	3	1	0	0	1
sed	1	1	0	0	0	0	0
addr2line	1	1	0	0	0	0	0
ar	0	0	0	0	0	0	0
as	6	7	4	0	0	0	0
cxxfilt	0	0	0	0	0	0	0
gprof	0	0	0	0	0	0	0
ld	3	3	5	3	1	1	2
nm	2	2	2	1	0	0	1
objcopy	0	0	0	0	0	0	0
objdump	0	0	0	0	0	0	0
ranlib	0	0	0	0	0	0	0
readelf	0	0	0	0	0	0	0
size	1	1	1	0	0	0	0
Total	24	22	24	6	3	3	5

Please also note that all the numbers in Table 8.2 and Table 8.3 are the *common* results from the five repetitions of Macke, KLEE, AFL, AFLFast and Munch, i.e. we only list those vulnerabilities and chains that were reported by *all five runs* of the respective method.

The Flex exception We saw in Table 8.3 that AFL, AFLFast and Munch discovered more `main` vulnerabilities in *Flex* than Macke. In the code of Flex, the majority of the functionality is contained within a function that is called directly by the `main` function, viz. `flex_main` function. Due to this fact, Macke should have given more time to this large function than other smaller functions, because the baseline tools get much more overall time to analyse this single function close to the entry point than Macke. We leave time-scaling based on the size of isolated-functions as future work.

Thus, we have shown, using 20 benchmarks, that a scalable greybox fuzzing approach, such as Macke, makes it more likely to discover vulnerabilities in a considerably shorter time than basic and coverage-guided tools, by deliberately executing functions in isolation, and performing a bottom-up feasibility analysis.

8.7 Real Vulnerabilities in the Wild

In Section 8.5 and Section 8.6, we have shown how Macke outperforms state-of-the-art techniques on programs that have a single user-interface, i.e. *main* function. We will now show that Macke can also find vulnerabilities in open-source libraries that have many possible entry points, i.e. APIs, that increase their attack surface. Unlike the basic and coverage-guided tools, Macke can analyse these libraries *automatically*, without the need for manually writing API drivers.

To demonstrate that Macke can effectively be used to test libraries without writing test-drivers, such as is the case with baseline tools, viz. AFL, KLEE, AFLFast and Munch, we picked *three popular open-source libraries*, listed below.

1. Libtiff 4.0.9 [3] – A library used by application developers to process images of TIFF, and a few other, formats.
2. Libpng 1.6.35 [4] – A library used by application developers to process PNG images.
3. Libcurl 7.59.0 [5] – A library for transferring data using various secure and non-secure transfer protocols.

Our goal with these case studies was to find out if we can reproduce the vulnerabilities reported in the past for them and if we can find any new ones.

For finding vulnerabilities, we filtered the list of all reported vulnerabilities (by Macke) to those where there was at least one API function in the chain of vulnerability. Table 8.4 lists all previously known vulnerable functions in the respective versions of Libtiff, Libpng and Libcurl.

We obtained this list from NVD[6] and then filtered them by the name of the library and the corresponding latest version. The second column of Table 8.4 lists the known CVE identifier for the respective vulnerabilities. The last three columns show, for the three modes of analysing isolated components, whether Macke could find the same vulnerability in the given time-out. As we can see from Table 8.4, all the known vulnerabilities in Libtiff and Libpng, and all but one vulnerabilities in Libcurl, could be found by Macke under the given time-limit.

We also found *23* new vulnerabilities in these three libraries that could be exploited through at-least one function in the respective libraries' API. Table 8.5 lists the previously unknown vulnerabilities (of type "buffer errors") in Libtiff, Libpng and Libcurl that can be exploited by an improper (but valid) use of their APIs.

8.8 Synthesis of the Results

8.8.1 RQ1 and RQ2– Coverage

Several works in the past [60, 102, 133] have shown that the primary reason that state-of-the-art test-case generation techniques are unable to find many vulnerabilities is a lack of

[3] http://www.simplesystems.org/libtiff/
[4] https://libpng.sourceforge.io/
[5] https://curl.haxx.se/
[6] https://nvd.nist.gov/

Table 8.4: *Known Vulnerabilities in Libtiff 4.0.9, Libpng 1.6.35 and Libcurl 7.59.0*

		Found by Mode		
Function	**CVE**	**Symex**	**Fuzz**	**GBFuzz**
TIFFSetupStrips	CVE-2017-17095	✓	✓	✓
PackBitsEncode	CVE-2017-17942	✓	✓	✓
TIFFPrintDirectory	CVE-2017-18013	✓	✓	✓
TIFFSetDirectory	CVE-2018-5784	✓	✓	✓
TIFFPrintDirectory	CVE-2018-7456	✓	✓	✓
LZWDecodeCompat	CVE-2018-8905	✓	✓	✓
TIFFWriteDirectorySec	CVE-2018-10963	✓	✓	✓
png_set_text_2	CVE-2016-10087	✓	✓	✓
png_set_PLTE	CVE-2015-8126	✓	✓	✓
png_get_PLTE	CVE-2015-8126	✓	✓	✓
png_do_expand_palette	CVE-2013-6954	✓	✓	✓
png_free_data	CVE-2018-14048	✓	✓	✓
Curl_http_readwrite _headers	CVE-2018-1000301	✓	✓	✓
Curl_smtp_escape_eob	CVE-2018-0500	✓	✓	✓
Curl_auth_create_ntlm _type3_message	CVE-2019-3822	✗	✗	✗
Curl_pp_readresp	CVE-2018-1000300	✓	✓	✓

coverage in deeper parts of the code, often guarded by sophisticated checks for malformed inputs. We, therefore, hypothesised that forcing higher coverage in programs will also lead to discovering previously unknown vulnerabilities. Compared to dynamic analysis techniques of symbolic execution and fuzzing applied to only the program entry points, we showed in Section 8.5 that Macke achieves higher line- and function-coverage. The reason for higher in-depth coverage was merely the under-constrained nature of the analysis, where isolated functions were analysed directly. The basic and coverage-guided tools could not cover as much of the source-code or functions because they had to overcome complex *frontier nodes* [106] to execute these functions.

RQ1 – Our results for overall line-coverage (including coverage at every call-graph depth) could not be generalised to pick one strong winner. However, we found that symbolic execution achieved higher coverage for most analysed programs.

RQ2 – Our results show that, for the selected benchmarks, Macke achieves higher in-depth line coverage and function coverage than basic and coverage-guided baseline tools.

8.8.2 RQ3 and RQ4– Vulnerabilities

Our hypothesis for discovering vulnerabilities was based on several previous works [40, 59, 102] that use symbolic execution at the level of isolated functions and found more vulnerabilities and increased coverage. In Section 8.6, we found that the number of

Table 8.5: *New Vulnerabilities Discovered in Libtiff 4.0.9, Libpng 1.6.35 and Libcurl 7.59.0*

Vulnerable Function	**Affected API**
TIFFFindField	TIFFGetFieldDefaulted
unixErrorHandler	TIFFFdOpen
TIFFRGBAImageOK	TIFFReadRGBAImage
TIFFSwabArrayOfShort	TIFFSwabArrayOfShort
TIFFSwabArrayOfLong	TIFFSwabArrayOfShort
TIFFWriteBufferSetup	TIFFWriteTile
png_set_filler	png_set_add_alpha
png_warning	png_set_compression_method
png_colorspace_set_chromaticities	png_set_cHRM
png_error	png_set_compression_buffer_size
png_set_keep_unknown_chunks	png_image_skip_unused_chunks
png_icc_check_header	png_set_iCCP
png_rtran_ok	png_set_alpha_mode
png_get_y_pixels_per_meter	png_get_y_pixels_per_inch
png_get_y_offset_microns	png_get_y_offset_inches
curl_easy_cleanup	curl_easy_cleanup
curl_easy_perform	curl_easy_perform
curl_getdate	curl_getdate
curl_mime_init	curl_mime_init
curl_slist_append	curl_slist_append
curl_slist_free_all	curl_slist_free_all
curl_easy_escape	curl_easy_escape
curl_easy_unescape	curl_easy_unescape

vulnerabilities (including potential false-positives, as we will discuss soon) reported by Macke is always higher than state-of-the-art symbolic execution and fuzzing tools and comparable to the state-of-the-art compositional tool. Additionally, the number of vulnerabilities that could be exploited from the `main` function is also the same or higher.

Importantly, a compositional analysis framework can also help mitigate the problems induced by potential *false-positives*, as follows. In particular, not all reported vulnerabilities that cannot be exploited from an interface, such as the `main` function, are false-positives. We provide three reasons for this claim here. 1. If a vulnerability can be shown to be exploitable through multiple caller-callee pairs ($|chain| > 1$), then it could *potentially* be *true-positive* and, hence, should be fixed. Manually confirming all reported vulnerabilities with $|chain| > 1$ was not feasible in our work, but sorting vulnerabilities by $|chain|$ should be the first step in bug-triage. Targeted symbolic execution allows Macke also to report more chains ($chain \prec P2$ in Table 8.2) than from merely analysing the isolated functions and examining their stack-traces. Without targeted symbolic execution, as discussed in Section 8.6, we can not find many critical vulnerabilities, some of which were exploitable through the top-level interface. 2. There may be many other factors [50], such as the degree of connectedness of a function [93] and the distance to an interface such as `main` function

[97], that affect if a vulnerability may be exploited, even if an exploit from `main` could not be generated. 3. In practice, functions tend to be reused in unforeseen contexts and, hence, it may be advisable to fix vulnerabilities directly inside functions that may be reused.

RQ3 – Our results related to vulnerabilities could not be generalised to pick a winner amongst symbolic execution, fuzzing and greybox fuzzing. However, by a small margin, fuzzing was able to find more or the same, vulnerabilities than symbolic execution and greybox fuzzing in isolated functions.

RQ4 – Our results show that, for all but one selected benchmarks, Macke finds more vulnerabilities than basic and coverage-guided baseline tools. It also finds more, or the same number of, true-positives as the baseline tools.

8.8.3 RQ5– Testing Libraries

Our final research question was whether Macke could help in effectively testing libraries to find vulnerabilities, without the time-consuming process of writing drivers. By applying our framework to three popular open-source libraries, Libtiff, Libpng and Libcurl, we showed in Section 8.7 that all, but one, of the known vulnerabilities, could have been found by Macke. We were also able to find new vulnerabilities and report them to their development teams. This is an essential contribution of our research because open-source libraries with public APIs are often used as daemon or microservices on remote servers accepting input through standard protocols, such as HTTP. If a malicious user was to send malformed input to the API to trigger the discovered vulnerabilities, they could cause a denial-of-service resulting in a substantial monetary and functional loss[7]. Macke, helps in mitigating potential vulnerabilities by reporting the chains of feasible functions, thereby indicating its potential exploitability. For the vulnerabilities that could not be confirmed to be feasible from the API, i.e. potential false-positives, we argue for them in the same manner as earlier, viz. with reports containing chains of vulnerable functions, it makes it easier for developers and testers to triage the reported bugs.

RQ5 – Our results show that, for the selected open-source libraries, Macke can effectively find vulnerabilities in them without writing specialised drivers for automatically testing them.

8.9 Concluding Notes

In this part of our thesis, we described a scalable, and completely automated, vulnerability discovery method consisting of a three-step dynamic analysis procedure. First, as described

[7] Please note that we did not include evaluation of any network related libraries, such as OpenSSL [138], because most network libraries rely on function callbacks, which cannot be resolved by our compositional analysis engine and, hence, the exploitability of discovered vulnerabilities cannot be confirmed from API functions.

in Chapter 5, our framework automatically isolates components of a program by automatically adding entry points to the program, so that they may be analysed by a dynamic analysis technique. Next, as described in Chapter 6, we propose automated procedures to adapt the testing drivers created in the first step and analyse isolated components of the program-under-test using symbolic execution, fuzzing or a novel greybox fuzzing method. Lastly, after obtaining the list of vulnerabilities in isolated components, as described in Chapter 7, we propose to determine the feasibility of the discovered vulnerabilities by compositional analysis from their parent components.

