COVID-19 impact statement

The COVID-19 pandemic has reshaped many aspects of our lives, including the pursuit of knowledge and academic endeavors. As a Ph.D. student, my journey through this difficult period has been marked by unique challenges and adaptations that have significantly impacted my academic, personal, and professional life. Specifically, at the beginning of my second year, I found myself alone abroad while my husband and little girl were in a different country, greatly affecting my mental health. Being alone and in lockdown without the people I love and care about was incredibly difficult. Additionally, the uncertainty surrounding the progression of the pandemic and its impact on my academic schedules and plans has been a source of anxiety and stress.

After several months, I could finally return to my home country and settle with my family. Adapting to working remotely and having remote meetings in different time zones was challenging.

Black-Box Cooperative Verification Framework For Finding Software Vulnerabilities in Concurrent Programs

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

2023

Fatimah Khalid Aljaafari
Department of Computer Science

Contents

Word count: 30000

List of figures

List of tables

List of publications

Published and accepted papers:

F. Aljaafari, R. Menezes, E. Manino, F. Shmarov, M. A. Mustafa, and L. Cordeiro, “Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs,” *IEEE Access*, vol. 10, pp. 121 365–121 384, 2022. DOI: `10.1109/ACCESS.2022.3223359`

F. Aljaafari, F. Shmarov, E. Manino, R. Menezes, and L. Cordeiro, “EBF 4.2: Black-Box cooperative verification for concurrent programs (competition contribution),” in *Proc. TACAS (2)*, ser. LNCS, Springer, 2023

Submitted / In progress / co-authored papers:

K. Alshmrany, M. Aldughaim, A. Bhayat, F. Shmarov, F. Aljaafari, and L. Cordeiro, “FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis,” *The Formal Aspects of Computing Journal (FAC)*,

Abbreviations

BMC	Bounded Model Checking
SAT	Satisfiability Solver
SMT	Satisfiability Modulo Theories
PUT	Program Under Test
GBF	Gray-Box Fuzzing
OpenGBF	Open-source Gray-Box Fuzzing
EBF	Ensembles of Bounded Model Checking with Fuzzing
ESBMC	Efficient SMT-based Bounded Model Checker
AFL	American Fuzzy Lop
IR	Intermediate Representation
CFG	Control-Flow Graph
SSA	Static Single Assignment
AST	Abstract Syntax Tree
SV-COMP	Software Verification Competition
SAGE	Scalable Automated Guided Execution
SMC	Stateless Model Checking
POR	Partial Order Reduction
VFG	Value-Flow Graph
IoT	Internet of Things

Abstract

Detecting software vulnerabilities in concurrent programs poses a significant challenge due to the extensive state-space exploration required, with interleavings growing exponentially as the number of program threads and statements increases. Combining different verification and testing techniques, at least in theory, achieves better results than individual use. In theory, combining different verification and testing techniques to detect software vulnerabilities can improve results compared to using them individually. We propose and evaluate *EBF* (Ensembles of Bounded Model Checking with Fuzzing) – a technique that combines Bounded Model Checking (BMC) and Gray-Box Fuzzing (GBF) to detect software vulnerabilities in concurrent programs. Given the lack of publicly available GBF tools for concurrent programs, we first propose *OpenGBF*, a new open-source concurrency-aware gray-box fuzzer that explores different thread interleavings by instrumenting the program under test (PUT) with random delays. Then, we develop a cooperative framework that combines the BMC tool and *OpenGBF* as follows. On the one hand, we force the BMC tool to provide seed values to *OpenGBF* by injecting additional vulnerabilities (error statements) in the PUT, thus increasing the likelihood of *OpenGBF* executing paths guarded by complex mathematical expressions. On the other hand, we aggregate the results of the BMC and *OpenGBF* tools in the framework using a decision matrix, thus improving the accuracy of *EBF*. We evaluate the performance of *EBF* compared to state-of-the-art pure BMC tools and demonstrate that it can produce up to 14.9% more correct witnesses than the corresponding BMC tools alone. Moreover, we show the effectiveness of *OpenGBF* by illustrating its capability of finding 41.9% of the vulnerabilities within our evaluation suite, while non-concurrency-aware GBF tools can only find 0.55%. Finally, thanks to our concurrency-aware *OpenGBF*, *EBF* successfully detects a data race in the open-source *wolfMqtt* library and reproduces known bugs in several other real-world concurrent programs, which shows its efficacy in finding vulnerabilities in real-world concurrent software.

Declaration of originality

I hereby confirm that no portion of the work referred to in the 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 statement

Acknowledgements

Firstly, I would like to express my heartfelt gratitude to my supervisor, Dr. Lucas Cordeiro, for his constant encouragement, support, and guidance during my Ph.D. journey. I cannot imagine a better supervisor; he always supported me and was instrumental in both my research and personal life. He has also been generous in helping me rebuild confidence and overcome difficulties. I would also like to thank my co-supervisor, Dr. Mustafa A. Mustafa, for his guidance, assistance, and invaluable feedback.

Secondly, I would like to express my gratitude to my collaborators, Rafael Menezes, Edoardo Manino, and Fedor Shmarov, for their guidance, support, and valuable discussions we had throughout our collaboration. Working alongside them not only enhanced my professional journey but also facilitated a comfortable and inspiring working environment.

