Efficient Hybrid Fuzzing for Detecting Vulnerabilities and Achieving High Coverage in Software

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering

2023

By

Kaled M Alshmrany

Department of Computer Science

Contents

Word Count: 33,914

List of Figures

List of Tables

Abbreviations

SMB	Server Message Block
BMC	Bounded Model Checking
ESBMC	Efficient SMT-based Bounded Model Checker
AFL	American Fuzzy Lop
FA	Floating Point
SMT	Satisfiability Modulo Theories
IEEE	Institute of Electronics and Electrical Engineers
SAT	Satisfiability
TS	Transition System
VC	Verification Condition
PCSG	Probabilistic Context-Sensitive Grammar
CMU	Carnegie Mellon University
DoS	Denial of Service Attack
CDP	Cisco Discovery Protocol
KJ	Kilo Joule
CP	Core Point
GA	Genetic Algorithms
SAGE	Scalable Automated Guided Execution
PUT	Program Under Test
SSA	Static Single Assignment
AST	Abstract Syntax Tree
PCSG	Probabilistic Context Sensitive Grammar
SGF	Smart Grey-box Fuzzing
CFG	Control Flow Graph
CLP	Constraint Logic Programming
IL	Intermediate Language
MEIC	Maximum Expectation of Instruction Count

Abstract

Developing secure and bug-free software is an extraordinarily challenging task. Due to the devastating effects vulnerabilities may have on financial, security, or an individual's well-being. Detecting such issues is difficult because (i) many bugs manifest themselves only after a lengthy operation, and (ii) the search space to be explored becomes complex and extremely extensive. In this thesis, we describe and evaluate approaches for detecting vulnerabilities and achieving high coverage in C software using the combination of bounded model checking (BMC) and fuzzing. We present three significant novel contributions. First, we develop a method that generates initial inputs (seeds) that bypass sophisticated guards to enhance the fuzzer's exploration more profound into the target program. Furthermore, this method decreases the burden of the fuzzer in mutation processes through static analysis. As part of this contribution, we propose and design a tracer subsystem, which coordinates and analyses the processes and the connection between the employed techniques. Second, we present our new fuzzer, which has the benefit of performing a lightweight static program analysis to identify input verification. This improved fuzzer has the benefit of performing a lightweight static program analysis to identify input verification and to ensure that only seeds satisfying the conditions are chosen. This procedure reduces our method's dependence on a computationally expensive bounded model checker to discover high-quality seeds. Also, the improved fuzzer analyses the target program and identifies potential infinite loops using heuristics. The loops are then constrained to speed up the fuzzing process, depending on an approximate estimate of the number of program paths. In addition, we describe our new approach: a selective fuzzer that learns from test cases produced by BMC and a modified fuzzer to generate new test cases that successfully detect software vulnerabilities. Finally, we develop and evaluate *FuSeBMC*, an automated testing tool that exploits the combination of BMC and fuzzing to test software and increase code coverage. *FuSeBMC* has demonstrated advantages in resource management and consequently reduces the consumption of CPU and memory by exchanging essential information between engines in a manner that maximises the benefit of their cooperation. Additionally, it decreases the generation processes for execution paths that BMC may not reach or cause path explosion problems. As a result, *FuSeBMC* can mitigate the negative impact, generate effective seeds, and avoid the path explosion issue. *FuSeBMC* has been evaluated exhaustively and competitively by participating in the most prominent and competitive international software testing competition for two years, 2021 and 2022, winning six international prizes. *FuSeBMC* is currently the leading state-of-the-art software testing tool for C programs. We further hypothesise that *FuSeBMC* is currently the most robust automated testing tool in the literature.

Declaration

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

Copyright

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

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

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

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

List of Publications

Published and accepted papers:

Kaled M Alshmrany, Rafael S Menezes, Mikhail R Gadelha, and Lucas C Cordeiro. "FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs". In: *The 24th International Conference on Fundamental Approaches to Software Engineering (FASE)* 12649 (2020). https://doi.org/10.1007/978-3-030-71500-7_19, pp. 363–367.

Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. "FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in c programs". In: *The International Conference on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-030-79379-1_6. Springer. 2021, pp. 85–105.

Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. "FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing". In: *The 25th International Conference on Fundamental Approaches to Software Engineering (FASE)* 13241 (2022). https://doi.org/10.1007/978-3-030-99429-7_19, pp. 336–340.

Kaled Alshmrany, Ahmed Bhayat, Franz Braußе, Lucas Cordeiro, Konstantin Korovin, Tom Melham, Mustafa A. Mustafa, Pierre Olivier, Giles Reger, and Fedor Shmarov. "Position Paper: Towards a Hybrid Approach to Protect Against Memory Safety Vulnerabilities". In: *IEEE Secure Development Conference*. Aug. 2022.

Submitted / In progress / co-authored papers:

Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, Fedor Shmarov, Fatimah Aljaafari, and Lucas C Cordeiro. "FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis". In: *The Formal Aspects of Computing Journal (FAC)* (2022).

Mohannad Aldughaim, Kaled Alshmrany, Mohamed Mustafa, Lucas Cordeiro, and Alexandru Stancu. "Bounded Model Checking of Software Using Interval Methods via Contractors". In: *arXiv preprint arXiv:2012.11245, Submitted in International Conference on Software Engineering (ICSE 2023)* (2022).

Acknowledgements

First and foremost, I thank the Almighty Allah for giving me the strength to do my PhD. I am grateful for all your blessings.

I would not be where I am now without the folks that stood by my side along the journey. I want to express my sincere gratitude to my supervisor Dr. Lucas C. Cordeiro for his guidance and assistance throughout my PhD journey. Dr. Lucas is a great supervisor who inspired me to be a scientist and do the right thing, even when the road got tough. I faced numerous challenges during my PhD, but he was always encouraging and helped me see things from a different, more optimistic viewpoint; I'll miss his phrase "good student who follows what the supervisor says." Thank you for supporting me, for the long discussions and for lifting me when I was feeling upset. Your patience, insights, and good humour were invaluable and very much appreciated.

Thanks to the whole ESBMC team, including Lucas Cordeiro, Mikhail Gadelha, and Rafael Menezes. I learned something from all of them. Likewise, I thank the entire SCorCH team, especially Giles Reger, Fedor Shmarov, and Ahmed Bhayat. In particular, I benefited from conversations and collaboration with the hybrid approach.

The research community at the university of Manchester helped make the PhD an enjoyable experience. Thanks to my fellow PhD students, in particular Ahmed Bhayat, Mohannad Aldughaim, and Moteb Alghamdi, whom I learned from and shared my ideas during my PhD journey.

Thanks to my dear family, who has always been a supporter. Special thanks to my mother for all of her moral support, patience, and concern. Thank you for being there when I most needed you. Finally, big thanks to my great father for supporting me throughout my entire education. Thank you for making my life easy and assisting me in achieving my dream.

To my dearest mother and father . . .

Chapter 1

Introduction

> When you change the way you look at things, the things you look at change.
>
> *Max Planck*

Cybersecurity is a global phenomenon that presents researchers with a significant technical challenge. Cybersecurity challenges come in various forms, including ransomware, phishing, and malware attacks [7]. However, cybersecurity also requires the involvement of different techniques to address these challenges [8, 9, 10, 11]. While cybersecurity is one of the most serious challenges nowadays, access to information and knowledge has never been as easy as today. In this regard, technology plays a significant role.

Errors in computer systems can have a significant negative impact and are dangerous since they potentially cause monetary and human loss. For example, the crash of the Bloomberg terminal in 2015 caused substantial financial losses. It forced the government to postpone the sale of debts estimated at three billion pounds [12]. In addition, a flaw in the win32k system enabled Windows 10 users to escape from security sandboxes due to a security vulnerability [13]. Also, in 2015, a software bug rendered an F-35 fighter aircraft incapable of detecting targets accurately [14]. In terms of human lives, the China Airlines Airbus A300 crash on April 26, 1994, which was caused by a software error, claimed 264 lives [15].

Vulnerabilities are flaws or weaknesses in the design of a system that allow an attacker to exploit and violate the system's security policy. They are considered a significant cause of a wave of threats to cybersecurity [16, 17]. A vulnerability-based cyber attack can pose significant damage. For instance, the WannaCry ransomware attack in 2017 exploited a vulnerability in Server Message Block (SMB) protocol, which resulted in millions of pounds in losses and gross reputation damage to the UK medical sector [18]. It has infected over $200,000$ computers in 150 countries [19]. In addition, it has caused severe crisis management issues and enormous financial losses for numerous companies and governments. WannaCry ransomware attack is notorious for blocking user access to

files and systems by encrypting them until the victim pays a ransom for the decryption key [20]. Even though it has been nearly four years since the first WannaCry ransomware attack, WannaCry attacks have recently increased. Compared to January 2021, Check Point indicates a 53% increase in WannaCry ransomware attacks in March 2021.

Software testing is one of the most significant validation methods [21]. Research has shown that software testing may detect up to 90% of the errors detected during development [22, 23]. Software testing is a method of executing a program to locate software bugs and vulnerabilities, thus guaranteeing software product quality. The concept of software testing is detecting problems by running exhaustive tests known as test suites. However, establishing a test is difficult since it requires knowledge of the system's behaviour, how it might be violated, and the capacity to reach and cover deep system paths. In addition, software testing must be updated periodically during the system development phase to ensure its efficacy. Moreover, these tests may result in a significant drain on effort, negatively reflecting on the system development.

In recent years, research has been devoted to reducing the impact of development expenses by relying on automated techniques. *Fuzzing* is one of the most common techniques [24]. In general, the fuzzing technique works to automatically provide random or invalid data to test the system for various exceptions, such as system crashes, code failures, and monitoring behaviour, or to provide coverage statistics. It yielded interesting results even though it runs randomly during the detection process [25], which may require considerable effort to identify unacceptable behaviour. *Symbolic execution*, on the other hand, is another technique that analyzes the source code, identifies the inputs that execute each part of the program, and solves its constraints by assuming symbolic values rather than constant values [26]. However, symbolic execution faces challenges in resolving constraints containing non-linear arithmetic combinations and path explosion with loops that may result in infinite operations [27].

Moreover, *static analysis* techniques have been investigated, where they concentrate their efforts solely on the source code and disregard the actual program execution [28]. Static analysis techniques utilize an abstract domain to track the current program state. For example, it can determine the upper and lower bounds of a program variable rather than all the possible values it can assume throughout the program execution. Also, path, flow, and context-sensitive can be added to static analysis by differentiating between pathways, statement execution order, and method call [29]. Static analysis is often used to detect common errors such as overflow, array out-of-bound, memory leaks, and invalid arithmetic operations. Although static analysis techniques are the most widely used technique for program analysis, they may fail to demonstrate some safety properties and have high false-positive rates, resulting in inaccurate reporting [30].

Formal verification methods also have a role in this field, which can be split into *deductive verification* and *model checking* [31]. *Deductive verification* utilizes computer-

assisted theorem provers to prove the correctness of a system [32]. It can handle the data-intensive elements of the design. Although it is time-consuming, it is scalable to large systems [33]. *Model checking* is a computer-aided formal method for verifying the correct functioning of a system design model [21, 34]. It examines all potential model behaviours. Also, model checking can address control-flow challenges, including concurrency and deadlock [31]. However, model checking faces difficulties when exploring the entire state space. It is highly resource-hungry, which might lead to the consumption of memory or time before providing an answer.

With software testing technologies, organizations seek testing completeness and quality metrics to establish test completion criteria. One such metric is code coverage [35]. Code coverage is a crucial component because the ability to cover all paths in a program increases the rate of error detection or, at the very least, ensures that an execution path in the program does not contain any errors [36]. If coverage is less than 100%, further tests can be created to test the elements that were missed, hence increasing coverage. Test coverage can assist in assessing the quality of testing and directing test generators to produce tests that cover previously untested areas [37].

This thesis uses two techniques above - *Fuzzing* and *Bounded Model Checking* - to verify real-world C programs and increase code coverage. These techniques were selected due to their impact on the field and simplicity in understanding and development. Also, the main motivation behind combining such complementary techniques is to leverage the strengths of concolic execution and bounded model checking in generating inputs satisfying complex branch conditions, which are challenging to derive for mutation-based fuzzing. At the same time, fuzzing can quickly explore deep paths with simple checks that can offset the large resources consumption of concolic execution and bounded model checking. These techniques are summarized below and discussed in further detail in Sections 2.3, and 2.4. We also experimentally evaluated their performance against the proposed hybrid fuzzer algorithm in Chapter 3 and Chapter 4.

Bounded Model Checking (BMC) is a method of unwinding a program and limiting bound until it detects a property violation. It relies on the symbolic implementation of unrolling the program's loops [38]. BMC has been applied in many single- and multi-threaded programs to detect subtle errors [39, 40] and demonstrate its effectiveness [41]. Although BMC can detect shallow bugs efficiently, the BMC method may not detect profound errors efficiently unless the bound is large enough to reach all cases in the state space. However, widening the limit may cause the BMC method to become slow and resource-intensive for error detection and code coverage [42]. Therefore, we employ BMC as a component in our automated approach to detect shallow bugs and help the fuzzer to explore profound bugs in the program. Furthermore, we evaluate BMC when

verifying and obtaining code coverage on general and various criteria that include much real-world software.

Fuzzing has been relied on in many works and has demonstrated extraordinary efficacy, making it the standard in commercial software development processes [43]. Nevertheless, despite its advancements, fuzzing still faces challenges that it suffers. One of them is its inability to explore deep paths. Also, it causes an overhead if the initial seed is inefficient because its mutation is dependent on the seed. Moreover, fuzzing may encounter difficulties while exploring program execution paths with complex programming protections. Therefore, we considered these challenges in our automated approach to reduce the negative impact and benefit from our analysis in alleviating overhead and providing effective seeds.

1.1 | Problem Statement

In order to effectively combine the fuzzing and BMC to generate test cases that verify C programs and achieve high coverage, we first need to focus on and understand the challenges and shortcomings. Therefore, this PhD thesis presents the following challenges and shortcomings of fuzzing and BMC: First, a fuzzer finds it hard to explore program sections occurring behind complex guards [44]. Second, the instrumentation of the target program brings a significant overhead to the program execution, affecting the fuzzer's execution speed. Third, the straightforward method by which a fuzzer generates seeds might result in the fuzzer being stuck in one portion of the code and failing to explore other branches. Four, BMC explores a potentially exponential number of paths in the source code which may lead to path explosion. Finally, BMC takes enormous time when there are many loops, making it slow and resource-intensive [42].

To illustrate the main shortcomings of both fuzzing and BMC, we introduce a short C program in Figure 1.1. The presented program accepts coefficients of a quadratic polynomial and an integer candidate solution in the range [1,100] as input from the user. It terminates successfully if the provided candidate solves the equation. However, the program returns an error if the given equation does not have real solutions or the input candidate value is outside the [1,100] range.

The program takes four integer inputs (lines $5, 6, 7,$ and 10): a, b, c, and x. On line-8, it checks if the condition $(b*b >= 4*a*c)$ is true. After that, there is a loop that iterates an unknown number of times with an if-condition inside (line-11). Lastly, an else-condition on line-17 can be reached after the loop terminates.

Let us suppose we need to generate test cases to cover all the branches in this program. Fuzzing struggles to generate the inputs that satisfy the if-condition because of the complex mathematical guards on line-8. Such guards pose a challenge to a fuzzer [44] as it relies on mutating the given seed randomly and is therefore unlikely to satisfy the

```
 1 #include <assert.h>
 2 void reach_error() { assert(0); }
 3
 4 int main() {
 5   int a = input();
 6   int b = input();
 7   int c = input();
 8   if(b*b == 4*a*c) {       // fuzzer struggles
 9     while(1) {             // unknown # iterations
10       int x = input();
11       if(x <= 0 || x > 100) // easy to reach by both
12         reach_error();
13       else
14         return 0;
15     }                      // BMC struggles
16   }
17   else
18     reach_error();
19
20 return 0;
21 }
```

Figure 1.1: Illustrative example

guard condition. At the same time, a fuzzer does not have to deal with loop termination on line-9 as it only needs to generate random test inputs, in contrast to BMC, which needs to unwind each loop a sufficient number of times. This brings challenges to BMC. In order to illustrate these challenges, we expanded the execution of line-9 in figure 1.2. For example, the if-statements on lines 1 and 8 represent the first and second unwindings of the loop. Observe that there are three paths between lines 1 and 14. In detail, three paths through the if-statement on line-1, followed by three paths through the if-statement on line-8. Therefore, k unwindings of the loop will result in k if-else statements paths. This indicates that there will be 3^k possible paths, resulting in an exponential explosion that makes the model checking process intractable [45].

This shows that there could be parts of a program that are hard for fuzzing and parts that are hard for BMC, inhibiting both techniques from producing complete test inputs. Therefore, the problem statement focuses on overcoming challenges, integrating the strengths of fuzzing and BMC technologies, and mitigating their negative effects to generate test cases that verify C programs, achieve high coverage in a sufficient time, and reduce energy consumption.

1.2 | Scope of this Thesis

This thesis focuses on automated software testing and code coverage for programs written in the C programming language. In particular, this thesis will also concentrate on the methodologies and tools that enhance the efficiency of the used techniques while reducing used hardware resource consumption further.

Despite the prevalence of high-level programming languages, we chose to focus on the C programming language because it is widely used [46], and most system software

```
 1 if(*) {                      // unwinding# 1 for the loop
 2    int x = input();
 3    if(x <= 0 || x > 100)
 4       reach_error();
 5    else
 6       return 0;
 7 }
 8 if(*) {                      // unwinding# 2 for the loop}
 9    int x = input();
10    if(x <= 0 || x > 100)
11       reach_error();
12    else
13       return 0;
14    }
15       .                      // unwinding# ?? for the loop
16       .                      // unwinding# until unknown !!!
17       .                      // 2^ #unwinding paths... explosion!!
18 }
```

Figure 1.2: BMC unwinding of the loop at line 9 of figure 1.1

like the Unix computer operating system uses it [46]. Moreover, there are only a few computer architectures for which no C compiler exists [47]. Also, it is used to program complex and essential data because of its adaptability and efficacy [48].

The C programming language is generally involved in many domains, such as (Operating Systems, the Development of New Languages, and Embedded Systems). For example, C is the programming language used to create the Unix-Kernel, Microsoft Windows utilities, operating system programs, and a significant portion of the Android operating system.

The growth of the C programming language has an impact on programming languages as well. Examples include C++, C#, Python, Java, JavaScript, PHP, and the C shell of the Unix operating system, among others. Every language makes use of C to varying degrees. For example, language syntax and control structures like C++, PHP, and Perl are based on C, whereas Python leverages C to provide standard libraries.

On the other hand, C is the language of preference for creating embedded systems. This language's popularity can be attributed to its availability of machine-level hardware Interfaces, C compilers, and deterministic resource utilization.

Therefore, providing methods and techniques for testing any C software and presenting statistics analysis of code coverage will significantly benefit correcting the systems.

Automated testing is applied directly to the program's source code, thus reducing potential impediments to its implementation [28]. Additionally, it often saves time because the tester may execute a significant number of tests in a short period. Also, automation testing saves money and effort, enhances the quality of testing tasks [49], and contributes to software quality [50].

In the context of software testing and achieving high code coverage for C programs, we define our research question in three parts:

- How can fuzzing and BMC techniques be enhanced and combined to allow the validation of real-world programs?

The Clang front-end offers many advantages for analyzing and validating programs. Furthermore, it uses much less memory than other compilers, allowing taking more code into memory at a time [51]. Clang, a state-of-the-art compiler, has been widely used in industry because it is fast, light, and reduces interface issues. In addition, it provides clear and concise diagnostics [51]. Finally, an incremental bounded model checking is an adaptive and evolving method for other methodologies by taking advantage of the outputs provided by this method to support other technologies, such as selective fuzzer in our approach. This is the focus of Chapter 3.

This thesis introduced a novel approach implemented in a new tool named *FuSeBMC* to automate test generation that combines Fuzzing and BMC technologies to detect security vulnerabilities in C programs. The work conducted during this thesis successfully explored and analysed the targeted C program through the Clang compiler [51] to inject labels incrementally, which will be used later to guide the Fuzzing and BMC techniques to produce test cases. In addition, a new fuzzing algorithm was introduced as a part of this research, which learns from the test cases produced by the implemented techniques to produce new test cases to explore the remaining uncovered program paths. Lastly, a novel algorithm was presented for managing the time allocated to fuzzing and BMC to improve *FuSeBMC*'s energy consumption.

- How can resource consumption be reduced during testing while achieving high accuracy results and better code coverage on the targeted software?

Software Testing is time-consuming and resource-hungry [52]. It accounts for approximately 40% to 50% of total resources and 30% of total effort [53, 54]. Since resource consumption accounts for a significant portion of test generation cost, it may also be important to consider energy consumption efficiency. In addition, consider performance and its associated cost. Performance is problematic for validators because of the state explosion problem. The state explosion problem can consume a great deal of time and resources and detract from their primary responsibility [55].

We present our green testing approach that uses low energy consumption and maintains high performance and code coverage by linking BMC with fuzzing to work collaboratively and monitor the processes in each technique. This approach is designed and evaluated in Chapter 3.

- How can the quality of test cases be enhanced by employing additional analysis to eliminate technique overheads and reduce the time necessary to achieve high code coverage?

Testing tools have worked on making many methods to improve testing quality and coverage by producing a high-quality test suite and achieving coverage of most execution paths in the program. Nonetheless, the production of this test suite is an expensive burden, which causes industry tools to produce unsound and incomplete testing tools because their requirements are low, and they give results quickly. However, these results are inaccuracy and could not achieve a high percentage of code coverage.

We present our *FuSeBMC* v4 approach, which relies on smart seeds, static analysis, and subsystem tracer to generate high-quality test suites with accurate results and high coverage while simultaneously reducing the costs associated with the fuzzing and BMC techniques. Our new approach is described in Chapter 4.

1.3 | Contributions

The main contribution of this thesis is the development, implementation, and evaluation of a hybrid fuzzer to automatically verify C programs and achieve high coverage. In this respect, this thesis provides three significant novel contributions.

First, we introduce an automated approach based on BMC technology for seed generation that bypasses complex guards and thus helps fuzzers explore deep paths within the targeted program. Also, it simplifies the target program through static analysis to reduce the overhead of fuzzers when managing the mutation process to speed up the fuzzing process. In addition, it presents a new method for producing seeds that makes the fuzzers process effective and fast due to its reliance on the subsystem that coordinates the process and static analysis. Finally, since loop unwinding causes an exponential path burst, we limit the decoding depth of each loop to a small number, depending on a rough estimate of the number of program paths.

Second, we describe the new fuzzing technique based on the popular American Fuzzy Lop tool [56]. The newly presented approach can perform a lightweight static program analysis to recognize input verification. It analyzes the code for conditions on the input variables and ensures that seeds are only selected if they pass these conditions. This reduces the dependence on the computationally expensive bounded model checker for finding quality seeds. Another interesting feature of the modified fuzzer is that it analyses the PUT and identifies potentially infinite loops heuristically. It then bounds these loops in an attempt to speed up fuzzing. During the several rounds of fuzzing, these bounds are incremented. Furthermore, we introduce our new method, a selective fuzzer based on learning from test cases generated by BMC/Fuzzing to produce new test cases that can successfully detect new vulnerabilities.

Finally, we developed *FuSeBMC*, an automated test generation tool that exploits the combination of Fuzzing and BMC to verify real-world C programs and increase code coverage. *FuSeBMC* was distinguished by its novel algorithm for managing the time al-

located to each engine and goal. The advantage of this algorithm is to prevent *FuSeBMC* from wasting time finding test cases for challenging goals. At the same time, the information gathered before any prevention is used later by a selective fuzzer. Furthermore, this approach can reduce generation processes for paths we believe BMC cannot access. As a result, we can generate high-coverage test cases to avoid the path explosion issue, reduce the negative impact, and benefit from our analysis in alleviating overhead and providing effective seeds. Also, we produce test cases that can detect errors leading to crashes. Our tool was evaluated by participating in the international competition in software testing (Test-Comp) for two years, 2021 and 2022. The competition contained approximately 3173 benchmarks in 2021 and 4236 in 2022 from the largest and most diverse open-source repository of software verification tasks. In addition, we provide a detailed analysis of the evaluation's results and a comparison of the state-of-the-art- tools in this field. This contribution shows our effective methods that successfully earn six international awards.

1.4 | Overview of this thesis

This thesis is structured as a journal/alternative format with permission from the Department of Computer Science supervisory team. The thesis core chapters (i.e., chapters 3 to 5) represent research papers published and under review by various international conferences and journals. In addition, a section titled "Thesis context" was inserted at the beginning of each chapter. It connects the chapters and highlights their contributions to the entire work. Furthermore, it provides a straightforward narrative for the thesis and turns it into a storyline. The chapters in this thesis are organized and described as follows:

- **Chapter 2** provides a background in software testing and code coverage research. Also, it presents a brief background of the employed technologies and discusses their effectiveness in code coverage and software testing research. Then, it reviews related works and identifies shortcomings and challenges.

- **Chapter 3** provides an overview of the latest science in software testing. In addition, it presents a summary of the state-of-the-art tools in this field and explains the techniques for these tools. Finally, it describes our approach *FuSeBMC* to find security vulnerabilities in C programs. This chapter explains the techniques used in *FuSeBMC* and how they have been employed and connected to work collaboratively. Furthermore, this chapter shows how our approach has been classified as a low energy consumption tool. The content of this chapter is adapted from: Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. "FuSeBMC: An energy-efficient test generator for finding security vul-

nerabilities in c programs". In: *The International Conference on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-030-79379-1_6. Springer. 2021, pp. 85–105.

- **Chapter 4** discusses the importance of code coverage, its role in developing validation methods, and how the science of code coverage directly relates to detecting vulnerabilities. In addition, it describes the various types of code coverage and their differences. Furthermore, this chapter summarizes the common and utilized techniques in code coverage, as well as the obstacles and challenges these technologies may face. Then, we introduce our new approach *FuSeBMC* v4, which relies on smart seeds and static analysis to achieve high code coverage. Also, it demonstrates how the smart seeds produced by *FuSeBMC* v4 were used to speed up the coverage process and maintain the quality of software testing. This chapter is based on our submitted paper: Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, Fedor Shmarov, Fatimah Aljaafari, and Lucas C Cordeiro. "FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis". In: *The Formal Aspects of Computing Journal (FAC)* (2022).

- **Chapter 5** concludes this thesis by summarising the contributions, results, and awards this research has received and mentioning current and future work.

Chapter 2

Background

> Success on any major scale requires you to accept responsibility... in the final analysis, the one quality that all successful people have... is the ability to take on responsibility.
>
> *Michael Korda*

The purpose of this chapter is to provide definitions and background needed to understand this thesis. In addition, it presents related works to the method and an overview of the combined techniques, including their history, pros, and cons. This thesis primarily focuses on developing novel C program testing techniques built on top of bounded model checking and fuzzing. This chapter is structured as follows: First, we discuss the field of software testing in section 2.1. After that, section 2.2 describes the various techniques used to verify C programs. Then, section 2.4.3 discusses code coverage and its relationship to software testing. Next, section 2.3 explains the bounded model checking technique and its evolution in support of software testing. Afterward, section 2.4 illustrates the fuzzing technique and discusses its types in software testing. Section 2.5 then reviews related work focused on hybrid fuzzer and identifies its current challenges and shortcomings. The chapter concludes with section 2.6, which overviews the combined techniques.

2.1 | Software Testing

The literature on software testing dates back to the early 1970s [57]. Although it is plausible that the concept of testing emerged along with the earliest programming experiences: Hetzel dates the first program testing conference to 1972 [58]. Testing was envisioned as an art and was demonstrated as the "destructive" process of executing a program to detect errors. Dijkstra's most frequently cited adage concerning software testing is that it can only demonstrate the presence of faults, never their absence [59].

Definition 1 (***Software Testing***)**.** Software testing is a process or series of operations meant to ensure that computer code performs as intended and does not perform any unintended actions [60].

There are many different types of testing in software. In this thesis, we will highlight the most common and most widely used software testing strategies, which differ in terms of classification and purpose. For example, correctness testing can be divided into a white box, black box, and gray box testing techniques. This type is used to test the correct behavior of the system. Also, there is performance testing, which is divided into load testing and pressure testing, in which the testing takes place in the form of the life cycle of a process. In addition, reliability testing focuses in its content on the fact that errors are detected before the system is deployed. Finally, security testing is one of the most popular tests because it is useful for the tester to find and fix problems. This type of testing aims to find loopholes and vulnerabilities in the system that could cause significant damage to any system.