In this chapter, we proposed an evaluation scheme to find out how effective and efficient our proposed vulnerability discovery technique is, compared to state-of-the-art dynamic analysis methods. To investigate our research questions, we applied Macke on 20 open-source UNIX-based programs, typically run from the command-line, and three open-source libraries with APIs allowing third-party application developers to perform tasks such as image-processing or web-resource handling. We also applied well-known symbolic execution and fuzzing tools, and their coverage-guided variants that promise improvement in terms of coverage and vulnerabilities. We found in our results that Macke (all three modes of analysis) was effective in achieving higher line and function coverage, and finding more vulnerabilities than state-of-the-art tools. Macke was also effective in finding vulnerabilities in libraries without the manual effort of writing test drivers. In terms of time-efficiency, we claim that Macke was able to achieve the above results in less than 10% of the time taken by the state-of-the-art techniques, due to its capability of analysing in parallel isolated components and performing compositional analysis for discovered vulnerability. The results, therefore, demonstrate clearly that our scalable greybox fuzzing framework, Macke, is superior to the state-of-the-art tools in terms of coverage, vulnerabilities and ease of use for libraries.

At the end of the vulnerability discovery part, however, our framework may still have many discovered vulnerabilities for which it could not be determined whether they are ultimately feasible in a program entry point, such as the `main` function or an API. These discovered vulnerabilities could, optionally, be discarded as being *false-positive* reports, i.e. not feasible to be exploited in real-world usage of the program. However, this might underestimate the effect they might have in the real-world usage, especially when dynamic analysis of the isolated components has not achieved 100% path coverage. In the next part of this thesis, we will discuss the above, and other, implications of the vulnerability reports generated by Macke and how to prioritise them.

Part III

Vulnerability Analysis

9 Assessing Discovered Vulnerabilities for Effective Triage

This chapter introduces some ideas combining output generated by the vulnerability discovery steps with various impact factors and heuristics to assess the discovered vulnerabilities. Parts of this chapter have previously appeared in [97], where the author of this thesis was the first author.

In this thesis so far, we have discussed how our framework can find vulnerabilities in programs that contain many interacting components. In Chapter 7, we described how we could confirm whether a vulnerability in a component may be feasible in its parent components. If our framework can confirm feasibility up to a program entry point, then the corresponding input can be reported as a possible exploit. However, as mentioned briefly in Section 7.6 when it cannot be confirmed that a discovered vulnerability is feasible in the parent components, we require an assessment framework to prioritise them because they might not always be false-positives.

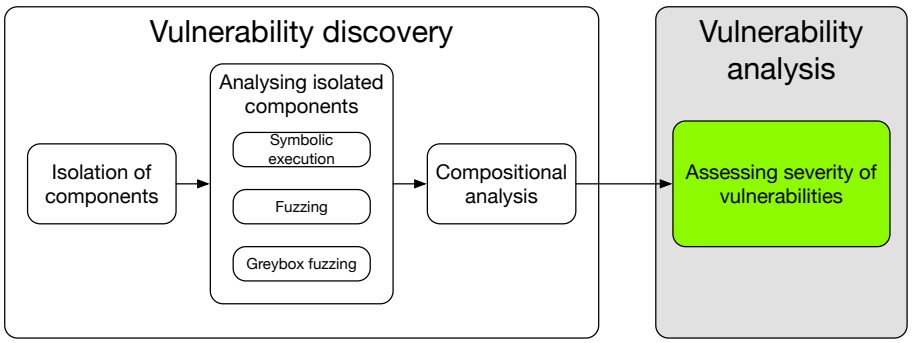


Figure 9.1: *Vulnerability assessment in the solution framework*

In this chapter, we will list some ideas about assessing the vulnerabilities discovered by our compositional (and scalable) analysis framework and prioritising them to assist developers and testers in the triage process. This is the last step in managing vulnerabilities in real-world programs, as shown in Figure 9.1. We will start this chapter by recalling the results from the steps of vulnerability discovery, in Section 9.1. In it, we will expand on the terminology of *false-positive* vulnerabilities and discuss in detail why assessing their impact, instead of dismissing them from vulnerability reports, leads to a more effective triage process. In Section 9.2, we discuss a common scale used in the research of vulnerability assessment. Then, in Section 9.3, we will discuss some factors which may affect the severity of discovered vulnerabilities in different usage context. Finally, in Section 9.4, we will conclude the chapter.

9.1 Consolidating Reports of Discovered Vulnerabilities

We, first of all, recall the output from vulnerability discovery methodology described in Part II. This output includes vulnerabilities in isolated components, chain of vulnerable components (where the discovered vulnerabilities are feasible) and inputs (test cases) that exploit vulnerabilities in vulnerable components. In addition to these, there are also other test cases generated by our framework that execute in-component paths *not* leading to vulnerable instructions. However, in this part of the thesis, we will only focus on the discovered vulnerabilities.

For all vulnerabilities discovered by our framework, the corresponding test cases can be seen as "*proofs-of-exploitation*" for vulnerable components. Additionally, chains of vulnerable components list the components that are affected by a vulnerability. However, let us consider the chain of vulnerable components depicted in Figure 9.2.

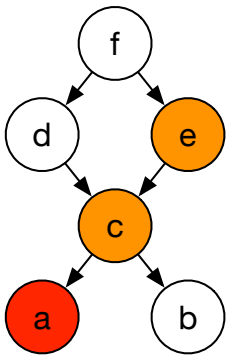


Figure 9.2: *Chains of vulnerable components reported by the vulnerability discovery framework*

We see from this figure that a vulnerability in a was found to be feasible in c and e ($c \in parent_L(a) \wedge e \in parent_L(c)$). However, let us suppose that, according to compositional analysis step described in Chapter 7, the vulnerability in a was found neither to be feasible in components d and, consequently, f, nor in f due to e. This can happen because there exists no in-component path in d that may lead to the vulnerability path in c. An example of such a scenario may be as follows – let us assume that we are dealing with a C-program where the definitions of the functions a, c and d are as listed in Listing 9.1. We can see from the definition of function `d` there is a check (so-called, "*sanitisation*") on the value of `num` which prevents the divide-by-zero vulnerability on line 2 to be exploited by an input to the isolated function, `d`. In such a scenario, it is correct that compositional analysis reports that the vulnerability in a is not feasible in d.

However, as we showed in the previous chapters, dynamic analysis techniques, such as symbolic execution, fuzzing or greybox fuzzing, may saturate in terms of generating test cases to execute paths in parent components, such as d, that lead to vulnerability discovered in isolated vulnerable components, such as a. An example of such a case is listed in Listing 9.2, where we have only changed the definition of function `d` from Listing 9.1. By looking at the definition of `d` in Listing 9.2, we can see that the vulnerability in `a` is feasible in `d` because there is no sanitisation of the variable `num` before calling the `c`. When

```
1  int a(int x, int num) {
2      return x/num; /* Divide-by-zero vulnerability */
3  }
4  int c(int num) {
5      return b(100, num)
6  }
7  int d(int num) {
8      if (num != 0)
9          return c(num);
10     else
11         return 0;
12 }
```

Listing 9.1: *Functions a, c and d. Input to c being sanitised in d*

```
1  int d(int num) {
2      FILE * file;
3      file = fopen("/tmp/dummy.txt", "r");
4      int num = atoi((char)fgetc(file));
5
6      for (int i=0; i<num; i++)
7          printf("%d", num);
8
9      return c(num);
10 }
```

Listing 9.2: *Function d where, input to c has not been sanitised in d*

analysing the isolated function `d` a dynamic analysis technique such as symbolic execution may not be able to generate test cases for the path to `a` from `d` that executes the vulnerable instruction on line 2 of Listing 9.1 because it might get stuck in path-explosion due to 1. system-call (`fgetc`) on line 4, or 2. input-dependant loop on lines 6-7. Regardless of the underlying reason for path-explosion, if compositional analysis is unable to report that the vulnerability in `a` is feasible in function `d`, it is *not necessarily* because there does not exist a path in `d` leading to the divide-by-zero vulnerability. Such a path *might* exist, as we saw in the example of Listing 9.2, but may not be reported by symbolic execution, fuzzing or greybox fuzzing, due to saturation.

9.1.1 What is a False-positive?

Description of the two scenarios above leads us to ask the following question w.r.t. reporting of vulnerabilities by our framework – "*For vulnerabilities whose feasibility could not be determined in a program entry point, should our framework report them as false-positives?*" To answer this question, let us first state the definition of a false-positive vulnerability or, simply, a false-positive.

Definition 9.1.1. *(False-positive Vulnerability) A vulnerability that was discovered by analysing an isolated component using a dynamic analysis technique, but that can never be exploited in any real-world usage of the program-under-test, is called a false-positive vulnerability.*

Unlike static analysis [10], state-of-the-art dynamic analysis techniques, e.g. symbolic execution and fuzzing, do not report vulnerabilities that cannot materialise in a real-world usage of a program. This is because, unlike the techniques described in this thesis, existing

techniques usually perform the analysis only at one program entry point. As a result, all vulnerabilities discovered by them can be exploited in a real-world usage also. However, due to the reasons discussed above, we argue that not all vulnerabilities whose feasibility may not be determined by compositional analysis of their parent components are false-positives because we recognise explicitly the saturation problems associated with our analysis methods. Therefore, in our framework, we *do not exclude any discovered vulnerabilities*, regardless of whether they were found to be feasible from parent components or not. We call these vulnerabilities as "*unconfirmed vulnerabilities*" instead of false-positives.

9.1.2 Vulnerability Prioritisation as the Antidote

At the same as claiming that not all unconfirmed vulnerabilities are false-positives, we also realise that some of the unconfirmed vulnerabilities may affect the program less severely than the others, were they to be exploited in a real-world usage. Assigning severities to discovered vulnerabilities may be more effective than discarding them altogether because, in many development contexts, it cannot be determined in advance how a developed component may be used or reused. In case that a vulnerable component is reused in an unexpected way, it might be possible to exploit the vulnerability more easily than previously envisaged.

We will, therefore, extend our discovery framework with an automated assessment step to prioritise the discovered vulnerabilities in the order in which they ought to be fixed by the developers. In the following sections, we will discuss some ideas on how reported, but unconfirmed, vulnerabilities may be assessed and ranked to make it easier for a developer or tester to prioritise and triage them appropriately.

9.2 Scale for Scoring Vulnerabilities

To triage the reported vulnerabilities by our discovery framework, we must rank them based on the context of their development, exploitability and potential damage to the underlying assets. Several scales for scoring vulnerabilities in programs have been presented in the past, of which *Bugzilla's* severity scale [58] and *Common Vulnerability Severity Scale (CVSS)* [121] are two examples. Bugzilla is an openly available platform to report and track bugs and vulnerabilities in popular open-source projects. The bugs are reported with nominal categories for prioritising the fixing process. However, from past works [9] and our own experience, we note that the context-free ranking system of Bugzilla results in a ranking that does not truly represent the severities of bugs. The first reason for this is that, for most of the reported bugs, the development community ignored the "priority" field of the reports, and used only the "severity" field as a proxy for both, priority and severity [97]. Secondly, we also found many instances in the analysed programs where the severity values in Bugzilla were changed by the developers when, either, the underlying assets were not considered important enough, or the bug was not fixed because it was too complex to exploit it.

In CVSS [87], Mell et al. proposed a standardised way to capture the principal characteristics of a vulnerability and assign a numerical score reflecting its severity. CVSS consists

of a base-score that is a combined scale, including the metrics listed in Table 9.1.