Thirdly, I would like to extend my heartfelt gratitude to my wonderful family for making my journey to success possible and rewarding. My special thanks go to my dear husband, Abdulateef Alnouaim, who has been my unwavering source of encouragement, motivating me to reach for my dreams and offering his support whenever needed. I would also like to express my deep appreciation to my parents, Khalid and Mona, for their constant love and endless support. Their guidance and belief in me have been instrumental in my achievements. To my grandparents, whose prayers and words of wisdom have guided my path to success. To my caring brothers and sisters, thank you for your continuous encouragement. To my beautiful little daughter, Norah, who has been my constant motivation and source of inspiration. Her patience during my busiest moments is a testament to her remarkable understanding and support.

Finally, I would like to thank King Faisal University for awarding me the scholarship that made my education possible. I also sincerely thank my home country for generously funding my educational journey, enabling me to pursue my dreams and acquire the knowledge and skills that have shaped my future.

.

Chapter 1

Introduction

Concurrency is becoming increasingly widespread in present-day software systems due to the performance benefits provided by multi-core hardware [4][5][1]. Examples of using concurrency include online banking, auto-pilots, computer games, and railway ticket reservation systems [6]. Such programs are relatively complex, posing unique challenges compared to sequential programs. The main reason for such complexity is that it handles multiple threads sharing the same resources, each with a different execution.

For example, in online banking [7], the issue may occur when two or more users access their account and transfer money to the same recipient simultaneously, and both transactions are processed concurrently. Hence, the bank system should ensure that the funds are deducted correctly from the users' accounts and transferred to the recipient account without any conflicts. So, in this case, the system should protect the read and write operations; failure to do so will result in software vulnerabilities [7]. Another real-world example is the Heartbleed[1] vulnerability (CVE-2014-0160) in the OpenSSL cryptographic library, allowing the attacker to read the memory and obtain sensitive information from the affected system. This information should be protected.

Software vulnerabilities are the flaws or weaknesses within a program that allows an attacker to perform malicious activities and compromise the security and integrity of the system [8]. Concurrent programs also contain software vulnerabilities that an attacker can exploit [9]. Concurrent programs involve the simultaneous operation of multiple processes or threads on shared computing resources [10]. Consequently, such programs can feature vulnerabilities that are specific to concurrency, such as data races, deadlocks, and thread leaks [11], as well as vulnerabilities that are common to sequential programs, like invalid memory accesses and memory leaks [12]. Some software vulnerabilities are more severe than others. For instance, out-of-bounds (an example of invalid memory access) was ranked as the top issue in the 2022 MITRE ranking[2], while data races were placed 22nd. However, ensuring the correctness and safety of such systems or software is crucial [13].

[1]https://heartbleed.com/

[2]`https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html`

Software testing is essential to software development; it executes a program or system to find flaws or errors [14]. The main objectives of software testing are to ensure software quality, functionality, and performance and to produce reliable results [15]. The process involves executing exhaustive test cases, which is challenging due to the knowledge required of the systems' behaviour. Despite the significant resources devoted to software testing, many existing software still encounter software vulnerabilities [16]. In the case of concurrent programs, the different possible threads' interleavings cause the program's execution to be nondeterministic [17]. Consequently, some vulnerabilities are difficult to find because they only happen in a specific thread order, making testing concurrent programs an inherently difficult task compared with sequential programs [5].

Due to this complex nature of concurrent programs [18], manual testing of such programs is not always adequate [19]; automated testing and verification are often used [20]. In this context, there is a wide range of different automated techniques for finding vulnerabilities in concurrent programs [21][22]. One of the techniques is *abstract interpretation* [19], a program analysis technique based on computing fixed-point lattices over abstract semantics of the system. The classical rule of signs for multiplication can be seen as an example of the technique where the abstract domain is the sign of a number; one can compute the sign of multiplication without doing any concrete computation (e.g., two negative numbers will result in a positive). Another technique is *data-flow analysis* [23][24], which is the generalization of the process of computing flow equations for all nodes of a Control-Flow Graph (CFG) [25]. This technique is commonly applied for compiler optimizations (as it is polynomial for bit-vector problems [26]) [27]. It can be used for more precise safety properties (e.g., privacy leaks). Among these, two techniques have substantially developed in recent years: Bounded Model Checking (BMC) and fuzzing [28].

BMC [29] is a formal verification technique that searches for program violations by unwinding the program to a given bound k. If no property violation is detected, then the value of K is increased until the violation is detected, the verification problem becomes intractable, or the predefined upper bound is reached. Although many different bounded model checkers [30]–[34] have been successfully used to verify sequential and concurrent programs, BMC has several fundamental drawbacks. Specifically, BMC frequently encounters difficulties in achieving high path coverage (especially for concurrent programs) and reaching deep statements within the code due to the state-space explosion and its reliance on Boolean Satisfiability (SAT) [35] or Satisfiability Modulo Theories (SMT) solvers [36].