Typically, verification consists of a sequence of tests [61]. For many projects, the acceptance criterion is that the product executes specific tasks correctly, and the only way to verify this is to test the product against these tasks. In order to facilitate this operation, test suites incorporate into software projects. The test suites are compilations of independent tests. Importantly, test suites can be automatically used in the codebase. This will enable developers to validate the target code without individual effort during the development cycle. However, writing tests is difficult because they require substantial system knowledge. In addition, test suites must be maintained in tandem with system development to prevent incompatibilities. This alone requires a considerable amount of engineering time [62].

As a result of the testing industry's evolution over decades and the contributions of numerous authors, the majority of testing literature contains confusing and sometimes inconsistent terminology [63]. Therefore, this thesis derives the terminologies from the Institute of Electronics and Electrical Engineers IEEE Computer Society standards [63].

Definition 2 (***Error***)**.** When programmers make mistakes (errors), we refer to them as bugs. There is a tendency for errors to spread where a requirements error may be increased during the design and coding phases.

Definition 3 (***Fault***)**.** A fault is the representation of an error. The representation refers to the manner of expressions. When a designer commits an error, something that should have been included in the representation is missed. A fault occurs when we enter incorrect data into a representation or fail to enter the correct information.

Definition 4 (***Vulnerability***)**.** A vulnerability is a weakness or error in the security design, implementation, or methods of software that can be exploited and lead to a security violation [64]. Software may contain a vulnerability in one or more components [64].

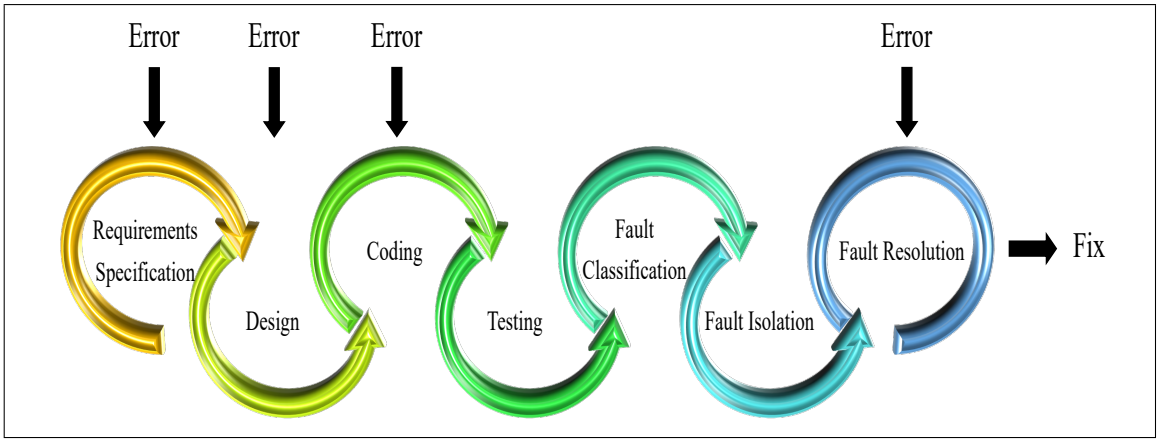


Figure 2.1: A testing life cycle model. The initial three steps are the development phase, the fourth step is the testing phase, and the final three steps are the error-fixing phase [63].

Definition 5 (***Test Case***)**.** A test case is a collection of actions performed on a target software under which a tester will determine whether the software satisfies requirements and functions correctly [63]. Each test case has a unique identifier linked to specific program activity. In addition, a test case also consists of inputs and expected outputs.

Definition 6 (***Test***)**.** Testing is concerned with failures, errors, and faults. A test is a process of utilizing test cases to exercise software. A test serves two essential purposes: detecting errors and validating correct execution.

Definition 7 (***Test Suite***)**.** A test suite, also known as a test set, is a sequence of test cases gathered for test execution. Also, a test suite includes instructions and details regarding the system configuration to be utilized during testing. It is a good practice for organizing test cases so that developers can classify them according to analytical or planning requirements.

Figure 2.1 illustrates a testing life cycle model, which can be divided into three phases. The initial is the development phase, followed by the testing phase, and finally, the debugging phase. This cycle demonstrates that the probability of errors increases in the first phase and may spread to the rest of the other phases. In addition, there is a possibility that an error will arise during the debugging phase if the fix is insufficient or may cause misbehaviour of the correct program earlier. However, a notable tester [63] characterized the life cycle as beginning with Putting Bugs IN, then Finding Bugs, and ending with Getting Bugs OUT. This series of terms demonstrates that test cases play a significant role in testing. The testing process can be divided into three steps: test design, test case development, and run.

Test Cases. The core of software testing is specifying a set of test cases for the target software to be tested. A test case usually has inputs that may be preconditions or the actual inputs identified by a particular testing method. To be effective, each test should include the appropriate preconditions and test case inputs. In addition, it must monitor the outputs

and validate the expected postconditions to determine whether the test was successful. Finally, other information might be helpful to include in the test case to support testing management, such as the execution history of a test case, ID, date, by whom it was run, and the version of software on which it was run. All of this demonstrates that test cases are valuable, if not more valuable than source code. Therefore, test cases must be developed and reviewed continuously to ensure that test coverage is sufficient, potential impacts are identified, and the test data is accurate.

2.2 | Testing Techniques

Techniques for software testing are the strategies, procedures, and templates used to accomplish software testing activities successfully and efficiently [65]. Methods based on metrics for test estimation, black-box or white-box techniques for test design, and static testing techniques are examples of software testing techniques. At the same time, tools for dynamic analysis, coverage analysis, and test design are examples of software testing tools that provide automated or semi-automatic support for software testing techniques and processes. Software testing tools are developed to support software testing techniques [66] and fully or partially automate the processes.

In light of many studies and reviews on many aspects of software testing methodologies and challenges [57, 67, 68, 69, 70], the current software testing practices are far from satisfactory. These studies claim that advanced tools and seamless integration between development and testing [69] are still required [57, 67]. Also, there remain gaps between industry practices and testing research [68]. These deficiencies present opportunities for the enhancement of testing processes [71], testing procedures [72], and supporting tools [73]. Therefore, numerous studies have been conducted to determine which testing techniques and selection methods are employed [74] and which approaches and tools are prevalent [75]. In addition, Automated software testing techniques are observed by Kasurinen et al. [73] and identify factors that influence software testing automation.

2.3 | Bounded Model Checking

The process of applying the ideas behind model checking to software is a complex task. For instance, if we seek to verify properties such as the reachability property, the verification problem becomes undecidable [76] because it is impossible for an algorithm to determine whether or not a specific program state can be achieved and always finished [77]. In addition, the amount of memory that generic programs can potentially require is essentially limitless. Therefore, to implement a model checker, one must accept the possibility that the software will never finish or use an unsound or incomplete approximation.

Bounded Model Checking (BMC) can be used instead of transforming the program

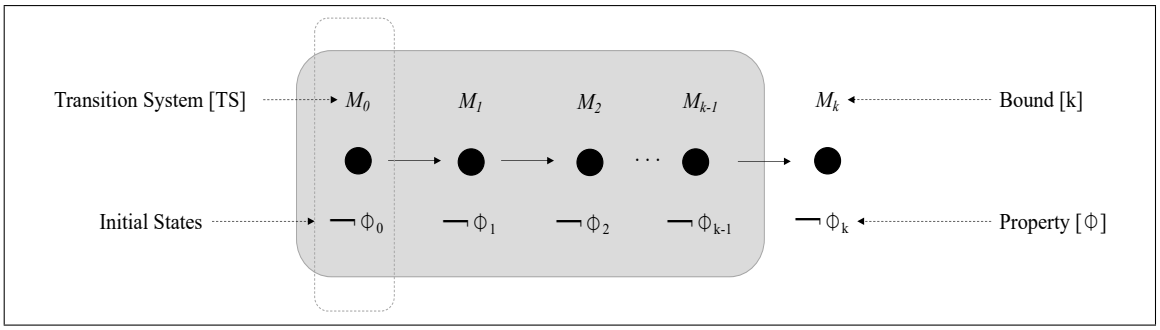


Figure 2.2: An illustration of a transition system when the program is modelled in BMC, where M represents a transition system and ϕ represents a property.

under test (PUT) into a state machine for evaluation, as explained by Biere et al. [78]. BMC is based on Boolean Satisfiability (SAT) [79] or Satisfiability Modulo Theories (SMT) [80]. Initial proposals using BMC based on SAT [79] to verify hardware designs were made in the early 2000s [78, 79]. Studies carried out by a team from Carnegie Mellon University (CMU) demonstrated the success of verifying large digital circuits by BMC, which depends on standard SAT solvers [78]. In contrast, Armando et al. [81] initially proposed BMC based on SMT [80] to address the ever-increasing complexity of software verification. BMC based on SAT or SMT has been successfully applied to detect subtle vulnerabilities in real programs [39, 82, 83, 84]. In BMC, the set of program states is still formalized as an evaluation of all program variables and the program counter's placement because the program statements are considered as transitions from one state to the next [61]. These transitions are viewed as constituting a sequence within the program, where each is transitioning the current state of the program to a new state. Exploration terminates with the completion of k transitions, where k represents the number of bounds. The final state is examined to determine whether it violates a property by finding if the variable evaluation of the final state satisfies the negation of the verification property ϕ. The concept underlying BMC is to examine the negation of certain properties at a specified depth. Figure 2.2 describes how BMC operates.

The program is modelled in BMC as a state transition system (TS) derived from its control-flow graph [85]. Then, it converts to a *Static Single Assignment* form (SSA). Each control graph node will be transformed into an assignment, or a guard will be created from a conditional expression. Each edge indicates a change in the control position of the program [86]. Kripke structure [87] is utilized as TS $M = (S, T, S_0)$ when modeling the program. A Kripke is an abstract machine consisting of a collection of states S, initial states $S_0 \subseteq S$, and transition relation $T \subseteq S \times S$. The collection of states $S = \{s_0, s_1, ...s_n\} : n \in \mathbb{N}$ includes all the states. Each state contains the values of all program variables and a program counter pc. Each transition is represented by $\gamma(si, si + 1) \in T$, where it represents a logical formula encoding all the changes in variables and pc from s_i to s_{i+1}. Then, a *Verification Condition (VC)* denoted by Ψ is computed. The *Verification*

Condition is a quantifier-free formula in a decidable subset of first-order logic. As input, BMC takes three components: transition system M, a property ϕ, and a bound k. Then, BMC unfolds the transition system M k times and translates it into a verification condition Ψ_k, where Ψ_k is satisfiable if and only if ϕ contains a counterexample of depth less than or equal to k. Formally, the bounded model checking procedure may be formulated as follows:

$$\Psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{2.1}$$

The initial states of M are represented by I in the above formula (2.1), and the relation between the two states in M is represented by $\gamma(s_j, s_{j+1})$. ϕ represents the safety properties that must not be compromised. $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents the execution of M with the i length. If some $i \leq k$ satisfies Ψ_k at time-step i, then there exists a state in which ϕ is violated. Then, an SMT solver takes Ψ_k to check for satisfiability. Then, if Ψ_k is satisfiable, the SMT solver will provide an assignment that satisfies it. The counterexample is produced using this assignment's values extracted from the program variables. A counterexample for a property ϕ consists of a sequence of states $\{s_0, s_1, .., s_k\} | s_0 \in S_0$ and $s_i \in S | 0 \leq i < k$ and $\gamma(s_i, s_{i+1})$. If Ψ_k is unsatisfiable, then no error state is reachable in k or fewer steps, indicating that no property was broken.

Two quantifier-free formulas encode the constraints and properties. The first formula (C) serves as the first part of Ψ_k, which is $I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i1} \gamma(s_j, s_{j+1})$. And the second formula (P) the second part of Ψ_k which is $\bigvee_{i=0}^{k} \neg\phi(s_i)$. The SMT solver, after that, examines $C \models_{\mathcal{T}} P$ in the form of $C \wedge \neg P$.

Bounded Model Checking analyzes only bounded program runs. However, it produces verification conditions that specify the precise execution path of a statement, the context in which a particular function is invoked, and the bit-exact representation of expressions [88]. In this context, a verification condition is a logical formula derived from a bounded program and desired correctness properties, the validity of which indicates that the program's behaviour conforms to its specification [76]. Within the context, users can describe correctness attributes using assert statements or code created automatically from a specification language [89]. If all verification conditions of a bounded program are valid, then the program conforms to its specification up to the provided bound. Although BMC was developed about two decades ago, considerable developments in SMT [80] have only recently made it practical and feasible. Due to the current size of source codes and complex software systems, however, the impact of this methodology is still restricted in reality [90].

Furthermore, BMC tools typically fail because of the limitation of memory or time. This limitation is often observed in programs with loops whose bounds cannot be statically identified and validated or whose bounds are too large. Moreover, even if a program does not include a violation up to a specific bound k, we cannot ensure its safety af-

ter bound k (k+1). As a result of these limitations, researchers have been motivated to design new techniques to search deeply into a program's search space while simultaneously demonstrating global correctness [91]. In particular, hybrid techniques have been proposed to mitigate these limitations, which are explained briefly in Section 2.6.

2.4 | Fuzzing

The term "fuzz" was invented in 1990 by Miller et al. [24]. It refers to a program that produces a stream of random inputs for the target program to consume [24]. Numerous contexts, such as penetration testing [92], grammar-based test case generation [93, 94], and dynamic symbolic execution [95, 96], have used the term "fuzz" or "fuzzing."

Fuzzing is an effective and widely-used method for detecting software security vulnerabilities and bugs. The purpose of fuzzing is to identify security bugs, such as buffer overflows and software crashes, by repeatedly executing the software with diverse inputs [97]. Fuzzing is simple to implement compared to other techniques and may be conducted with or without the source code. In addition, fuzzing requires less knowledge of target programs than other testing methods and can be quickly scaled up to accommodate large-scale applications [17].

In the last two decades, fuzzing has become the most effective and efficient state-of-the-art vulnerability detection technique, despite its many disadvantages, such as low efficiency and low code coverage. The positives have outweighed the negatives, and fuzzing has become the industry standard for software development procedures in the commercial sector [43]. For example, Microsoft's Security Development Lifecycle [98] requires fuzzing on all untrusted product interfaces.

2.4.1 | Fuzzing Process

Fuzzing is the process of executing a Program Under Test (PUT) with fuzz inputs. Miller et al. [24] consider a fuzz input to be an unanticipated input that the PUT may receive. The following definition describes the concept of fuzzing.

Definition 8 (***Fuzzing***)**.** Fuzzing is the PUT's execution using inputs sampled from a fuzz input space that extends beyond the expected input space of the PUT.

Definition 9 (***Fuzz Testing***)**.** Fuzz testing is the practice of using fuzzing to determine whether a program under test violates a correctness policy.

Definition 10 (***Fuzzer***)**.** A fuzzer is a program that conducts fuzz testing on a program under test.

Algorithm 1 A Fundamental fuzzing algorithm.

Require: program P , timeout T
1: $corpus \leftarrow initial\_inputs\_(seeds)$
2: $B \leftarrow \emptyset$; `// a collection of detected bugs`
3: $Q \leftarrow \emptyset$; `// a queue for preparing seed to fuzz`
4: **while** ¬isDone(B, Q) **do**
5: $candidate \leftarrow Select(Q, B)$;
6: $mutated \leftarrow Mutate(candidate, B)$;
7: $B \leftarrow Evaluate(mutated)$;
8: **if** $is\_unexpected\_behavior\_\mathbf{OR}\_system\_crash(B, Bs)$ **then**
9: $Q \leftarrow Q \cup mutated$
10: $B \leftarrow Bs \cup B$
11: **end if**
12: **end while**
13: **return** B

2.4.1.1 Fuzzing Algorithm:

We present a fundamental fuzzing approach in Algorithm 1. It is sufficiently generic to understand the basic idea of the fuzzing process and accommodate existing fuzzing methods. Algorithm 1 accepts a target program P and a timeout T as input and returns a collection of detected bugs B. The testing of the target program begins with selecting a corpus of initial inputs (seeds). Then, the fuzzer repeatedly modifies these inputs and evaluates the target program. If the outcome creates unexpected behaviour or a system crash, the fuzzer maintains the input made for future processing. Finally, the fuzzer terminates by achieving a certain objective, such as detecting a bug or exceeding a timeout.

The general process of fuzzing is illustrated in Figure 2.3. Five modules are contained within the fuzzer system: Target program, test case generator, monitor, bug detector, and bug filter. Generally, it starts with the target program and seed files as inputs. The target program can be a program under test or any program being tested. It could be software for a network service, an operating system, application software, or binary code with or without source code. The test case generator then typically mutates a sample and generates inputs for the program being tested. The generated inputs may consist of a specific file type or a network data stream. The generator can use many mutation strategies for initial inputs to improve the efficacy of fuzzing. Then, during the execution of the PUT, the fuzzer monitors the execution state to detect crashes or unusual behaviour. Utilizing techniques such as taint analysis and code instrumentation, the monitor obtains code coverage and useful runtime data of the target program. When the target program crashes, the bug detector reports and analyzes pertinent data to determine if a bug is present. The bug detector is built and implemented within a fuzzer to assist debuggers in discovering potential bugs in a target program. Finally, filtering vulnerabilities is typically conducted manually, making it time-consuming and complex to resolve. Recent tools

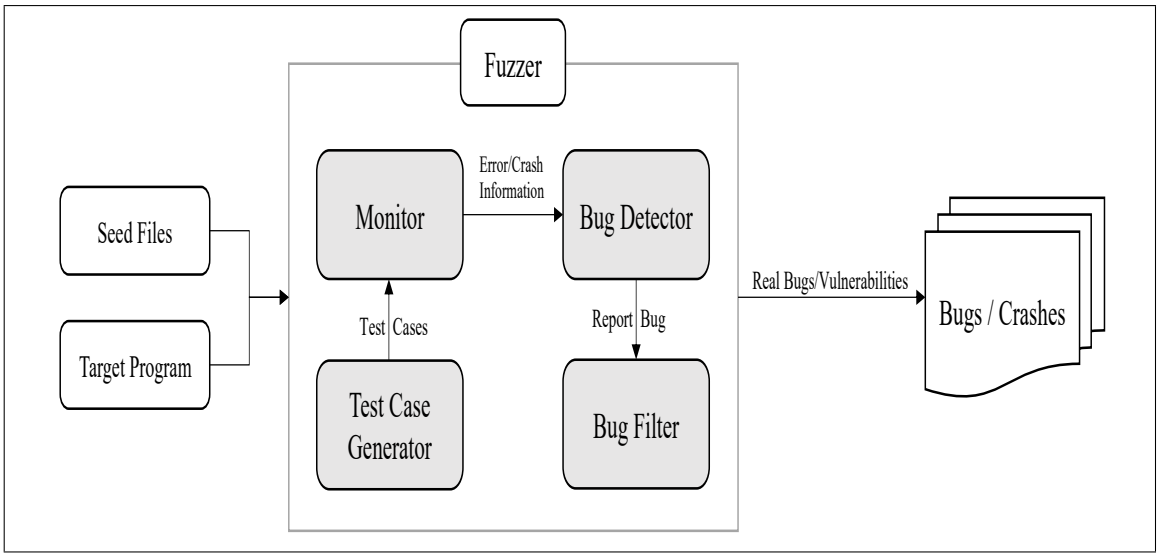


Figure 2.3: The general process of fuzzing. It takes the target program and seeds as inputs and then executes the fuzz processes, outputting a report when the target program crashes.

such as Chen et al. [99], Francis et al. [100], and Zalewski [101] have alleviated the issue. Some relied on sorting the fuzzer's outputs (bug-inducing test cases), while others relied on evaluating the exploitability of a bug [102].

2.4.2 | Types of Fuzzers

Fuzzing techniques can be categorized from several angles. First, they can be classed as black-box, white-box, or grey-box fuzzing according to their understanding of the target program [103]. Also, fuzzing can be mutation-based or generation-based, depending on the sort of data generation. Lastly, it can be categorized by the feedback type (i.e., feedback and no-feedback fuzzing).

2.4.2.1 Mutation-based and generation-based

Fuzzing techniques are straightforward to use. Nevertheless, weaknesses in fuzzing have emerged over time, such as the inability to explore paths outside narrow-ranged input constraints [104] and the production of specific inputs to pass complex paths [44]. These weaknesses prompted developers to seek new techniques to improve fuzzing, such as mutation-based and generation-based approaches, to address some of these weaknesses.

Mutation-based fuzzers: For the mutation-based fuzzers technique, collecting multiple samples of the target program type is necessary. The fuzzer then applies mutations to these samples and sends them to the parser of the target program. One type of mutation is the replacement of data bytes with bytes. This is also possible for several byte ranges, such as two- and four-byte ranges. Thus, mutation-based fuzzers produce test cases by manipulating input data that feeds a target. However, this approach of fuzzers has a few disadvantages. It can take some time to complete fuzzing on a single sample because it is

Table 2.1: Common Black-box, Grey-box, and White-box Fuzzers. The table represents the types of fuzzers and the approach used for each type.

Fuzzer types	Mutation-based	Generation-based
Black-box fuzzers	SAGE [106], Libfuzzer [107]	CLsmith [25], LangFuzz [94] QuickFuzz [108]
Grey-box fuzzers	AFL [56], Driller [44] Vuzzer [110], Mayhem [111]	Syzkaller [109]
White-box fuzzers	Miller [112]	Sulley [113], SPIKE [114] Peach [115]

a very inefficient method [97]. Also, a significant component of functionality will almost always be missed [97]. In general, mutation-based fuzzers are unaware of the required input format or specifications [105].

Generation-based fuzzers: For generation-based fuzzers, it is necessary to conduct a preliminary investigation into the file specifications. Generation-based fuzzers do not need any sample file (initial inputs - seed). Nonetheless, it will rely on user-supplied configuration files to make the process smarter [97]. Typically, these files contain metadata that describes the types of variables and language of the target program. Consider these templates as lists of data structures, relative placements, and potential values. Generally speaking, generation-based fuzzers produce inputs based on a specification. Although generation-based fuzzers are an effective fuzzing technique that capitalizes on the user's knowledge and inventiveness, it is arduous, time-consuming, and error-prone because constructing specifications is so broad. There are so many potential fuzzing rule configurations [43].

2.4.2.2 White-box, black-box and grey-box

Fuzzers could be classified as white-box, grey-box, or black-box based on their reliance on program source code and the depth of their program analysis. Table 2.1 contains a selection of common black-box, grey-box, and white-box fuzzers. Beginning with rudimentary techniques applied in the early days of fuzzing and moving to progressively complicated techniques, the following is an overview of fuzzing techniques. Also, we examine the benefits and drawbacks of each approach.

Black-box fuzzers do not consider the program's internal logic; instead, they continuously feed input data and examine the output results [116]. In terms of understanding the program, black-box fuzzers lie at one extreme. Black-box fuzzers are the most straightforward sort of fuzzing [117]. They modify initial inputs randomly and then test the target program with these modified inputs. The efficacy of black-box random fuzzers is reliant on a good collection of initial inputs to initiate the fuzzing process. Good initial inputs will rapidly exercise more code in the program to be fuzzed. In contrast, initial inputs that are not well-formed may lead to expenditures and inefficient resource use, and they may not produce any benefits [43].

White-box fuzzers were first proposed by Godefroid et al. [118]. They can get comprehensive program information, such as source code, design specifications, and runtime information. Then, they use this information to enhance the efficacy of the fuzzing process. White-box fuzzing may efficiently and exhaustively search the target program through the use of dynamic symbolic execution and a coverage-maximizing guided search technique [119]. Theoretically, white-box fuzzing is capable of producing test cases that cover all program paths. However, the code coverage of white-box fuzzing cannot approach 100 percent in practice because of issues such as the various execution pathways in real software and the imprecision of satisfying a constraint during symbolic execution [119].

Grey-box fuzzers enhance black-box fuzzing with approaches for white-box fuzzing. They closely resemble white-box fuzzing by removing some of its components to reduce expenses and complexity [43]. Code instrumentation is the most frequent technique for grey-box fuzzers [120]. They gather certain information, such as runtime coverage information. Then, they use this information to modify mutation strategies to produce test cases that cover more paths or quickly detect bugs. However, no assurance that using this information would build improved test cases to cover additional paths or trigger bugs [119].

2.4.2.3 Feedback and no-feedback fuzzers

Some fuzzers employ static or dynamic program analysis approaches to improve fuzzing efficiency. It can obtain useful information about the target program and then produce test cases based on this information. This type of work was categorized as Feedback fuzzers, where Feedback refers to the runtime information. It can be used to guide production test cases in the next loop. Path coverage is the foundation of most feedback techniques [116]. In contrast, no information is collected during the program's execution using no-feedback fuzzers.

2.4.3 | Code Coverage

Test cases are critical software project artefacts because they enable developers to validate software and generate software with a low probability of errors. Assigning a quality measure to test cases is one of the challenges in software testing. For example, higher-quality test cases detect more bugs than low-quality test cases. These measures of quality are known as test adequacy criteria. Code coverage is frequently applied in software testing because it indicates which portions of the PUT have been tested. Also, code coverage is the commonly used test adequacy criterion [121]. In general, code coverage is a measure of identifying the parts of the program under test (PUT) that execute when the program is executed and the extent to which a test case covers the PUT [122]. Furthermore, code coverage could assist figure out the portions of code or paths that have been covered and have not. Thus, code coverage information helps focus testing on the uncovered portions. In addition, it is usually utilized as a measure of the effectiveness and sufficiency of a test [123]. Also, it could assist the developers in understanding how much code is covered in quantitative measurement.

Code coverage has been utilized in various contexts, such as software testing, integration testing, and unit testing. Therefore, a code coverage tool is usually integrated into various tools [60]. Code coverage can indicate how much code has been tested, although good code coverage does not necessarily indicate effective testing. For instance, even if the code coverage shows that testing has covered 100% of the code coverage, it may be unable to reveal some or all of the existing vulnerabilities. Therefore, test coverage does not necessarily imply exhaustiveness.

Nevertheless, code coverage provides a quantifiable assessment of a test's thoroughness. Also, code coverage could be helpful to determine whether given test cases do not cover portions of a PUT. In addition, the information on code coverage can generate additional test cases to increase coverage [124].

Coverage measurements can be control- or data-flow oriented. The idea of control flow criteria is to examine the execution flow of test cases in the target code. Statement, branch, and path coverage are examples of common control-flow code coverage metrics. These coverage criteria specify which statements must be executed to achieve full coverage of the target code [125].

Data-flow coverage criteria are based on examining the relationships between values and variables and how these relationships may affect how the program is executed [35]. Data-flow coverage criteria aim to examine how value-assigning and value-utilizing statements interact. Def-use pair coverage is one example of a data-flow coverage criterion. The def-use pair coverage criteria check how well a variable's use has been evaluated in relation to its potential definition points [126].

While control flow-related coverage criteria are frequently employed in practice, data

flow-related coverage criteria are used less often since they are more difficult to automate and more expensive to conduct [127]. Our research concentrated on control flow-related coverage criteria because they were comparable and supported by the many state-of-the-art tools [125].