Table 9.1: *CVSS base-score values*

Base-score value	Description	Allowed values
Attack Vector (AV)	The context in which exploiting the vulnerability is possible.	*network*, *adjacent*, *local* and *physical*
Attack Complexity (AC)	The complexity of the attack process, if possible.	*low* and *high*
Privileges Required (PR)	The level of privileges an attacker must have to carry out an exploit.	*none*, *low* and *high*
User Interaction (UI)	The amount of direct user interaction required for the attacker to carry out an exploit.	*none* and *required*
Scope (S)	Whether or not other components (changed scope) than the vulnerable one can be affected if the vulnerability is exploited.	*unchanged* and *changed*
Confidentiality (C)	The amount of confidential data that will be exposed if the vulnerability is exploited.	*none*, *low* and *high*
Integrity (I)	The amount of information that the attacker can modify in the exploited component.	*none*, *low* and *high*
Availability (A)	The amount of information to which the attacker can deny access.	*none*, *low* and *high*

CVSS3 [121], a later improvement, also includes optional temporal and environmental scores, in addition to the base-score. Unlike Bugzilla, for calculating CVSS and CVSS3 scores, scoring for *all* base-score values is mandatory. In this way, CVSS base-scores explicitly take into account underlying assets and attack complexity, thereby eliminating the need for further arbitrary adjustment.

9.3 Factors Impacting Priority of Vulnerabilities

In the severity scales discussed in Section 9.2, we listed some standard metrics based on which we may be able to assign a score to a discovered vulnerability. Most of these metrics, or base-score values for CVSS, depend on various factors related to the discovered vulnerability, the vulnerable component, other interacting components, the context of usage and, sometimes, properties of the entire program itself. Overall, we divide the impact factors of the discovered vulnerabilities in three categories – structural, organisational and

asset factors. We will now briefly describe these categories below.

Structural Factors

Structural factors refer to the characteristics of the program and, particularly, the vulnerable component itself, that may affect the manifestation, exploitation and severe effect of vulnerability. In the previous chapters, we described in much detail the concept of compositionality (Section 5.2) and how any program-under-test can be seen as being composed of interacting components. Compositionality itself can also form the basis of structural factors that affect the exploitability of a vulnerability by a malicious actor. Concretely, below are some structural factors that may be related to the priority of vulnerabilities

1. *Function complexity*, based on McCabe's complexity metric [93],
2. Simple *function size*, based on instruction count,
3. *Number of components possibly interacting* with a vulnerable component according to static code analysis,
4. *The number of distinct vulnerabilities* (based on vulnerable lines-of-code) in a vulnerable component, and others.

Some past works have inspired this category of factors influencing the priority of vulnerabilities. For example, El Emam et al. [50] utilised the object-oriented design of programs and predicted faulty behaviour based on metrics related to object interactions. Similarly, Nagappan et al. [92] and Nagappan et al. [93] used metrics such as the complexity of functions and call-graph structures, derived from static analysis of the program-under-test to predict the order in which discovered bugs in them might be fixed in the future. We have drawn inspiration in our framework from all of these past works for structural factors for vulnerability assessment. Intuitively, structural factors not only affect how a vulnerability in one component may affect itself and other components interacting with it but also affect the triage process by masking in complexity the latent effect of vulnerability.

Organisational Factors

For using structural factors to determine exploitability of vulnerabilities, we need internal information about the program, such as the ones listed above. However, organisational factors may only be extracted from outside the program and, often, capture the context in which it is developed and may be used. It is important to note that organisational factors do not *only* take into account the characteristics of the teams and organisations (e.g. software-development companies) but how the development, use and reuse of a program are organised.

To clarify the above point, we list below some examples of organisational factors affecting priority of vulnerabilities in a program's components

1. *Component reuse* is an important factor to be accounted for during the development of the component itself. Reuse of a component, especially in case of APIs or libraries,

may result in a vulnerability being easier to exploit compared to the initial scenario where the inputs were assumed to be sanitised in other parent components.
2. *Release history* refers to the previous iterations of a vulnerable component (or other components interacting with it). It may be an important factor affecting priority because, firstly, new vulnerabilities may have been introduced while adding or removing new functionality and, secondly, many assumptions about handling input to, and from, other components may no longer be true for a changed component.
3. *Engineer metrics* are related to the team of developers responsible for a vulnerable component. It has been argued and shown in the past [94] that the number of engineers and their professional experience contributes to the reliability of software developed by them which, consequently, might affect the assessment of the vulnerabilities discovered in components.

Past work by Nagappan et al. [94] and Murphy-Hill et al. [90] have also found that factors related to the organisation of software development, combined with the metrics related to the developmental history of the program-under-test can have a tangible effect on the bugs in the program, whether they be newly-introduced ones or existing from older components.

Asset Factors

Finally, we consider impact factors related to the underlying assets that are at stake from the information system being considered for vulnerabilities. In particular, the *confidentiality*, *integrity* and *availability* base-score values are related solely to access to the data that can be breached if a vulnerability were to be exploited in the system. Below we list some potential asset factors that impact the severity of a discovered vulnerability

1. *Trustworthiness of storage*, based on, e.g., whether the data is stored in private storage (on-site) or third-party cloud storage. Another factor that affects the trustworthiness is whether the data is encrypted before storage and, hence, it cannot be used for unintended purposes by potentially malicious actors.
2. *Organisation-wide access control* is an important factor affecting who can access data and assets (authentication) of an organisation and what an authentic user is allowed to do with the access (authorisation).
3. *Organisation-wide integrity protection* is also an important factor for determining how severely the integrity of an organisational asset may be affected if a vulnerability were exploited.

We note for these factors that, most likely, their impact factor values will remain unchanged w.r.t. the particular vulnerability discovered in a program.

9.3.1 Drawing on Past Knowledge

In this section, so far, we have listed many factors that may potentially affect the priority of vulnerability discovered by analysing components of a program in isolation and, then, compositionally. The challenge we face is to assign *importance* to these factors in relative

order, to be able to combine them to arrive at a ranking (or base-score values, in case of CVSS).

Our approach in this thesis is to learn these functions to score vulnerabilities based on past-knowledge available openly for many real-world programs. The goal is to correlate the structural, organisational and asset factors to the assessment scales for discovered vulnerabilities. In this way, we will be able to learn how important these factors are for predicting the priority, and how to combine them for the best possible prediction power. In the next chapter (Chapter 10), we will instantiate this idea in a case-study and evaluate how a score predictor performs in the given scenario.

However, we recognise various caveats associated with such an approach based on the correlation of factors and priority of vulnerabilities. For example, we may not be able to measure all interesting factors precisely using nominal or categorical scales, especially the more subjective ones. Secondly, if not all factors are equally important for all general-purpose software, then we claim that it is a better idea to learn from prior knowledge related to a single project or programs related to a particular domain. In all such scenarios, it may be effective to consider our approach on a case-to-case basis instead of a gold-standard of vulnerability assessment.

9.4 Concluding Notes

In this chapter, we recollected the output from vulnerability discovery step that includes a list of vulnerabilities, vulnerable components and related components that are affected by them, in the form of a chain of vulnerable components. Based on the reasons discussed in Section 9.1, we showed that not all vulnerabilities that were not found to be feasible in parent components are *false-positives*, meaning that it cannot be assumed that they will *not* have any adverse effect if left untreated. In fact, a more useful approach in cases of unconfirmed vulnerabilities is to assign the discovered vulnerabilities appropriate priority values to be used as a proxy for prioritisation.

With these ideas as our base, we then discussed some existing scales for assessing vulnerabilities in real-world software and saw that CVSS [87] provides a more inclusive scale based, from the point-of-view of a potential attacker and the underlying assets. After that, we discussed various structural and organisational factors which may impact how severely a discovered vulnerability can affect a program and its attached components. We proposed an idea that utilises existing knowledge in terms of these factors extracted from old programs and predicts the base-scores of vulnerabilities based on how well combinations of certain factors correlate with them.

In the next chapter, we will expand on the ideas of this chapter by presenting a case study where we aim to predict the CVSS3 scores for discovered vulnerabilities using automatically extracted impact factors from many open-source projects.

10 Case Study – Machine Learning Based Score Predictor

This chapter instantiates some of the ideas presented in Chapter 9 in a case study involving vulnerability discovery, a vulnerability scoring scale and a prediction model generated by machine learning. Parts of this chapter have previously appeared in [97], where the author of this thesis was the first author.

In the previous chapter, we discussed in detail the need for an assessment process to determine how severely a discovered vulnerability may affect a program and its underlying assets if it was exploited, intentionally or unintentionally. We motivated the need for such a process by showing that some vulnerabilities may not be reported as feasible because of path-explosion in the parent components. This is why it may be more effective to accurately assess the discovered vulnerabilities even if they may seem like false-positives at first glance.

In this chapter, we will present a case study based on the ideas presented in Chapter 9. We designed and implemented an automated framework for assessing severity scores (based on CVSS3) for vulnerabilities discovered in many open-source programs and evaluated, and improved, our results based on experts' feedback. An overview of all the steps described in this chapter is depicted in Figure 10.1.

This chapter is organised as follows – In Section 10.1, we describe how we collected data from open-source software and bugs repositories and structured them for this study. Then, we describe the application and output of vulnerability discovery on these collected open-source programs, in Section 10.2. In Section 10.3, we present the design and intuition behind some structural factors that we chose as indicators for predicting the severities of discovered vulnerabilities. Using these automatically extracted features, we describe a machine learning-based predictor in Section 10.4. The presentation of the predicted

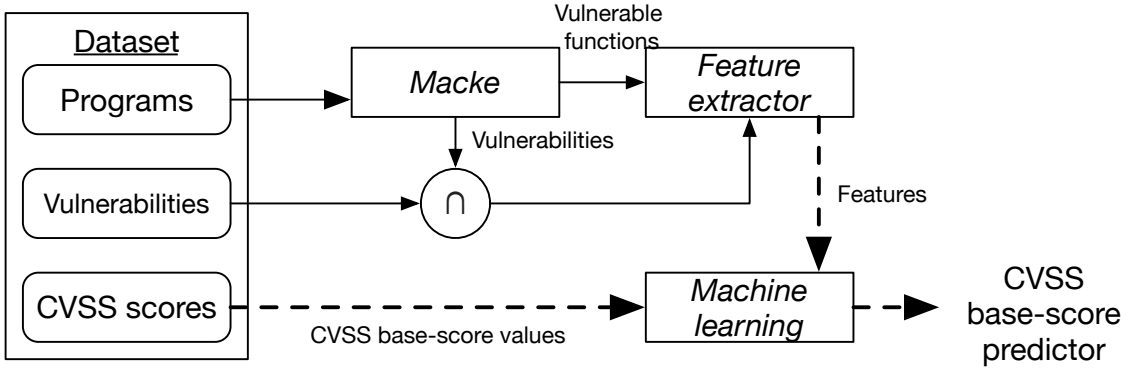


Figure 10.1: *Overview of steps to generate a CVSS base-score predictor. Steps depicted by dashed lines are repeated after receiving feedback from experts (Section 10.5).*

results and gathering feedback from experts is described in Section 10.5. Based on their feedback, we improve our machine learning models by adding more features in Section 10.6 and re-training our models in Section 10.7. Finally, we conclude with some high-level interpretation of the results in Section 10.9.

10.1 Collecting Data

We start this chapter by describing our data collection procedure for carrying out the case study. The proposed technique in this chapter, as we will describe in detail later, is based on machine learning. For this technique, we firstly need a list of vulnerabilities reported in the past. Such a list must include CVSS3 base-score values for all reported vulnerabilities. We rule out Bugzilla repositories, such as [58], because of reasons listed in Section 9.2. Instead, our choice of repositories of bugs is the National Vulnerabilities Database (NVD) [95]. NVD's lists of common vulnerabilities and exposures (CVEs) are always reported with CVSS scores (sometimes in CVSS version 3) and relevant references to detailed bug reports and proofs-of-concept.

For CVEs obtains from NVD, we further filter them to only include reports that satisfy the following criteria

1. Report follows CVSS3 notation, and not just CVSS version 1.0 or 2.0,
2. Reports a memory-related bug in a C-language program, and
3. Report specifies the name of the vulnerable C function.

After applying the above filtering criteria, the next step is to download the source code for all affected programs.

In Section 10.1.1, we will present the size of dataset and ground-truth for machine learning.

10.1.1 Data Collection Results

We included *21 open-source programs* in this study[1]. These programs are listed in Table 10.1, along with the lines of code in them (second column). The third column in this table shows the number of *connected functions*, i.e. functions that are reachable from the `main` function or the programs' APIs. The fourth column in this table shows the number of vulnerable functions, i.e. functions where *at least* one vulnerability was found by Macke. The number of vulnerable functions reported in this column is an *augmented value* that includes vulnerabilities reported in NVD and *some* vulnerabilities discovered by Macke and manually analysed and scored by us.