The term "fuzz" was introduced by Barton Miller in the 1990s [37]. Miller *et al.* conducted an experiment where they tested 90 programs by running them with random inputs. They found that over 24% of these programs crashed. As a result, they called the term "fuzz" for the programs that generated these random inputs. Since then, "fuzz testing" or "fuzzing" has become the name of the technique used to find vulnerabilities through extensive testing with numerous test cases or inputs. Fuzzing [22] is an automated software testing technique used to detect software vulnerabilities, bugs, and undefined behaviour, mainly for sequential programs. It involves inputs repeatedly generated and provided to a Program Under Test (PUT); these inputs often start with some initial guess (seed values). Then, the PUT is executed for each given sequence of input values; its behaviour is examined for abnormalities, such as crashes or failures [38]. Fuzzing has several advantages, including its relatively straightforward integration with existing testing frameworks, high scalability, and, most importantly, the ability to explore deep execution paths that are not as costly as in BMC [39]. However, fuzzing has some drawbacks; it often suffers from low branch coverage since the input generation is based on random mutations [40]. It typically happens when a program features conditional statements with complex conditions (e.g., input validation functions). Consequently, providing a good initial seed value for the fuzzing process is essential. Another challenge facing the fuzzing process is finding vulnerabilities in concurrent programs [41] because the existing fuzzing techniques do not focus on thread interleavings that affect execution states.

In the past, efforts have been made toward developing a combined verification technique that harnesses the strengths of both BMC and fuzzing. For example, Ognawala et al. [42] developed a hybrid framework that combines fuzzing and symbolic execution to improve function coverage and apply it to general-purpose software. Alshmrany et al. [43] developed a technique that uses BMC to guide a fuzzer in analyzing sequential C programs. Chowdhury et al. [44] developed a technique that improves the seed generation of Gray-Box Fuzzing (GBF) by using BMC as a constraint solver to find execution paths within complex blocks of code. Nevertheless, given the current knowledge in software verification, no techniques harness BMC and fuzzing to verify concurrent programs. *The question of whether combining BMC and fuzzing improves bug finding in concurrent programs remains open.*

The challenge in answering this fundamental question is threefold. First, while many open-source BMC tools exist in the literature, all existing concurrency fuzzers are (at least partially) closed-source. As a result, using any of these concurrency-aware fuzzers requires a major reproducibility effort. Second, combining BMC and fuzzing techniques for concurrency is not straightforward, given the lack of existing baselines. Third, BMC and fuzzing are very different techniques; their cooperation inside the framework has to be carefully coordinated. We take inspiration from a trend in automated software verification called *cooperative verification* [45], [46]. In this context, the main idea is to implement a communication interface between different tools (i.e., a common information exchange format), which allows the exchange of partial results (artifacts). By doing so, we can harness the strengths of different verification techniques and solve more complex problems [47]–[49].

1.1 Research motivation and challenges

Generally, BMC and fuzzing detect software vulnerabilities in fundamentally different ways. As a result, it is natural to ask whether combining the two techniques can lead to better coverage of the search space and whether combining them can lead to finding more vulnerabilities than other existing approaches. More precisely, in this Ph.D. thesis, we ask the following fundamental research question:

Research Question. *Does combining bounded model checkers and gray-box fuzzers discover more concurrency vulnerabilities, and does it do it faster than either approach on their own?*

In addressing this question, we are confronted with many practical design challenges, the solution of which is central to this PhD thesis:

1. **Concurrency-aware gray-box fuzzer.** Although there have been recent attempts to fuzz concurrent programs, no mature, fully open-source tool exists. As a result, designing such a tool is an important step toward answering our research question. In doing so, we aim to draw from the lessons learned in the existing literature and implement our *OpenGBF*. This tool is representative of a concurrency-aware gray-box fuzzer [1].

   However, developing such a concurrency-aware fuzzer tool to fuzz concurrent programs poses sub-challenges, which are outlined as follows; this challenge and the following sub-challenges are tackled in Chapter 3:

   (a) **Thread interleavings:** concurrent programs can have different thread interleavings, which leads to different behaviour. Therefore, such a tool requires exploring different thread interleavings and being thread-aware.

   (b) **Data races and atomicity violations:** concurrent programs can be exposed to different vulnerabilities besides the bugs encountered in sequential programs. Thus, the concurrency-aware fuzzer should detect these concurrency-related bugs (e.g., data race, deadlock, and thread leak) beside the memory-related bugs (e.g., memory leak, buffer overflow) by generating test inputs that can trigger such vulnerabilities.

   (c) **Scalability:** the large number of thread interleavings in concurrent programs makes fuzzing computationally exhausting. Hence, limiting the number of active threads during fuzzing is important to reduce the computation overhead.

   (d) **Non-Deterministic behaviours:** reproducing a specific bug in a concurrent program is challenging due to the non-deterministic behaviour of concurrent programs. Thus, the fuzzer requires an approach to reproduce the exact input and interleavings that lead to the bug.

2. **Input seed generation.** Fuzzers rely on initial seed values as their starting points for generating (mutating) test cases or inputs. These seeds are critical because they provide the fuzzer with a foundation to initiate its testing process. Furthermore, seed values are essential in guiding the fuzzer's exploration of the program's search space. However, when exploring complex path conditions, such as (`if(x*x -2*x +1 == 0)`), fuzzers often struggle to explore such program paths effectively. Consequently, providing good initial seeds that cover most of the program search space is essential to improve the fuzzer bug-finding capabilities. This challenge is tackled in Section 4.3.1.2.