In general, coverage measures the quantity of testing conducted depending on a particular criterion [128]. A coverage criterion is a rule or group of rules determining the test requirements for a given test set. Typically, these metrics are calculated by dividing the number of items tested by the total number of program items. Methods for code coverage can be classified into three standard criteria:

- **Statement Coverage:** each statement must be executed at least once for statement coverage to be complete. It is the most fundamental coverage criterion in white-box testing. The industry's most prevalent type of coverage is statement coverage [124]. Developers utilize it frequently to evaluate software quality. It is a method for ensuring that each line of source code gets tested at least once. While measuring Statement Coverage, this may appear straightforward, but care must be taken. This is because a particular condition in the source code may not be executed depending on the input values. This would imply that testing would not cover all lines of code. Therefore, employing distinct input values may be necessary to account for all such scenarios in the source code. To calculate statement coverage, we divide the number of executed statements in the PUT by the number of statements in the PUT. Figure 2.4 illustrates a code fragment. To illustrate the calculation of statement coverage for this code, consider the cases where variables $x = 2$ and $y = 3$. The statements on lines 1-5 and 11 are performed in this case. Therefore, the number of statements executed is 6, and the total number of statements in 2.4 is 11. Thus, the statement coverage for this code fragment in Figure 2.4 with inputs $x = 2$ and $y = 3$ is $\frac{6}{11}(100) = 55\%$. However, the "if" part of the code would not be executed if $x = 3$ and $y = 2$. This indicates that Statement Coverage would not be 100% with either set of values. In such a scenario, it may be necessary to execute the tests with all three sets of inputs [($x = 2$, $y = 3$), ($x = 3$, $y = 2$), and ($x = 0$, $y = 0$)] to achieve 100% Statement Coverage of the code.

$$\texttt{Statement Coverage} = \frac{\texttt{(Number of executed statements)}}{\texttt{(Total number of statements)}} \star 100$$

- **Branch coverage:** tests all program branches to ensure that each decision point (e.g., if statements, loops) has been executed in the code at least once during testing. Branch coverage is extensively utilized due to its simplicity of implementation and negligible impact on the execution of the PUT. Figure 2.4 illustrates an example of code for calculating branch coverage. This code contains three branches: two

```
1 Add (int x, int y){
2       if (y > x){
3           y = y - x;
4           printf(``%d'', y);
5       }
6       if (x > y){
7           y = x - y;
8           printf(``%d'', y);
9       }
10      else: printf(``0'');
11 }
```

Figure 2.4: An illustrative code fragment containing an (Add) function that receives two integer arguments.

conditional branches from lines $2-5$ and $6-9$, and one unconditional branch from line 10. If $x = 2$ and $y = 3$, the condition on line 2 will be true and executed. Consequently, the branch coverage is $\frac{1}{3}(100) = 33\%$. Likewise, if $x = 4$ and $y = 2$, then the condition on line 6 will be executed too, and the branch coverage will be 33%. Finally, if $x = 1$ and $y = 1$, the "else" branch in line 9 will be examined and executed. Providing these various values for x and y indicates that we have covered all three branches in the code, achieving 100% (each branch 33% $*3$) branch coverage.

$$\texttt{Branch Coverage} = \frac{\texttt{((Number of executed branches)}}{\texttt{(Total number of branches)}} * 100$$

- **Function coverage:** measures the extent to which PUT's functions are covered during testing [129]. During test execution, all functions found in PUT are examined. It must be ensured that these functions are extensively tested using a variety of input values. Because there may be several functions within PUT's source code, and based on the input values used, a function may or may not be executed. Thus, function coverage aims to ensure that we call every function in PUT. Each function in the PUT must be covered by at least one input to achieve 100% of function coverage. Code coverage is the most specific criterion to measure. Consider figure 2.5, which contains two functions (add and foo). If $a = 10$, then only the function "add" will be called, making the function coverage 50% ($\frac{1}{2}(100)$).

$$\texttt{Function Coverage} = \frac{\texttt{((Number of functions called)}}{\texttt{(Total number of function)}} * 100$$

2.4.4 | Types of Vulnerabilities

Software, in general, is prone to vulnerabilities caused by developer errors, which include: *buffer overflow*, where a running program attempts to write data outside the mem-

```
1 int x;
2 if (x>7)
3 {
4   add(x);
5 else:
6   foo(x);
7 }
```

Figure 2.5: An example code fragment containing (add and foo) functions that receive an integer argument.

ory buffer, which is intended to store this data [130]; *memory leak*, which occurs when programmers create a memory in a heap and forget to delete it [131]; *integer overflows*, when the value of an integer is greater than the integer's maximum size in memory or less than the minimum value of an integer. It usually occurs when converting a signed integer to an unsigned integer and vice-versa [132]. Another example is *string manipulation*, where the string may contain malicious code and is accepted as an input; this is reasonably common in the C programming language [133]. *Denial-of-service attack* (DoS) is a security event that occurs when an attacker prevents legitimate users from accessing specific computer systems, devices, services, or other IT resources [134]. For example, a vulnerability in the Cisco Discovery Protocol (CDP) module of Cisco IOS XE Software Releases 16.6.1 and 16.6.2 could have allowed an unauthenticated, adjacent attacker to cause a memory leak, which could have lead to a DoS condition [135]. Part of our motivation is to mitigate the harm done by these vulnerabilities by the proposed method *FuSeBMC*.

2.5 | Related Work

This section presents various existing research to discuss appropriate tools and methodologies. The purpose of this section is to clarify the context of our research. Most related techniques include fuzzing, bounded model checking, or combining the two.

2.5.1 | Fuzzers

In the 1990s, Barton Miller [136] proposed fuzzing at the University of Wisconsin, and it quickly gained popularity as a technique for identifying software vulnerabilities [97]. American fuzzy lop (AFL) [56, 137] is one of the most popular fuzzing tools. AFL is a coverage-based fuzzer designed to detect software vulnerabilities. AFL employs an evolutionary strategy to learn mutations based on measurements of code coverage. AFL yields high code coverage by employing genetic algorithms and guided fuzzing. Another implemented fuzzing technique is Vuzzer [110], a fuzzer with an application-aware strategy. The primary benefit of this approach is that it does not require prior knowledge of the application or input format. Vuzzer leverages control- and data-flow features derived from

static and dynamic analysis to infer fundamental properties of the application in order to increase coverage and investigate deeper paths. This allows significantly faster production of interesting inputs compared to a technique that is application-agnostic. Furthermore, similar to AFL, Syzkaller [109] utilizes coverage-guided methodologies to conduct fuzzing testing on the target program. Also, LibFuzzer [107] uses code coverage information produced by LLVM's (SanitizerCoverage) instrumentation to generate test cases. LibFuzzer is ideally suited for testing libraries with tiny input and a runtime of milliseconds per input to ensure that library code does not crash due to invalid input[1]. Wang et al. [138] proposed a method employing data-driven seed generation (Skyfire). It processes and generates well-distributed seed inputs for fuzzing algorithms by extracting the knowledge of grammar. However, Skyfire is designed to identify syntax features and semantic rules using probabilistic context-sensitive grammar (PCSG). AFLFast [139] is an upgraded version of AFL that exercises a low-frequency path using many techniques. The tool was found to be seven times faster than AFL [139].

Smart grey-box fuzzing (SGF) [140] is a fuzzer that generates high-impact seeds by employing a high-level structural representation of the original seeds. Likewise, AFLSmart [140] is a structure-aware fuzzing that combines the AFL fuzzer and the PEACH fuzzer engine. GTFuzz [141] is a tool for prioritizing inputs based on the extraction of syntax tokens that guard the target location. These tokens are extracted using the technique of backward static analysis. Additionally, this extraction helps GTFuzz enhance its mutation algorithm. Instrim [142] is a control flow graph CFG-aware fuzzer. It analyzes software to maintain the fuzzing speed in specific blocks selected according to the Control Flow Graph CFG. As a consequence, Instrim enhanced speed by a maximum of 1.75 times. Moreover, there is Peach [115]. This tool's advanced and reliable fuzzing framework is one of its primary features. This framework can generate an XML file for defining a data model and state model. In addition, numerous fuzzers have been produced, and each with its unique improvements. For example, DGF [137] searches for directed paths, and SYMFUZZ [143] controls the selection of paths, while Alexandre Rebert's method [144] employs guided seed selection. AutoFuzz [145] is a verification approach for network protocols that employs fuzzing. It begins with locating the protocol's specifications, then uses fuzzing to detect vulnerabilities.

The common weakness of pure fuzzing approaches is their inability to provide test cases that explore program code that occurs beyond complicated guards. In addition, because fuzzers essentially work by randomly mutating seeds, they struggle to find inputs required to satisfy complicated guards.

[1] `https://llvm.org/docs/LibFuzzer.html`

2.5.2 | BMC and Symbolic execution

In recent years, bounded model checking has been successfully applied to verifying C programs. There are various state-of-the-art bounded model checkers. CBMC [82] is likely one of the most popular bounded model checkers currently available. It is a bounded model checker based on SAT [79] for sequential and concurrent programs [146, 147]. Another approach is ESBMC [148] which is derived from CBMC. It employs SMT [80] solvers instead of SAT solvers to validate C programs. Also, LLBMC is a sequential bounded model checker for C and C++ programs. Unlike ESBMC, LLBMC [83] performs model checking on LLVM bytecode rather than the clang-generated abstract syntax tree (AST). Although this method simplifies the verification and reduces costs, it may also create flaws, such as losing context information (e.g., mangled class and function names and source location information) [149].

Symbolic execution is a popular technique for detecting errors in software [150]. It has demonstrated proficiency in generating test cases with high coverage and detecting bugs in complicated software. In contrast to BMC, which depends on its concept of encoding execution paths up to a specific length, symbolic execution explores each path symbolically and independently. SAT or SMT solvers are applied to the constraints determined for each path. KLEE [151] is one of the most used symbolic execution engines. It is a tool that exploits the LLVM compiler infrastructure and dynamic symbolic execution to explore the search space path-by-path. KLEE has been used in numerous specialized tools as a reliable symbolic execution engine. Additionally, BAP [152] is constructed on top of Vine [153], which relies on symbolic execution. BAP contains valuable analysis and verification methods. In its analysis, BAP relies on an intermediate language (IL). A Tracer [154] is a tool for verification that employs constraint logic programming (CLP) and interpolation techniques. DART [155] is another approach based on symbolic execution. It performs software analysis and employs random automated testing to identify software bugs. Avgerinos, Thanassis, et al. [150] proposed a method for enhancing symbolic execution with verification-based algorithms. It works to enhance the efficacy of dynamic symbolic execution. The method demonstrated its capacity to detect errors and obtain higher code coverage than existing methods for dynamic symbolic execution. Also, SymbexNet [156] and SymNet [157] are used to validate the implementation of network protocols. CoVeriTest [158] is a Cooperative Verifier Test generation that uses a hybrid method for test generation. It applies several conditional model checkers in iterations with numerous configurations for value analysis. CoVeriTest modifies the level of cooperation and assigns each verifier a time budget. However, the path explosion problem associated with loops and arrays makes BMC and symbolic execution impractical.

2.5.3 | Combination

Relatively recently, the combination of symbolic execution and BMC with fuzzing has been employed to leverage the power of both techniques. VeriFuzz [159] is among the most advanced tools that combine BMC with fuzzing. It is a program-aware fuzz tester that combines feedback-driven evolutionary fuzz testing with static analysis. In addition, it leverages grey-box fuzzing to exploit lightweight instrumentation for monitoring test-run behaviour. VeriFuzz won the top prize at Test-Comp 2020 [160]. FuSeBMC and VeriFuzz are similar in their general concept because they combine the BMC and fuzzing techniques. However, FuSeBMC differs in many aspects. Firstly, we adopted the ESBMC tool as a BMC engine and modified fuzzer (based on AFL) and the selective fuzzer as our fuzzing engines. Secondly, we apply Clang to analyze the target code, unlike VeriFuzz, which relies on a C and C++ parser. We also employ SMT-based encodings in FuSeBMC, as opposed to SAT-based encodings in VeriFuzz. Furthermore, in terms of seeds, our tool produces seeds during the detecting process, unlike VeriFuzz, which relies on pre-seeds. Moreover, instrumentation was developed in FuSeBMC to analyze the code, while VeriFuzz relied on IR. Finally, in our approach, we have developed a system called Tracer to coordinate the operations between the engines other than VeriFuzz, which employs the BMC as an initial step only and relies on fuzzing heavily.

Stephens et al. developed Driller [44], a hybrid vulnerability excavation tool. It detects deeply embedded bugs by combining guided fuzzing and concolic execution. It utilizes concolic execution to analyze the program and trace the inputs. Also, the concolic execution guides fuzzing along various paths by utilizing its constraint-solving engine. Stephens et al. combined the strengths of the two techniques and reduced their limitations by avoiding path explosion in concolic analysis and incompletion in fuzzing. First, Driller splits the program based on tests for specific input values. Then, applying the proficiency of fuzzing, Driller explores potential input values in a *compartment*. Although Driller has demonstrated its effectiveness in detecting more bugs, it may lead to the path explosion problem because it requires significant computational resources. MaxAFL [161] is a gradient-based fuzzer that is built on top of AFL. Initially, the developers determine the Maximum Expectation of Instruction Count (MEIC) using a lightweight static analysis. Then, they produce an objective function using MEIC. After that, a gradient-based optimization algorithm is used to generate efficient inputs by minimizing the objective function. Hybrid Fuzz Testing [162] is a tool that easily generates provably random test cases, ensuring the execution of unique paths. In addition, it identifies unique execution paths by utilizing symbolic execution to identify the border nodes that lead to such paths. Also, the tool gathers all possible border nodes based on resource restrictions to utilize fuzzing with provably random input, preconditioned to lead to each border node.

Badger [163] presents a hybrid testing approach for complexity analysis. It produces

new input using Symbolic PathFinder [164] and provides the Kelinci fuzzer with worst-case analysis. Badger utilizes fuzz testing to generate a diverse set of inputs to increase coverage and the resource-related cost associated with each path. He et al. [165] developed a method for learning a fuzzer via symbolic execution. It begins by phrasing the learning task within the context of imitation learning. Then, it leverages symbolic execution to provide high-quality inputs with high coverage, while a neural network-based fuzzer learns to fuzz new programs. LibKluzzer [166] is a novel implementation that combines symbolic execution and fuzzing. Its strength is derived from the combination of coverage-guided fuzzing and white-box fuzzing. LibKluzzer is constructed of LibFuzzer, and an extension of KLEE called KLUZZER [167]. Munch [129] is a hybrid framework tool. It utilizes fuzzing with seed inputs produced by symbolic execution and focuses on symbolic execution when fuzzing becomes saturated. It aims to decrease the number of queries sent to the SMT solver to concentrate on the paths that may lead to uncovered functions. The developers designed Munch to increase function coverage. SAGE (Scalable Automated Guided Execution) is a hybrid fuzzer developed by Godefroid et al. [106]. Microsoft makes considerable use of SAGE, which has successfully detected security-related bugs. It utilizes generational search to expand dynamic symbolic execution and improve code coverage by negating and solving path predicates. In addition, SAGE relies on DART's random test methodology to mutate good inputs using grammar. FairFuzz [168] is a grey-box fuzzer employing guided mutation. It employs a mutation mask for each pair of seeds and rare branches to guide the fuzzing to each uncommon branch using coverage. SAFL [169] is an effective fuzzer for C/C++ programs. It uses symbolic execution in a lightweight approach to producing initial seeds that can be used to determine the proper fuzzing direction.

The combination of fuzzing and symbolic execution for software verification has proven to be the most effective. Our approach uses fuzzing and BMC in tandem to leverage their strengths and overcome their weaknesses. It expands on this combination by utilizing our main novelties, which we explained in chapters 3 and 4. The tracer module and smart seed generation distinguish our approach (*FuSeBMC*) from other approaches and tools. In addition, by utilizing shared memory and analyzing the graph goal in order to select an effective strategy, *FuSeBMC* was able to win six international awards.

2.5.4 | Existing Solutions & their Limitations

Numerous kinds of research have been conducted to find solutions to software problems based on C language because of its popularity in many common systems in governments and large companies. Table X reviews the most popular state-of-the-art tools that work on C programs, where the most efficient techniques and tools in each technique are divided on the table. Throughout our research, we focused on employing requirements relating

Approach \ Requirements	R1	R2	R3	R4
Fuzzing				
SPIKE [137]	Y	Y	N	N
AutoFuzz [153]	*	Y	N	*
AFL [66]	Y	*	N	*
VeriFuzz [167]	Y	Y	*	*
Symbolic execution				
SYMBEXNET [164]	Y	Y	Y	*
Klee [159]	Y	*	N	Y
LibKluzzer [174]	Y	*	*	N
SymNet [165]	Y	*	Y	N
Model checking				
ESBMC [156]	Y	N	N	*
CBMC [105]	*	N	N	*

Symbol	Description
Y	Requirement met
N	Requirement not met
*	Partial functionality

- R1. Capacity to detect vulnerabilities.
- R2. Ability to achieve high code coverage.
- R3. Avoiding the path explosion
- R4. How fast to detect vulnerabilities

Figure 2.6: Limitations of Existing Related Solutions.

to the fundamental elements and highlighting the limitations of most methodologies. Our requirements revolve around the following: 1) Capacity to detect vulnerabilities. 2) Ability to achieve high code coverage. 3) Avoiding the path explosion. 4) How fast to detect vulnerabilities. The table evaluates whether these approaches can or cannot satisfy those requirements. The table also identifies the solutions and their constraints.

2.6 | Overview of hybrid fuzzing

Since 2007, numerous researchers have incrementally improved hybrid fuzzing. Table 2.2 outlines the stages of development and the essential tools for combining the two techniques. In 2007, Miller B.P. et al. [112] introduced the notion of hybrid testing. The author proposes a mixed test method by combining the two techniques. The experiment demonstrates that the effect of the mixed test is significantly superior to that of the single instance. However, the author did not clarify if the techniques could compensate for each other's weaknesses. Even though the author did not explicitly propose the hybrid fuzzing technique in this study, the random testing notion employed is consistent with the fuzzing. Therefore this research has become the conceptual prototype for numerous hybrid fuzzing techniques that have followed [170]. Then, TaintScope [171], a checksum-aware directed fuzzing solution, was introduced in 2010. Combining concrete and symbolic execution approaches, TaintScope can automatically correct the checksum fields of malformed test cases. In 2012 Pak B S. et al. [162] introduced the idea of hybrid fuzzing. They showed the advantages of the hybrid fuzz testing technique for improving code coverage with reasonably low overhead. After that, Sword [172], an automatic fuzzing solution for software vulnerability identification, was proposed in 2014. It utilizes a combination of fuzzing,

symbolic execution, and taint analysis techniques to address the various issues. Sword can produce test cases with the highest likelihood of triggering deep-seated program vulnerabilities by the guidance of symbolic execution and taint analysis. In 2015, Binary-oriented hybrid fuzz testing [173] was developed, which combines the benefits of fuzz testing and symbolic execution. The concept behind this strategy is that when fuzz testing cannot raise the code coverage ratio, symbolic execution will begin to operate and create new program input. The introduction of AFL [56] introduced innovative concepts to the corporate and academic communities. It is a significant step forward in developing fuzzing and a turning point for the hybrid fuzzing technique [170], which can bring design ideas. Therefore, in 2016, N.J. et al. presented Driller [44], a novel hybrid fuzzing technique. Driller combines the most efficient fuzzing and symbolic execution techniques using the fork server concept. In 2018, hybrid fuzzing expanded, attracting domestic and international researchers. This year, QSYM [104] proposed by Yun is a comprehensive analysis of the benefits and drawbacks of hybrid fuzzing. It demonstrated remarkable results. In 2019, Xie Xiaofei et al. proposed Afleer [174]. It integrates AFL with KLEE [151] via a branch coverage methodology, produces many test cases with AFL, and then searches for the coverage knowledge gained from AFL using Klee. In 2020, hybrid fuzzing technology emerged as one of the most significant coverage-oriented fuzzing branches [170]. By adjusting the constraint-solving portion of symbolic execution, the Pangolin [175] approach described by Rongxin Wu et al. enhances the effectiveness of vulnerability mining.

The developments of hybrid fuzzing recently are due to the advancement with symbolic execution and the advancement with fuzzing. For example, KLEE has dramatically improved the efficiency of symbolic execution, making it possible to combine symbolic execution and fuzzing. On the other hand, tools such as AFL and Libfuzzer [107] have made coverage-oriented fuzzing the standard, allowing hybrid fuzzing to introduce new concepts. In the years that followed, the combination of constraint solutions and coverage became widespread [170]. There have been numerous excellent academic and commercial tools for hybrid fuzzing to date.

In this section, we compare the performance and coverage of each software testing methodology. Figure 2.7 shows a comparison of Code coverage for each software testing technique. Fuzzing tools are fast, enabling them to explore PUT branches more simply and deeply. However, the fuzzer frequently generates test inputs that cannot explore new paths. This causes the fuzzer to become time-consuming and less effective. On the one hand, symbolic execution can cover and analyze all fundamental program blocks. On the other hand, symbolic execution may not be theoretically scalable due to the vast number of paths in the target program. Also, the path explosion problem poses a substantial obstacle for symbolic execution users, which may impede the coverage process. In contrast, the hybrid fuzzer can cover all branches effectively, as shown in Figure 2.7c. The fundamental concept underlying hybrid fuzzers is integrating both techniques to alleviate each other's

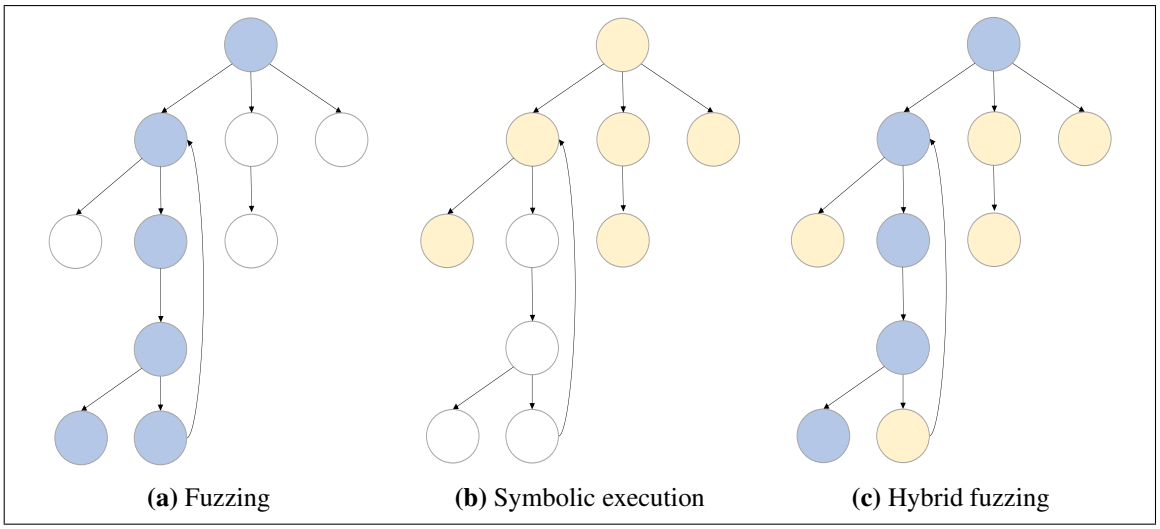
Table 2.2: A history of research on hybrid fuzzers.

	Most proposals/tools for hybrid fuzzers
2007	Miller B.P. et al. [112] introduced the notion of hybrid testing
2010	TaintScope [171], a checksum-aware directed fuzzing solution
2012	Pak B S. et al. [162] introduced the idea of hybrid fuzzing for the first time
2014	Sword [172], an automatic fuzzing solution for software vulnerability identification
2015	Binary-oriented hybrid fuzz testing [173]
2016	Driller [44], a novel hybrid fuzzing technique
2018	QSYM [104] is a comprehensive analysis of the benefits and drawbacks of hybrid fuzzing.
2019	Xie Xiaofei et al. proposed Afleer [174]
2020	Pangolin [175] approach enhances the effectiveness of vulnerability mining.

limitations and obtain high code coverage.

Table 2.3 provides a comprehensive comparison of software testing techniques. In addition, the purpose and objective of the combination, as well as the most popular tools utilized, are described. Furthermore, it illustrates the pros and cons of each technique and its usability, scalability, and accuracy. The table demonstrates that hybrid fuzzing approaches are superior to other methods in several aspects. Moreover, numerous software testing competitions are being dominated by hybrid fuzzing tools, demonstrating the effectiveness of this technique.

The effectiveness of each technique's execution rate and code coverage is compared in Figure 2.8. Although symbolic execution has a high code coverage, its execution rate speed is one of its major limitations. In addition, the fuzzing approach is fast in finding bugs, but they have low code coverage results. The result of a low code coverage renders it ineffective for finding all program bugs. The hybrid fuzzer approach is the most effective due to its speed and capacity to test and cover large systems, even though it has a lower code coverage than symbolic execution. Therefore, we present our hybrid fuzzer FuSeBMC that has been used recently as a comparison tool in many research papers [45, 176]. FuSeBMC differs from many hybrid fuzzer in that it is based on three engines (modified AFL, BMC, and selective fuzzer) that are co-operated by the subsystem tracer. Moreover, we provided lightweight instrumentation on AFL to help prevent the fuzzing process from getting stuck. Also, in our fuzzing, we have been working on



(a) Fuzzing **(b)** Symbolic execution **(c)** Hybrid fuzzing

Figure 2.7: Code coverage comparison for each software testing technique. Circles represent the paths in the target program and their depth, while the colours indicate the ability of each technology to cover the paths.

what we call life push; when we see that the fuzzing process is stuck in a certain process, we push the bits to go to the following process, preventing the fuzzer from wasting time or becoming stuck in one. In terms of BMC, our BMC here is characterized by providing instrumentation and labels, which helps guide and run the BMC engine to reduce resource consumption. Finally, we apply static analysis to mitigate the downsides when integrating the technologies. As a result, FuSeBMC showed its efficiency compared to our closest competitors, Klee, Symbiotic, and VeriFuzz.

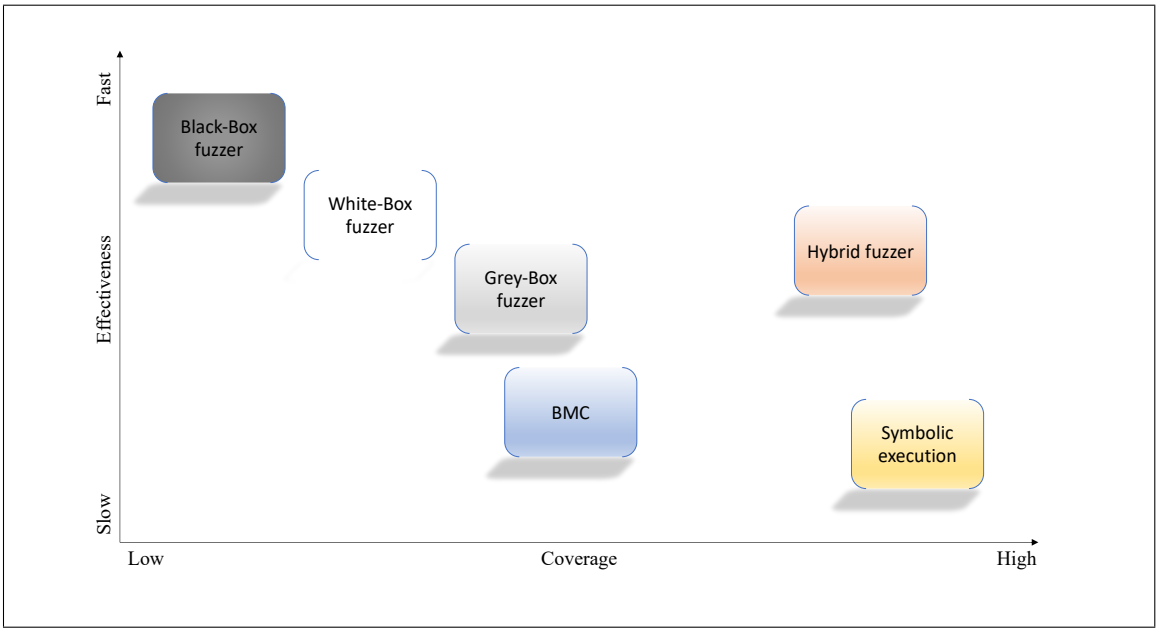


Figure 2.8: Techniques comparisons in code coverage and execution process. The x-axis shows the capacity of coverage achieved, while the y-axis shows the effectiveness.

Table 2.3: Comparing the performance of techniques

	Fuzzing	**BMC**	**Symbolic execution**	**Hybrid**
Purpose of integrate with fuzzing	-	Seed generation	Path exploration	Path exploration - Guiding
Example Tool	AFL [56] Libfuzzer [107]	CBMC [82] ESBMC [148]	KLEE [151] DART [155]	Driller [44] QSYM [104]
Pros	- Simple to design - Perform automated tests	- Largely automatic and fast - Better for certain problems	- Fast/Accurate results - High coverage	- Bypass quickly conditions - Detect more bugs
Cons	- Struggle at complex conditions - Miss program paths	- Suffer from large loops - Path explosion/Slow	- Path-explosion/Expensive resources - Slow with large constraint systems	- Inefficient in large-scale software - Expensive in coverage
Objective	Testing / Coverage	Testing / Verification	- Testing / Coverage - Verification	- Testing / Coverage - Verification
Accuracy	Moderate	Moderate	High	Highest
Scalability	Good	Good	Bad	Best
How complex is it to use?	Easy	Hard	Hard	Easy

2.7 | Summary

This chapter has outlined the background of software testing, code coverage, and techniques used in this thesis. Also, it provides research relevant to the method and an overview of the combined techniques, including their history, pros, and cons. In Section 2.1, we began by providing a historical overview of software testing, its origins, and its significance in the industry. Then, we defined numerous concepts, including errors, test cases, and test suites, necessary to understand a technique for detecting software bugs. In addition, we provided a summary of the testing life cycle model to illustrate test processes and potential mistake locations. Then, we described the components of test cases and their function in software testing. Next, in Section 2.4.3, we discussed code coverage and its function in finding vulnerabilities and ensuring system correctness. Moreover, we explored various viewpoints regarding the relationship between code coverage and bug detection. After that, we described the various code coverage types and presented an example for each.

In section 2.2, we provided an overview of the testing techniques and how software testing practices are far from satisfactory [67, 68, 69, 70]. After that, we presented the BMC technique, which consists of unwinding the design and the correctness property k times. Then, we explained its concept and operation in detail. In Section 2.4, we discussed the fuzzing approach, its concept, and its algorithm. Also, we described types of fuzzing, including their advantages and disadvantages, and reasons for classifying each type.