[1] The programs used in this case study are different from the programs used to analyse our vulnerability discovery framework in Part II. The reason for this is that we could not find enough vulnerability reports in NVD for the programs used for our previous evaluation. We could, therefore, not collect enough ground-truth to train our machine learning predictors described in this chapter.

Table 10.1: *Programs analysed and vulnerabilities in them*

Program and Version	LOC	Connected functions	Vulnerable functions from NVD, (with CVSS)$\|\|N\|\|$	Vulnerable functions CVSS scored manually, $\|\|M\|\|$	Vulnerable functions found by us, $\|\|X\|\|$
BlueZ 5.42	286,206	49	3	2	6
AutoTrace 0.31.1	18,581	23	3	0	3
GraphicsMagick 1.3	324,422	22	4	4	10
Icoutils 0.31.1	40,093	45	2	3	5
ImageMagick 6.0.4-8	476,747	51	1	3	8
Jasper 1.900.27	46,578	33	3	4	19
Jasper 2.0.10	46,622	33	2	3	6
Libarchive 3.2.1	204,993	62	1	4	15
Libass 0.13.3	18,745	46	1	3	29
Libmad 0.15.1	12,866	22	1	1	4
Libplist 1.12	6,075	69	1	5	27
Libsndfile 1.0.28	85,189	153	1	3	40
Libxml2 2.9.4	334,796	36	2	3	22
Lrzip 0.631	18,622	115	1	1	7
Openslp 2.0.0	55,545	27	1	3	17
Potrace 1.12	12,928	28	1	3	13
Rzip 2.1	2,651	34	1	3	19
Tcpdump 4.9.0	103,152	13	1	1	7
Tiff 4.7.0	82,725	125	3	2	6
Virglrenderer 0.5.0	57,213	70	2	0	21
Ytnef 1.9.2	4,818	70	6	1	12
		Total	**41**	**52**	**296**

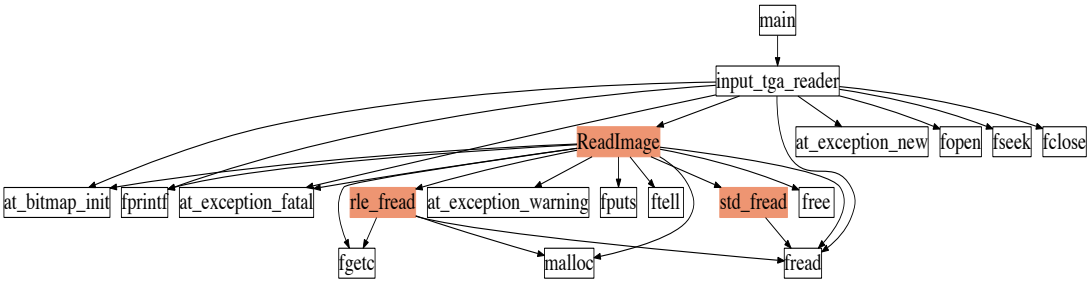


Figure 10.2: *Call-graph of Autotrace 0.31.1 program to convert a TGA bitmap to vector graphics format*

10.2 Discovering Vulnerabilities

The next step after collecting lists of vulnerabilities is to analyse the associated programs using our vulnerability discovery framework, *Macke*.

The output of vulnerability discovery includes a JSON file that lists

1. discovered vulnerabilities,
2. the vulnerable instruction, and
3. chains of feasibility for all discovered vulnerabilities.

The results of the vulnerability discovery step are presented now.

10.2.1 Vulnerability Discovery Results

In the last column of Table 10.1, we have listed *all* the vulnerable functions discovered by Macke in *30 minutes*. We found that the total number (296) uniquely vulnerable functions included *all* the vulnerabilities reported in NVD as well (demonstrating the strength of our techniques). The extra vulnerabilities (ones not rated on NVD) may not be true positives and, hence, require extra assessment.

For all the analysed programs we, next, obtained the *call-graphs*. An example of this is in Figure 10.2, where the call-graph of Autotrace 0.31.1 is shown. In this call-graph, those functions are highlighted where at least one vulnerability was found by Macke.

10.3 Extracting Features

In this step, we process the results from the data-collection procedure and vulnerability discovery by Macke, to extract some features related to the vulnerabilities and vulnerable functions. In this section, we will describe these features and their corresponding intuitions. We assume for all analysed programs in this study that the likelihood and ease of sanitising input are equal for all functions in a path that *uses* the said input. The readers must note that in the following description, we will use the terms “nodes” and “functions” interchangeable, as we wish to employ some concepts related to graph-theory (where terms such as “nodes” and “edges” are used heavily).

1. *Node degree (`d_in, d_out`)*, defined as the number of callers or callees (called, henceforth, also as *neighbours*) for a function in the call-graph. A higher node degree, as also explained by El Emam et al. [50], and Nagappan et al. [93], may make it more likely that a vulnerability in it may infect other functions. E.g. For the function `rle_fread` (Figure 10.2) the values of incoming node degree, `d_in`, is 1 and outgoing node degree, `d_out`, is 3.
2. *Distance to interface (`di`)*, defined as the length of the shortest path from a program entry point to the given function. The shorter this distance, the less likely it may be that a pointer argument was sanitised before being accessed. E.g. The value of `di` for `std_fread`, as seen from Figure 10.2, is 3.
3. *Clustering coefficient (`cc`)*, for a node is defined as the ratio of neighbouring nodes that are also mutually connected (as caller-callee pair, in our context). The intuition for this feature is that the bigger the clustering coefficient for a vulnerable function, the more likely it is that the vulnerability may be exploited by another function in the cluster. E.g. The value of `cc` for the function `rle_fread` (Figure 10.2) is 0.5 (out of 6 pairs of neighbouring nodes of `rle_fread`, 3 are connected in a caller-callee relationship).
4. *Node path length (`nl`)* is defined as the average distance from a node to all the nodes that are reachable from it. The intuition behind this feature is that the lower the value for `nl`, the higher the likelihood that input to it is *not* sanitised before being passed. E.g. Three functions are reachable from the function `rle_fread` (Figure 10.2) with respective distances as 1, 1 and 1. Therefore, $nl = 1$ for `rle_fread`. For calculating node path length, we deal with recursion by detecting loops in the call-graph and stopping to count distance when we find one.
5. *Vulnerabilities discovered (`nv`)* is a feature unrelated to the call-graph but is a simple count of vulnerabilities discovered in a function by Macke. A high value for `nv` might indicate [92] that an expected sanitisation for one or more function parameters was not done by a caller to the function. E.g. Macke discovered 1 vulnerability in the function, `rle_fread` and, therefore, `nv` is 1.
6. *Maximum length of infection (`li`)* is obtained from Macke's results. As we discussed in Chapter 7, one part of the output of compositional analysis is the chain of feasibility for a vulnerability in a function. The feature `li` takes exactly these chains into account and represents the length of the longest chain of vulnerability feasibility. E.g. For the function `rle_fread` in Autotrace (Figure 10.2), the chain of feasibility of vulnerabilities discovered in this function was found to be feasible in `ReadImage` function. Therefore, the value of `li` for `rle_fread` is 2.

10.4 Predicting Base-scores

10.4.1 Preparing Data for Prediction Models

Having obtained the basis for machine learning (ground-truth) in the form of features related to functions, vulnerabilities and a program's call-graph, we will now describe the training procedure for learning prediction models for CVSS3 of vulnerabilities discovered

by Macke. We first bring to the notice of the reader that we learn individual predictors for *each base-score value of CVSS3* by using them as targets for prediction based on the features listed above. The final CVSS3 severity score, *severity*, can be calculated with the general formula

$$severity = calc\_cvss3(y_{AV}, y_{AC}, y_{PR}, y_{UI}, y_S, y_C, y_I, y_A) \tag{10.1}$$

where the subscripts, *base*, are the CVSS3 base-scores described in Chapter 9, Concretely,

$$y_{base} = f^L_{base}(V) \tag{10.2}$$

where,

$$V = \langle d\_in, d\_out, di, cc, nl, nv, li \rangle \tag{10.3}$$

The superscript, L, in Equation (10.2) stands for "learned", denoting the learned model, f^L_{base}.

10.4.2 Machine Learning Models

For learning the CVSS3 scores, we used the following two machine learning algorithms to learn the functions, f^L

1. Random-forest classifier,
2. Naive Bayes classifier,

We used *scikit-learn* [110], a popular machine learning toolkit for Python. For all learning algorithms, we employ K-fold cross-validation for training. Then, we applied the model with the *best validation score* on the test dataset to calculate the test scores.

10.4.3 Machine Learning Results

We will now discuss the test scores of the learned predictors for both our machine learning models. The *ground-truth* to be used for machine learning is $G = N \cup M$, where N and M are as shown in Table 10.1.

The complete dataset, G, is split into training (75%) and testing (25%) sets. The training set is split for 4-fold cross-validation, i.e. 4 machine-learning models are trained by holding out folds one-by-one, and validation score calculated on the held-out fold. To remove the effect of random initial states, for all iterations, we trained 10 models generated with different seeds and perform majority voting for predicted base-score values. We calculated the *accuracy measure* for all base-scores values, which is the ratio of correct predictions out of all the predictions. The results of learned models on testing dataset are shown in Table 10.2, where the best accuracy scores are highlighted in bold.

From Table 10.2, we can see that, except *attack complexity* and *privileges required*, the best accuracy scores were obtained by us for random-forest classifier. We note that accuracy scores for *attack vector*, *confidentiality impact* and *integrity impact* (0.59, 0.64 and 0.55

Table 10.2: *Prediction results on test dataset – with original features (Section 10.3) only*

	Random Forest	**Naive Bayes**	**Random Guessing**
AV	**0.59**	0.27	0.25
AC	0.55	**0.59**	0.50
PR	0.91	**0.95**	0.33
UI	**0.73**	0.45	0.50
S	**0.91**	0.91	0.50
C	**0.64**	0.45	0.33
I	**0.55**	0.27	0.33
A	**0.82**	0.55	0.33

respectively) are better than randomly guessing them (because these base-score values may be in one of 4, 3 and 3 classes respectively).

10.5 Reporting and Gathering Feedback from Experts

The next step in this case study was to present the obtained results on the analysed programs to experts in secure software development (listed in the evaluation in Section 10.5.3). To do this, we applied the best of all learned models (based on their accuracy scores on testing dataset) to predict scores for previously unreported vulnerabilities (those vulnerabilities that were not part of the ground-truth) found by Macke.

10.5.1 Interactive Reporting of Vulnerabilities

The choice of medium to present the prediction results in our case study was a web application. The requirements for such an application are

1. Displaying the results of vulnerability discovery as a (clickable) call-graph, with the ability to focus (through zooming) on functions and viewing their source code.
2. Upon focussing on a vulnerable function, displaying the CVSS3 base- and aggregate scores.
3. Allowing a user/developer to change CVSS3 base-score values, leading to an automatic update of the aggregate value.
4. Allowing the user to send individual feedback in the form of textual information.
5. Logging all relevant interactions by a participating user (only for this case study, and after having informed them), including base-score values changed, function nodes clicked, and source code expanded.

To meet the above requirements, we decided to implement our web-application on a NodeJS [46] server, with a React [144] frontend. The implemented application's interface for Autotrace program is shown in Figure 10.3.