3. **Aggregating BMC and GBF results.** By implementing a cooperative framework of different tools, we risk them returning conflicting results. The main reason is that BMC depends on abstractions of program execution states and symbolic execution (see Section 2.3.1), whereas the fuzzer tests concrete inputs and execution schedules. Hence, when the two techniques disagree, generating a bug trace from these tools is vital, which allows us to obtain a more accurate and comprehensive understanding of the bugs. Also, we can make an informed choice about the final verification result. This challenge is tackled in Section 4.3.1.4.

4. **Resource allocation trade-off.** The main disadvantage of using a cooperative framework of different tools is that they all share the same computational resources. We must decide how many resources to allocate to each tool for a program. Generally, this decision depends not only on the problem at hand but also on the partial results we obtain from the tools in the cooperative framework. Specifically, in our cooperative framework, we combined the tools sequentially, and since the fuzzer requires initial seed values, it waits for the seeds to be extracted from the BMC, and accordingly, the fuzzing process starts. We discuss a strategy to optimize our cooperative framework in Section 4.3.2.

1.2 Scope of the thesis

The scope of this Ph.D. thesis is exploring, developing, and evaluating a novel technique that combines two well-known software verification techniques, Bounded Model Checking (BMC) and fuzzing, to efficiently address the challenges introduced when verifying concurrent programs (i.e., multi-threaded programs written in C language). The main objective is to harness the strength of both BMC and fuzzing to improve the bug-detection capabilities and the verification process in general for concurrent programs. Mainly, this thesis aims to:

- **Explore the challenges of verifying concurrent programs:** we explore the challenges associated with verifying concurrent programs. We present the limitations of existing verification techniques and show the need for a technique that can address these challenges posed by concurrent programs.

- **Propose a cooperative verification framework:** we build a framework that combines dissimilar techniques (BMC and fuzzing) to obtain the strength of both techniques to improve bug-finding capabilities.
- **Design and implement *OpenGBF*:** we provide a detailed explanation of *OpenGBF*. We provide the implementation challenges of fuzzing concurrent programs and how *OpenGBF* addresses them.
- **Design and implement *EBF* tool:** we provide a detailed explanation of *EBF* framework. We explain the cooperation and the implementation of the communication interface between BMC and *OpenGBF*.
- **Address the combination challenges:** we address the challenges of combining two different techniques. We explore how we provide seeds to the fuzzer, divide the time allocation for each technique, and how we incorporate the final result.
- **Evaluate *EBF* efficiency:** we evaluate *EBF* on different benchmark suites to show its effectiveness in finding concurrency and memory-related bugs. We also compare *EBF* with different state-of-the-art BMC tools to illustrate its bug-finding capabilities, scalability, and correctness.
- **Evaluate *EBF* on real-world programs:** we apply *EBF* framework to real-world concurrent programs, highlighting its scalability of finding bugs in large programs. We will also show *EBF* ability to reproduce known bugs in real-world concurrent programs.
- **Future work:** we conclude each chapter with ideas to improve concurrent program verification, specifically focusing on improving *EBF*.

1.3 Contributions

The main contribution of this Ph.D. thesis is developing, implementing, and evaluating a cooperative framework to detect concurrency and memory-related vulnerabilities in concurrent programs. In this respect, this thesis makes three major novel contributions.

First, we develop a new fully open-source state-of-the-art concurrency-aware gray-box fuzzer called *OpenGBF* [50]. Our main approach is instrumenting the PUT with random delays obtained from a random number generator whose seed value is controlled by the fuzzer. This method allows us to find different thread interleavings and explore deep execution paths. Additionally, our fuzzer can generate crash reports containing the entire program execution path, including the thread interleavings that lead to the crash.

Second, we present *EBF* – Ensembles of Bounded Model Checking with Fuzzing. This technique combines the strengths of BMC in resolving complex conditional guards with the flexibility of our concurrency-aware gray-box fuzzer *OpenGBF*. *EBF* introduce initial seeds by repeatedly introducing error statements for the BMC to extract such seeds from their counterexample for the fuzzer. These extracted seeds help the fuzzer mutate and generate more effective seeds to explore different paths. Furthermore, *EBF* incorporates a result decision matrix for coping with the potential conflict verdicts produced by the tools in the cooperative. In addition, *EBF* distributes the available computational resources between the tools to improve its bug-finding capabilities efficiently.