In Section 2.5, we discussed and classified related work based on the employed techniques. Also, we briefly discussed state-of-the-art tools for each technique. In addition, we outlined the disadvantages and challenges of these tools and approaches.

Finally, we surveyed hybrid fuzzing tools in Section 2.6. First, this survey examined the development of this type of technology across time. Then, we demonstrated the advantages of hybrid tools and how they leverage strengths and overcome weaknesses in each technique. Following that, we created a comparison between the hybrid techniques and other software testing techniques, comparing them in numerous areas, such as accuracy, usability, and scalability. As a result, the survey showed the advantages of hybrid testing approaches for detecting software errors and achieved high coverage.

Chapter 3

FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs

> A man who dares to waste one hour of time has not discovered the value of life.
>
> *Charles Darwin*

3.0 | Chapter Overview

3.0.1 | Thesis Context

The previous chapter provided an overview of the software testing field and introduced its basic terminology. The section then illuminated the primary obstacles associated with the employed techniques. In addition, it clarified related works and their challenges and shortcomings. Lastly, a survey was conducted on hybrid techniques, their impact on software testing, and how they have become attractive. In this chapter, we explore hybrid techniques that combine both techniques' strengths to optimize the vulnerability detection process. In addition, we aim to reduce the effort and consumption caused by hybrid techniques. We then conducted a literature review on theories and state-of-the-art tools. Next, we discuss our hybrid approach based on fuzzing and symbolic execution via bounded model checking. Also, we provide our selective fuzzer that learns from the test cases produced by fuzzing/BMC to produce new test cases for the uncovered goals. Finally, we present a novel algorithm for managing the time allocated to fuzzing and BMC to improve *FuSeBMC*'s energy consumption.

FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs

Kaled M. Alshmrany[1,2(✉)], Mohannad Aldughaim[1], Ahmed Bhayat[1], and Lucas C. Cordeiro[1]

[1] University of Manchester, Manchester, UK
kaled.alshmrany@postgrad.manchester.ac.uk
[2] Institute of Public Administration, Jeddah, Saudi Arabia

The content of this chapter is adapted from: Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. "FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in c programs". In: *The International Conference on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-030-79379-1_6. Springer. 2021, pp. 85–105.

3.0.2 | Author's Contributions

I designed the main idea of the research paper, developed and validated the approach, conducted experiments and evaluation, analysed results, investigated related work, provided and edited all graphics, participated in the entire writing process, and addressed the reviewer's comments. In addition, Muhannad and Ahmed provided critical feedback, proofread the paper, and made revisions and amendments during the writing process. My supervisor, Lucas, also contributed to the idea and proofread the paper. Moreover, he guided the entire research procedure.

3.0.3 | Abstract

We describe and evaluate a novel approach to automated test generation that exploits fuzzing and Bounded Model Checking (BMC) engines to detect security vulnerabilities in C programs. We implement this approach in a new tool *FuSeBMC* that explores and analyzes the target C program by injecting labels that guide the engines to produce test cases. *FuSeBMC* also exploits a selective fuzzer to produce test cases for the labels that fuzzing and BMC engines could not produce test cases. Lastly, we manage each engine's execution time to improve *FuSeBMC*'s energy consumption. We evaluate *FuSeBMC* by analysing the results of its participation in Test-Comp 2021 whose two main categories evaluate a tool's ability to provide *code coverage* and *bug detection*. The competition results show that *FuSeBMC* performs well compared to the state-of-the-art software testing

tools. *FuSeBMC* achieved 3 awards in the Test-Comp 2021: first place in the *Cover-Error* category, second place in the *Overall* category, and third place in the *Low Energy Consumption* category.

keywords Automated Test Generation and Bounded Model Checking and Fuzzing and Security.

3.1 | Introduction

Developing software that is secure and bug-free is an extraordinarily challenging task. Due to the devastating effects vulnerabilities may have, financially or on an individual's well-being, software verification is a necessity [177]. For example, Airbus found a software vulnerability in the A400M aircraft that caused a crash in 2015. This vulnerability created a fault in the control units for the engines, which caused them to power off shortly after taking-off [178]. A software vulnerability is best described as a defect or weakness in software design [179]. That design can be verified by Model Checking [180] or Fuzzing [43]. Model-checking and fuzzing are two techniques that are well suited to find bugs. In particular, model-checking has proven to be one of the most successful techniques based on its use in research and industry [181]. This paper will focus on fuzzing and bounded model checking (BMC) techniques for code coverage and vulnerability detection. Code coverage has proven to be a challenge due to the state space problem, where the search space to be explored becomes extremely large [181]. For example, vulnerabilities are hard to detect in network protocols because the state-space of sophisticated protocol software is too large to be explored [182]. Vulnerability detection is another challenge that we have to take besides the code coverage. Some vulnerabilities cannot be detected without going deep into the software implementation. Many reasons motivate us to verify software for coverage and to detect security vulnerabilities formally. Therefore, these problems have attracted many researchers' attention to developing automated tools.

Researchers have been advancing the state-of-the-art to detect software vulnerabilities, as observed in the recent edition of the International Competition on Software Testing (Test-Comp 2021) [183]. Test-Comp is a competition that aims to reflect the state-of-the-art in software testing to the community and establish a set of benchmarks for software testing. Test-Comp 2021 [183], had two categories *Error Coverage* (or *Cover-Error*) and *Branch Coverage* (or *Cover-Branches*). The *Error Coverage* category tests the tool's ability to discover bugs where every C program in the benchmarks contains a bug. The aim of the *Branch Coverage* category is to cover as many program branches as possible. Test-Comp 2021 works as follows: each tool task is a pair of an input program (a program under test) and a test specification. The tool then should generate a test suite according to the test specification. A test suite is a sequence of test cases, given as a directory of

files according to the format for exchangeable test-suites[1]. The specification for testing a program is given to the test generator as an input file (either coverage-error-call.prp or coverage branches.prp for Test-Comp 2021) [183].

Techniques such as fuzzing [112], symbolic execution [184], static code analysis [185], and taint tracking [186] are the most common techniques, which were employed in Test-Comp 2021 to cover branches and detect security vulnerabilities [183]. Fuzzing is generally unable to create various inputs that exercise all paths in the software execution. Symbolic execution might also not achieve high path coverage because of the dependence on Satisfiability Modulo Theories (SMT) solvers and the path-explosion problem. Consequently, fuzzing and symbolic execution by themselves often cannot reach deep software states. In particular, the deep states' vulnerabilities cannot be identified and detected by these techniques in isolation [187]. Therefore, a hybrid technique involving fuzzing and symbolic execution might achieve better code coverage than fuzzing or symbolic execution alone. VeriFuzz [159] and LibKluzzer [166] are the most prominent tools that combine these techniques. VeriFuzz combines the power of feedback-driven evolutionary fuzz testing with static analysis, where LibKluzzer combines the strengths of coverage-guided fuzzing and dynamic symbolic execution.

This paper proposes a novel method for detecting security vulnerabilities in C programs that combines fuzzing with symbolic execution via bounded model checking. We make use of coverage-guided fuzzing to produce random inputs to locate security vulnerabilities in C programs. Separately, we make use of BMC techniques [188, 189]. BMC unfolds a program up to depth k by evaluating (conditional) branch sides and merging states after that branch. It builds one logical formula expressed in a fragment of first-order theories and checks the satisfiability of the resulting formula using SMT solvers. These two methods are combined in our tool *FuSeBMC* which can consequently handle the two main features in software testing: *bug detection* and *code coverage*, as defined by Beyer et al. [190]. We also manage each engine's execution time to improve *FuSeBMC*'s efficiency in terms of verification time. Therefore, we raise the chance of bug detection due to its ability to cover different blocks of the C program, which other tools could not reach, e.g., KLEE [151], CPAchecker [191], VeriFuzz [159], and LibKluzzer [166].

Contributions. This paper extends our prior work [8] by making the following original contributions.

- We detail how *FuSeBMC* guides fuzzing and BMC engines to produce test cases that can detect security vulnerabilities and achieve high code coverage while massively reducing the consumption of both CPU and memory. Furthermore, we discuss using a custom fuzzer we refer to as a *selective fuzzer* as a third engine that learns from the test cases produced by fuzzing/BMC to produce new test cases for the uncovered goals.

[1] `https://gitlab.com/sosy-lab/software/test-format/`

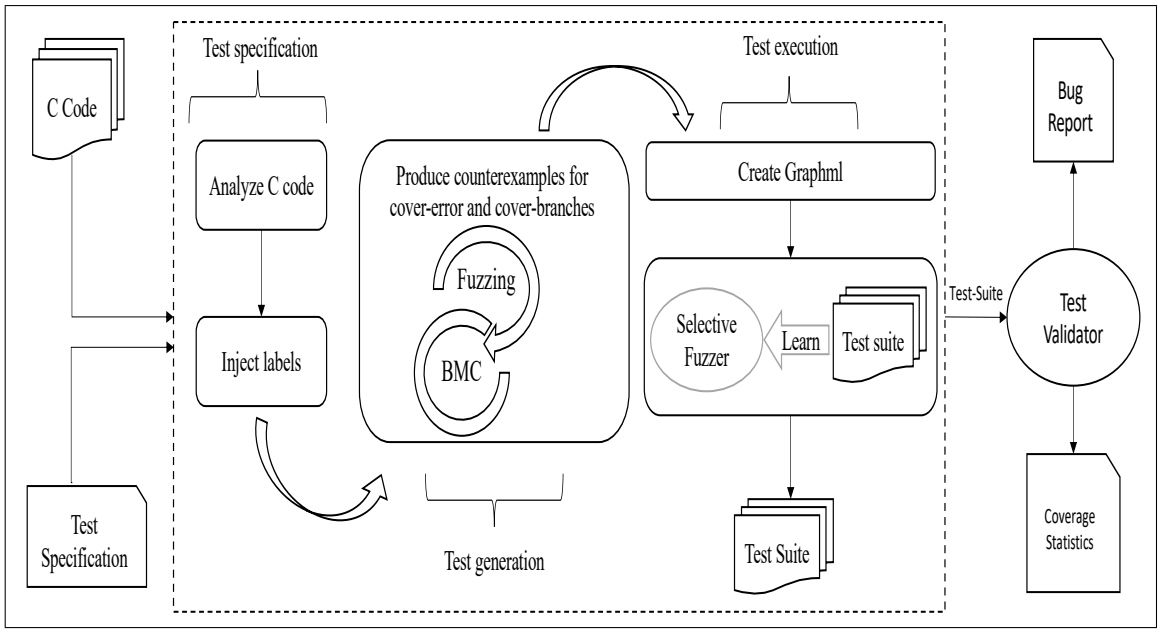


Figure 3.1: *FuSeBMC*: An Energy-Efficient Test Generator Framework.

- We provide a detailed analysis of the results from *FuSeBMC*'s successful participation in Test-Comp 2021.*FuSeBMC* achieved first place in *Cover-Error* category and second place in *Overall* category. *FuSeBMC* achieved first place in the subcategories *ReachSafety-BitVectors, ReachSafety-Floats, ReachSafety-Recursive, ReachSafety-Sequentialized and ReachSafety-XCSP*. We analyse the results in depth and explain how our research has enabled *FuSeBMC*'s success across these categories as well its low energy consumption.

3.2 | *FuSeBMC*: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs

We propose a novel verification method named *FuSeBMC* (cf. Fig. 3.1) for detecting security vulnerabilities in C programs using fuzzing and BMC techniques. *FuSeBMC* builds on top of the Clang compiler [51] to instrument the C program, uses Map2check [192, 193] as a fuzzing engine, and ESBMC (Efficient SMT-based Bounded Model Checker) [90, 194] as BMC and symbolic execution engines, thus combining dynamic and static verification techniques. We have employed Map2check as our fuzzing engine for several reasons. First, it relies on a fuzzer based on libFuzzer. The fuzzer in this place is quick and lightweight when providing inputs for programs with large arrays, relieving much of the load on our main engine BMC. In addition, the joint collaboration between developers in *FuSeBMC* and Map2check is a good motivation for integrating Map2check easily into our approach.

The method proceeds as follows. First, *FuSeBMC* takes a C program and a test specification as input. Then, *FuSeBMC* invokes the fuzzing and BMC engines sequentially to

find an execution path that violates a given property. It uses an iterative BMC approach that incrementally unwinds the program until it finds a property violation or exhausts time or memory limits. In code coverage mode, *FuSeBMC* explores and analyzes the target C program using the clang compiler to inject labels incrementally. *FuSeBMC* traverses every branch of the Clang AST and injects a label in each of the form $GOAL_i$ for $i \in \mathbb{N}$. Then, both engines will check whether these injected labels are reachable to produce test cases for branch coverage. After that, *FuSeBMC* analyzes the counterexamples and saves them as a *graphml* file. It checks whether the fuzzing and BMC engines could produce counterexamples for both categories *Cover-Error* and *Cover-Branches*. If that is not the case, *FuSeBMC* employs a second fuzzing engine, the so-called selective fuzzer (cf. Section 3.2.6), which attempts to produce test cases for the rest of the labels. The selective fuzzer produces test cases by learning from the two previous engines' output.

FuSeBMC introduces a novel algorithm for managing the time allocated to its component engines. In particular, *FuSeBMC* manages the time allocated to each engine to avoid wasting time for a specific engine to find test cases for challenging goals. For example, let us assume we have 100 goals injected by *FuSeBMC* and 1000s to produce test cases. In this case, *FuSeBMC* distributes the time per engine per goal so that each goal will have 10s and recalculate the time for the goals remaining after each goal passed. If an engine succeeds on a particular goal within the time limit, the extra time is redistributed to the other goals; otherwise, *FuSeBMC* kills the process that passes the time set for it.

Furthermore, *FuSeBMC* has a minimum time, which a goal must be allocated. If there are too many goals for all to receive this minimum time, *FuSeBMC* will select a subset to attempt using a quasi-random strategy (e.g., all even-numbered goals). *FuSeBMC* also manages the global time of the fuzzing, BMC, and selective fuzzing engines. It gives 13% of the time for fuzzing, 77% for BMC, and 10% for selective fuzzing. The timing of all engines was divided based on the evaluation of various prior experiments. *FuSeBMC* further carries out time management at this global level to maximize engine usage. If, for example, the fuzzing engine is finished before the time allocated to it, its remaining time will be carried over and added to the allocated time of the BMC engine. Similarly, we add the remaining time from the BMC engine to the selective fuzzer allocated time.

FuSeBMC prepares valid test cases with metadata to test a target C program using TestCov [195] as a test validator. The metadata file is an XML file that describes the test suite and is consistently named *metadata.xml*. Fig 3.2 illustrates an example metadata file with all available fields [195]. Some essential fields include the program function that is tested by the test suite $\langle entryfunction \rangle$, the coverage criterion for the test suite $\langle specification \rangle$, the programming language of the program under test $\langle sourcecodelang \rangle$, the system architecture the program tests were created for $\langle architecture \rangle$, the creation time $\langle creationtime \rangle$, the SHA-256 hash of the program under test $\langle programhash \rangle$, the producer of counterexample $\langle producer \rangle$ and the name of the target program

```
 1 <?xml version='1.0'>
 2 <!DOCTYPE test-metadata PUBLIC [...]>
 3 <test-metadata>
 4   <entryfunction>main</entryfunction>
 5   <specification>COVER(init(main()), FQL(COVER EDGES(@DECISIONEDGE
       )))
 6    </specification>
 7   <sourcecodelang>C</sourcecodelang>
 8   <architecture>32bit</architecture>
 9   <creationtime>2021-02-28 20:44:56.117416</creationtime>
10   <programhash>e8f2cf545726d8f791bfc137e9eca7e9de4cb696</
       programhash>
11   <producer>FuSeBMC</producer>
12   <programfile>sv-benchmarks/c/array-tiling/skippedu.c</
       programfile>
13 </test-metadata>
```

Figure 3.2: An example of a metadata.

$\langle programfile \rangle$. A test case file contains a sequence of tags $\langle input \rangle$ that describes the input values sequence. Fig 3.3 illustrates an example of the test case file.

Algorithm 2 describes the main steps we implemented in *FuSeBMC*. It consists of extracting all *goals* of a C program (line 1). For each goal, the instrumented C program, containing the goals (line 2), is executed on our verification engines (fuzzing and BMC) to check the reachability property produced by REACH(G) for that goal (lines 8 & 20). REACH indicates that it has been reached, while BMC refers to our BMC engine (ESBMC), and selective fuzzer is referred to our selective fuzzer engine. REACH is a function; it takes a goal (G) as input and produces a corresponding property for fuzzing/BMC (line 7 & 19). If our engines find that the property is violated, meaning that there is a valid execution path that reaches the goal (counterexample), then the goals are marked as covered, and the test case is saved for later (lines 9-11). Then, we continue if we still have time allotted for each engine. Otherwise, if our verification engines could not reach some goals, then we employ the selective fuzzer in attempt to reach these as yet uncovered goals. In the end, we return all test cases for all the goals we have found in the specified XML format (line 41). Goal labels demonstrate the effectiveness of our technique in guiding engines—moreover, the way of combining the two technologies and utilizing duties for each. The generated test cases are then used in the selective fuzzer to build additional test cases that could detect bugs or increase coverage.

3.2.1 | Analyze C Code

FuSeBMC explores and analyzes the target C programs as the first step using Clang [196]. In this phase, *FuSeBMC* analyzes every single line in the C code and considers the conditional statements such as the *if*-conditions, *for*, *while*, and *do while* loops in the code. *FuSeBMC* takes all these branches as path conditions, containing different values due to the conditions set used to produce the counterexamples, thus helping increase the code coverage. It supports blocks, branches, and conditions. All the values of the variables

Algorithm 2 Proposed *FuSeBMC* algorithm.

Require: program P

1: $goals \leftarrow clang\_extract\_goals(P)$; `// reachability property`
2: $instrumentedP \leftarrow clang\_instrument\_goals(P, goals)$
3: $reached\_goals \leftarrow \emptyset$
4: $tests \leftarrow \emptyset$
5: $FuzzingTime = 150$
6: **for all** $G \in goals$ **do**
7: $\phi \leftarrow REACH(G)$; `// ϕ : property for fuzzing/BMC`
8: $result, test\_case \leftarrow Fuzzing(instrumentedP, \phi, FuzzingTime)$
9: **if** $result = false$ **then**
10: $reached\_goals \leftarrow reached\_goals \cup \{G\}$
11: $tests \leftarrow tests \cup \{test\_case\}$
12: **end if**
13: **if** $FuzzingTime = 0$ **then**
14: $break$
15: **end if**
16: **end for**
17: $BMCTime = FuzzingTime + 700$
18: **for all** $G \in (goals - reached\_goals)$ **do**
19: $\phi \leftarrow REACH(G)$
20: $result, test\_case \leftarrow BMC(instrumentedP, \phi, BMCTime)$
21: **if** $result = false$ **then**
22: $reached\_goals \leftarrow reached\_goals \cup \{G\}$
23: $tests \leftarrow tests \cup \{test\_case\}$
24: **end if**
25: **if** $BMCTime = 0$ **then**
26: $break$
27: **end if**
28: **end for**
29: $SelectiveFuzzerTime = BMCTime + 50$
30: **for all** $G \in (goals - reached\_goals)$ **do**
31: $\phi \leftarrow REACH(G)$
32: $result \leftarrow selectivefuzzer(instrumentedP, \phi, SelectiveFuzzerTime)$
33: **if** $result = false$ **then**
34: $reached\_goals \leftarrow reached\_goals \cup \{G\}$
35: $tests \leftarrow tests \cup \{test\_case\}$
36: **end if**
37: **if** $SelectiveFuzzerTime = 0$ **then**
38: $break$
39: **end if**
40: **end for**
41: **return** $tests$

```
<?xml version="1.0"?>
<!DOCTYPE testcase PUBLIC [...]>
<testcase>
  <input>2</input>
  <input>1</input>
  <input>128</input>
  <input>0</input>
  <input>0</input>
  <input>1</input>
  <input>64</input>
  <input>0</input>
  <input>0</input>
</testcase>
```

Figure 3.3: An example of test case file.

within each path are taken into account. Parentheses and the *else*-branch are added to compile the target code without errors.

3.2.2 | Inject Labels

FuSeBMC injects labels of the form $GOAL_i$ in every branch in the C code as the second step. In particular, *FuSeBMC* adds $else$ to the C code that has an *if*-condition with no $else$ at the end of the condition. Additionally, *FuSeBMC* will consider this as another branch that should produce a counterexample for it to increase the chance of detecting bugs and covering more statements in the program. For example, the code in Fig. 3.4 consists of two branches: the *if*-branch is entered if condition $x < 0$ holds; otherwise, the *else*-branch is entered implicitly, which can exercise the remaining execution paths. Also, Fig. 3.4 shows how *FuSeBMC* injects the labels and considers it as a new branch.

```
#include <stdio.h>
int example () {
   int x;
   if ( x < 0 ){
      //...
   }
}
```

(a) Original C code.

```
#include <stdio.h>
int example () {
   int x;
   if ( x < 0 ){
      GOAL_1:;
      //...
   }
   else{
      GOAL_2:;
   }
   return 0;
}
```

(b) Code instrumented.

Figure 3.4: Original C code vs code instrumented.

3.2.3 | Produce Counterexamples

FuSeBMC uses its verification engines to generate test cases that can reach goals amongst $GOAL_1$, $GOAL_2$, ..., $GOAL_n$ inserted in the previous phase. *FuSeBMC* then checks whether all goals within the C program are covered. If so, *FuSeBMC* continues to the next phase; otherwise, *FuSeBMC* passes the goals that are not covered to the selective

fuzzer to produce test cases for it using randomly generated inputs learned from the test cases produced from both engines. Fig. 3.5 illustrates how the method works.

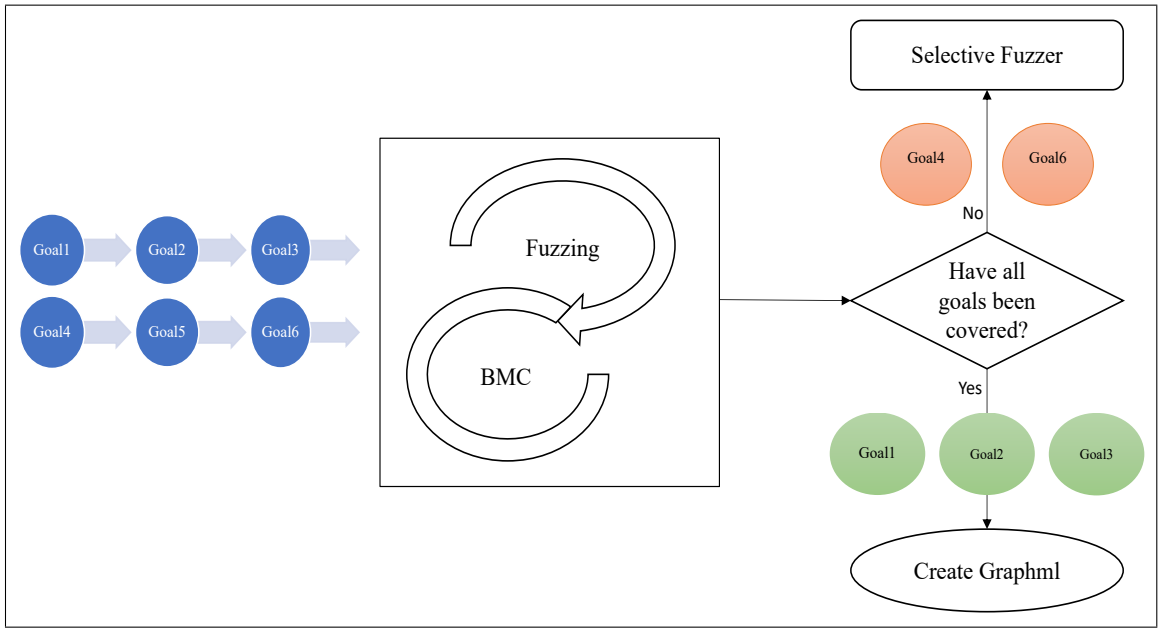


Figure 3.5: Produce Counterexamples.

3.2.4 | Create Graphml

FuSeBMC will generate a *graphml* for each goal injected and then name it. The name of the *graphml* takes the number of the goal extended by the *graphml* extension, e.g., (*GOAL1. graphml*). The *graphml* file contains data about the counterexample, such as data types, values, and line numbers for the variables, which will be used to obtain the values of the target variable.

3.2.5 | Produce test cases

In this phase, *FuSeBMC* will analyze all the *graphml* files produced in the previous phase. Practically, *FuSeBMC* will focus on the `<edge>` tags in the *graphml* that refer to the variable with a type non-deterministic. These variables will store their value in a file called, for example, (*testcase1.xml*). Fig. 3.6 illustrates the edges and values used to create the test cases.

3.2.6 | Selective Fuzzer

In this phase, we apply the selective fuzzer to learn from the test cases produced by either fuzzing or BMC engines to produce test cases for the goals that have not been covered by the two. The selective fuzzer uses the previously produced test cases by extracting from each the number of assignments required to reach an error. For example, in Fig. 3.7, we

```
 1    <edge id="E2" source="N2" target="N3">
 2      <data key="startline">3</data>
 3      <data key="assumption"> a = -2147483647;</data>
 4      <data key="threadId">0</data>
 5    </edge>
 6
 7    <edge id="E4" source="N4" target="N5">
 8      <data key="startline">4</data>
 9      <data key="assumption">b = 0;</data>
10      <data key="threadId">0</data>
11    </edge>
```

Figure 3.6: An example of target edges

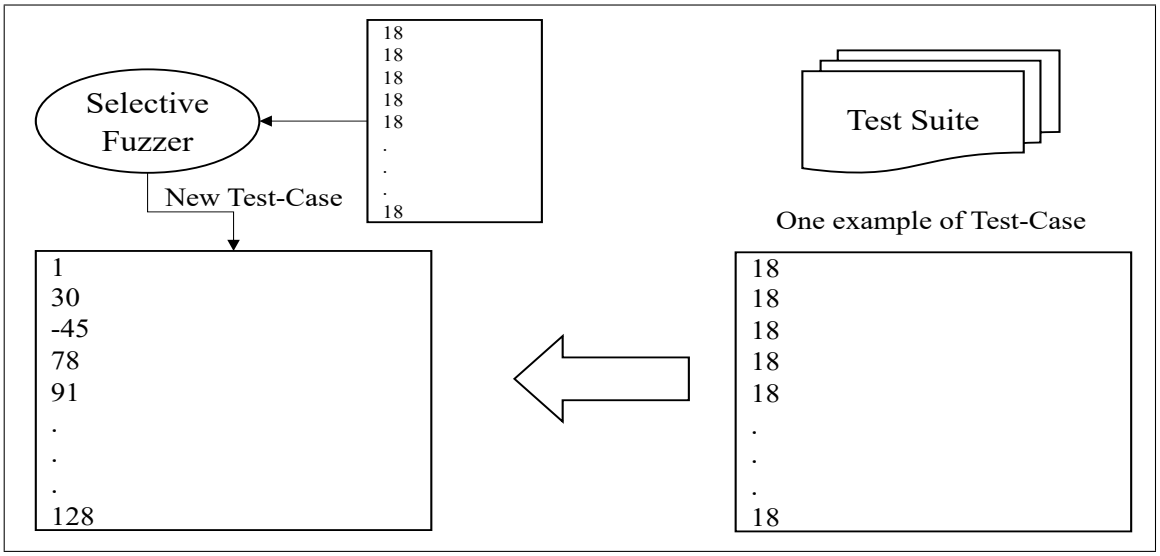


Figure 3.7: The Selective Fuzzer

assumed that the fuzzing/BMC produced a test case that contains values 18 (1000 times) generated from a random seed. The selective fuzzer will produce random numbers (1000 times) based on the test case produced by the fuzzer. In several cases, the BMC engine can exhaust the time limit before providing the information needed by the selective fuzzer, such as the number of inputs, when large arrays need to be initialized at the beginning of the program.

3.2.7 | Test Validator

The test validator takes as input the test cases produced by *FuSeBMC* and then validates these test cases by executing the program on all test cases. The test validator checks whether the bug is exposed if the test was bug-detection, and it reports the code coverage if the test was a measure of the coverage. In our experiments, we use the tool TESTCOV [195] as a test validator. The tool provides coverage statistics per test. It supports block, branch, and condition coverage and covering calls to an error function. TESTCOV uses the XML-based exchange format for test cases specifications defined by Test-Comp [188]. TESTCOV was successfully used in recent editions of Test-Comp 2019, 2020, and 2021 to execute almost 9 million tests on 1720 different programs [195].