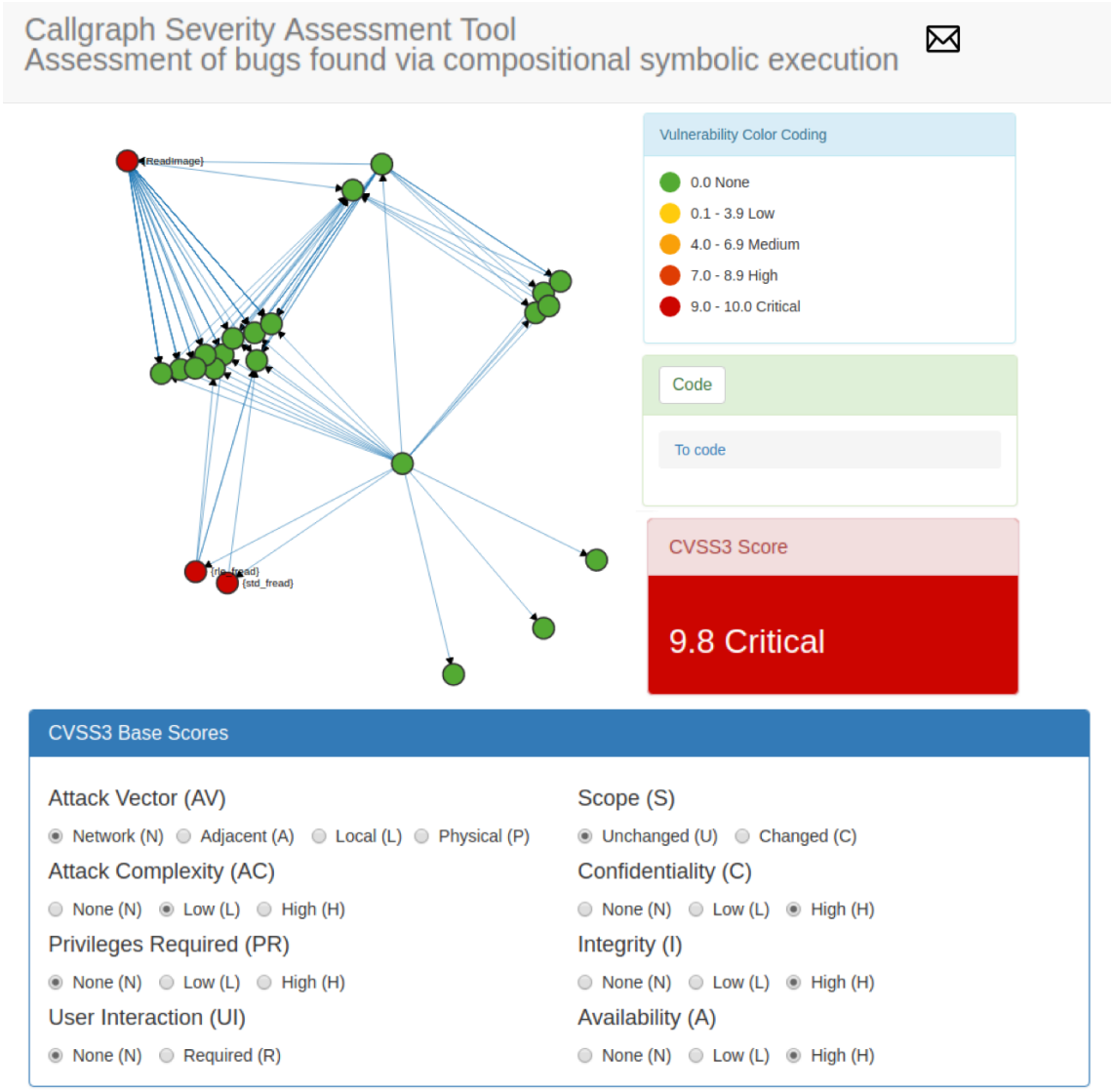


Figure 10.3: *Severity assessment interface for Autotrace 0.31.1 with interactive call-graph*

10.5.2 Gathering Feedback

We will now describe the process of presenting the interactive vulnerability reports to our target audience and obtaining feedback from them, to improve our framework's overall effectiveness in predicting CVSS3 base-scores. Please note that the distinction between previously reported (on NVD) and unreported vulnerabilities is transparent to the users. Our intuition for doing this is so that their feedback is not biased by past assessment for certain vulnerabilities. For previously unreported vulnerabilities, if the experts disagree with the predicted values, we request that they provide reasons for disagreement through a textual input field. As explained in the requirements, we collect the following information from all experts who participated in this study

1. Function nodes that were expanded (by clicking on them, as shown in Figure 10.3).
2. Function nodes for which any CVSS3 base-score values were changed (including old and updated values),
3. Function nodes for which the source code was expanded,
4. (optional) Any extra feedback from the expert
5. (optional) The expert's full name and email if they agreed to be contacted by us.

Above information is stored in a MySQL backend for manual processing at the end. For evaluating the effectiveness of our framework, we, first of all, used the feedback provided by the experts.

10.5.3 Feedback Results

We will now provide the actual results of applying the steps listed in Section 10.5.2. The predictions were made on, as described earlier, the dataset $(X - G)$ (previously unreported vulnerabilities or, the so-called, test dataset for machine learning). For getting feedback, we contacted

1. developer mailing-lists of the analysed programs,
2. students of a Master's level course, titled "Security Engineering", who had sufficient background in secure software development principles and symbolic execution, and
3. two technical staff's members at our organisation, one of whom has a doctoral degree in a security-related field.

The call-graphs, respective CVSS3 base-scores, and final scores are, then, presented using the framework described in Section 10.5.

The unique feedback items received from the participants of this study are listed in Table 10.3. In this table, anonymised users *1, 2 and 7* (first column) were technical staff's members at the organisation of this thesis's author. Users *3 and 6* were developers of the analysed programs who left feedback regarding the assessment of vulnerabilities discovered therein. Users *4 and 5* were students of *Security Engineering* course, offered at the author's university. The second column in lists the number of programs analysed by the respective participant. The third column lists the number of *unique* functions in the programs on which they either viewed the predicted CVSS3 scores or, additionally,

expanded the source code. The last column lists the number of optional comments left by the respective participant. The last row of Table 10.3 lists the number of *unique* analysed programs (hence, the values in the column do not add to 5), functions and feedback items received by us.

Table 10.3: *Summary of feedback received by experts*

Expert ID	Programs analysed	Functions expanded	Comments left
1	1	1	1
2	1	4	1
3	1	1	1
4	1	1	1
5	1	2	2
6	1	8	4
7	3	6	3
Unique	**4**	**20**	**13**

In listing 10.1, we have listed all the feedback received from the experts who participated in our survey[2]. Some of the original comments have not been included in this list because they were either irrelevant to the task at hand (e.g. "you must examine the latest version of this program because some vulnerabilities were fixed later"), or were identical to the comments left for other vulnerable functions or programs.

```
Program: Jasper 2.0.10
Functions selected: jpc_dec_decodepkt
Comment:
Without pinpointing the vulnerable instruction, the
function is very hard to analyse manually. Looking
at the size of this function (250 lines), it might be a
good idea to keep the score high, because it's likely
to be reused somewhere. I also think the signature
of the function suggests the incoming parameters are
very varied and, hence, might be prone to being unsanitised

Functions selected: jpc_dec_decodepkt, main, jpc_dec_lookahead
Comment:
The tool looks great, but it would be really useful if
for each function you would also indicate the number
of the LOC where the buffer overflow vulnerability
occurs. Otherwise, for large functions, it is difficult
to pinpoint the vulnerability manually. Of course, it is
also easier to analyse the code of commented functions
in comparison to functions without any comments.

Program: Rzip 2.1
Functions selected: read_buf, write_u16, BZ2_bzBuffToBuffCompress, write_buf
Comment:
All OK.

Functions selected: read_u8
Comment:
All is OK but for this file I'm not sure the result of confidentiality
```

[2]We have only omitted repetitive feedback items, i.e. where the feedback was identical to a feedback in listing 10.1 verbatim by the same author.

```
Functions selected: read_stream, write_stream, write_u32
Comment:
I think the file is used locally.

Program: Libass 0.13.3
Functions selected: ass_pre_blur1_vert_c
Comment:
To me this function does not seem to be exploitable via the network.

Program: ImageMagick 6.0.4-8
Functions selected: ReadRLEImage
Comment:
Here it looks to me like those code will be exploitable via the network as
    imagemagic is often used to parse network data
```

Listing 10.1: *Feedback from experts*

Synthesizing Feedback

We can now qualitatively analyse the comments listed in listing 10.1 to determine if, generally, our framework was successful in helping the security experts assess the discovered vulnerabilities. Overall, our observations from their feedback may be summarised as follows

- Most experts found the assessment framework to be useful.
- Bug-triage process is significantly affected by the size of the source code being analysed.
- In the absence of relevant comments, the perceived score of functions is affected by how "complex" it is.
- The perceived score of a vulnerable function, somehow, depends on what parameters are passed to it.
- Experts prefer pinpointing of the vulnerable instructions, rather than only the affected function.

10.6 Adding More Features

Drawing inspiration from the high-level feedback provided to us by the participants of this case-study, we moved to add more features to our initial dataset, to improve the accuracy of predictions. Below, we have listed the new features and some intuition behind each

1. *Function size (`s`)*: As some participants noted, the size of a vulnerable function points to a possibility that its functionality may have been wrongly implemented, whereas it should have been implemented as smaller functions. Therefore, we included *function size*, which is the number of LLVM instructions in the compiled version of a function.
2. *Approximate function complexity (`fx`)*: Another high-level feedback from the participants was that the *perceived* score of a vulnerability is related to the complexity of the function. To include this criterion, we added, in addition to function size, a count of LLVM *basic-blocks* in the function. A basic-block [67] is defined as a straight-line of instruction-sequence that contains no branches inside it, other than the entry or

exit points of the block. Past research [93] have also shown with case studies that even though no single set of indicators work equally well over different programs, the number of basic-blocks and arcs in a function's control-flow graph (CFG) did correlate with the possibility of a component's failure.

3. *Pointer parameters (`pt`)*: Based on the observation that many functions with pointer parameters were included in the set of functions whose vulnerability scores were changed by the participants of our study, we added another feature to our list of features that provides the number of parameters that are of *pointer type*. The intuition behind this feature is that, because we are dealing with memory-related vulnerabilities such as buffer-overflows, a higher number of pointer parameters may be correlated with a higher possibility of unsafe pointer manipulations or indexing, leading to errors.

10.7 Re-learning Predictor

With the original features (Section 10.3) and newly added features (Section 10.6), we re-train the same machine learning algorithms as listed in Section 10.4 to learn new predictive functions, f^L, for all CVSS3 base-score values. Consequently, the next effectiveness measure of our assessment framework is the accuracy of these re-trained models.

10.7.1 Machine Re-learning Results

Using the additional features of *function size*, *approximate function complexity* and *pointer parameters*, we trained our machine learning models, viz. random-forest classifier and naive Bayes classifier, each with 4-fold cross-validation. In Table 10.4, we have listed the accuracy scores on the test set with the newly learned machine learning models. From

Table 10.4: *Prediction results on test dataset – with original and added features (Section 10.6)*

	Random Forest	**Naive Bayes**	**Random Guessing**
AV	**0.64**	0.50	0.25
AC	**0.82**	0.55	0.50
PR	**1.00**	0.95	0.33
UI	**0.95**	0.95	0.50
S	**1.00**	0.95	0.50
C	**0.91**	0.91	0.33
I	**0.73**	0.50	0.33
A	**0.91**	0.82	0.33

this table, we can see that the accuracy of predicting CVSS3 base-score values increase for both random-forest and naive Bayes classifiers, in general. We can also see that the best accuracy scores for all base-score values were obtained by random-forest classifier. This increase in prediction accuracy was particularly noticeable for *privileges-requires* and *scope change* after including features based on feedback received from security experts and

developers and could be predicted with a 100% accuracy in the test dataset. The accuracy for predicting *User-interface* was also close to 100%.