Finally, we present our evaluation of *EBF*, focusing on its bug-finding capabilities, scalability, and correctness. First, we demonstrate that combining BMC and fuzzing improves verification results compared to either technique applied separately. More precisely, *EBF* enhances the bug-finding capabilities of all state-of-the-art concurrent BMC tools considered in this thesis by up to 14.9%. Similarly, *EBF* can detect 24.2% of the vulnerabilities in our evaluation benchmark suit. In contrast, the state-of-the-art gray-box fuzzer *AFL++* can only detect 0.55%. Second, we demonstrate the scalability of *EBF* in detecting vulnerabilities in real-world programs by applying *EBF* to the *wolfMQTT* open-source library that implements the *MQTT* messaging protocol, and we detect a data race bug. We reported the bug to the developers of the *wolfMQTT* library; they fixed the bug after reporting it in June 2021. Also, *EBF* successfully reproduced known bugs in several real-world concurrent programs (i.e., *pfscan* [51], *bzip2smp* [52] and *swarm 1.1* [53]). Lastly, we reported that the bug-finding capabilities of *EBF* are stable across a wide range of parameter values. Specifically, we conduct a comparison experiment along three different axis: the distribution time between the BMC tool and *OpenGBF*, the maximum delay introduced by *OpenGBF*, and the maximum number of threads allowed by *OpenGBF*. Our findings show a large sweet spot of parameter values that allows *EBF* to find nearly 75-fold more bugs than the worst setting.

1.4 Thesis structure

The remainder of this thesis is arranged in the following structure (the thesis's visual structure is also illustrated in Figure 1.1).

Chapter 2 presents the background of the common software vulnerabilities and static analysis techniques, precisely Bounded Model Checking (BMC) and automated software testing techniques, precisely Gray-box fuzzing. It also presents the related work of the BMC, Fuzzing, and hybrid techniques, including their advantages and disadvantages.

Chapter 3 provides the details about the design of our concurrency-aware gray-box fuzzer *OpenGBF*, which include the challenges of fuzzing concurrent programs and the design choices made to address these challenges. Specifically, it describes each component inside *OpenGBF* framework, which involves instrumenting the PUT by inserting functions that control and monitor the thread interleavings in concurrent programs. The content of this chapter is derived from [1].

Chapter 4 describes *EBF*, which combines BMC and *OpenGBF*. It contains *EBF* framework that includes four stages, which precisely describe the cooperation between the concurrency-aware fuzzer and BMC tool. The stages include seed generation for helping the fuzzer, aggregating the final results, and generating the bug report. It also describes the CPU time allocation inside *EBF*. The content of this chapter is derived from [1] and [2].

Chapter 5 presents *OpenGBF* design choices. Also, it presents the implementation details of *EBF*, including the programming language used, how to run the tool, and the flags that can be used, and also describes the tool's output. In addition, it describes the use of *OpenGBF* alone without using *EBF*.

Chapter 6 analyses the results of both *EBF* in general and *OpenGBF* in specific. It describes the benchmark suit used for the evaluation. Also, it evaluates three versions of *EBF* on *Concurrency Safety* category from SV-COMP 2022. Furthermore, it evaluates *EBF* on the real-world concurrent program and provides an evaluation of *EBF* optimization settings. The content of this chapter is adapted from [1].

Chapter 7 summarizes the research contributions, highlighting the significance of the proposed framework in software verification for concurrent programs, the evaluation, and the future work in concurrent programs' verification.