3.3 | Evaluation

3.3.1 | Description of Benchmarks and Setup

We conducted experiments with *FuSeBMC* on the benchmarks of Test-Comp 2021 [197] to check the tool's ability in the previously mentioned criteria. Our evaluation benchmarks are taken from the largest and most diverse open-source repository of software verification tasks. The same benchmark collection is used by SV-COMP [198]. These benchmarks yield 3173 test tasks, namely 607 test tasks for the category *Error Coverage* and 2566 test tasks for the category *Code Coverage*. Both categories contain C programs with loops, arrays, bit-vectors, floating-point numbers, dynamic memory allocation, and recursive functions.

The experiments were conducted on the server of Test-Comp 2021 [197]. Each run was limited to 8 processing units, 15 GB of memory, and 15 min of CPU time. The test suite validation was limited to 2 processing units, 7 GB of memory, and 5 min of CPU time. Also, the machine had the following specification of the test node was: one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86-64-Linux, Ubuntu 20.04 with Linux kernel 5.4).

FuSeBMC source code is written in C++; it is available for downloading at GitHub,[2] which includes the latest release of *FuSeBMC* v3.6.6. *FuSeBMC* is publicly available under the terms of the MIT license. Instructions for building *FuSeBMC* from the source code are given in the file *README.md*.

3.3.2 | Objectives

This evaluation's main goal is to check the performance of *FuSeBMC* and the system's suitability for detecting security vulnerabilities in open-source C programs. Our experimental evaluation aims to answer three experimental goals:

EG1 **(Security Vulnerability Detection)** Can *FuSeBMC* generate test cases that lead to more security vulnerabilities than state-of-the-art software testing tools?

EG2 **(Coverage Capacity)** Can *FuSeBMC* achieve a higher coverage when compared with other state-of-the-art software testing tools?

EG3 **(Low Energy Consumption)** Can *FuSeBMC* reduce the consumption of CPU and memory compared with the state-of-the-art tools?

[2] `https://github.com/kaled-alshmrany/FuSeBMC`

3.3.3 | Results

First, we evaluated *FuSeBMC* on the *Error Coverage* category. Table 3.1 shows the experimental results compared with other tools in Test-Comp 2021 [197], where *FuSeBMC* achieved the 1st place in this category by solving 500 out of 607 tasks, an 82% success rate.

In detail, *FuSeBMC* achieved 1st place in the subcategories *ReachSafety-BitVectors, ReachSafety-Floats, ReachSafety-Recursive, ReachSafety-XCSP and ReachSafety- Sequentialized*. *FuSeBMC* solved 10 out of 10 tasks in *ReachSafety-BitVectors*, 32 out of 33 tasks in *ReachSafety -Floats*, 19 out of 20 tasks in *ReachSafety-Recursive*, 53 out of 59 tasks in *ReachSafety-XCSP* and 101 out of 107 tasks in *ReachSafety-Sequentialized*.

FuSeBMC outperformed the top tools in Test-Comp 2021, such as KLEE [151], CPAchecker [191], Symbiotic [199], LibKluzzer [166], and VeriFuzz [159] in these subcategories. However, *FuSeBMC* did not perform as well in the *ReachSafety-ECA* subcategory if compared with leading tools in the competition. We suspect that this is due to the prevalence of nested branches in these benchmarks. The *FuSeBMC*'s verification engines and the selective fuzzer could not produce test cases to reach the error due to the existence of too many path conditions, making the logical formula hard to solve and making it difficult to create random inputs to reach the error. Also, we had an issue with our BMC engine, which we rely heavily on in this subcategory. The issue is reported in the ESBMC repository on GitHub [3]. This issue is related to the "witnesses.h" file, where we define the variable as "unsigned short int start_line." Our BMC engine cannot reach the input "non-deterministic input" if this input is located in the line number higher than 13,372 in the target code. For example, in the program "problem07-lable10.c" in the benchmarks of Test-Comp 2020, our BMC engine could not perform well. This issue happened since the variable responsible for storing the line number in our BMC engine could not approximately store a value higher than approximately 13,372. In the "problem07-lable10.c" program, the non-deterministic variable is located in line 215,423, which caused our BMC engine to be unable to produce counterexamples for these goals.

> Overall, the results show that *FuSeBMC* produces test cases that detect more security vulnerabilities in C programs than state-of-the-art tools, which successfully answers **EG1**.

FuSeBMC also participated in the *Branch Coverage* category at Test-Comp 2021. Table 3.2 shows the experimental results from this category. *FuSeBMC* achieved 4th place in the category by successfully achieving a score of 1161 out of 2566, behind the 3rd place system by 8 scores only. In the subcategory *ReachSafety-Floats*, *FuSeBMC* obtained the

[3] `https://github.com/esbmc/esbmc/issues/291`
[4] `https://test-comp.sosy-lab.org/2021/results/results-verified/`

Table 3.1: *Cover-Error* Results[4]. We identify the best for each tool in bold.

Cover-Error	Task-Num	*FuSeBMC*	CMA-ES Fuzz	CoVeriTest	HybridTiger	KLEE	Legion	LibKluzzer	PRTest	Symbiotic	Tracer-X	VeriFuzz
ReachSafety-Arrays	100	93	0	59	69	88	67	**96**	11	73	75	95
ReachSafety-BitVectors	10	**10**	0	8	6	9	0	9	5	8	7	9
ReachSafety-ControlFlow	32	8	0	8	8	10	0	**11**	0	7	9	9
ReachSafety-ECA	18	8	0	2	1	14	0	11	0	15	2	**16**
ReachSafety-Floats	33	**32**	0	16	22	6	0	30	3	0	0	30
ReachSafety-Heap	57	45	0	37	38	46	0	**47**	9	**47**	44	**47**
ReachSafety-Loops	158	131	0	35	53	96	4	**138**	102	82	78	136
ReachSafety-Recursive	20	**19**	0	0	5	16	0	17	1	17	14	13
ReachSafety-Sequentialized	107	**101**	0	61	93	86	0	83	0	79	57	99
ReachSafety-XCSP	59	**53**	0	46	52	37	0	3	0	41	31	25
SoftwareSystems-BusyBox-MemSafety	11	0	0	0	0	0	0	0	0	0	0	0
DeviceDriversLinux64-ReachSafety	2	0	0	0	0	0	0	0	0	0	0	0
Overall	607	**405**	0	225	266	339	35	359	79	314	246	385

first place by achieving 103 out of 226 scores. Thus, *FuSeBMC* outperformed the top tools in Test-Comp 2021. Further, *FuSeBMC* obtained the first place in the subcategory *ReachSafety-XCSP* by achieving 97 out of 119 scores. However, *FuSeBMC* did not perform well in the subcategory *ReachSafety-ECA* compared with the leading tools in the Test-Comp 2021. Again we suspect the cause to be the prevalence of nested branches in these benchmarks.

These results validate **EG2**. *FuSeBMC* proved its capability in *Branch Coverage* category, especially in the subcategories *ReachSafety-Floats and ReachSafety-XCSP*, where it ranked first.

In both *Cover-Error* and *Cover-Branches* categories, various test cases produced by *FuSeBMC* are validated successfully. The majority of our test cases were produced by the BMC engine and the selective fuzzer. Incremental BMC allows *FuSeBMC* to keep unwinding the program until a property violation is found or time or memory limits are exhausted. This approach is advantageous in the `Cover-Error` category, as finding one error is the primary goal. Another strength of *FuSeBMC* is that it can accurately model C programs that use the IEEE floating-point arithmetic [148, 200]. The floating-point encoding layer in our BMC engine extends the support for the SMT `FP` theory to solvers that do not support it natively. *FuSeBMC* can test programs with floating-point arithmetic using all currently supported solvers in BMC engine (ESBMC), including Boolector [201], which does not support the SMT `FP` theory natively. On the other hand, our fuzzing engine did not produce many test cases because it does not model the C library, so it mostly guesses the inputs. For example, in the *Cover-Error* category, TestCov confirms 500 test cases produced by *FuSeBMC*, where our fuzzing engine produces 13 (Map2Check), BMC engine produces 393 (ESBMC), while our selective fuzzer produces 94 test cases (selective).

However, note that our fuzzing engine is not limited to only producing test cases. It helps our selective fuzzer by providing information about the number of inputs required to trigger a property violation, i.e., the number of assignments required to reach an error. In several cases, the BMC engine can exhaust the time limit before providing such information, e.g., when there are large arrays that need to be initialized at the beginning of the program. For example, consider the following code fragment extracted from the `standard_copy1_ground-2.c` benchmark, as illustrated in Fig. 3.8.

In this particular example, ESBMC exhausts the time limit before checking the assertion $a1[x] == a2[x]$. Apart from that, our employed verification engines also demonstrate a certain level of weakness in producing test cases due to the many optimizations we perform when converting the program to SMT. In particular, two techniques affected the

[5] `https://test-comp.sosy-lab.org/2021/results/results-verified/`

Table 3.2: *Cover-Branches* Results[5]. We identify the best for each tool in bold.

Cover-Branches	Task-Num	*FuSeBMC*	CMA-ES Fuzz	CoVeriTest	HybridTiger	KLEE	Legion	LibKluzzer	PRTest	Symbiotic	Tracer-X	VeriFuzz
ReachSafety-Arrays	400	284	139	229	225	96	195	**296**	119	226	223	295
ReachSafety-BitVectors	62	37	23	39	13	28	29	**40**	27	37	37	38
ReachSafety-ControlFlow	67	15	4	16	3	8	8	16	5	**18**	15	**18**
ReachSafety-ECA	29	5	0	6	2	7	3	10	2	10	7	**12**
ReachSafety-Floats	226	**103**	51	98	84	16	64	90	41	50	48	99
ReachSafety-Heap	143	88	19	79	74	81	69	**90**	40	84	86	86
ReachSafety-Loops	581	412	152	402	338	274	271	419	252	383	385	**424**
ReachSafety-Recursive	53	36	19	31	31	18	20	36	9	**38**	34	35
ReachSafety-Sequentialized	82	62	0	61	39	26	1	55	8	36	41	**71**
ReachSafety-XCSP	119	**97**	0	80	80	81	2	80	79	93	69	88
ReachSafety-Combinations	210	15	0	31	8	82	18	139	2	135	99	**180**
SoftwareSystems-BusyBox-MemSafety	72	1	0	5	4	6	0	6	4	7	4	**8**
DeviceDriversLinux64-ReachSafety	290	35	13	**60**	6	25	56	58	16	44	56	57
SoftwareSystemsSQLite-MemSafety	1	0	0	0	0	0	0	0	0	0	0	0
Termination-MainHeap	231	202	138	193	189	119	166	199	51	178	185	**204**
Overall	2566	1161	411	1128	860	784	651	1292	519	1169	1087	**1389**

```
1  #define N 100000
2    ...
3    int a, a1[N], a2[N];
4    for (a = 0 ; a < N ; a++) {
5       a1[a] = __VERIFIER_nondet_int();
6       a2[a] = __VERIFIER_nondet_int();
7    }
8    ...
9    for (int x = 0 ; x < N ; x++)
10     __VERIFIER_assert(a1[x] == a2[x]);
```

Figure 3.8: Code fragment that contains a large array.

test-case generation significantly: *constant folding* and *slicing*. *Constant folding* evaluates constants (which includes nondeterministic symbols) and propagates them throughout the formula during encoding, and *slicing* removes expression not in the path to trigger a property violation. These two techniques can significantly reduce SMT-solving time. However, they can remove the expressions required to trigger a violation when the program is compiled, i.e., variable initialization might be optimized away, forcing *FuSeBMC* to generate a test case with undefined behavior.

Regarding our fuzzing engine, we identified a limitation in handling programs with pointer dereferences. The fuzzing engine keeps track of variables throughout the program but has issues identifying when they go out of scope. When we try to generate a test case that triggers a pointer dereference, our fuzzing engine provides thrash values, and the selective fuzzer might create test cases that do not reach the error.

FuSeBMC achieved 2nd place overall at Test-Comp 2021, with a score of 1776 out of 3173. Table 3.4 and Fig. 3.9 shows the overall results compared with other tools in the competition. Overall, *FuSeBMC* performed well compared with top tools in the subcate-

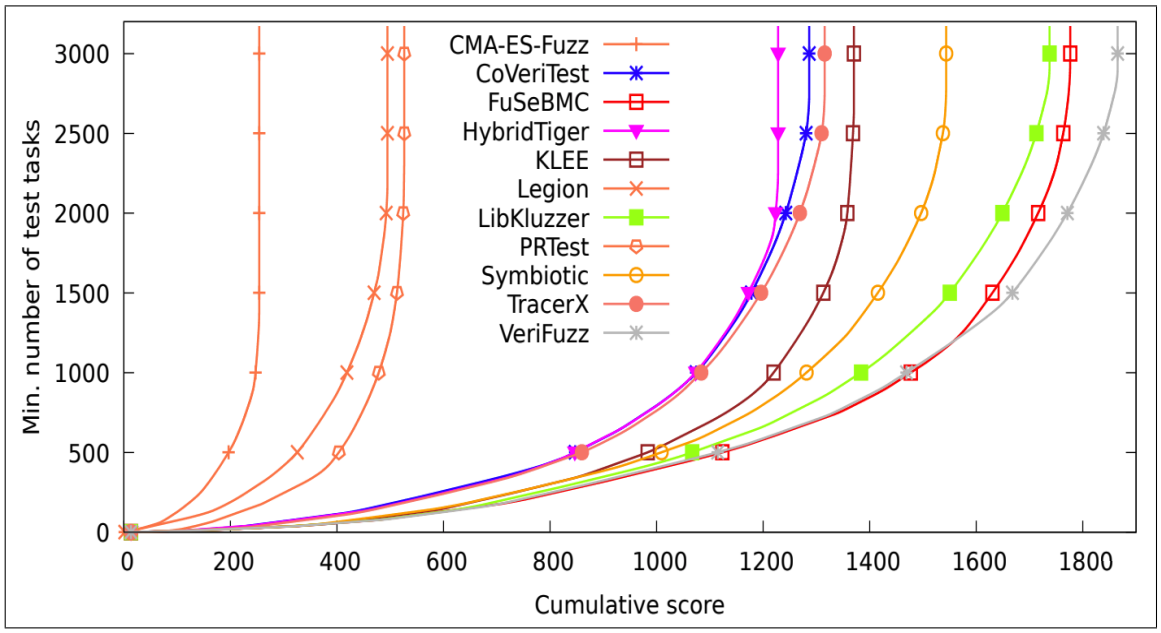


Figure 3.9: Quantile functions for category *Overall*. [183]

gories *ReachSafety-BitVectors, ReachSafety-Floats, ReachSafety-Recursive, ReachSafety-Sequentialized and ReachSafety -XCSP.*

Test-Comp 2021 also considers energy efficiency in rankings since a large part of the cost of test generation is caused by energy consumption. *FuSeBMC* is classified as a Green-testing tool - Low Energy Consumption tool (see Table. 3.3). *FuSeBMC* consumed less energy than many other tools in the competition. This ranking category uses the energy consumption per score point as a rank measure: CPU Energy Quality, with the unit kilo-joule per score point (kJ/sp). It uses CPU Energy Meter [202] for measuring the energy.

Table 3.3: The Consumption of CPU and Memory [183].

Rank	Test Generator	Quality(sp)	CPU Time(h)	CPU Energy(kWh)	Rank Measure
Green Testing					(kj/sp)
1	TRACERX	1315	210	2.5	6.8
2	KLEE	1370	210	2.6	6.8
3	*FuSeBMC*	1776	410	4.8	9.7
worst					51

These experimental results showed that *FuSeBMC* could reduce the consumption of CPU and memory efficiently and effectively in C programs, which answers **EG3**.

[6] `https://test-comp.sosy-lab.org/2021/results/results-verified/`

Table 3.4: Test-Comp 2021 *Overall* Results[6].

Cover-Error and Branches	Task-Num	*FuSeBMC*	CMA-ES Fuzz	CoVeriTest	HybridTiger	KLEE	Legion	LibKluzzer	PRTest	Symbiotic	Tracer-X	VeriFuzz
OVERALL	3173	1776	254	1286	1228	1370	495	1738	526	1543	1315	**1865**

3.4 | Tool Setup and Configuration

FuSeBMC can be run using the command below. The user is required to set the architecture, the property file path, the competition strategy, and the benchmark path, as:

```
fusebmc.py [-a {32, 64}] [-p PROPERTY_FILE]
               [-s {kinduction,falsi,incr,fixed}]
               [BENCHMARK_PATH]
```

where `-a` sets the architecture to 32 or 64, `-p` sets the property file to `PROPERTY_FILE`, where it has a list of all the properties to be tested. `-s` sets the BMC strategy to one of the listed strategies`{kinduction,falsi,incr,fixed}`. For Test-Comp'21, *FuSeBMC* uses `incr` for incremental BMC, which relies on the ESBMC's symbolic execution engine to increasingly unwind the program loops using an iterative technique. The `incr` strategy verifies the program for each unwind bound up to a maximum default value of 50 or indefinitely (until it exhausts the time or memory limits). The Benchexec tool info module is `fusebmc.py` and the benchmark definition file is `FuSeBMC.xml`.

3.5 | Software Project

FuSeBMC is implemented using C++, and it is publicly available under the terms of the MIT License at GitHub[7]. The repository includes the latest version of *FuSeBMC* (version 3.6.6). *FuSeBMC* dependencies and instructions for building from source code are all listed in the `README.md` file. Test-Comp 2021 provides the script, benchmarks, and *FuSeBMC* binary to reproduce the competition's results[8].

3.6 | Conclusions and Future work

We proposed a novel test case generation approach that combined Fuzzing and BMC and implemented it in the *FuSeBMC* tool. *FuSeBMC* explores and analyzes the target C programs by incrementally injecting labels to guide the fuzzing and BMC engines to produce test cases. We inject labels in every program branch to check for their reachability, producing test cases if these labels are reachable. We also exploit the selective fuzzer to produce test cases for the labels that fuzzing and BMC could not produce test cases. *FuSeBMC* achieved two significant awards from Test-Comp 2021. First place in the *Cover-Error* category and second place in the *Overall* category. *FuSeBMC* outperformed the leading state-of-the-art tools because of two main factors. Firstly, the usage of the selective fuzzer as a third engine that learns from the test cases of fuzzing/BMC to produce new test cases for the as-yet uncovered goals. Overall, it substantially increased

[7] `https://github.com/kaled-alshmrany/FuSeBMC`
[8] `https://test-comp.sosy-lab.org/2021/`

the percentage of successful tasks. Secondly, we apply a novel algorithm of managing the time allocated for each engine and goal. This algorithm prevents *FuSeBMC* from wasting time finding test cases for difficult goals so that if the fuzzing engine is finished before the time allocated to it, the remaining time will be carried over and added to the allocated time of the BMC engine. Similarly, we add the remaining time from the BMC engine to the selective fuzzer allocated time. As a result, *FuSeBMC* raised the bar for the competition, thus advancing state-of-the-art software testing. Future work will investigate the extension of *FuSeBMC* to test multi-threaded programs [203, 204] and reinforcement learning techniques to guide our selective fuzzer to find test cases that path-based fuzzing and BMC could not find.

Chapter 4

FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis

> If you believe in yourself and have dedication and pride -and never quit, you'll be a winner. The price of victory is high, but so are the rewards.
>
> *Vince Lombardi*

4.0 | Chapter Overview

4.0.1 | Thesis Context

In the previous chapter, we described our approach *FuSeBMC*, an automated test generating tool that leveraged the combination of Fuzzing and BMC, as well as its scientific contributions and impact on the field of software testing. It has also won multiple international awards and introduced a method that reduces the consumption of resources. In addition to these contributions, we noticed several areas that require improvement in our previous methods, such as code coverage and the significance of contributing to it. In this regard, we set out to develop the method to maintain its effectiveness in detecting vulnerabilities while simultaneously obtaining high code coverage. In this chapter, we introduce our improved approach *FuSeBMC* v4, improving code coverage with smart seeds via fuzzing and static analysis. It includes smart seeds generation, shared memory, a new fuzzer, and a "Tracer" subsystem, among other enhancements and features. This approach relies on smart seed to improve performance in hybrid fuzzers and achieve high C program coverage. Besides, *FuSeBMC* employs shared memory to coordinate the engines and seed distribution by the Tracer. Together, these features turn *FuSeBMC* into a

FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis

Kaled M. Alshmrany[1 2], Mohannad Aldughaim[1], Ahmed Bhayat[1], Fedor Shmarov[1], Fatimah Aljaafari[1] and Lucas C. Cordeiro[1]

[1]University of Manchester, Manchester, UK
[2]Institute of Public Administration, Jeddah, Saudi Arabia

leading fuzzer, as demonstrated by our evaluation results.

The content of this chapter is adapted from: Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, Fedor Shmarov, Fatimah Aljaafari, and Lucas C Cordeiro. "FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis". In: *The Formal Aspects of Computing Journal (FAC)* (2022).

4.0.2 | Author's Contributions

I designed the main idea of the research paper, developed and validated the approach, conducted experiments and evaluation, analysed results, investigated related work, provided and edited all graphics, participated in the entire writing process, and addressed the reviewer's comments. In addition, Mohannad, Ahmed, Fedor and Fatimah provided critical feedback, proofread the paper, and made revisions and amendments during the writing process. My supervisor, Lucas, also contributed to the idea and proofread the paper. Moreover, he guided the entire research procedure.

4.0.3 | Abstract

Bounded model checking (BMC) and fuzzing techniques are among the most effective methods for detecting errors and security vulnerabilities in software. However, there are still shortcomings in detecting these errors due to the inability of extant methods to cover large areas in target code. We propose *FuSeBMC* v4, a test generator that synthesizes seeds with useful properties, that we refer to as *smart seeds*, to improve the performance of its hybrid fuzzer thereby achieving high C program coverage. *FuSeBMC* works by first analyzing and incrementally injecting goal labels into the given C program to guide BMC and Evolutionary Fuzzing engines. It ranks these goal labels according to a user-defined strategy. After that, the engines are employed for an initial period to produce the so–called smart seeds. Finally, the engines are run again, with these smart seeds as starting seeds, in

an attempt to achieve maximum code coverage / find bugs. During both seed generation and normal running, coordination between the engines is aided by the *Tracer* subsystem. This subsystem carries out additional coverage analysis and updates a shared memory with information on goals covered so far. Furthermore, the *Tracer* evaluates test cases dynamically to convert cases into seeds for subsequent test fuzzing. Thus, the BMC engine can provide the seed that allows the fuzzing engine to bypass complex mathematical guards (e.g., input validation). As a result, we received three awards for participation in the fourth international competition in software testing (Test-Comp 2022), outperforming all state-of-the-art tools in every category, including the coverage category.

keywords Code Coverage and Coverage Branches and Automated Test Generation and Bounded Model Checking and Fuzzing and Security

4.1 | Introduction

Fuzzing is one of the essential techniques for discovering software bugs and is used by major corporations such as Microsoft [106], and Google [205]. Fuzzers work by constructing inputs known as *seeds* and then running the program under test (PUT) on these seeds. The goal is to discover a bug by causing the PUT to crash. The main disadvantage of fuzzers is that due to the random manner in which they generate inputs, they are often unable to explore program paths with complex guards. BMC, on the other hand, are very good at using program information to circumvent guards but are often slow and resource-intensive to run.

Hybrid fuzzing attempts to circumvent this issue with more significant program-specific analysis. One common technique is *concolic fuzzing*, which involves using a theorem prover to solve path constraints and thereby helps the fuzzer to explore deeper into the program [44, 206, 207]. However, they still have some fundamental weaknesses; the most important is that the straightforward way they generate seeds can lead to the fuzzer becoming stuck in one part of the code and not exploring other branches.

This paper presents *FuSeBMC*, a state-of-the-art hybrid fuzzer that incorporates various innovative features and techniques. This journal paper is based on several published conference papers [8, 9, 10]. In practice, we concentrated on the enhancements made to *FuSeBMC* between 2021 (when our TAP paper [9] was published) and 2022. We discussed these enhancements briefly in our FASE '22 paper [10], but were unable to provide all of the details due to the limited number of pages allowed. In this journal paper, we are able to expand on these enhancements, such as the use of the Tracer subsystem, shared memory, and the method of analyzing and ranking goals. In addition, we demonstrate the advancement achieved by carrying out a more thorough experimental evaluation. To summarise, we extend those papers by (i) discussing *FuSeBMC* in greater detail (Section 4.2)

(ii) providing more examples, and (iii) providing a thorough and up-to-date experimental evaluation of the tool (Section 4.3).

An important *FuSeBMC* subsystem discussed in this paper is the *Tracer* which coordinates the bounded model checker and the various fuzzing engines. The Tracer monitors the test cases produced by the fuzzers. It selects those with the highest impact (as measured by a couple of metrics discussed in Section 4.2) to act as seeds for future rounds of fuzzing. Further, as discussed above, ESBMC produces test cases to cover particular branches. However, a test case it produces may also cover branches other than the one targeted. In order to ascertain precisely which branches a test case covers and thereby prevent ESBMC from running multiple times unnecessarily, the Tracer takes a test case produced by ESBMC and runs the PUT on it, recording all goals covered.

Bounded model checking can be slow and resource-intensive. To mitigate against this, *FuSeBMC* does not make use of an off-the-shelf fuzzer for its grey box fuzzing, but instead uses a modified version of the popular American Fuzzy Lop tool. One of the features of this modified fuzzer is its ability to carry out lightweight static analysis of a program to recognize input verification. It analyzes the code for conditions on the input variables and ensures that seeds are only selected if they pass these conditions. This reduces the dependence on the computationally expensive bounded model checker for finding quality seeds. Another interesting feature of the modified fuzzer is that it analyses the PUT and heuristically identifies potentially infinite loops. It then bounds these loops in an attempt to speed up fuzzing. These bounds are incremented during the multiple fuzzing rounds. In this version, *FuSeBMC* relied on this fuzzer instead of the previous one because our modified fuzzer is consistent with our new theory of producing and using seeds. In contrast, the previous fuzzer (Map2Check) does not support that. Also, we wanted to integrate our selective fuzzer to be within the stages in the AFL and thus benefit from the power and analysis of AFL, generating effective test cases.

Together, these features turn *FuSeBMC* into a leading fuzzer. In the 2022 edition of the Test-Comp software testing competition, *FuSeBMC* achieved first place in both the main categories, *Cover-Error* and *Cover-Branches*. In the *Cover-Branches* category, it achieved first place in 9 out of the 16 subcategories that it participated in. In the *Cover-Error* category, it achieved first place, or joint first place, in 8 out of the 14 subcategories that it participated in.

Contributions. This journal paper explains the latest developments to the *FuSeBMC* fuzzer. The work presented here is a substantial extension of our previous published conference papers [8, 9, 10]. *FuSeBMC*'s main new features can be summarised as follows:

- The use of lightweight static analysis to recognize some forms of input validation on variables, thereby enabling fuzzing to produce more effective seeds and speed up the fuzzing process.

- The prioritization of deeper goals with regards to finding test cases as this can result in providing higher code coverage and generating fewer test cases.
- The setting of a loop unwinding depth during seed generation and fuzzing. As loop unwinding leads to exponential path explosion, we restrict the unwinding depth of each loop to a small number, depending on an approximate estimate of the number of program paths.

We also extend our previous papers by:

- Explaining the working of the *FuSeBMC* tool in greater depth and clarity than previously.
- Providing a detailed analysis of our participation in the international competition on software testing (Test-Comp 2022), where our tool *FuSeBMC* was able to achieve three significant awards. *FuSeBMC* earned first place in all the categories by the improvements described in this manuscript. We also provide a thorough comparison between version 4 of the tool and the previous iteration, version 3, thereby demonstrating the effectiveness of our extensions.

4.2 | FuSeBMC v4 Framework

FuSeBMC combines dynamic and static verification techniques for improving code coverage and bug discovery. It utilizes the Clang compiler [51] front-end to perform various code transformations, ESBMC (Efficient SMT-based Bounded Model Checking) [148, 194] as a BMC and symbolic execution engine, and a modified version of the American Fuzzy Lop (AFL) tool [56, 137] as well as a custom selective fuzzer [9] as fuzzing engines.

FuSeBMC takes a C program as input and produces a set of test cases, maximizing code coverage while also checking for various bugs. Users can choose to check for several types of bugs that are supported by ESBMC (such as array bounds violations, divisions by zero, pointers safety, arithmetic overflows, memory leaks, and other user-defined properties). Figure 4.1 illustrates the *FuSeBMC* architecture and its workflow, and Algorithm 3 presents the main stages of the *FuSeBMC* execution.