10.8 Intuitively Analysing Case Study Results

In this case study, we have seen that even though the initial set of features, derived from analysed programs and discovered vulnerabilities, led to a good prediction score for all base-score values, the addition of more features after incorporating feedback from experts improved the prediction power even more. However, it should be emphasised that not *all base-score* values are correlated with the included features intuitively. While some base-score values, such as attack vector, complexity and change in scope may be predicted by considering structural features of the program, such as the degree of connectedness, others such as confidentiality and integrity of the affected data may not. One of the internal threats to validity of our case study is that we did not guarantee that an almost-uniform distribution in base-score values of CVSS3, e.g. our dataset does not have an approximately equal number of vulnerabilities with confidentiality impact of *low*, *medium* and *high*. This means that any classified based on machine learning may have predicted correctly for a majority of previously unseen examples, without necessarily finding a correlation between features and base-score values. We note that this is especially true for base-score values related to underlying data of a system, i.e. confidentiality, integrity and availability. However, for open-source programs whose usage context is impossible to know beforehand, it was also impossible for us to validate the ground-truth without also knowing the organisation factors and asset factors for their future usage.

10.9 Concluding Notes

In this chapter, we instantiated our vulnerability analysis framework in a case-study, by manually designing impact factors and automatically extracting them from some programs-under-analysis. We utilised popular machine learning models to train predictors for individual base-score values for CVSS3 scale to assess the vulnerabilities discovered by our framework. Later, we improved the prediction by consulting and incorporating feedback from several experts in secure software development.

From Tables 10.2 and 10.4, we can see that, while some CVSS3 base-score values could be predicted by our framework with high accuracy, there are others for which the framework does not perform reasonably well. We want to stress in this work that we don't claim that *all features* of extracted functions may correlate with *all base-score values*. Based on these results, we claim that our chosen features can be used to assign *most* of the base-score values with high accuracy for previously reported bugs. However, for other base-score values, where the accuracy of prediction is not as high, we should use other sources, such as function or requirements specifications, or even manual intervention for increasing effectiveness assessment.

The comments, as listed in listing 10.1, indicate that most experts who used our tool were satisfied by the format of the tool and agreed with the predicted values for base-scores

of previously unseen vulnerabilities. Some of the feedback, as seen in listing 10.1, was not concerned with the features of the analysed programs and functions but instead with the presentation of the results. Therefore, by qualitatively analysing feedback received from the experts, we can claim that such a tool for predicting CVSS3 severities can effectively aid vulnerability assessment.

An important feature of the interactive tool designed by us for this case study is that it does not depend on the technique used to discover the vulnerable functions. All the features used for machine learning, except `li`, can be just as easily extracted using any static analyser or even a manual code review. A comparison of the effectiveness of severity assessment based on different vulnerability scanners is left as future work.

Threats to validity: There is an external threat to validity of this case study whereby it cannot be said in general that the predictor learned using the features engineered by us can predict CVSS3 scores of vulnerabilities for any program. Our attempt to deal with potentially learning a limited predictor based on a single program's structural factors was to include as many and diverse programs from NVD as we could gather. In addition to the above, there is also an internal validity in our experiment design, that we mentioned in Section 10.8. By extracting a limited set of features (both, before and after collecting feedback from experts), we forced the machine learning models to learn correlations between *just those features* and the CVSS3 base-score values. However, as mentioned above, many base-score values may not even be intuitively related to any of the features picked by us, even if our results suggest that the prediction power of the learned model is sufficiently high. For this reason, we state that a predictive assessment framework that can effectively assist in triaging bugs discovered by our framework must include other organisational and asset factors (Chapter 9). The training and inference of the machine learning models should also be done in a particular context of development and usage. This case study could only demonstrate prediction accuracy based on the features that could be included from the open-source programs analysed.

Therefore, we have shown in this chapter using a specialised case study that the vulnerabilities discovered by our framework can be effectively assessed using an automated process, like the one described here. With an effective vulnerability assessment framework, software developers and testers can better handle and prioritise discovered vulnerabilities than devoting energy to classify whether certain reported vulnerabilities may be false-positives, i.e. never materialise in real-world usage.

Part IV

Conclusion

11 Conclusion

This chapter concludes this thesis by discussion resolutions to our original research questions, contributions of this thesis to the state-of-the-art, limitations of our work and future work.

In this doctoral thesis, we propose to design a framework for vulnerability discovery and analysis that aims to help developers catch potential bugs in programs, using a combination of symbolic execution and fuzzing. We started this thesis by describing at length the background of two dynamic analysis techniques, viz. *symbolic execution* and *fuzzing*. After describing the concepts of symbolic inputs and path conditions, we listed in detail the benefits and drawbacks of symbolic execution and explained how it suffers from the well-known problems of path explosion and constraint solving issues. Similarly, by describing the concepts of seed inputs and input mutation, we showed that fuzzing, while good at generating a large number of test-cases at high speed, fails to achieve high structural coverage in programs due to their reliance on diverse seed inputs. Additionally, we also listed, through a systematic mapping study, a large set of academic works proposed in the recent past that have tried to tackle the above issues related to symbolic execution and fuzzing by combining their technical aspects in meaningful ways to increase structural coverage and vulnerability discovery power for real-world programs.

We identified relevant gaps in research and built upon it by describing a scalable approach of dynamically analysing programs using a novel compositional approach involving symbolic execution and fuzzing. We proposed to, first, automatically isolate components of a program by generating test-drivers allowing their direct dynamic analysis. Then, we proposed three ways to analyse isolated components, viz. symbolic execution, fuzzing and greybox fuzzing. Greybox fuzzing is a novel analysis method that actively monitors if symbolic execution or fuzzing has saturated (unable to find further coverage) and switches over to fuzzing or concolic execution, respectively, and sharing inputs between the methods all this while. Finally, we proposed determining the feasibility of vulnerabilities discovered in isolated components by performing a two-phase compositional analysis – collating and matching the results of analysis and performing targeted symbolic execution towards automatically generated summary of the vulnerable components.

After performing compositional analysis for vulnerable components, we still might have many reported vulnerabilities whose feasibility could not be determined from a program's entry-point. However, instead of discarding those reports as "false-positive", we proposed a severity assessment approach in this thesis to prioritise the discovered vulnerabilities to assist the bug-triage process. Our proposed approach takes into account various structural and organisational factors for the program-under-test and uses prior domain knowledge to correlate these factors to various base-score values of CVSS3, a popular vulnerability scoring system.

We instantiated all the steps of our approach for C-language programs, where functions were considered as components of programs, and the parent relationship was described as the *caller* relationship (function calling). Our goal was to evaluate our scalable compositional

greybox fuzzing approach in terms of its effectiveness and efficiency in achieving high coverage and finding vulnerabilities in programs. We compared these values with state-of-the-art symbolic execution and fuzzing tools, many of which are advanced coverage-driven techniques proposed in the recent past.

In this chapter, we will, first, recall the research questions from the earlier parts of this thesis and answer them, having performed the evaluations. Then, we will list some limitations of the proposed approaches in this thesis. Finally, we will end this thesis with some ideas for future work in this field of research.

11.1 Revisiting Research Questions

We may summarise the findings of this thesis by recalling the research questions listed in Chapter 1 and providing answers for them.

RQ1: What are the concrete shortcomings and gaps in the state-of-the-art in solutions related to symbolic execution and fuzzing?

We, first, listed in Chapter 2 and Chapter 3 the existing problems affecting the state-of-the-art in symbolic execution and fuzzing. Then, in Chapter 4, we listed hybrid solution proposals that improve basic symbolic execution and fuzzing by modifying the various technical aspects of these techniques to increase coverage and find more vulnerabilities in programs-under-test. However, we also saw in this chapter that various avenues had not been addressed and investigated as possible solutions. Hence, we may now answer this research question as follows

> **State-of-the-art symbolic execution and fuzzing techniques are unable to achieve high coverage, especially for structures lying deep inside a program's control-flow-graph. As a result, they are not able to find vulnerabilities in uncovered parts.**

RQ2: How is structural coverage of components related to vulnerability discovery in them, and how may dynamic analysis exploit it?

In Chapter 5, we described an automated procedure to isolate components in a program by artificially removing the branching conditions for their entry, allowing them to be analysed independently of their parent components. Then, in Chapter 6, we described three dynamic analysis techniques, viz. symbolic execution, fuzzing and a novel greybox fuzzing approach, to find vulnerabilities in isolated components that do so by covering those instructions that could not be covered in a reasonable amount of time by state-of-the-art dynamic analysis techniques (without isolation of components). In Chapter 8, we saw that our hypothesis was indeed correct and, using our isolation approach, three dynamic analysis techniques were able to cover more instructions and find more vulnerabilities than state-of-the-art techniques. Hence, we may now answer this research question as follows

The vulnerabilities that were not discovered by state-of-the-art symbolic execution and fuzzing techniques can be discovered by adding more program entry points and allowing a dynamic analysis technique to analyse all components in isolation.

RQ3: How may the exploitability of discovered vulnerabilities be determined using the compositional nature of a program?

We asked in Chapter 6 a pertinent question about the feasibility of the vulnerabilities discovered in isolated components, i.e. how may we determine feasibility (exploitability) of a discovered vulnerability in isolated components? To answer this question, we described a two-phase automated compositional analysis procedure in Chapter 7 and instantiated it for C-language programs. Our evaluation (Chapter 8) showed that we were able to find *chains of vulnerable components* using the above procedure, by, first, comparing reported errors (stack-trace matching) and, then, targeted symbolic execution. Hence, we may now answer this research question as follows

Using compositional analysis the exploitability of the discovered vulnerability may be determined by summarising vulnerable components and targeting them from higher-level (in terms of *parent* relationship) components.

RQ4: For all discovered vulnerabilities, how may we prioritise the process of fixing them?

The need for a severity assessment portion of our framework arose from the observation that even though not all discovered vulnerabilities could be confirmed to affect a program entry point, they might still materialise in a yet unknown usage scenario in the future. By recognising various structural and organisational impact factors, as described in Chapter 9, we may be able to draw on prior knowledge to determine how severely the underlying assets may be affected if a vulnerability were to be exploited. We instantiated this idea in Chapter 10, where we designed a machine learning framework to predict the severity of vulnerabilities discovered in various open-source programs. We concluded for this case-study that, based on the precision of predicted severity values and feedback obtained from experts (developers of open-source programs, security researchers and students of security engineering), that such an assessment framework will, indeed, help in increasing the effectiveness of the programmers in prioritising the bug-triage process. Hence, we may now answer this research question as follows

The vulnerabilities discovered by our proposed approach may be assessed by considering various impact factors and combining them with knowledge of the domain and context of usage of the program-under-test.

11.2 Contributions

In this thesis, we have made the following contributions w.r.t. the gaps in the state-of-the-art as elaborated in Chapter 4.

1. In Part II, we described a dynamic analysis technique that can analyse isolated components to look for vulnerabilities in them and, then, compositionally determine the feasibility of the discovered vulnerabilities. We have shown in Chapter 4 that this was a gap in the research of hybrid symbolic execution and fuzzing works, i.e. programs' compositional aspects were not targeted by the proposed solutions in the past. Our compositional analysis approach is, therefore, a novel contribution of this thesis. Additionally, our open-source tool Macke [98] is the first of its kind in hybrid compositional analysis tool that can be used out-of-the-box for any general-purpose C programs.
2. In Chapter 6, we described a novel greybox fuzzing technique based on active saturation monitoring for symbolic execution and fuzzing. W.r.t. the state-of-the-art in fuzzing tools, this solution addresses the gap in fuzzing techniques that employ symbolic execution to increase line coverage in programs-under-test. Our results have indicated that, for many programs, we see increased coverage in isolated components when fuzzing is combined with symbolic execution in the particular way described in this thesis.
3. In addition to the improvement in the state of the art in fuzzing, our proposed greybox fuzzing approach also contributes to the field of symbolic execution by combining it with fuzzing, thereby taking the load off constraint solvers by using existing solutions from the fuzzer for easy-to-reach branches in a program.
4. Our final contribution is described in Part III, where we propose a generic methodology for assessing the vulnerabilities discovered by dynamic analysis with compositional analysis. The assessment framework relies on various organisational, structural and asset related factors to prioritise bugs discovered by our framework. This vulnerability analysis framework is a novel contribution as none of the existing research in the state-of-the-art have proposed a method to assess the discovered vulnerabilities for effective bug triage.