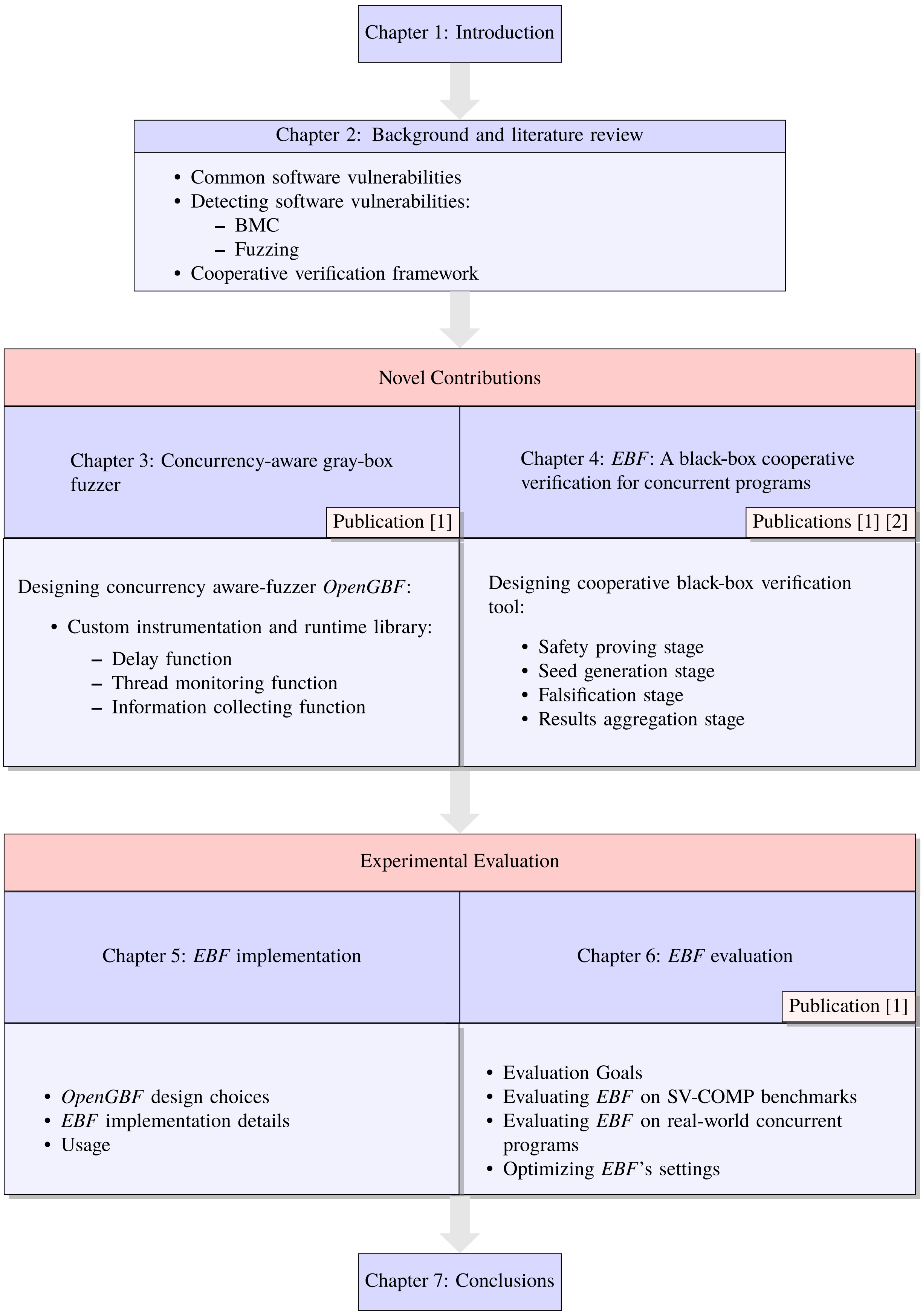

Chapter 1: Introduction
Chapter 2: Background and literature review
• Common software vulnerabilities
• Detecting software vulnerabilities:
– BMC
– Fuzzing
• Cooperative verification framework
Novel Contributions
Chapter 3: Concurrency-aware gray-box fuzzer
Publication [1]
Chapter 4: *EBF*: A black-box cooperative verification for concurrent programs
Publications [1] [2]
Designing concurrency aware-fuzzer *OpenGBF*:
• Custom instrumentation and runtime library:
– Delay function
– Thread monitoring function
– Information collecting function
Designing cooperative black-box verification tool:
• Safety proving stage
• Seed generation stage
• Falsification stage
• Results aggregation stage
Experimental Evaluation
Chapter 5: *EBF* implementation
Chapter 6: *EBF* evaluation
Publication [1]
• *OpenGBF* design choices
• *EBF* implementation details
• Usage
• Evaluation Goals
• Evaluating *EBF* on SV-COMP benchmarks
• Evaluating *EBF* on real-world concurrent programs
• Optimizing *EBF*'s settings
Chapter 7: Conclusions

Figure 1.1. Thesis structure.

Chapter 2

Background and literature review

2.1 Chapter introduction

This chapter introduces the core concepts behind the verification of concurrent programs, especially the background information behind designing *EBF* framework. It includes the common software vulnerabilities *EBF* supports, the cooperative framework architecture, LLVM Pass instrumentation, Bounded model checking, and Gray-box fuzzing. Also, it presents the related work in BMC, fuzzing, hybrid techniques, and other techniques for finding vulnerabilities in concurrent programs.

2.2 Common software vulnerabilities

Software vulnerabilities, also known as software flaws, security vulnerabilities, and bugs (in this thesis, we will refer to them as either software vulnerabilities or bugs), have become the root cause of threats in cybersecurity [49]. Software vulnerabilities are any flaws or weaknesses in the system design or implementation that an attacker can exploit, resulting in a breach of the system's security policy and causing severe damage. Software vulnerabilities can be classified into different types of bugs; we will focus on two types. The first type, *memory*-related bugs, occur in concurrency programs because specific program inputs with specific threads interleavings can trigger these bugs [54], [55]. The second type is *Concurrency*-related bugs, which occur because of the non-determinism behaviour of the thread interleavings [16]. However, predefined assertions also check if the specific condition holds during the test execution. Both of these bugs (concurrency bugs and memory-related in concurrency context) may lead the program to produce abnormal behaviours or unexpected hangs.

2.2.1 Memory-related vulnerabilities

Memory-related vulnerabilities refer to security flaws or weaknesses in programs caused by improper handling or manipulation of a program's memory. Which can be classified as the following:

Invalid memory accesses comprise a large family of memory safety violations, which include accessing memory outside the bounds of the intended buffer for either reading (potentially exposing some sensitive data to the attacker) or writing (resulting in memory corruptions or injections of executable code), accessing previously freed memory (known as "user-after-free"), or pointing to a memory location after deallocation (known as "Dangling Pointer") [56].

Uninitialized variables occur when a program attempts to access a declared variable but is not initialized with a value (e.g., `int x; int sum=x+5;`), resulting in obtaining garbage data or sensitive information in the memory left from other processes. Also, dereferencing invalid pointers or *NULL* pointers can be considered a type from the uninitialized variable vulnerabilities family, which can cause the program to crash or exit unexpectedly [57].