4.2.1 | Overview

FuSeBMC begins by injecting goal labels into the given C program (based on the code coverage criteria that we introduce in Section 4.2.2.1) and ranking them according to one of the strategies described in Section 4.2.2.2 (i.e., depending on the goal's origin or depth in the PUT). From then on, *FuSeBMC*'s workflow can be divided into two main stages: *seed generation* (the preliminary stage) and *test generation* (the full coverage analysis stage). During *seed generation*, *FuSeBMC* applies the fuzzers and BMC to the

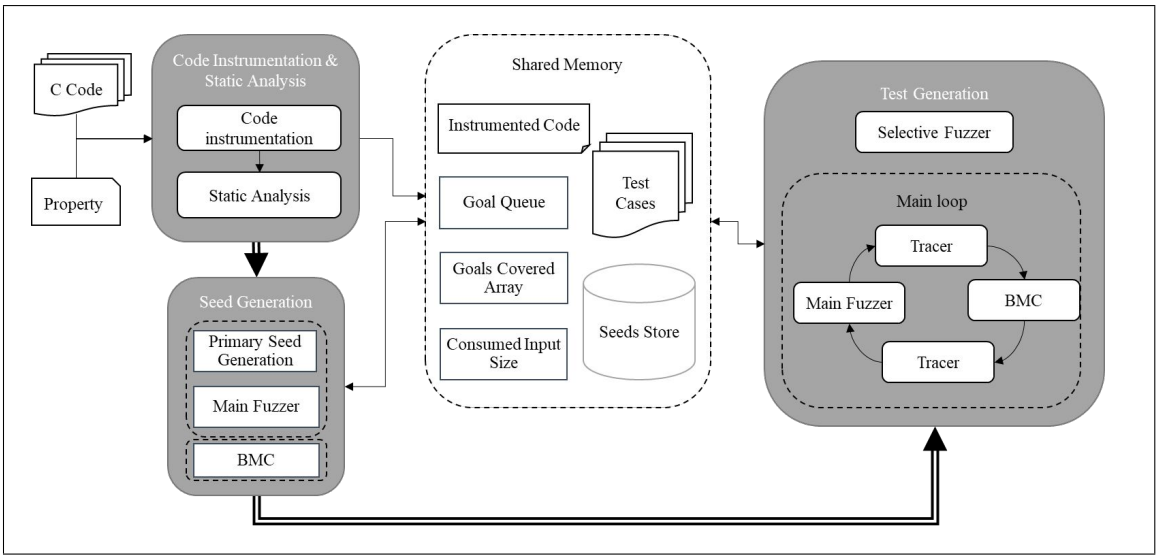


Figure 4.1: The Framework of *FuSeBMC* v4. This figure illustrates the main components of *FuSeBMC*. Our tool starts by instrumenting and analyzing the source code, then performs coverage analysis in two stages: seed generation and test generation.

instrumented code once for a short time to produce seeds that are used by the fuzzers at the *test generation* stage and test cases that may provide coverage of some "shallow" goals. The intuition behind this divide is to quickly generate some meaningful seeds for the fuzzer that could increase the chances of exploring the PUT past the entry point, which often contains restrictive input validators that are hard to negotiate for the fuzzers. During *test generation*, the above engines are applied with a longer timeout while accompanied by another analysis engine called *Tracer*. It helps the execution of the fuzzers and the bounded model checker by recording which goal labels in the PUT have been covered by the test cases produced by these engines. This is done to prevent the computationally expensive BMC engine from trying to reach an already covered goal. *FuSeBMC* continues with the *test generation* stage until all goals are covered or a timeout is reached.

In Figure 4.2 we introduce a short C program which we use as a running example to demonstrate the main code transformations throughout this section. The presented program accepts coefficients of a quadratic polynomial and an integer candidate solution in the range [1,100] as input from the user. It terminates successfully if the provided candidate solves the equation. However, the program returns an error if the given equation does not have real solutions or the input candidate value is outside the [1,100] range.

Algorithm 3 *FuSeBMC* algorithm

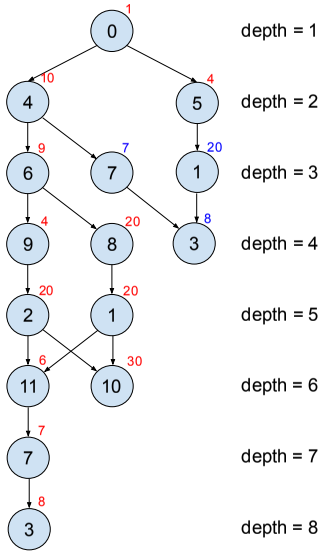
1: $P := get\_input\_PUT()$
   // Code Instrumentation & Analysis
2: $P' := inject\_goal\_labels(P)$
3: $G := get\_list\_of\_sorted\_goals(P')$ // Seed Generation
4: $T := \emptyset; B := \emptyset;$ // initializing queues for test cases and bug reports
5: $\{S, G_{cov}, T\} := generate\_seeds(P');$ // see Algorithm 4
6: $G.remove\_goals(G_{cov})$
   // Test Generation
7: **while** $G \neq \emptyset$ **or** $timeout$ **do**
8: $g := G.pop()$; // Start of main loop
9: $\{output, G_{cov}\} := run\_fuzzer(P', g, S, fuzz_{timeout})$
10: **if** $G_{cov} \neq \emptyset$ **then**
11: $\{T, G, S\} := run\_tracer(P', output, T, G, G_{cov}, S);$ // see Algorithm 5
12: **end if**
    // current goal has been covered, so skip to the next iteration
13: **if** $g \in G_{cov}$ **then**
14: **continue**
15: **end if**
    // BMC's output can be a reachability witness or a bug trace
16: $\{output, res\} := run\_bmc(P', g, bmc_{timeout})$
17: **if** $res = success$ **then**
18: $\{T, G, S\} := run\_tracer(P', output, T, G, \emptyset, S);$ // see Algorithm 5
19: **else**
20: **if** $output \neq \emptyset$ **then**
21: $B := B \cup generate\_bug\_report(P', output)$
22: **else**
23: $\{T, G, S\} := run\_tracer(P', output, T, G, \emptyset, S);$ // see Algorithm 5
24: **end if**
25: **end if**
26: **end while**; // End of main loop
27: **if** $G \neq \emptyset$ **then**
28: $\{testcases, G_{cov}\} := run\_selective\_fuzzer(P')$
29: $T := T \cup testcases$
30: **end if**
31: **return** $\{T, B, G\}$

```c
#include <assert.h>
void reach_error() { }
bool check(int a, int b, int c, int x) {
 return (a*x*x + b*x + c == 0);}
int main() {
 int a = __VERIFIER_nondet_int();
 int b = __VERIFIER_nondet_int();
 int c = __VERIFIER_nondet_int();
 if(b*b >= 4*a*c) {
  while(1) {
   int x = __VERIFIER_nondet_int();
   if(x <= 0 || x > 100)
    reach_error();
   if(check(a, b, c, x))
    return 0;
  }
 }
 else
  reach_error();
return 0;
}
```

(a)

```c
#include <assert.h>
void reach_error() {
 GOAL_1:;}
bool check(int a, int b, int c, int x) {
 GOAL_2:;
 return (a*x*x + b*x + c == 0);
}
int main() {
 GOAL_0:;
 int a = __VERIFIER_nondet_int();
 int b = __VERIFIER_nondet_int();
 int c = __VERIFIER_nondet_int();
 if(b*b >= 4*a*c) {
  GOAL_4:;
  while(1) {
   GOAL_6:;
   int x = __VERIFIER_nondet_int();
   if(x <= 0 || x > 100) {
    GOAL_8:;
    reach_error();}
   else {
    GOAL_9:;}
   if(check(a, b, c, x)) {
    GOAL_10:;
    return 0;}
   else {
    GOAL_11:;}
  }
  GOAL_7:;}
 else {
  GOAL_5:;
  reach_error();}
 GOAL_3:;
 return 0;
}
```

(b)



(c)

Figure 4.2: An example of a) a C program, b) the corresponding instrumented code, and c) the resulting goals tree, their depth in the code, and resulting rank values.

4.2.2 | Code Instrumentation & Static Analysis

At this stage, *FuSeBMC* instruments the PUT and performs multiple static analyses. It takes the PUT (i.e., a C program) and a property file as inputs and produces three files: the instrumented program, *Goal Queue*, and *Consumed Input Size*.

4.2.2.1 Code Instrumentation

FuSeBMC uses Clang tooling infrastructure [51] at its front-end to parse the input C program and traverse the resulting Abstract Syntax Tree (AST), recursively injecting goal labels into the PUT. This process is guided by the *FuSeBMC* code coverage criteria. Namely, *FuSeBMC* inserts labels inside conditional statements, loops, and functions as follows.

- For conditional statements: the label is inserted at the beginning of the block whether the statement is an `if`, `else`, or an instrumented empty `else`.
- For loops: the label is placed at the beginning of the loop body and right after exiting the loop.
- For functions: labels are injected at the beginning and at the end of the function body.

Furthermore, *FuSeBMC* adds declarations for several standard C library functions, such as *"printf, "strcpy", "memset"* and other C language functions, to ensure that we cover the majority of the functions that we may encounter in large programs while also maintaining the proper operation of our approach. The resulting *instrumented code* is functionally equivalent to the original C program. Figure 4.2b demonstrates an example of the described code instrumentation for the program in Figure 4.2a.

4.2.2.2 Static Analysis

Apart from the required code instrumentation, Clang produces compilation error and warning messages and utilizes its static analyzer to simplify the input program (e.g., calculating the sizes of expressions, evaluating static asserts, and performing constants propagation) [208].

Furthermore, the Clang static code analyzer produces the *Consumed Input Size*, which represents the minimum number of bytes required for fuzzing. This information plays an important role in enhancing the fuzzing process (see Section 4.2.4.1).

Another function of static analysis is to identify the ranges of the input variables. This is performed by collecting branch conditions that match a pattern $(x \circ val)$, where x is the name of the variable, val is a numeric value, and $\circ \in \{>, \geq, <, \leq, =, \neq\}$. This information is used by both fuzzers for generating inputs only from the identified ranges.

Finally, *FuSeBMC* analyzes the instrumented code and ranks the injected goal labels. Each goal label is attributed with its origin information (i.e., if statement, while loop, end

of function) and its depth in the instrumented program. Then *FuSeBMC* sorts all goals using one of the two strategies (line 3 of Algorithm 3): (1) based on their depth (i.e., depth-first search), or (2) based on their *rank* scores calculated as a product of a goal's depth and its *power* score - a value between 1 and 5 describing the goal's branching power. Each power score has been decided via experimental analysis. The *if* statement goals are assigned a score of 5 (goals 4, 8 and 9 in Figure 4.2b), the *function* goals - 4 (goals 1 and 2 in Figure 4.2b, note that main function goals are scored differently), the *loop* goals - 3 (goal 6 in Figure 4.2b) and the *else* goals - 2 (goal 5 in Figure 4.2b). All remaining types of goals (i.e., *end-of-main* (goal 3 in Figure 4.2b), *empty-else*, and *after-loop* goals (goal 7 in Figure 4.2b)) are assigned a value of 1.

In general, the goal sorting improves overall *FuSeBMC* performance. Using the depth strategy, *FuSeBMC* attempts to cover the deeper goals first. This is beneficial since all preceding goals on the path to a deep goal can be ignored during subsequent fuzzing as the same test case covers them. On the other hand, the ranking strategy allows prioritizing conditional branches as they may lead to multiple goals increasing potential code coverage.

Figure 4.2c features the resulting goals tree for the instrumented code from Figure 4.2b (with `GOAL_0` representing the entry point of the program, i.e., the `main` function). Note that *FuSeBMC* builds it based on the original Clang AST without analyzing the code for trivially unreachable goals. For example, labels `GOAL_7` and `GOAL_3` can never be reached during the program's execution. However, this will not be reflected in the goals tree.

The goal's depth value is assigned at its highest depth. Therefore, labels `GOAL_1`, `GOAL_7`, and `GOAL_3` are assigned depth values of 5, 7, and 8, respectively. When the first ranking strategy is applied, two goals at the same depth are ordered in the ascending order of their label names. Using the second ranking strategy, two goals with the same *rank* value are processed in the "power score first" manner. For example, `GOAL_8` will be placed in front of `GOAL_1` and `GOAL_2` since it has a higher power score (5 vs 4). Hence, *FuSeBMC* will process the goal labels in the following orders {3,7,10,11,1,2,8,9,6,4,5}, {10,8,1,2,4,6,3,7,11,5,9} using the first and second sorting strategies, respectively. Finally, the list of goals is stored in the shared memory as the *Goal Queue*. This queue can be modified by the BMC and Tracer engines during the consecutive stages to remove the goal labels that have been covered.

4.2.2.3 Shared Memory

The set of data files that each component of *FuSeBMC* has access to (both for reading and writing) is called *Shared Memory*. Apart from *Instrumented Code*, *Consumed Input Size* and *Goal Queue* discussed above, it contains *Seeds Store* – a collection of seeds used by the fuzzer for test generation, *Test cases* – all test cases generated by *FuSeBMC*, and *Goal*

Covered Array – list of all goal labels that have been covered by the produced test cases.

4.2.3 | Seed Generation

Algorithm 4 $generate\_seeds()$ algorithm

Input: P' : instrumented file
Output: S : set of seeds, G_{cov} : goals covered, T: generated test cases

1: $S := generate\_primary\_seeds(P')$; `// 0's, 1's, and random`
2: $\{testcases, G_{cov}\} := run\_fuzzer(P', S, fuzz_{timeout})$; `// timeout = 5 seconds`
3: **if** $G_{cov} \neq \emptyset$ **then**
4: $T := T \cup testcases$
5: $S := S \cup generate\_seed(testcases)$
6: **end if**
7: $\{output, res\} := run\_light\_bmc(P', bmc_{timeout})$; `// BMC with partial loops & timeout = 15 seconds`
8: **if** $res = success$ **then**
9: $S := S \cup output$
10: $T := T \cup generate\_testcases(P', \{output\})$
11: **else**
12: **if** $output = \emptyset$ **then**
13: $T := T \cup generate\_testcases(P', S)$
14: **end if**
15: **end if**
16: **return** $\{S, G_{cov}, T\}$

Having ranked the goals, *FuSeBMC* carries out seed generation (line 5 of Algorithm 3 and Algorithm 4 where it is described in detail) as a preliminary step before full coverage analysis (i.e., *test generation*) begins. In this phase, *FuSeBMC* simplifies the target program by limiting loop bounds, and utilizes the information about the input ranges. Then *FuSeBMC* applies the fuzzer and the BMC engine (for 5 and 15 seconds, respectively) for a short time in succession.

Since the seed store is empty at this point, *FuSeBMC* performs *primary seed generation* (Line 1 Algorithim 4) to enable the fuzzing process. This procedure involves generating binary seeds (i.e., a stream of bytes) based on *Consumed Input Size* and the input constraints collected during static analysis. In detail, it generates three sequences of bytes, where: 1) all bytes have a value of 0, 2) all bytes have a value of 1, 3) all byte values drawn randomly from the identified input ranges. Then the fuzzer (see Section 4.2.4.1) is initialized with the primary seeds and is run for a short time to produce test cases that are then converted into new seeds and added to the *Seeds store* (see Figure 4.1 and lines 2 — 6 in Algorithm 4).

When the seed generation by fuzzing is finished, *FuSeBMC* executes the BMC engine (see Section 4.2.4.2) for each goal label in the *Goal Queue*. In order to minimize the execution time, it is run with "lighter" settings: all implicit checks (i.e., memory safety, arithmetic overflows) and assertion checks are disabled, and the bound for loop unwinding is reduced. If a goal label is reached successfully, the BMC engine produces a witness – a set of program inputs that lead to that goal label. The sequence of these input values is added as a seed to the *Seed Store*.

All new seeds produced by the fuzzer and the BMC engine are deemed *smart* due to their powerful effect on code coverage. Conceptually, bounded model checkers use SMT solvers to produce test cases that resolve complex branch conditions (i.e., guards). Such guards (for example, lines 5 and 12 in Figure 4.2a) pose a challenge to a fuzzer [44] as it relies on mutating the given seed randomly and is therefore unlikely to satisfy the branch condition. Seeds produced by BMC help solve this issue since they can be passed to a fuzzer, which can then advance deeper behind the complex guards into the target program (which is usually hard for a bounded model checker).

4.2.4 | Test Generation

Following seed generation, *FuSeBMC* begins the main coverage analysis phase (lines 7 — 30 of Algorithm 3). *FuSeBMC* incorporates three engines to carry out this analysis: two fuzzers (main fuzzer and selective fuzzer) and a bounded model checker. Here, both the main fuzzer and the BMC engine are run with longer timeouts than during the seed generation stage. Briefly, the fuzzers utilize the *smart* seeds produced at the previous stage, and generate test cases by randomly mutating the program's input and running it to analyze code coverage. The bounded model checker determines the reachability of particular goal labels similarly to the *seed generation* stage. *FuSeBMC*'s *Tracer* component aids the above engines by running a light-weight analysis (see Section 4.2.4.3) of the produced test cases and updating the *Shared Memory*.

In the following subsections, we discuss the *FuSeBMC* components involved in coverage analysis in greater detail.

4.2.4.1 Main Fuzzer

In *FuSeBMC* we implement a modified version of the American Fuzzy Lop (AFL) tool [56]. The modified AFL generates test cases based on the evolutionary algorithm implemented in AFL [137].

The standard algorithm implemented in the AFL tool works as follows. Firstly, an initial input stream of fixed size is generated using the provided seed (a random seed is used if not explicitly specified). Secondly, the target program is repeatedly executed with the randomly mutated input. If the target program does not reach any new states after

multiple input mutation rounds, a new byte is added to or removed from the input stream, and the mutation process restarts. The above algorithm continues until an internal timeout is reached or the fuzzer finds inputs that fully cover the program. In general, the AFL's mutation algorithm heavily relies on the quality of the initial seeds for providing higher code coverage. Therefore, generating seeds with higher coverage potential is crucial.

FuSeBMC modifies the original AFL fuzzer as follows.

1. It performs additional instrumentation to the PUT to minimize its execution overhead by limiting the bounds of loops heuristically identified as potentially infinite. Note that these bounds can be iteratively changed between the AFL runs.
2. The mutation operators are modified by accepting only inputs from the ranges identified during the static code analysis.
3. It controls the size of the generated test cases via the *Consumed Input Size*. In detail, the minimum size of the test cases produced by the fuzzer is set to the current value of the consumed input size. This allows counter-acting the size selection bias of the AFL mutation algorithms, which tend to favor a reduction of the number of bytes in the generated test cases (instead of adding extra bytes) between the mutation rounds. At the same time, the modified fuzzer can control the maximum size of the produced test cases. For example, when the *Consumed Input Size* starts growing gradually during the fuzzing process (a behaviour often observed in programs accepting input in an infinite loop), the maximum test-case size is set in order to prevent performance degradation.
4. It outputs the list of goals covered by the produced test cases and records them in *Goals Covered Array*.

4.2.4.2 Bounded Model Checker

FuSeBMC uses *ESBMC* to check for the reachability of a given goal label within the instrumented program (lines 16 - 25 of Algorithm 3). If it concludes that the current goal is reachable it produces a counterexample that can be turned into a witness – a sequence of inputs that leads the program's execution to that goal label – which is then used to generate a test case. Every new test case thus discovered is also added to the *Seed Store* to be used by the fuzzers. Even if the BMC runs out of time or memory, its progress in reducing the input ranges is saved as a *incomplete seed* – a sequence of input values that lead the PUT execution part of the way towards the given goal label.

4.2.4.3 Tracer

The *Tracer* subsystem determines the goals covered by test cases produced by the bounded model checker and the fuzzer. Whenever a test case is produced, *Tracer* compiles the instrumented program together with the newly generated test cases and runs the

Algorithm 5 $run\_tracer()$ algorithm

Input: P' : instrumented file, *output* : engines output , T: generated test cases , G : list of goals , G_{cov} : covered goals list, S : set of seeds

Output : T: generated test cases , G : list of goals , S : set of seeds

1: **if** $output$ from BMC and incomplete **then**
2: $testcase := complete\_testcase(output)$
3: **else**
4: $testcase := generate\_testcases(output, G_{cov})$
5: **end if**
6: $T := T \cup testcase$
7: $G_{cov} := run\_testcase(P', testcase)$
8: $G.remove\_goals(G_{cov})$
9: $S := S \cup testcase$
10: **return** $\{T, G, S\}$

resulting executable. Prior to the compilation, it performs additional instrumentation to the test case to output information about the PUT input size, the types of input variables, and the visited goals. This information is dynamically updated in the *Shared Memory* (i.e., *Goals Covered Array* and *Consumed Input Size*).

The Tracer also analyses the test cases produced by the other two engines to add the highest impact cases (i.e., the test cases leading to new goals or reaching the maximum analysis depth) to the *Seeds Store*.

Another responsibility of *Tracer* is to handle the partial output of the bounded model checker when it reaches the timeout outputting an incomplete counterexample. *Tracer* completes such counterexamples randomly and performs the coverage analysis and updates *Seeds Store* as described above.

4.2.4.4 Selective Fuzzer

The selective fuzzer's [9] main function is to attempt to reach the remaining uncovered goals after the iterative process of applying the fuzzer, and the BMC engine has finished. Similarly to the main fuzzer, it utilizes information about the identified ranges of the input variables to produce inputs for the PUT. At the same time, it implements a complementary test generation approach. It produces random values from the given input ranges – in contrast to the mutation-based approach used in the main fuzzer. The selective fuzzer terminates upon covering all the remaining goals or upon reaching the timeout.

4.3 | Evaluation

4.3.1 | Description of Benchmarks and Setup

In order to assess the performance of *FuSeBMC v4*, we evaluated its participation in Test-Comp 2022 [209], and also compared it to the results obtained by the previous version of the tool, *FuSeBMC v3*, in Test-Comp 2021 [210].

Test-Comp is a software testing competition where the participating tools compete in automated test case generation. All test case generation tasks in Test-Comp are divided into two categories: *Cover-Branches* and *Cover-Error*. The former requires producing a set of test cases that maximize *code coverage* (in particular, *branch coverage*) for the given C program, and the latter deals with *error coverage*: generating a test case that leads to the predefined error location (i.e., explicitly marked error function) within the given C program. In *Cover-Branches*, code coverage is measured by the TestCov [195] tool, which assigns a score between 0 and 1 for each task. For example, if a competing tool achieves 80% code coverage on a particular task, it is assigned a score of 0.8 for that task and so forth. Overall scores for the subcategories are calculated by summing the individual scores for each task in the subcategory and rounding the result. In *Cover-Error*, each tool earns a score of 1 if it can provide a test case that reaches the error function and gets a 0 score otherwise. Each category is further divided into multiple subcategories (see Tables 4.1 and 4.3) based on the most prominent program features and/or the program's origin. The vast majority of the programs present in Test-Comp are taken from SV-COMP [198] – the largest and most diverse open-source repository of software verification tasks. It contains hand-crafted and real-world C programs with loops, arrays, bit-vectors, floating-point numbers, dynamic memory allocation, and recursive functions, event-condition-action software, concurrent programs, and BusyBox[1] software.

Both Test-Comp 2021 and Test-Comp 2022 evaluations were conducted on servers featuring an 8-core (4 physical cores) Intel Xeon E3-1230 v5 CPU @ 3.4 GHz, 33 GB of RAM and running x86-64 Ubuntu 20.04 with Linux kernel 5.4. Each test suite generation run was limited to 8 CPU cores, 15 GB of RAM, and 15 mins of CPU time, while each test suite validation run was limited to 2 CPU cores, 7 GB of RAM, and 5 mins of CPU time. In 2021, *FuSeBMC* distributed its allocated time to its various engines as follows. The fuzzer received 150s when running on benchmarks from the *Cover-Error* category and 70s on benchmarks from the *Cover-Branches* category. The bounded model checker received 700s and 780s on benchmarks from the two categories, respectively. Finally, the selective fuzzer received 50s for benchmarks from both categories. In 2022, these figures were tweaked. The seed generation received 20s for benchmarks from both categories. The fuzzer received 200s and 250s on benchmarks from *Cover-Error* and *Cover-*

[1] https://busybox.net/

Branches, respectively, the bounded model checker 650s and 600s, and the allocation for the selective fuzzer was decreased from 50s to 30s from the previous year.

Despite the fact that the hardware setup remained unchanged across the two competition editions, the set of test generation tasks was significantly expanded. Namely, the task set in Test-Comp 2021 consisted of 3173 tasks: 607 in the *Cover-Error* category, and 2566 in the *Cover-Branches* category. By contrast, Test-Comp 2022 was expanded to contain 4236 test tasks: 776 in the *Cover-Error* category and 3460 in the *Cover-Branches* category (including a new subcategory *ProductLines* introduced into both categories). We have taken this into consideration when discussing the performance of two versions of *FuSeBMC* in Section 4.3.3.1. A detailed report of the results produced by the competing tools in both Test-Comp 2021[2] and Test-Comp 2022[3] is available online.

FuSeBMC source code is written in C++ and Python; it is available for download from GitHub[4]. The latest release of *FuSeBMC* is v4.1.14. *FuSeBMC* is publicly available under the terms of the MIT license. Instructions for building *FuSeBMC* from the source code are given in the file *README.md*.

4.3.2 | Objectives

The main goal of our experimental evaluation is to assess the improvements of *FuSeBMC v4* and its suitability for achieving high code coverage and error coverage in open-source C programs. As a result, we identify three key evaluation objectives:

O1 **(Performance Improvement)** Demonstrate that *FuSeBMC v4* outperforms *FuSeBMC v3* in both code coverage and error coverage.

O2 **(Coverage Capacity)** Demonstrate that *FuSeBMC v4* achieves higher code coverage for C programs than other state-of-the-art software testing tools.

O3 **(Error Detection)** Demonstrate that *FuSeBMC v4* finds more errors in C programs than other state-of-the-art software testing tools.

4.3.3 | Results

4.3.3.1 *FuSeBMC v4* vs *FuSeBMC v3*

Tables 4.1 and 4.3 contain the comparison of the *FuSeBMC v4* and *FuSeBMC v3* performances in *Cover-Branches* and *Cover-Error* categories of Test-Comp, respectively. *FuSeBMC v3* achieved first place in *Cover-Error*, fourth place in *Cover-Branches*, and placed second overall in Test-Comp 2021, while *FuSeBMC v4* reached first place in both

[2] `https://test-comp.sosy-lab.org/2022/results/results-verified/`
[3] `https://test-comp.sosy-lab.org/2021/results/results-verified/`
[4] `https://github.com/kaled-alshmrany/FuSeBMC`

Table 4.1: Comparison of the average coverage (per subcategory and the category overall) achieved by *FuSeBMC v4* and *FuSeBMC v3* in the *Cover-Branches* category in TestComp-2022 and TestComp-2021, respectively.

Subcategory	% average coverage		Improvement
	FuSeBMC v4	*FuSeBMC v3*	$\Delta\%$
Arrays	82%	71%	11%
BitVectors	80%	60%	20%
ControlFlow	64%	22%	42%
ECA	37%	17%	20%
Floats	54%	46%	8%
Heap	73%	62%	11%
Loops	81%	71%	10%
ProductLines	29%	-	-
Recursive	85%	68%	18%
Sequentialized	87%	76%	11%
XCSP	90%	82%	8%
Combinations	61%	7%	53%
BusyBox	34%	1%	32%
DeviceDrivers	20%	12%	8%
SQLite-MemSafety	4%	0%	4%
Termination	92%	87%	5%
Cover-Branches	61%	45%	16%

categories and overall in Test-Comp 2022. However, taking into account that the test generation task set has been significantly expanded in Test-Comp 2022, we analyze their relative performances in each subcategory. Namely, in *Cover-Branches*, we compare average code branch coverage, and in *Cover-Error*, we compare the percentages of successfully detected errors demonstrated by both tools in every subcategory and in the entire category (as well the improvements of *FuseBMC v4* in comparison to *FuSeBMC v3*), respectively.

Table 4.1 shows that *FuSeBMC v4* advanced in each individual subcategory, including the overall average improvement of 16% in the *Cover-Branches* category in comparison to *FuSeBMC v3*. The greatest increase (i.e., 53%) was demonstrated in the *Combinations* subcategory. *FuSeBMC v3* achieved eighth place in this subcategory in Test-Comp 2021, while *FuSeBMC v4* reached first place in *Combinations* in Test-Comp 2022. We attribute this success to the modifications in the seed generation phase of *FuSeBMC v4* (in particular, the introduction of *smart seeds*). Table 4.2 presents a subset of generation tasks from the *Combinations* subcategory where *FuSeBMC v4* demonstrated the most striking improvement. It can be seen that *FuSeBMC v3* provided very low code coverage of $\sim 6.52\%$ for these tasks on average, while *FuSeBMC v4* increased this number to $\sim 90.14\%$ (i.e., 83.62% average improvement).