11.3 Limitations

Pointer Analysis Currently, support for a few kinds of pointers in Macke is insufficient. Particularly, if a function parameter list contains at least one parameter of double- or more pointers (such as array-of-arrays), then the function will not be isolated. The same is true if there is at least one parameter of *function-pointer* type. In case of pointers to structures which may, themselves, contain members of pointer data-types, Macke attempts to allocate memory for them (using `malloc`) and extract values for them from the function arguments, as described in Chapter 6. However, if there are no explicit checks on pointers inside structures, then a crash resulting from their access will be, correctly, reported.

Global Variables Another limitation of Macke is that it does not take into account the values of global variables that might affect the internal states of isolated functions. However, as is also true for past works in compositional analysis [40, 61, 102], when including all possible global variables in the argument extraction procedure, the search space for possible executions explodes intractably.

Selection of Comparison Baseline For evaluating the performance of vulnerability discovery, we compared Macke with baseline symbolic execution (KLEE) and fuzzing (AFL) tools, as well as more advanced coverage-guided tools (AFLFast and Munch). The rationale behind picking these tools for comparison was, as described in Chapter 8 that these are the most widely referred to tools in research and have been shown to have state-of-the-art coverage and vulnerability performance for general-purpose C-language programs. Even though we took utmost care to include the most representative set of state-of-the-art tools, there may have been other tools or framework that we may have unintentionally ommitted in our study and may achieve comparable, or better, coverage and vulnerability detection for the selected programs.

Impact Factors for Severity Assessment In the case study described in Chapter 10, we utilised only some structural features and features based on the discovered vulnerabilities, to predict their CVSS3 scores. The reason for not picking additional features, possibly even organisational ones, was that these programs were open-source projects, and their development context, history and organisational structures were readily available for everybody. Therefore, the prediction results described in this chapter might not apply to an arbitrary program developed under different circumstances. In general, it should be clear from the descriptions and discussions in Part III that any prediction method that works for a program, or programs developed and maintained under similar circumstances, may not be useful as-is for dissimilar programs in another context.

Choice of Analysed Programs In this study, we have evaluated all our proposed techniques on open-source C-language software. Our goal, as mentioned in Chapter 8, was to include real-world programs that accept input through CLI. However, we excluded some network-related programs and libraries that relied solely on function callbacks for the majority of their functionality, e.g. *ngircd*, which is a lightweight IRC client.

External Validity Due to the above reasons, and possibly more, our results may not generalise to other kinds of C programs that may functionally or structurally differ from our evaluation set. However, we have taken care to, firstly, not exclude any programs by design in our study and, secondly, include programs varying from medium- to large-scale, in terms of the number of high-level source-code lines and functions. We applied these selection criteria to, both, open-source programs with a `main` entry point and open-source libraries that are popularly used by many third-party applications.

11.4 Future Work

We will now list, from our perspective, some hints on how the ideas described in this thesis may be expanded upon in the future to improve the state-of-the-art in static and dynamic vulnerability detection for large and real-world software.

Scaled Analysis Time We saw from the evaluation of the vulnerability discovery phase that, for certain programs such as Flex, our compositional approach was unable to achieve as high coverage, or find as many vulnerabilities, as the baseline tools used for comparison. The reason for this ineffectiveness was that our approach gave an *equal amount of time to all functions* in a C program. Due to this, the larger (in terms of LOC) functions close to the program entry point were analysed for the same amount of time as functions lying deeper in the call-graph but, possibly, being only called seldom for realistic program inputs. A solution to this problem, which we leave as future work, is to scale the analysis time for a function according to some notion of *complexity* and *likelihood of reachability* so that easier, and deeper, components receive less attention than larger and more-often utilised components.

Vulnerability Description Language The method of analysis used to analyse isolated components of a program is, in theory, independent of the steps coming before and after it. The compositional analysis step depends only on the output of the analysis, containing details of discovered vulnerabilities such as the vulnerable instruction and stack-trace of a crashing execution. Therefore, as future work, we can imagine developing a standard description format for vulnerabilities discovered in an isolated component such that, as long as any mode of analysis, dynamic or static, follows the description standard for reporting vulnerabilities, it may be used to analyse an isolated component in a plug-n-play fashion in Macke.

Automated Bug-fix Generation Some vulnerabilities discovered by Macke may be able to use repeatable patterns for fixing them, e.g. a simple check on index-bound to fix buffer-overflows. As future work, we may use such fix patterns to automatically fix low-level vulnerabilities discovered by Macke in isolated components.

All the above ideas and the rest introduced in this thesis may be applied to any general-purpose programming language, for discovering language and system-specific vulnerabilities, in a highly effective and efficient manner, as demonstrated by this thesis.

Bibliography

[1] *Address Sanitizer (ASAN).* `https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm`.

[2] *AFL Fuzzer.* `http://lcamtuf.coredump.cx/afl`. Accessed: 2017-09-09.

[3] P. Agrawal and V. D. Agrawal. "Probabilistic analysis of random test generation method for irredundant combinational logic networks". In: *IEEE Transactions on Computers* 100 (1975).

[4] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. "A taxonomy of software integrity protection techniques". In: *Advances in Computers*. Vol. 112. Elsevier, 2019.

[5] F. E. Allen and J. Cocke. "A program data flow analysis procedure". In: *Communications of the ACM* 19 (1976).

[6] S. Anand, P. Godefroid, and N. Tillmann. "Demand-driven compositional symbolic execution". In: *TACAS*. 2008.

[7] S. Anand, C. Păsăreanu, and W. Visser. "JPF–SE: A symbolic execution extension to java pathfinder". In: *TACAS*. 2007.

[8] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. "Enhancing symbolic execution with veritesting". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014.

[9] P. Ayari K.and Meshkinfam et al. "Threats on building models from cvs and bugzilla repositories: the mozilla case study". In: *Conference of the center for advanced studies on Collaborative research*. IBM Corp. 2007, pp. 215–228.

[10] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. "Using static analysis to find bugs". In: *IEEE software* 25 (2008).

[11] S. Banescu. "Characterizing the Strength of Software Obfuscation Against Automated Attacks". PhD thesis. Technische Universität München, 2017.

[12] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. "Software complexity and maintenance costs". In: *Communications of the ACM* 36.11 (1993), pp. 81–95.

[13] C. Barrett and S. Berezin. "CVC Lite: A new implementation of the cooperating validity checker". In: *CAV*. 2004.

[14] C. Barrett and C. Tinelli. "Satisfiability modulo theories". In: *Handbook of Model Checking*. Springer, 2018.

[15] C. W. Barrett, D. L. Dill, and J. R. Levitt. "A decision procedure for bit-vector arithmetic". In: *Proceedings of the 35th annual Design Automation Conference*. 1998.

[16] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. "A taint based approach for smart fuzzing". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012.

[17] T. Berners-Lee. *"I was devastated": The Man Who Created The World Wide Web Has Some Regrets*. https://www.vanityfair.com/news/2018/07/the-man-who-created-the-world-wide-web-has-some-regrets.

[18] T. Berners-Lee. *The web is under threat. Join us and fight for it.* https://webfoundation.org/2018/03/web-birthday-29/.

[19] M. Bishop. *Computer security: art and science*. Addison-Wesley Professional, 2003.

[20] M. Böhme, V-T. Pham, M-D. Nguyen, and A. Roychoudhury. "Directed greybox fuzzing". In: *ACM SIGSAC Conference on Computer and Communications Security, Proceedings of the*. 2017.

[21] M. Böhme, V-T. Pham, and A. Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *IEEE Transactions on Software Engineering* (2017).

[22] K. Böttinger and C. Eckert. "DeepFuzz: Triggering Vulnerabilities Deeply Hidden in Binaries". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2016.

[23] C. Boyapati, S. Khurshid, and D. Marinov. "Korat: Automated testing based on Java predicates". In: *ACM SIGSOFT Software Engineering Notes*. 2002.

[24] A. R. Bradley and Z. Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007.

[25] J. Burnim and K. Sen. *Heuristics for Scalable Dynamic Test Generation*. Tech. rep. UC Berkeley, 2008.

[26] C. Cadar, D. Dunbar, D. Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Operating Systems Design and Implementation*. 2008.

[27] C. Cadar and D. Engler. "Execution generated test cases: How to make systems code crash itself". In: *SPIN Workshops: Model Checking of Software*. 2005.

[28] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. "EXE: automatically generating inputs of death". In: *ACM Transactions on Information and System Security (TISSEC)* 12 (2008).

[29] C. Cadar and K. Sen. "Symbolic execution for software testing: three decades later". In: *ACM Communications* (2013).

[30] J. Cai, S. Yang, J. Men, and J. He. "Automatic software vulnerability detection based on guided deep fuzzing". In: *International Conference on Software Engineering and Service Science*. 2014.

[31] F. Camilo, A. Meneely, and M. Nagappan. "Do bugs foreshadow vulnerabilities?: a study of the Chromium project". In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. 2015.

[32] S. Cha, M. Woo, and D. Brumley. "Program-adaptive mutational fuzzing". In: *Security & Privacy*. 2015.

[33] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on binary code". In: *Proceedings - IEEE Symposium on Security and Privacy* (2012), pp. 380–394. ISSN: 10816011. DOI: 10.1109/SP.2012.31.

[34] J. Chen, H. Shu, and X. Xiong. "Ewap: Using Symbolic Execution to Exploit Windows Applications". In: *WRI World Congress on Computer Science and Information Engineering*. 2009.

[35] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: (2018). arXiv: 1803.01307. URL: https://arxiv.org/pdf/1803.01307.pdf.

[36] Z. Chen, S. Guo, and D. Fu. "A directed fuzzing based on the dynamic symbolic execution and extended program behavior model". In: *International Conference on Instrumentation, Measurement, Computer, Communication and Control*. 2012.

[37] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E : A Platform for In-Vivo Multi-Path Analysis of Software Systems". In: *Security* 46 (2011), pp. 1–14. ISSN: 0362-1340. DOI: 10.1145/1950365.1950396. URL: http://dl.acm.org/citation.cfm?id=1950396.

[38] C. Y. Cho, V. D'Silva, and D. Song. "Blitz: Compositional bounded model checking for real-world programs". In: *ASE*. 2013.

[39] T. S. Chow. "Testing software design modeled by finite-state machines". In: *IEEE transactions on software engineering* (1978).

[40] M. Christakis and P. Godefroid. "IC-Cut: A compositional search strategy for dynamic test generation". In: *Model Checking of Software*. 2015.

[41] M. Christakis and P. Godefroid. "Proving memory safety of the ANI Windows image parser using compositional exhaustive testing". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. 2015.

[42] *Clang: C language family frontend for LLVM*. https://clang.llvm.org/.

[43] E. Clarke, D. Kroening, and K. Yorav. "Behavioral consistency of C and Verilog programs using bounded model checking". In: *Design Automation Conference*. 2003.

[44] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977.

[45] B. Cui, Y. Ji, and J. Wang. "An instruction-level symbolic checksum system for windows x86 program". In: *Chinese Journal of Electronics* (2012).

[46] R. Dahl. *Node.js – A JavaScript runtime*. https://nodejs.org/en/.

[47] D. E. Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[48] J. W. Duran and S. C. Ntafos. “An evaluation of random testing”. In: *IEEE transactions on Software Engineering* (1984).

[49] N. Eén and N. Sörensson. “An extensible AT-solver”. In: *International conference on theory and applications of satisfiability testing*. 2003.

[50] K. El Emam, W. Melo, and J. Machado. “The prediction of faulty classes using object-oriented design metrics”. In: *Journal of Systems and Software* (2001).

[51] R. E. Fairley. “Tutorial: Static analysis and dynamic testing of computer software”. In: *Computer* 11 (1978).

[52] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski. “Guest editors’ introduction: Software protection”. In: *IEEE Software* 28 (2011).

[53] D. Fangquan, D. Chaoqun, Z. Yao, and L. Teng. “Binary-oriented hybrid fuzz testing”. In: *International Conference on Software Engineering and Service*. 2015.

[54] L. D. Fosdick and L. J. Osterweil. “Data flow analysis in software reliability”. In: *ACM Computing Surveys (CSUR)* 8 (1976).

[55] The OWASP Foundation. *Application Security Risks*. 2017.