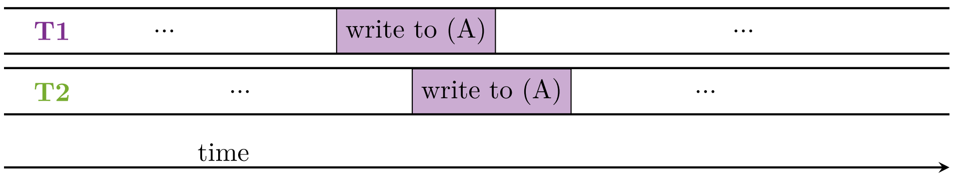
Memory leak occurs when a program incorrectly handles memory allocations so that memory no longer needed is not released. Double free can be considered a type of memory leak vulnerability when a program attempts to free a memory location already released. This may lead to memory exhaustion, resulting in the system hanging or crashing [58].

2.2.2 Concurrency-related vulnerabilities

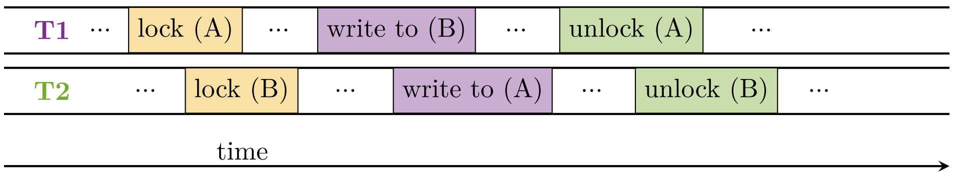
Concurrency-related vulnerabilities refer to security flaws or weaknesses in programs caused by mis-synchronization of multiple threads execution [59]. Which can be classified as the following:

Data race occurs when the program execution leads to an undesired behaviour because of a specific sequence and/or timing of the instructions executed by each thread. For example, when one thread modifies the shared memory without acquiring a lock first, it results in memory corruption when another thread attempts to update the exact memory location (see Figure 2.1a).

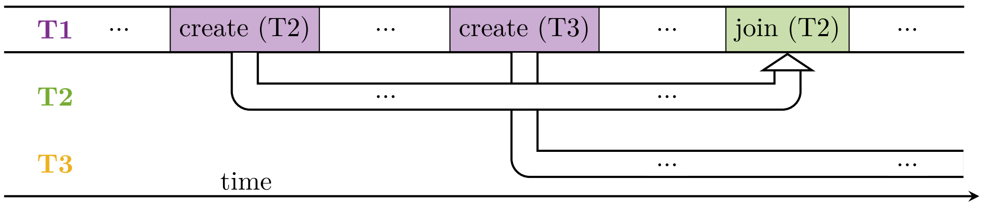
Deadlock occurs when the program is not in the final state and cannot progress to any other state. For example, when a thread fails to release a lock after accessing the shared memory, the program becomes stuck because the other thread is waiting to access the shared memory (see Figure 2.1b).



(a) The program contains **data race**, which occurs when T1 and T2 are trying to write to the memory region A simultaneously with no synchronization between the operations.



(b) The program ends in a **deadlock** since T1 acquires a lock for the memory region A and then tries to write to the memory region B. At the same time, T2 performs the opposite, acquiring a lock for B and attempting to write to A. This will result in both threads waiting indefinitely for each other to release their corresponding locks before the program's execution can continue.



(c) T3 is a source of **thread leak** since, unlike T2, it terminates but never joins T1. Hence, over time, the number of idle threads rises, leading to the possibility of resource exhaustion.

Figure 2.1. Concurrency-related vulnerabilities.

Thread leak occurs when a thread finishes and never joins the calling thread, therefore never releasing the occupied resources, a type of vulnerability specific to multi-threaded programs (see Figure 2.1c).

2.2.3 User-defined properties

User-defined properties refer to the conditions specified by the developer, defining the expected state the program should not reach during its execution (reachability statement). For example, an assert statement (e.g., `assert(0);`) is inserted at a specific point in the program to check whether this statement is reached during the program's execution or not.

In this Ph.D. thesis, our cooperative framework, called *EBF*, can detect all the vulnerabilities specified in this section.

2.3 Detecting software vulnerabilities

Detecting software vulnerabilities is crucial for ensuring the security and reliability of software programs. Therefore, several techniques and approaches are commonly used to find software vulnerabilities in concurrent programs, including Bounded Model Checking (BMC), symbolic execution, fuzzing, data-flow analysis, and machine learning. In our *EBF* framework, we employ two powerful software testing and verification techniques: BMC and fuzzing.

2.3.1 Bounded model checking

Bounded model checking (BMC) is a formal verification technique successfully employed in software and hardware verification over the past decades [29]. BMC operates with the underlying program's mathematical model, represented as a finite state transition system. It analyzes the model's behaviour up to a finite positive bound k. It determines whether the specified safety property (e.g., absence of data races, deadlocks, buffer overflows, assertion violations, etc.) holds.

In brief, BMC symbolically executes the program up to the specified bound k and encodes all the obtained traces C with the given safety property P as an SAT/SMT formula [60] $C \wedge \neg P$. A decision procedure often called an *automated theorem prover* or *solver* checks the generated formula and provides a verdict on its satisfiability. If the formula is satisfiable, it indicates a violation of the safety property, along with the generation of a witness (counterexample [61]). In contrast, if the formula is unsatisfiable, it proves the program is safe within the provided bound k.