As for the *Cover-Error* category, *FuSeBMC v4* progressed by 14% on average in comparison to *FuSeBMC v3* (see Table 4.3). *FuSeBMC v4* improved in the majority of subcategories while showing no change in three subcategories: both *FuSeBMC* ver-

Table 4.2: Comparison of code coverage achieved by *FuSeBMC v4* and *FuSeBMC v3* in a subset of tasks from the *Combinations* subcategory.

Task name	% coverage		Improvement
	FuSeBMC v4	*FuSeBMC v3*	Δ%
pals_lcr.3.1.ufo.BOUNDED-6.pals+Problem12_label01	94.90%	13.30%	81.60%
pals_lcr.3.1.ufo.UNBOUNDED.pals+Problem12_label02	84.40%	5.19%	79.21%
pals_lcr.4.1.ufo.BOUNDED-8.pals+Problem12_label04	94.10%	4.44%	89.66%
pals_lcr.4_overflow.ufo.UNBOUNDED.pals+Problem12_label05	94.00%	11.50%	82.50%
pals_lcr.5.1.ufo.UNBOUNDED.pals+Problem12_label05	86.20%	0.78%	85.42%
pals_lcr.5_overflow.ufo.UNBOUNDED.pals+Problem12_label09	94.00%	4.82%	89.18%
pals_lcr.6.1.ufo.BOUNDED-12.pals+Problem12_label09	92.90%	5.18%	87.72%
pals_lcr.7_overflow.ufo.UNBOUNDED.pals+Problem12_label09	92.60%	5.31%	87.29%
pals_lcr.8.ufo.UNBOUNDED.pals+Problem12_label08	78.20%	8.17%	70.03%
Average value	90.14%	6.52%	83.62%

sions achieved the highest possible result of 100% in *BitVectors*, *FuSeBMC v4* failed to advance the number of detected errors past 95% in *Recursive*, while *FuSeBMC v4* could not identify any errors in *DeviceDrivers* similarly to *FuSeBMC v3*. Also, *FuSeBMC v4* demonstrated a performance degradation of 2% in the *XCSP* subcategory.

Additionally, we compared the performance of *FuSeBMC v4* utilizing smart seeds with the version of *FuSeBMC v4* using only primary seeds (*i.e.* all zeros, all ones, and randomly chosen values) on the *ECA* (which stands for *event-condition-action* systems) subcategory in *Cover-Error* (where *FuSeBMC v4* demonstrated 28% improvement in comparison to *FuSeBMC v3* in the competition settings; see Table 4.3). It contains 18 test case generation tasks with C programs featuring input validation that involves relatively complex mathematical expressions. Such a program feature is notoriously difficult for the fuzzers whose initial seed is based on a random choice. Table 4.4 presents the results obtained by the versions of *FuSeBMC v4* with smart seeds and with primary seeds. It can be seen that smart seeds allow detecting 5 more bugs than the version of *FuSeBMC* using standard seeds.

Overall, the results presented in Tables 4.1 and 4.3 provide sufficient evidence that the evaluation objective **O1** has been achieved.

4.3.3.2 *FuSeBMC v4* vs state-of-the-art

FuSeBMC v4 achieved the overall first place at Test-Comp 2022, obtaining a score of 3003 out of 4236 with the closest competitor, VeriFuzz [159], scoring 2971 and significantly outperforming several state-of-the-art tools such as LibKluzzer [166], KLEE [151], CPAchecker [191] and Symbiotic [199] (see Table 4.5).

Table 4.6 demonstrates the code coverage capabilities of *FuSeBMC v4* in comparison to other state-of-the-art software testing tools. It can be seen that *FuSeBMC* achieved

Table 4.3: Comparison of the percentages of the successfully detected errors (per category and the category overall) by *FuSeBMC v4* and *FuSeBMC v3* in the *Error Coverage* category in TestComp-2022 and TestComp-2021, respectively.

Subcategory	% errors detected		Improvement
	FuSeBMC v4	*FuSeBMC v3*	$\Delta\%$
Arrays	99%	93%	6%
BitVectors	100%	100%	0%
ControlFlow	100%	25%	75%
ECA	72%	44%	28%
Floats	100%	97%	3%
Heap	95%	80%	14%
Loops	93%	83%	10%
ProductLines	100%	-	-
Recursive	95%	95%	0%
Sequentialized	95%	94%	1%
XCSP	88%	90%	-2%
BusyBox	15%	0%	15%
DeviceDrivers	0%	0%	0%
Cover-Error	81%	67%	14%

Table 4.4: Comparison of *FuSeBMC v4* performance with smart seeds and with standard seeds, where TRUE shows that the bug has been detected successfully, UNKNOWN means otherwise.

Task name	*FuSeBMC v4*	
	Smart Seeds	Primary Seeds
`eca-rers2012/Problem05_label00.yml`	TRUE	TRUE
`eca-rers2012/Problem06_label00.yml`	TRUE	TRUE
`eca-rers2012/Problem11_label00.yml`	TRUE	TRUE
`eca-rers2012/Problem12_label00.yml`	TRUE	TRUE
`eca-rers2012/Problem15_label00.yml`	TRUE	TRUE
`eca-rers2012/Problem16_label00.yml`	TRUE	UNKNOWN
`eca-rers2012/Problem18_label00.ymll`	TRUE	TRUE
`eca-rers2018/Problem10.yml`	TRUE	TRUE
`eca-rers2018/Problem11.yml`	TRUE	TRUE
`eca-rers2018/Problem12.yml`	TRUE	UNKNOWN
`eca-rers2018/Problem13.yml`	TRUE	UNKNOWN
`eca-rers2018/Problem14.yml`	TRUE	UNKNOWN
`eca-rers2018/Problem15.yml`	TRUE	UNKNOWN
`eca-rers2018/Problem16.yml`	UNKNOWN	UNKNOWN
`eca-rers2018/Problem17.yml`	UNKNOWN	UNKNOWN
`eca-rers2018/Problem18.yml`	UNKNOWN	UNKNOWN
`eca-programs/Problem101_label00.yml`	UNKNOWN	UNKNOWN
`eca-programs/Problem103_label32.yml`	UNKNOWN	UNKNOWN

Table 4.5: Test-Comp 2022 *Overall* Results. The table illustrates the scores obtained by all state-of-art tools overall, where we identify the best tool in bold.

Total # tasks	Tool											
	CMA-ES Fuzz	CoVeriTest v2.0.1	*FuSeBMC* v4.1.14	HybridTiger v1.9.2	KLEE v2.2	Legion v1.0	Legion/SymCC	LibKluzzer v1.0	PRTest v2.2	Symbiotic v9.0	Tracer-X v1.2.0	VeriFuzz v1.2.10
4236	382	2293	**3003**	1830	2125	787	-	2658	945	2367	1069	2971

first place with an overall score of 2104 out of 3460. *FuSeBMC* participated in all 16 subcategories, in 9 of which (*i.e. Arrays, BitVectors, Floats, Heap, Loops, ProductLines, Recursive, Combinations* and *Termination*) it achieved first place and in 6 of which it reached second place. The results presented in Table 4.6 allow us concluding that the evaluation objective **O2** has been achieved.

Table 4.6: *Cover-Branches* category results at *Test-COMP 2022*. The best score for each subcategory is highlighted in bold.

Subcategory	Total # tasks	Tool										
		CMA-ES Fuzz	CoVeriTest v2.0.1	*FuSeBMC* v4.1.14	HybridTiger v1.9.2	KLEE v2.2	Legion v1.0	Legion/SymCC	LibKluzzer v1.0	PRTest v2.2	Symbiotic v9.0	Tracer-X v1.2.0	VeriFuzz v1.2.10
Arrays	400	159	257	**328**	247	104	210	263	323	160	250	243	307
BitVectors	62	26	**49**	**49**	16	33	34	45	48	33	**49**	**49**	48
ControlFlow	67	5	36	43	14	25	17	41	40	6	43	39	**44**
ECA	29	0	6	11	2	7	3	3	10	2	10	8	**12**
Floats	226	53	113	**122**	92	19	72	62	103	46	55	56	119
Heap	143	23	100	**104**	84	93	81	78	**104**	49	98	101	101
Loops	727	211	574	**591**	467	380	357	542	575	359	538	544	587
ProductLines	263	19	**77**	**77**	56	74	70	74	**77**	48	69	**77**	**77**
Recursive	53	25	41	**45**	39	21	26	27	43	11	**45**	40	41
Sequentialized	103	0	79	90	58	35	1	43	75	11	51	57	**91**
XCSP	119	0	116	107	**119**	102	2	102	118	102	114	96	110
Combinations	671	63	238	**401**	167	196	179	224	292	79	338	295	351
BusyBox	75	0	12	25	6	21	0	0	24	15	19	18	**29**
DeviceDrivers	290	13	**60**	59	6	25	56	47	57	16	42	56	57
SQLite-MemSafety	1	0	0	0	0	0	0	0	0	0	0	0	0
Termination	231	143	212	**213**	195	118	168	145	204	60	179	192	202
Cover-Branches	3460	624	1860	**2104**	1406	1242	1033	1487	1990	896	1802	1746	2075

Similarly, Table 4.7 demonstrates demonstrates the error detecting abilities of *FuSeBMC v4*. In particular, *FuSeBMC* achieved first place in 9 subcategories (*i.e. Arrays, BitVectors, ControlFlow, Floats, Heap, Loops, ProductLines, Recursive* and *BusyBox*) reaching the first overall place in this category with the result of 628 out of 776 ($\sim 81\%$ success rate). Overall, the results show that *FuSeBMC* produces test cases that detect more security vulnerabilities in C programs than state-of-the-art tools, which successfully demonstrates that the evaluation objective **O3** has been achieved.

Table 4.7: *Cover-Error* category results at Test-Comp 2022. The best score for each subcategory is highlighted in bold.

Subcategory	Total # tasks	Tool										
		CMA-ES Fuzz	CoVeriTest v2.0.1	*FuSeBMC* v4.1.14	HybridTiger v1.9.2	KLEE v2.2	Legion v1.0	Legion/SymCC	LibKluzzer v1.0	PRTest v2.2	Symbiotic v9.0	Tracer-X v1.2.0	VeriFuzz v1.2.10
Arrays	100	0	73	**99**	69	89	67	-	97	37	74	0	**99**
BitVectors	10	0	8	**10**	6	9	0	-	**10**	5	8	0	**10**
ControlFlow	32	0	18	**32**	16	27	0	-	27	0	24	0	30
ECA	18	0	3	13	1	13	0	-	11	0	14	0	**15**
Floats	33	0	25	**33**	23	6	0	-	30	3	0	0	32
Heap	56	0	49	**53**	42	52	3	-	**53**	13	**53**	0	**53**
Loops	157	0	75	**146**	53	95	4	-	136	102	81	0	142
ProductLines	169	0	160	**169**	53	**169**	34	-	**169**	92	159	0	**169**
Recursive	20	0	7	**19**	5	16	0	-	17	1	17	0	16
Sequentialized	107	0	61	102	92	86	0	-	81	0	79	0	**104**
XCSP	59	0	50	52	52	37	0	-	5	0	41	0	**55**
BusyBox	13	0	0	**2**	0	1	0	-	0	0	0	0	**2**
DeviceDrivers	2	0	0	0	0	0	0	-	0	0	0	0	0
Cover-Error	776	0	0	**628**	355	500	57	-	528	145	463	0	623

4.4 | Conclusion

In this paper, we presented *FuSeBMC* v4, a test generator that relies on smart seed generation to improve the state-of-the-art in hybrid fuzzing and achieve high coverage for C programs. First, *FuSeBMC* analyses and injects goal labels into the given C program. Then, It ranks these goal labels according to the given strategy. After that, the engines are employed to produce smart seeds for a short time to use them later. Then, *FuSeBMC* coordinates between the engines and seed distribution by the Tracer. This Tracer will generally manage the tool to record the goals covered and deal with the transfer of information between the engines by providing a shared memory to harness the power and take advantage of the power of each engine. So that the BMC engine helps give the seed that makes the fuzzing engine not struggle with complex mathematical guards. Furthermore, Tracer evaluates test cases dynamically to convert high-impact cases into seeds for subsequent test fuzzing. This approach was evaluated by participating in the fourth international competition on software testing Test-Comp 2022. Our approach *FuSeBMC* showed its effectiveness by achieving first place in the *Cover-branches* category, first place in the *Cover-Error* category and first place in the *Overall* category. This performance is due to various features of our tool, the most important of which are the following. First the generation of smart seeds, which help harness the power of the fuzzers and allow them to fuzz deeper. Second, simplifying the target program by limiting the bounds of potentially infinite loops to avoid the path explosion problem and produce seeds faster. Third, utilizing static analysis to manage the mutation process by limiting the range of values input variables can take, speeding up the fuzzing process. In the future, we are planning on developing the tool to deal with different types of programs, such as multi-threaded programs. Furthermore, we work with the SCorCH project[5] to improve our performance in detecting memory safety bugs by incorporating SoftBoundCETS [211] into *FuSeBMC*.

[5] `https://scorch-project.github.io/about/`

Chapter 5

Conclusion & Future Work Directions

In this thesis, I have presented three contributions to automated software testing. Firstly, I have developed an approach that generates seeds that bypass complex guards to aid the fuzzer in exploring deeper into the target program. Furthermore, this approach reduces the burden of the fuzzer in mutation processes through static analysis. As part of this contribution, I proposed, developed, and evaluated a tracer subsystem, which coordinates and analyses the processes within the approach and the link between the employed techniques.

Secondly, I introduced our new fuzzer, which shares the general concept of the AFL fuzzer. This modified fuzzer has the advantage of performing a lightweight static program analysis in order to recognise input verification. Therefore, the condition code is parsed against the input variables to guarantee that only seeds satisfying the conditions are selected. This reduces our approach's reliance on a computationally expensive bounded model checker to discover high-quality seeds. Also, the modified fuzzer provides analysis of the target program and identifies potential infinite loops through heuristics. It then constrains these loops to speed up the fuzzing process, with the level of constraint increasing as rounds go. Furthermore, I introduced our new approach: a selective fuzzer that relies on learning from test cases produced by BMC and a modified fuzzer to generate new test cases that can successfully detect software vulnerabilities.

Finally, I developed and evaluated *FuSeBMC*, an automated testing tool that exploits the combination of BMC and fuzzing to test software and increase code coverage. *FuSeBMC* has demonstrated advantages in managing the use of resources and consequently reduces the consumption of BMC by exchanging data between engines in a manner that maximises the benefit of their cooperation. Also, it decreases the generation processes for execution paths that BMC might not reach or cause path explosion issues. As a result, *FuSeBMC* can reduce the negative impact, generate effective seeds, and avoid the path explosion issue. *FuSeBMC* has been evaluated comprehensively and competitively by participating in the most powerful and complex international software testing competition for two years, 2021 and 2022, in which our tool won six international awards.

FuSeBMC is currently the leading state-of-the-art software testing tool (Chapter 4 and Table 4.5. I also hypothesise that *FuSeBMC* is currently the strongest automated testing tool in the literature.

5.1 | Future Work Directions

This thesis's work opens numerous avenues of investigation, but I will discuss the most promising and intriguing ones here.

In light of the tool *FuSeBMC*'s accomplishments, it invited many projects to be considered and the prospect of adopting it in their projects. One of the promising new projects is SCorCH[1]. It is a collaboration between the University of Manchester and the University of Oxford, two leading automated verification and testing centres. The SCorCH project aims to use a range of modern formal analysis technology for reasoning about capability-based systems and verifying their security properties. SCorCH invited *FuSeBMC* to be part of the hybrid approach to protect against memory safety vulnerabilities. Therefore, I developed and evaluated *FuSeBMC* to provide additional software security properties for detecting vulnerabilities, such as memory leaks. Future efforts will be made to assist the hybrid technique in identifying temporal memory vulnerabilities. In particular, we intend to combine *FuSeBMC* and SoftBoundCETS [211, 212] in order to incorporate the program's memory statistics during execution to guide the fuzzer toward inputs that violate spatial memory safety. Our published paper [4] highlighted the current status and plans for the project.

In this paper [4], we conducted experiments on various open-source programs, e.g., bftpd, which is an FTP server for Unix systems. One of the primary objectives of our hybrid fuzzer was to develop it to verify open-source software and provide what enhances and contributes to this field on the ground. The results demonstrated that our approach works well with open-source software and achieves the intended results.

Future implementation of *FuSeBMC* will include a verification technique based on interval methods via contractors [6]. It will help to reduce the domains of variables representing the search space. *FuSeBMC*'s current implementation prevents the BMC engine from entering large loops, hence decreasing the probability of passing the path. Instead, reliance will be placed solely on the fuzzing engine. Thus, *FuSeBMC*'s involvement in a verification approach based on interval methods via contractor project will benefit both sides. *FuSeBMC* can provide analysis for the approach, while the verification approach reduces resource consumption and enhances the BMC engine's effectiveness.

[1] `https://scorch-project.github.io/`

5.2 | Concluding Remarks

Over the years, vulnerabilities increased almost continuously. 2017 saw its peak and severity due to the emergence of a ransomware attack [18]. Simultaneously, software and programs compete to provide several services and benefits. Expanding the software and increasing the number of required functions makes the software massive and susceptible to overlapping and conflicting functions; hence, vulnerabilities may occur. This required hardware and software developers to deliver periodic updates to fix software bugs and security vulnerabilities. In contrast, verifying a program and covering its source code has become increasingly complex and challenging. This thesis, in particular, proposed an efficient hybrid fuzzing for detecting vulnerabilities and achieving high coverage in software. The hybrid fuzzer combines BMC and fuzzing techniques to minimize each approach's drawbacks, verify deep paths in software, and reduce the consumption of resources. However, the development of reliable software is a complex problem, and software testing techniques still need to evolve to keep pace with the rapid expansion of software and programs.

Bibliography

[1] Kaled M Alshmrany, Rafael S Menezes, Mikhail R Gadelha, and Lucas C Cordeiro. “FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs”. In: *The 24th International Conference on Fundamental Approaches to Software Engineering (FASE)* 12649 (2020). https://doi.org/10.1007/978-3-030-71500-7_19, pp. 363–367 (page 12).

[2] Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. “FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in c programs”. In: *The International Conference on Tests and Proofs (TAP)*. https://doi.org/10.1007/978-3-030-79379-1_6. Springer. 2021, pp. 85–105 (pages 12, 23, 52).

[3] Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. “FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing”. In: *The 25th International Conference on Fundamental Approaches to Software Engineering (FASE)* 13241 (2022). https://doi.org/10.1007/978-3-030-99429-7_19, pp. 336–340 (page 12).

[4] Kaled Alshmrany, Ahmed Bhayat, Franz Brauße, Lucas Cordeiro, Konstantin Korovin, Tom Melham, Mustafa A. Mustafa, Pierre Olivier, Giles Reger, and Fedor Shmarov. “Position Paper: Towards a Hybrid Approach to Protect Against Memory Safety Vulnerabilities”. In: *IEEE Secure Development Conference*. Aug. 2022 (pages 12, 96).

[5] Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, Fedor Shmarov, Fatimah Aljaafari, and Lucas C Cordeiro. “FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis”. In: *The Formal Aspects of Computing Journal (FAC)* (2022) (pages 12, 24, 72).

[6] Mohannad Aldughaim, Kaled Alshmrany, Mohamed Mustafa, Lucas Cordeiro, and Alexandru Stancu. “Bounded Model Checking of Software Using Interval Methods via Contractors”. In: *arXiv preprint arXiv:2012.11245, Submitted in In-*

ternational Conference on Software Engineering (ICSE 2023) (2022) (pages 12, 96).

[7] Mamoona Humayun, Mahmood Niazi, NZ Jhanjhi, Mohammad Alshayeb, and Sajjad Mahmood. "Cyber security threats and vulnerabilities: a systematic mapping study". In: *Arabian Journal for Science and Engineering* 45.4 (2020), pp. 3171–3189 (page 15).

[8] Kaled M Alshmrany, Rafael S Menezes, Mikhail R Gadelha, and Lucas C Cordeiro. "FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs". In: *In 24th International Conference on Fundamental Approaches to Software Engineering (FASE)* 12649 (2020). https://doi.org/10.1007/978-3-030-71500-7_19, pp. 363–367 (pages 15, 54, 73, 74).

[9] Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. "FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in c programs". In: *International Conference on Tests and Proofs*. https://doi.org/10.1007/978-3-030-79379-1_6. Springer. 2021, pp. 85–105 (pages 15, 73–75, 84).

[10] Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C Cordeiro. "FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing". In: *In 25th International Conference on Fundamental Approaches to Software Engineering (FASE)* 13241 (2022). https://doi.org/10.1007/978-3-030-99429-7_19, pp. 336–340 (pages 15, 73, 74).

[11] Kaled M Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, Fedor Shmarov, Fatimah Aljaafari, and Lucas C Cordeiro. "FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis". In: *In the Formal Aspects of Computing Journal (FAC 2022)* (2022) (page 15).

[12] Tom CW Lin. "Financial weapons of war". In: *Minn. L. Rev.* 100 (2015), p. 1377 (page 15).

[13] J. Butler and Nils. *MWR Labs Pwn2Own 2013 Write-up – Kernel Exploit*. 2013. URL: https://labs.mwrinfosecurity.com/blog/2013/09/06/mwr-labs-pwn2own-2013-write-up---kernel-exploit/ (visited on 04/25/2022) (page 15).

[14] Daniel Moore. "Targeting technology: Mapping military offensive network operations". In: *2018 10th International Conference on Cyber Conflict (CyCon)*. IEEE. 2018, pp. 89–108 (page 15).

[15] Kathy Abbott, Stephen M Slotte, Donald K Stimson, et al. "The interfaces between flightcrews and modern flight deck systems". In: (1996) (page 15).

[16] Robert Shirey. *Internet security glossary*. 2000 (page 15).

[17] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1.1 (2018), pp. 1–13 (pages 15, 31).

[18] E Cert. "Wannacry ransomware campaign exploiting smb vulnerability". In: *Retrieved from Cert Europa Website: https://cert.europa.eu/static/SecurityAdvisories /2017/CERT-EU-SA201* (2017), pp. 7–012. URL: https://cert.europa.eu/static/SecurityAdvisories/2017/CERT-EU-SA2017-012.pdf (pages 15, 97).

[19] Wikipedia. *Wannacry ransomware attack (2017)*. 2017. URL: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack (visited on 04/25/2022) (page 15).

[20] Savita Mohurle and Manisha Patil. "A brief study of wannacry threat: Ransomware attack 2017". In: *International Journal of Advanced Research in Computer Science* 8.5 (2017), pp. 1938–1940 (page 16).

[21] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008 (pages 16, 17).

[22] Lionel C Briand. "A critical analysis of empirical research in software testing". In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE. 2007, pp. 1–8 (page 16).

[23] Lucas C. Cordeiro. "SMT-based bounded model checking of multi-threaded software in embedded systems". PhD thesis. University of Southampton, UK, 2011. URL: http://eprints.soton.ac.uk/186011/ (page 16).

[24] Barton P Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (1990), pp. 32–44 (pages 16, 31).

[25] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. "Many-core compiler fuzzing". In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 65–76 (pages 16, 34).

[26] Tao Zhang, Pan Wang, and Xi Guo. "A survey of symbolic execution and its tool klee". In: *Procedia Computer Science* 166 (2020), pp. 330–334 (page 16).

[27] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". In: *Communications of the ACM* 56.2 (2013), pp. 82–90 (page 16).

[28] Daniel Marjamäki. *Cppcheck: a tool for static C/C++ code analysis*. 2013 (pages 16, 20).

[29] Anjana Gosain and Ganga Sharma. “Static analysis: A survey of techniques and tools”. In: *Intelligent Computing and Applications*. Springer, 2015, pp. 581–591 (page 16).

[30] J Aaron Pendergrass, Susan C Lee, and C Durward McDonell. “Theory and practice of mechanized software analysis”. In: *Johns Hopkins APL Technical Digest* 32.2 (2013), pp. 499–508 (page 16).

[31] Yonit Kesten, Amit Klein, Amir Pnueli, and Gil Raanan. “A Perfecto Verification: combining model checking with deductive analysis to verify real-life software”. In: *International Symposium on Formal Methods*. Springer. 1999, pp. 173–194 (pages 16, 17).

[32] Jussi Lahtinen, Janne Valkonen, Kim Björkman, Juho Frits, Ilkka Niemelä, and Keijo Heljanko. “Model checking of safety-critical software in the nuclear engineering domain”. In: *Reliability Engineering & System Safety* 105 (2012), pp. 104–113 (page 17).

[33] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. “Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation”. In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 414–429 (page 17).

[34] Edmund M Clarke. “Model checking”. In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1997, pp. 54–56 (page 17).

[35] Hong Zhu, Patrick AV Hall, and John HR May. “Software unit test coverage and adequacy”. In: *Acm computing surveys (csur)* 29.4 (1997), pp. 366–427 (pages 17, 36).

[36] Youngjoon Kim and Jiwon Yoon. “MaxAFL: Maximizing Code Coverage with a Gradient-Based Optimization Technique”. In: *Electronics* 10.1 (2021), p. 11 (page 17).

[37] Raanan Grinwald, Eran Harel, Michael Orgad, Shmuel Ur, and Avi Ziv. “User defined coverage-a tool supported methodology for design verification”. In: *Proceedings 1998 Design and Automation Conference. 35th DAC.(Cat. No. 98CH36175)*. IEEE. 1998, pp. 158–163 (page 17).

[38] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. “Bounded Model Checking.” In: *Handbook of satisfiability* 185.99 (2009), pp. 457–481 (page 17).

[39] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. “SMT-based bounded model checking for embedded ANSI-C software”. In: *IEEE Transactions on Software Engineering* 38.4 (2011), pp. 957–974 (pages 17, 29).

[40] Phillipe A. Pereira, Higo F. Albuquerque, Isabela da Silva, Hendrio Marques, Felipe R. Monteiro, Ricardo Ferreira, and Lucas C. Cordeiro. “SMT-based context-bounded model checking for CUDA programs”. In: *Concurr. Comput. Pract. Exp.* 29.22 (2017). DOI: 10.1002/cpe.3934 (page 17).

[41] Lucas C. Cordeiro, Eddie Batista de Lima Filho, and Iury Valente de Bessa. “Survey on automated symbolic verification and its application for synthesising cyber-physical systems”. In: *IET Cyper-Phys. Syst.: Theory & Appl.* 5.1 (2020), pp. 1–24. DOI: 10.1049/iet-cps.2018.5006 (page 17).

[42] Lucas C. Cordeiro. “Exploiting the SAT Revolution for Automated Software Verification: Report from an Industrial Case Study”. In: *10th Latin-American Symposium on Dependable Computing, LADC 2021, Florianópolis, Brazil, November 22-26, 2021 - Companion Volume*. Brazilian Computing Society, 2021, pp. 8–9. DOI: 10.5753/ladc.2021.18531 (pages 17, 18).

[43] Patrice Godefroid. “Fuzzing: Hack, art, and science”. In: *Communications of the ACM* 63.2 (2020), pp. 70–76 (pages 18, 31, 34, 35, 53).

[44] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” In: *NDSS*. 2016, pp. 1–16 (pages 18, 33, 34, 42, 45, 46, 49, 73, 82).

[45] Ravindra Metta, Raveendra Kumar Medicherla, and Samarjit Chakraborty. “BMC+ Fuzz: Efficient and effective test generation”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022, pp. 1419–1424 (pages 19, 46).

[46] Nikolaos S Papaspyrou. “A formal semantics for the C programming language”. In: *Doctoral Disseration. National Technical University of Athens. Athens (Greece)* 15 (1998) (pages 19, 20).

[47] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988 (page 20).

[48] Steve Oualline. *Practical C programming*. " O’Reilly Media, Inc.", 1997 (page 20).

[49] Macario Polo, Pedro Reales, Mario Piattini, and Christof Ebert. “Test automation”. In: *IEEE software* 30.1 (2013), pp. 84–89 (page 20).

[50] Hrushikesha Mohanty, JR Mohanty, and Arunkumar Balakrishnan. *Trends in software testing*. Springer, 2017 (page 20).

[51] *Clang Documentation*. `http://clang.llvm.org/docs/index.html`. [Online; accessed August-2019]. 2015 (pages 21, 55, 75, 79).

[52] George Candea, Stefan Bucur, and Cristian Zamfir. "Automated software testing as a service". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 155–160 (page 21).

[53] Edward Kit and Susannah Finzi. *Software testing in the real world: improving the process*. ACM Press/Addison-Wesley Publishing Co., 1995 (page 21).

[54] Rudolf Ramler and Klaus Wolfmaier. "Economic perspectives in test automation: balancing automated and manual testing with opportunity cost". In: *Proceedings of the 2006 international workshop on Automation of software test*. 2006, pp. 85–91 (page 21).