[56] V. Ganesh and D. L. Dill. “A decision procedure for bit-vectors and arrays”. In: *CAV*. 2007.

[57] V. Ganesh, T. Leek, and M. Rinard. “Taint-based directed whitebox fuzzing”. In: *International Conference on Software Engineering*. 2009.

[58] *GNOME Bugzilla*. `https://bugzilla.gnome.org/`.

[59] P. Godefroid. “Compositional dynamic test generation”. In: *ACM Sigplan Notices*. 2007.

[60] P. Godefroid, A. Kiezun, and M. Levin. “Grammar-based whitebox fuzzing”. In: *Sigplan Notices*. 2008.

[61] P. Godefroid, N. Klarlund, and K. Sen. “DART: directed automated random testing”. In: *ACM Sigplan Notices*. 2005.

[62] P. Godefroid, M. Levin, and D. Molnar. “Automated Whitebox Fuzz Testing”. In: *Network and Distributed System Security Symposium*. 2008.

[63] P. Godefroid, M. Levin, and D. Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Queue* (2012).

[64] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. “A classification of SQL-injection attacks and countermeasures”. In: *Proceedings of the IEEE International Symposium on Secure Software Engineering*. 2006.

[65] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. “Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations.” In: *Usenix*. 2013.

[66] R. Hamlet. “Random testing”. In: *Encyclopedia of software Engineering* (2002).

[67] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[68] G. Hoglund and G. McGraw. *Exploiting software: How to break code*. Pearson Education India, 2004.

[69] K. Inkumsah and T. Xie. "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution". In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 2008.

[70] S. Keele. "Guidelines for performing systematic literature reviews in software engineering". In: *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. sn, 2007.

[71] J. King. "Symbolic execution and program testing". In: *ACM Communications* (1976).

[72] B. Kitchenham, O. Brereton, D. Budgen, M. Turner, J. Bailey, et al. "Systematic literature reviews in software engineering–a systematic literature review". In: *Information and software technology* (2009).

[73] C. Kolb. *Constraint-size Thresholding in Symbolic Execution for Broader Path Coverage*. Technische Universität München. Bachelorarbeit. 2018.

[74] S. Krishnamoorthy, M. Hsiao, and L. Lingappan. "Strategies for scalable symbolic execution-driven test generation for programs". In: *China Information Sciences* (2011).

[75] D. Kroening and O. Strichman. *Decision procedures*. Springer, 2016.

[76] N. P Kropp, P. J. Koopman, and D. P. Siewiorek. "Automated robustness testing of off-the-shelf software components". In: *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. 1998.

[77] I.V. Krsul. *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[78] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. "Efficient state merging in symbolic execution". In: *Acm Sigplan Notices*. 2012.

[79] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization (CGO)*. 2004.

[80] C. Lemieux and K. Sen. "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018.

[81] Y. Lin, T. Miller, and H. Søndergaard. "Compositional Symbolic Execution using Fine-Grained Summaries". In: *ASWEC*. 2015.

[82] *LLVM Opt*. `http://llvm.org/docs/CommandGuide/opt.html`.

[83] K. Ma, K. Phang, J. Foster, and M. Hicks. "Directed symbolic execution". In: *Static Analysis* (2011).

[84] R. Majumdar and K. Sen. "Hybrid concolic testing". In: *ICSE*. 2007.

[85] R. Majumdar and R. Xu. “Reducing test inputs using information partitions”. In: *CAV*. 2009.

[86] B. Marczak and J. Scott-Railton. “The million dollar dissident: NSO group’s iPhone zero-days used against a UAE human rights defender”. In: *Citizen Lab* (2016).

[87] P. Mell, K. Scarfone, and S. Romanosky. “A complete guide to the common vulnerability scoring system version 2.0”. In: *Published by FIRST-Forum of Incident Response and Security Teams*. 2007.

[88] C Miller. “Babysitting an army of monkeys”. In: *CanSecWest* (2010).

[89] *Multiprocessing – The Python standard library*. `https://docs.python.org/3/library/multiprocessing.html`.

[90] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. “The design space of bug fixes and how developers navigate it”. In: *IEEE Transactions on Software Engineering* 41 (2015).

[91] D. Musliner, J. Rye, and T. Marble. “Using concolic testing to refine vulnerability profiles in FUZZBUSTER”. In: *Self-Adaptive and Self-Organizing Systems Workshops*. 2012.

[92] N. Nagappan and T. Ball. “Static analysis tools as early indicators of pre-release defect density”. In: *International conference on Software engineering*. 2005.

[93] N. Nagappan, T. Ball, and A. Zeller. “Mining metrics to predict component failures”. In: *International conference on Software engineering*. 2006.

[94] N. Nagappan, B. Murphy, and V. Basili. “The influence of organizational structure on software quality”. In: *Software Engineering, 2008. ICSE’08. ACM/IEEE 30th International Conference on*. IEEE. 2008.

[95] *National Vulnerability Database (NVD)*. `https://nvd.nist.gov/`.

[96] Y. Noller, R. Kersten, and C. S. Păsăreanu. “Badger: complexity analysis with fuzzing and symbolic execution”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018.

[97] S. Ognawala, R. N. Amato, A. Pretschner, and P. Kulkarni. “Automatically assessing vulnerabilities discovered by compositional analysis”. In: *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*. ACM. 2018.

[98] S. Ognawala, T. Hutzelman, F. Kilger, and E. Vintilla. *Macke – Modular and Compositional Analysis with KLEE (and AFL) Engine*. `https://github.com/tum-i22/macke`.

[99] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner. “Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach”. In: *Proceedings of the Symposium on Applied Computing*. ACM. 2018.

[100] S. Ognawala, T. Hutzelmann, and E. Vintilla. *KLEE22: A custom adaptation of KLEE for targeted symbolic execution*. `https://github.com/tum-i22/klee22`.

[101] S. Ognawala, F. Kilger, and A. Pretschner. "Compositional Analysis Aided by Targeted Symbolic Execution". In: *arXiv preprint arXiv:1903.02981* (2019).

[102] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer. "MACKE: compositional analysis of low-level vulnerabilities with symbolic execution". In: *International Conference on Automated Software Engineering*. 2016.

[103] S. Ognawala, A. Petrovska, and K. Beckers. "An Exploratory Survey of Hybrid Testing Techniques Involving Symbolic Execution and Fuzzing". In: *arXiv preprint arXiv:1712.06843* (2017).

[104] S. Ognawala, A. Pretschner, T. Hutzelmann, E. Psallida, and R. N. Amato. "Reviewing KLEE's Sonar-Search Strategy in Context of Greybox Fuzzing". In: *1st International KLEE Workshop* (2018).

[105] *Open data repository*. `https://osf.io/df87r/?view_only=f195375c8fd24c1aa46334e6dc2b7781`.

[106] B. Pak. "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution". PhD thesis. CMU, 2012.

[107] J. Pan. "Software testing". In: *Dependable Embedded Systems* 5 (1999).

[108] C. S. Păsăreanu and W. Visser. "A survey of new trends in symbolic execution for software testing and analysis". In: *International Journal on Software Tools for Technology Transfer* (2009).

[109] *Peach fuzzing platform*. `http://peachfuzzer.com`. Accessed: 2017-09-09.

[110] F. Pedregosa, G. Varoquaux, et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* (2011).

[111] J. Petley. "Panic stations: surveillance in the UK". In: *Index on Censorship* 42 (2013).

[112] V. Pham, M. Böhme, and A. Roychoudhury. "Model-based whitebox fuzzing for program binaries". In: *International Conference on Automated Software Engineering*. 2016.

[113] V. Pham, W. Ng, K. Rubinov, and A. Roychoudhury. "Hercules: reproducing crashes in real-world application binaries". In: *International Conference on Software Engineering*. 2015.

[114] B. Potter and G. McGraw. "Software security testing". In: *IEEE Security & Privacy* 2 (2004).

[115] A. Pretschner. "Classical search strategies for test case generation with Constraint Logic Programming". In: *FATES*. 2001.

[116] Y. Qin, Q. Wang, Y. J. Zeng, and Q. Xi. "Malware Behavior Analysis Technique Based on Approach to Sensitive Behavior Functions". In: *Applied Mechanics and Materials*. 2013.

[117] J. Radatz et al. "IEEE standard glossary of software engineering terminology". In: *IEEE Std* 610121990 (1990).

[118] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. "Vuzzer: Application-aware evolutionary fuzzing". In: *NDSS*. 2017.

[119] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. "Optimizing Seed Selection for Fuzzing." In: *USENIX Security Symposium*. 2014.

[120] R. S. Sandhu and P. Samarati. "Access control: principle and practice". In: *IEEE communications magazine* 32 (1994).

[121] K. Scarfone et al. *Common Vulnerability Scoring System v3.0: Specification Document*. Tech. rep. FIRST.Org, Inc, 2016.

[122] E. J. Schwartz, T. Avgerinos, and D. Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)". In: *Security and privacy (SP), 2010 IEEE symposium on*. 2010.

[123] K. Sen, D. Marinov, and G. Agha. "CUTE: a concolic unit testing engine for C". In: *SIGSOFT Software Engineering Notes*. 2005.

[124] K. Sen, G. Necula, L. Gong, and W. Choi. "MultiSE: Multi-path symbolic execution using value summaries". In: *FSE*. 2015.

[125] K. Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *Cybersecurity Development (SecDev), IEEE*. 2016.

[126] K. Serebryany. "OSS-Fuzz-Google {\textquoteright} s continuous fuzzing service for open source software". In: (2017).

[127] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." In: *USENIX*. 2012.

[128] M. Shahzad, M. Z. Shafiq, and A. X. Liu. "A large scale exploratory analysis of software vulnerability life cycles". In: *Software Engineering (ICSE), International Conference on*. 2012.

[129] C. Shortt and J. Weber. "Hermes: A Targeted Fuzz Testing Framework". In: *International Conference on Intelligent Software Methodologies, Tools, and Techniques*. 2015.

[130] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. "Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware." In: *NDSS*. 2015.

[131] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. "Alternate and learn: Finding witnesses without looking all over". In: *CAV*. 2012.

[132] International Organization for Standardization/International Electrotechnical Commission et al. "Information technology – Trusted Platform Module – Part 1: Overview". In: *International Standard, ISO/IEC* (2009).

[133] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, et al. "Driller: Augmenting fuzzing through selective symbolic execution". In: *Network and Distributed System Security Symposium*. 2016.

[134] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[135] R. Swiecki. *Honggfuzz*. `http://code.google.com/p/honggfuzz`. 2016.

[136] G. Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Tech. rep. National Institute of Standards and Technology, 2002.

[137] *The GNU C Library (glibc*. `https://www.gnu.org/software/libc/`.

[138] The OpenSSL Project. *OpenSSL: The Open Source toolkit for SSL/TLS*. `www.openssl.org`. 2003.

[139] *ThreadSanitizer*. `https://clang.llvm.org/docs/ThreadSanitizer.html`.

[140] N. Tillmann and J. De Halleux. "Pex–white box test generation for. net". In: *International conference on tests and proofs*. 2008.

[141] *Tool. American fuzzy lop (AFL)*. `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[142] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. "Chopped Symbolic Execution". In: *ACM/IEEE International Conference on Software Engineering*. 2018.

[143] *UndefinedBehaviourSanitizer*. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`.

[144] J. Walke. *React – A JavaScript library for building user interfaces*. `https://reactjs.org/`.

[145] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun. "SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 2018.

[146] T. Wang, T. Wei, G. Gu, and W. Zou. "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection". In: *Security & Privacy*. 2010.

[147] C. Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering". In: *International Conference on Evaluation and Assessment in Software Engineering*. 2014.

[148] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. "Fitness-guided path exploration in dynamic symbolic execution". In: *DSN*. 2009.

[149] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "Q SYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: (2018).

[150] B. Zhang, C. Feng, A. Herrera, V. Chipounov, G. Candea, and C. Tang. "Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing". In: *Iet Software* 12 (2018).

[151] J. Zimmerman. *Principles of Imperative Computation, 15-122*. 2013.

Index

List of Figures

List of Tables

List of Algorithms