[55] Hubert Garavel, Radu Mateescu, and Irina Smarandache. "Parallel state space construction for model-checking". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2001, pp. 217–234 (page 21).

[56] American fuzzy lop, `https://lcamtuf.coredump.cx/afl/`. 2021 (pages 22, 34, 39, 45, 49, 75, 82).

[57] Antonia Bertolino. "Software testing research: Achievements, challenges, dreams". In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 85–103 (pages 25, 28).

[58] Bill Hetzel. *The complete guide to software testing*. QED Information Sciences, Inc., 1988 (page 25).

[59] Edsger Wybe Dijkstra et al. *Notes on structured programming*. 1970 (page 25).

[60] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011 (pages 26, 36).

[61] Jeremy Morse. "Expressive and efficient bounded model checking of concurrent software". PhD thesis. University of Southampton, 2015 (pages 26, 29).

[62] Frederick P Brooks Jr. *The mythical man-month: essays on software engineering*. Pearson Education, 1995 (page 26).

[63] Paul C Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2013 (pages 26, 27).

[64] Fokke Heikamp. "Gray-box Network Fuzzing using Genetic Algorithms and Code Coverage". MA thesis. University of Twente, 2018 (page 26).

[65] Jihyun Lee, Sungwon Kang, and Danhyung Lee. "Survey on software testing practices". In: *IET software* 6.3 (2012), pp. 275–282 (page 28).

[66] IEEE Computer Society Professional Practices Committee et al. “SWEBOK: Guide to the Software Engineering Body of Knowledge, 2004 version”. In: *IEEE Computer Society* (2004) (page 28).

[67] Robert L Glass, Ross Collard, Antonia Bertolino, James Bach, and Cem Kaner. “Software testing and industry needs”. In: *IEEE Software* 23.4 (2006), pp. 55–57 (pages 28, 50).

[68] Natalia Juristo, Ana M Moreno, and Wolfgang Strigel. “Guest editors’ introduction: Software testing practices in industry”. In: *IEEE software* 23.4 (2006), pp. 19–21 (pages 28, 50).

[69] Antonia Bertolino. “Software testing research and practice”. In: *International Workshop on Abstract State Machines*. Springer. 2003, pp. 1–21 (pages 28, 50).

[70] Ram Chillarege. *Software testing best practices*. IBM Thomas J. Watson Research Division, 1999 (pages 28, 50).

[71] Ossi Taipale and Kari Smolander. “Improving software testing by observing practice”. In: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. 2006, pp. 262–271 (page 28).

[72] Natalia Juristo, Ana M Moreno, and Sira Vegas. “Reviewing 25 years of testing technique experiments”. In: *Empirical Software Engineering* 9.1 (2004), pp. 7–44 (page 28).

[73] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. “Software test automation in practice: empirical observations”. In: *Advances in Software Engineering* 2010 (2010) (page 28).

[74] Mats Grindal, Jeff Offutt, and Jonas Mellin. “On the testing maturity of software producing organizations”. In: *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART’06)*. IEEE. 2006, pp. 171–180 (page 28).

[75] Vahid Garousi and Tan Varma. “A replicated survey of software testing practices in the Canadian province of Alberta: What has changed from 2004 to 2009?” In: *Journal of Systems and Software* 83.11 (2010), pp. 2251–2262 (page 28).

[76] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007 (pages 28, 30).

[77] Alan Mathison Turing et al. “On computable numbers, with an application to the Entscheidungsproblem”. In: *J. of Math* 58.345-363 (1936), p. 5 (page 28).

[78] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. “Symbolic model checking without BDDs”. In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer. 1999, pp. 193–207 (page 29).

[79] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009 (pages 29, 41).

[80] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, A Biere, M Heule, H van Maaren, and T Walsh. “Handbook of satisfiability”. In: *Satisfiability modulo theories* 185 (2009), pp. 825–885 (pages 29, 30, 41).

[81] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. “Bounded model checking of software using SMT solvers instead of SAT solvers”. In: *International Journal on Software Tools for Technology Transfer* 11.1 (2009), pp. 69–83 (page 29).

[82] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A tool for checking ANSI-C programs”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2004, pp. 168–176 (pages 29, 41, 49).

[83] Florian Merz, Stephan Falke, and Carsten Sinz. “LLBMC: Bounded model checking of C and C++ programs using a compiler IR”. In: *International Conference on Verified Software: Tools, Theories, Experiments*. Springer. 2012, pp. 146–161 (pages 29, 41).

[84] Dirk Beyer, Matthias Dangl, and Philipp Wendler. “Boosting k-induction with continuously-refined invariants”. In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 622–640 (page 29).

[85] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers Inc., 1997. ISBN: 1558603204 (page 29).

[86] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. “SMT-based bounded model checking for embedded ANSI-C software”. In: *IEEE Transactions on Software Engineering* 38.4 (2012), pp. 957–974. ISSN: 00985589. DOI: 10.1109/TSE.2011.59. URL: http://www.esbmc.org (page 29).

[87] Saul A. Kripke. “Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi”. In: *Mathematical Logic Quarterly* 9.5-6 (1963), pp. 67–96. ISSN: 15213870. DOI: 10.1002/malq.19630090502 (page 29).

[88] Lucas Cordeiro. “SMT-based bounded model checking for multi-threaded software in embedded systems”. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 2. IEEE. 2010, pp. 373–376 (page 30).

[89] T Ball and SK Rajamani. *SLIC: A Specication Language for Interface Checking (of C), Microsoft Research, Technical Report, MSR-TR-2001-21*. 2002 (page 30).

[90] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. "ESBMC 5.0: an industrial-strength C model checker". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 888–891 (pages 30, 55).

[91] Omar M Alhawi, Herbert Rocha, Mikhail R Gadelha, Lucas C Cordeiro, and Eddie Batista. "Verification and refutation of C programs based on k-induction and invariant inference". In: *International Journal on Software Tools for Technology Transfer* 23.2 (2021), pp. 115–135 (page 31).

[92] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. "{SMS} of Death: From Analyzing to Attacking Mobile Phones on a Large Scale". In: *20th USENIX Security Symposium (USENIX Security 11)*. 2011 (page 31).

[93] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-based whitebox fuzzing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2008, pp. 206–215 (page 31).

[94] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with code fragments". In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 445–458 (pages 31, 34).

[95] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated whitebox fuzz testing." In: *NDSS*. Vol. 8. 2008, pp. 151–166 (page 31).

[96] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. "Fitness-guided path exploration in dynamic symbolic execution". In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009, pp. 359–368 (page 31).

[97] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007 (pages 31, 34, 39).

[98] Michael Howard and Steve Lipner. *The security development lifecycle*. Vol. 8. Microsoft Press Redmond, 2006 (page 31).

[99] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. "Taming compiler fuzzers". In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 197–208 (page 33).

[100] Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. "Tree-based methods for classifying software failures". In: *15th International Symposium on Software Reliability Engineering*. IEEE. 2004, pp. 451–462 (page 33).

[101] Michał Zalewski. *American Fuzzy Lop (AFL) fuzzer*. 2015. URL: http://lcamtuf.coredump.cx/afl/ (visited on 05/12/2019) (page 33).

[102] Team Microsoft. *!exploitable*. 2013. URL: https://www.microsoft.com/security/blog/2013/06/13/exploitable-crash-analyzer-version-1-6/ (visited on 06/17/2022) (page 33).

[103] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. "The art, science, and engineering of fuzzing: A survey". In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2312–2331 (page 33).

[104] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. Baltimore, MD, USA: USENIX Association, 2018, pp. 745–761. ISBN: 9781931971461 (pages 33, 45, 46, 49).

[105] Munea, Tewodros Legesse, Lim, Hyunwoo, Shon, and Taeshik. "Network protocol fuzz testing for information systems and applications: a survey and taxonomy". In: *Multimedia Tools and Applications* 75.22 (2016), pp. 14745–14757 (page 34).

[106] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing". In: *Queue* 10.1 (2012), pp. 20–27 (pages 34, 43, 73).

[107] Kostya Serebryany. "libFuzzer–a library for coverage-guided fuzz testing". In: *LLVM project* (2015) (pages 34, 40, 45, 49).

[108] Gustavo Grieco, Martıén Ceresa, and Pablo Buiras. "Quickfuzz: An automatic random fuzzer for common file formats". In: *ACM SIGPLAN Notices* 51.12 (2016), pp. 13–20 (page 34).

[109] D. Vyukov. *Syzkaller—Linux Kernel Fuzzer*. 2016. URL: https://github.com/google/syzkaller (visited on 06/19/2022) (pages 34, 40).

[110] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing." In: *NDSS*. Vol. 17. 2017, pp. 1–14 (pages 34, 39).

[111] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing mayhem on binary code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394 (page 34).

[112] Miller and et al. Barton. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Tech. rep. UW-Madison, 1995 (pages 34, 44, 46, 54).

[113] Pedram Amini and Aaron Portnoy. *Sulley fuzzing framework*. 2010 (page 34).

[114] *Fuzzing with spike*. `https://samsclass.info/127/proj/p18-spike.htm`. [Online; accessed August-2022]. 2015 (page 34).

[115] Michael Eddington. "Peach fuzzing platform". In: *Peach Fuzzer* 34 (2011) (pages 34, 40).

[116] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. "A systematic review of fuzzing techniques". In: *Computers & Security* 75 (2018), pp. 118–137 (page 35).

[117] Barton P Miller, Gregory Cooksey, and Fredrick Moore. "An empirical study of the robustness of macos applications using random testing". In: *Proceedings of the 1st international workshop on Random testing*. 2006, pp. 46–54 (page 35).

[118] Patrice Godefroid. "Random testing for security: blackbox vs. whitebox fuzzing". In: *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. 2007, pp. 1–1 (page 35).

[119] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. "Fuzzing: State of the art". In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1199–1218 (page 35).

[120] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices* 40.6 (2005), pp. 190–200 (page 35).

[121] Hadi Hemmati. "How Effective Are Code Coverage Criteria?" In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, pp. 151–156. DOI: 10.1109/QRS.2015.30 (page 36).

[122] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. "Code coverage at Google". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 955–963 (page 36).

[123] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems". In: *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE. 2015, pp. 560–564 (page 36).

[124] Baoying Lou and Jia Song. "A Study on Using Code Coverage Information Extracted from Binary to Guide Fuzzing." In: *International Journal of Computer Science and Security (IJCSS)* 14.5 (2020), pp. 200–210 (pages 36, 37).

[125] André Baresel, Mirko Conrad, Sadegh Sadeghipour, and Joachim Wegener. “The interplay between model coverage and code coverage”. In: *Proc. EuroCAST*. 2003, pp. 1–14 (pages 36, 37).

[126] Hadi Hemmati. “How effective are code coverage criteria?” In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE. 2015, pp. 151–156 (page 36).

[127] Raul Santelices and Mary Jean Harrold. “Efficiently monitoring data-flow test coverage”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 343–352 (page 37).

[128] Cem Kaner. “Software negligence and testing coverage”. In: *Proceedings of STAR* 96 (1996), p. 313 (page 37).

[129] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. “Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1475–1482 (pages 38, 43).

[130] Paul E Black and Irena Bojanova. “Defeating Buffer Overflow: A Trivial but Dangerous Bug”. In: *IT professional* 18.6 (2016), pp. 58–61 (page 39).

[131] Sen Zhang, Jingwen Zhu, Ao Liu, Weijing Wang, Chenkai Guo, and Jing Xu. “A Novel Memory Leak Classification for Evaluating the Applicability of Static Analysis Tools”. In: *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*. IEEE. 2018, pp. 351–356 (page 39).

[132] Willy Jimenez, Amel Mammar, and Ana Cavalli. “Software vulnerabilities, prevention and detection methods: A review1”. In: *Security in model-driven architecture* 215995 (2009), p. 215995 (page 39).

[133] El Habib Boudjema, Christèle Faure, Mathieu Sassolas, and Lynda Mokdad. “Detection of security vulnerabilities in C language applications”. In: *Security and Privacy* 1.1 (2018), e8 (page 39).

[134] US-CERT. *Understanding Denial-of-Service Attacks | CISA*. 2009. URL: `https://www.us-cert.gov/ncas/tips/ST04-015` (page 39).

[135] Cisco. *Cisco IOS XE Software Cisco Discovery Protocol Memory Leak Vulnerability*. 2018. URL: `https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20180926-cdp-memleak` (page 39).

[136] Barton, James H., Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. “Fault injection experiments using FIAT”. In: *IEEE Trans. Comput.* 39.4 (1990), pp. 575–582 (page 39).

[137] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. “Directed greybox fuzzing”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2329–2344 (pages 39, 40, 75, 82).

[138] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. “Skyfire: Data-driven seed generation for fuzzing”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 579–594 (page 40).

[139] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-based greybox fuzzing as markov chain”. In: *IEEE Transactions on Software Engineering* 45.5 (2017), pp. 489–506 (page 40).

[140] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. “Smart greybox fuzzing”. In: *IEEE Transactions on Software Engineering* 47.9 (2019), pp. 1980–1997 (page 40).

[141] Rundong Li, HongLiang Liang, Liming Liu, Xutong Ma, Rong Qu, Jun Yan, and Jian Zhang. “GTFuzz: Guard Token Directed Grey-Box Fuzzing”. In: *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE. 2020, pp. 160–170 (page 40).

[142] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. “Instrim: Lightweight instrumentation for coverage-guided fuzzing”. In: *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*. 2018 (page 40).

[143] Sang Kil Cha, Maverick Woo, and David Brumley. “Program-adaptive mutational fuzzing”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 725–741 (page 40).

[144] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. “Optimizing seed selection for fuzzing”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 861–875 (page 40).

[145] Serge Gorbunov and Arnold Rosenbloom. “Autofuzz: Automated network protocol fuzzing framework”. In: *IJCSNS* 10.8 (2010), p. 239 (page 40).

[146] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. “JBMC: A bounded model checking tool for verifying Java bytecode”. In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 183–190 (page 41).

[147] Lucas C. Cordeiro, Daniel Kroening, and Peter Schrammel. “JBMC: Bounded Model Checking for Java Bytecode - (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 11429. LNCS. Springer, 2019, pp. 219–223 (page 41).

[148] Mikhail R Gadelha, Rafael Menezes, Felipe R Monteiro, Lucas C Cordeiro, and Denis Nicole. “ESBMC: Scalable and Precise Test Generation based on the Floating-Point Theory:(Competition Contribution)”. In: *Fundamental Approaches to Software Engineering* 12076 (2020), p. 525 (pages 41, 49, 65, 75).

[149] Kevin Atkinson, Matthew Flatt, and Gary Lindstrom. “ABI compatibility through a customizable language”. In: *Proceedings of the ninth international conference on Generative programming and component engineering*. 2010, pp. 147–156 (page 41).

[150] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. “Enhancing symbolic execution with veritesting”. In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 1083–1094 (page 41).

[151] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224 (pages 41, 45, 49, 54, 63, 88).

[152] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. “BAP: A binary analysis platform”. In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 463–469 (page 41).

[153] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. “BitBlaze: A new approach to computer security via binary analysis”. In: *International conference on information systems security*. Springer. 2008, pp. 1–25 (page 41).

[154] Joxan Jaffar, Vijayaraghavan Murali, Jorge A Navas, and Andrew E Santosa. “TRACER: A symbolic execution tool for verification”. In: *International Conference on Computer Aided Verification*. Springer. 2012, pp. 758–766 (page 41).

[155] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed automated random testing”. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223 (pages 41, 49).

[156] Song, JaeSeung, Cristian Cadar, and Peter Pietzuch. “SYMBEXNET: testing network protocol implementations with symbolic execution and rule-based specifications”. In: *IEEE TSE* 40.7 (2014), pp. 695–709 (page 41).

[157] Sasnauskas, Raimondas, Philipp Kaiser, Russ Lucas Jukić, and Klaus Wehrle. "Integration testing of protocol implementations using symbolic distributed execution". In: *ICNP*. IEEE. 2012, pp. 1–6 (page 41).

[158] Dirk Beyer and Marie-Christine Jakobs. "CoVeriTest: Cooperative Verifier-Based Testing." In: *FASE*. 2019, pp. 389–408 (page 41).

[159] Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and R Venkatesh. "VeriFuzz: Program aware fuzzing". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2019, pp. 244–249 (pages 42, 54, 63, 88).

[160] Dirk Beyer. "Second Competition on Software Testing: Test-Comp 2020." In: *FASE*. 2020, pp. 505–519 (page 42).

[161] Youngjoon Kim and Jiwon Yoon. "Maxafl: Maximizing code coverage with a gradient-based optimization technique". In: *Electronics* 10.1 (2020), p. 11 (page 42).

[162] Brian S Pak. "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution". In: *School of Computer Science Carnegie Mellon University* (2012) (pages 42, 44, 46).

[163] Yannic Noller, Rody Kersten, and Corina 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, pp. 322–332 (page 42).

[164] Corina S Păsăreanu and Neha Rungta. "Symbolic PathFinder: symbolic execution of Java bytecode". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 179–180 (page 43).

[165] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. "Learning to fuzz from symbolic execution with application to smart contracts". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 531–548 (page 43).

[166] Hoang M Le. "LLVM-based Hybrid Fuzzing with LibKluzzer (Competition Contribution)." In: *FASE*. 2020, pp. 535–539 (pages 43, 54, 63, 88).

[167] Hoang M. Le. "LLVM-based Hybrid Fuzzing with LibKluzzer (Competition Contribution)". In: *Fundamental Approaches to Software Engineering*. Ed. by Heike Wehrheim and Jordi Cabot. Cham: Springer International Publishing, 2020, pp. 535–539. ISBN: 978-3-030-45234-6 (page 43).

[168] Caroline Lemieux and Koushik 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, pp. 475–485 (page 43).

[169] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang 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, pp. 61–64 (page 43).

[170] Tao Zhang, Yu Jiang, Runsheng Guo, Xiaoran Zheng, and Hui Lu. “A survey of hybrid fuzzing based on symbolic execution”. In: *Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies*. 2020, pp. 192–196 (pages 44, 45).

[171] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. “TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 497–512 (pages 44, 46).

[172] Jun Cai, Shangfei Yang, Jinquan Men, and Jun He. “Automatic software vulnerability detection based on guided deep fuzzing”. In: *2014 IEEE 5th International Conference on Software Engineering and Service Science*. IEEE. 2014, pp. 231–234 (pages 44, 46).

[173] Dong Fangquan, Dong Chaoqun, Zhang Yao, and Lin Teng. “Binary-oriented hybrid fuzz testing”. In: *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE. 2015, pp. 345–348 (pages 45, 46).

[174] XF Xie, XH Li, X Chen, GZ Meng, and Y Liu. “Hybrid testing based on symbolic execution and fuzzing. Ruan Jian Xue Bao”. In: *J. Softw* 30 (2019), pp. 3071–3089 (pages 45, 46).

[175] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. “Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1613–1627 (pages 45, 46).

[176] Dirk Beyer, Po-Chun Chien, and Nian-Ze Lee. “Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator”. In: () (page 46).

[177] M. Rodriguez, M. Piattini, and C. Ebert. “Software Verification and Validation Technologies and Tools”. In: *IEEE Software* 36.2 (2019), pp. 13–24. DOI: 10.1109/MS.2018.2883354 (page 53).

[178] *Airbus issues software bug alert after fatal plane crash*. the Guardian https://tinyurl.com/xw67wtd9. [Online; accessed March-2021]. 2015 (page 53).

[179] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. "Software vulnerability discovery techniques: A survey". In: *2012 fourth international conference on multimedia information networking and security*. IEEE. 2012, pp. 152–156 (page 53).

[180] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic". In: *25 Years of Model Checking*. Ed. by Orna Grumberg and Helmut Veith. 2008, pp. 196–215 (page 53).

[181] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. "Model checking and the state explosion problem". In: *LASER Summer School* 7682 LNCS.2005 (2012), pp. 1–30 (page 53).

[182] Wen Shameng and et al. Feng Chao. "Testing Network Protocol Binary Software with Selective Symbolic Execution". In: *CIS*. IEEE. 2016, pp. 318–322 (page 53).

[183] Dirk Beyer. *3rd Competition on Software Testing (Test-Comp 2021)*. 2021 (pages 53, 54, 67, 68).

[184] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394 (page 54).

[185] JP Faria. "Inspections, revisions and other techniques of software static analysis". In: *Software Testing and Quality, Lecture* 9 (2008) (page 54).

[186] Qin, S, and K. "Lift: A low-overhead practical information flow tracking system for detecting security attacks". In: *MICRO'06*. IEEE. 2006, pp. 135–148 (page 54).

[187] Saahil Ognawala, Fabian Kilger, and Alexander Pretschner. "Compositional fuzzing aided by targeted symbolic execution". In: *arXiv preprint arXiv:1903.02981* (2019) (page 54).

[188] Armin Biere. "Bounded Model Checking". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 457–481. DOI: 10.3233/978-1-58603-929-5-457. URL: https://doi.org/10.3233/978-1-58603-929-5-457 (pages 54, 61).

[189] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. "SMT-Based Bounded Model Checking for Embedded ANSI-C Software". In: *IEEE Trans. Software Eng.* 38.4 (2012), pp. 957–974. DOI: 10.1109/TSE.2011.59. URL: https://doi.org/10.1109/TSE.2011.59 (page 54).

[190] Dirk Beyer. “Second Competition on Software Testing: Test-Comp 2020”. In: *FASE*. Ed. by Heike Wehrheim and Jordi Cabot. Vol. 12076. LNCS. Springer, 2020, pp. 505–519 (page 54).

[191] Dirk Beyer and M Erkan Keremoglu. “CPAchecker: A tool for configurable software verification”. In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 184–190 (pages 54, 63, 88).

[192] Herbert Rocha, Raimundo Barreto, and Lucas C. Cordeiro. “Hunting Memory Bugs In C Programs With Map2Check”. In: *Tools And Algorithms For The Construction And Analysis Of Systems*. Vol. 9636. LNCS. 2016, pp. 934–937 (page 55).

[193] Herbert Rocha, Rafael Menezes, Lucas C. Cordeiro, and Raimundo S. Barreto. “Map2Check: Using Symbolic Execution and Fuzzing - (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 12079. LNCS. 2020, pp. 403–407 (page 55).

[194] Mikhail R Gadelha, Felipe Monteiro, Lucas Cordeiro, and Denis Nicole. “ESBMC v6. 0: Verifying C Programs Using k-Induction and Invariant Inference”. In: *International Conference on TACAS*. Springer. 2019, pp. 209–213 (pages 55, 75).

[195] Dirk Beyer and Thomas Lemberger. “TestCov: Robust test-suite execution and coverage measurement”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1074–1077 (pages 56, 61, 85).

[196] Bruno Cardoso Lopes and Rafael Auler. *Getting started with LLVM core libraries*. Packt Publishing Ltd, 2014 (page 57).

[197] Dirk Beyer. “Status Report on Software Testing: Test-Comp 2021”. In: *Proc. FASE. LNCS* 12649 () (pages 62, 63).

[198] Dirk Beyer. “Software Verification: 10th Comparative Evaluation (SV-COMP 2021)”. In: *Proc. TACAS (2). LNCS* 12652 () (pages 62, 85).

[199] Chalupa, Marek, Novák, J, Strejček, and Jan. “Symbiotic 8: Parallel and targeted test generation (competition contribution)”. In: *FASE*. Vol. 12649. LNCS. 2021 (pages 63, 88).

[200] Mikhail R Gadelha, Lucas C Cordeiro, and Denis A Nicole. “An efficient floating-point bit-blasting API for verifying C programs”. In: *Software Verification*. Springer, 2020, pp. 178–195 (page 65).

[201] Aina Niemetz, Mathias Preiner, and Armin Biere. “Boolector 2.0”. In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 53–58. DOI: 10.3233/sat190101. URL: https://doi.org/10.3233/sat190101 (page 65).

[202] Dirk Beyer and Philipp Wendler. “CPU Energy Meter: A tool for energy-aware algorithms engineering”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2020, pp. 126–133 (page 68).

[203] Lucas C. Cordeiro. “SMT-based bounded model checking for multi-threaded software in embedded systems”. In: *International Conference on Software Engineering*. ACM, 2010, pp. 373–376 (page 70).

[204] Phillipe A. Pereira, Higo F. Albuquerque, Hendrio Marques, Isabela da Silva, Celso B. Carvalho, Lucas C. Cordeiro, Vanessa Santos, and Ricardo Ferreira. “Verifying CUDA programs using SMT-based context-bounded model checking”. In: *Annual ACM Symposium on Applied Computing*. Ed. by Sascha Ossowski. ACM, 2016, pp. 1648–1653 (page 70).

[205] Tavis Ormandy Chris Evans Matt Moore. URL: https://security.googleblog.com/2011/08/fuzzing-at-scale.html (visited on 10/04/2021) (page 73).

[206] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657 (page 73).

[207] Xianya Mi, Baosheng Wang, Yong Tang, Pengfei Wang, and Bo Yu. “SHFuzz: Selective Hybrid Fuzzing with Branch Scheduling Based on Binary Instrumentation”. In: *Applied Sciences* 10.16 (2020), p. 5449 (page 73).

[208] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: *CGO*. San Jose, CA, USA, 2004, pp. 75–88 (page 79).

[209] D. Beyer. “Advances in Automatic Software Testing: Test-Comp 2022”. In: *Proc. FASE*. LNCS 13241. Springer, 2022 (page 85).

[210] Dirk Beyer. “Third Competition on Software Testing: Test-Comp 2021”. In: *Fundamental Approaches to Software Engineering*. Vol. 12076. LNCS. 2021, pp. 505–519 (page 85).

[211] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. “SoftBound: Highly compatible and complete spatial memory safety for C”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 245–258 (pages 94, 96).

[212] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. “CETS: compiler enforced temporal safety for C”. In: *Proceedings of the 2010 International Symposium on Memory Management*. 2010, pp. 31–40 (page 96).

Appendix A

Extensions

A.1 | Appendix

A.1.1 | Artifact

We have set up a zenodo entry that contains the necessary materials to reproduce the results given in this paper: `https://doi.org/10.5281/zenodo.4710599`. Also, it contains instructions to run the tool.

A.1.2 | Tool Availability

FuSeBMC contents are publicly available in our repository in GitHub under the terms of the MIT License. *FuSeBMC* provides, besides other files, a script called *fusebmc.py*. In order to run our *fusebmc.py* script, one must set the architecture (i.e., 32 or 64-bit), the competition strategy (i.e., k-induction, falsification, or incremental BMC), the property file path, and the benchmark path. *FuSeBMC* participated in the 3rd international competition, Test-Comp 21, and met all the requirements each tool needs to meet to qualify and participate. The results in our paper are also available on the Test-Comp 21 website. Finally, instructions for building *FuSeBMC* from the source code are given in the file README.md in our GitHub repository, including the description of all dependencies.

A.1.3 | Tool Setup

FuSeBMC is available to download from the link.[1] To generate test cases for a C program a command of the following form is run:

```
fusebmc.py [-a {32, 64}] [-p PROPERTY_FILE]
           [-s {kinduction,falsi,incr,fixed}] [<file>.c]
```

[1] `https://doi.org/10.5281/zenodo.4710599`

where `-a` sets the architecture (either 32- or 64-bit), `-p` sets the property file path, `-s` sets the strategy (one of `kinduction`, `falsi`, `incr`, or `fixed`) and <file>.c is the C program to be checked. *FuSeBMC* produces the test cases in the XML format.

Appendix **B**

FuSeBMC in Open-Source Software

B.1 | Appendix

B.1.1 | Open-Source Software

Implementation of a project on the ground is crucial and indicative of its success. In the field of software testing, open-source systems are referred to here. Open-source provides developers with foundations to build upon, so they do not need to start from scratch. Additionally, developers can write code, report an issue, and solve it themselves, which is challenging in a proprietary system. Herein lies the significance of evaluating our approach to open-source software. In addition, we desired to accomplish the objectives of performing such experiments, the most essential of which was the development of our hybrid fuzzer to verify open-source software and deliver what improves and contributes to this field on the ground.

B.1.2 | Experiments of *FuSeBMC*

We conducted experiments on benchmarks taken from the 2021 memory safety category of SV-COMP, which contain various open-source applications, e.g., Amazon AWS C commons library and bftpd, which is an FTP server for Unix systems. Table B.1 shows the results of our experiments where our approach successfully got 3 scores out of 5 in the FTP server and archived 141 scores out of 174 in Amazon AWS programs. The outcomes demonstrated that our hybrid fuzzer *FuSeBMC* works effectively with open-source software and achieves the desired outcomes.

Table B.1: *FuSeBMC*'s results on open-source software.

Software	LOC (avg)	Result	Property	TIME (avg)	# of Programs
FTP server	727	3	Memory Safety	2s	5
AWS program	7174	141	Unreachability of Error Function	30s	174