In state-of-the-art bounded model checker, the program under test is modeled as a state transition system, which is constructed by extracting its behaviour from the control-flow graph (CFG) [62]. This graph is then translated into a static single assignment (SSA) form. In the case of multi-threaded programs, each thread is represented as a CFG in which nodes represent control points, and edges represent transitions (or program statements). Each transition is enabled if the condition guards are true and the associated process is at the corresponding control point. For example, Figure 2.2 presents the CFG of two threads (T(A) and T(B)). Each thread includes two transitions (A0, A1 and B0, B1), the guards ($x > 2$ and $x > 3$), and the control points that determine whether a transition is enabled or disabled (T(A)0, T(A)1 and T(B)0, T(B)1).

A transition system, denoted as $M = (S, R, S_0)$, represents an abstract machine consisting of a set of states S (where $S_0 \subseteq S$ represents the set of initial states) and transitions R between states. For each $\gamma \in R$, $\gamma \subseteq S \times S$. A state $s \in S$ comprises the program counter value pc and

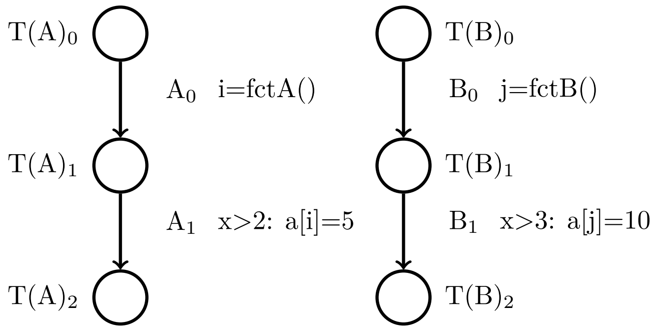


Figure 2.2. Control-flow graph of two threads (T_A and T_B). T represents the node (program location), and A and B represent the edges (program instructions).

all the values assigned to the program variables. The initial state, represented as s_0, assigns the initial program location of the CFG to the pc. Each transition is represented by $\gamma = (s_i, s_{i+1})$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$. This formula captures the constraints on the values of pc and the program variables. Given a transition system M, a property ϕ, and a bound k, BMC unrolls the transition system k times and translates it into a verification condition ψ, where ψ is satisfiable if ϕ contains a counterexample with a depth less than or equal to k. The BMC procedure can then be formulated as follows:

$$\psi^k = \underbrace{I(s_0) \wedge \bigwedge_{i=0}^{k-1} \gamma(s_i, s_{i+1})}_{\text{constraints}} \wedge \overbrace{P(s_k)}^{\text{property}} \tag{2.1}$$

In Formula 2.1, I represents the function defining the set of initial states of M, $\gamma(s_j, s_{j+1})$ represents the transition relation of M at time steps j and $j+1$, and $P(s_k)$ represents an LTL property ϕ at step k. Thus, the formula $\bigwedge_{i=0}^{k-1} \gamma(s_i, s_{i+1})$ represents all executions of M with lengths less than or equal to k. $P(s_k)$ is derived from the safety property being checked and represents the condition that a bounded execution of M with a length of k or less would violate it. The SMT solver then evaluates ψ_k for satisfiability. If ψk is satisfiable, the SMT solver provides an SSA that satisfies it. Consequently, a counterexample is created using the values extracted from the program variables in this assignment. A counterexample for a property ϕ consists of a sequence of states $s_0, s_1, \ldots, s_k$, where $s_0 \in S_0$ and $s_i \in S$ for $0 \leq i < k$, along with $\gamma(s_i, si+1)$. Suppose ψ_k is unsatisfiable. In that case, it indicates that the state is reachable within a length of k or less, suggesting that the property is not violated.

The constraints and properties are encoded as two quantifier-free formulas derived from ψk. The first formula, denoted as C, represents the constraints ($I(s_0) \wedge \bigwedge i = 0^{k-1} \gamma(s_i, s_{i+1})$), while the second formula, denoted as P, represents the properties ($P(s_k)$) as shown in Formula 2.1. Then, the SMT solver evaluates the formula as $C \wedge \neg P$.

Considering a concurrency environment, the BMC procedure can be context-bounded [63].

Context-bounded means limiting the number of threads up to a certain threshold. In other words, a schedule can yield a thread up to the threshold before it needs to be executed again. This algorithm takes advantage of the fact that most concurrency bugs in real applications are shallow, requiring only a few context switches to expose them [63].

BMC suffers from several drawbacks. One major challenge is the state-space explosion, which occurs when the verification depth increases. This challenge becomes even more severe for multi-threaded programs due to the need to explore the combined search space of thread interleavings and program states. Furthermore, the verification of logical formulas consumes more CPU time and computer memory as the size of the formula grows with increasing verification depth. Another concern arises from BMC's reliance on the underlying program's symbolic abstraction (over-approximation), which may lead to incorrect results when the devised model does not precisely represent the given program. For example, this situation can be caused by external libraries whose implementation in the language supported by the given BMC tool does not exist. Accordingly, their behaviour must be modeled (approximated) inside the BMC tool. As a result, existing BMC tools, such as *ESBMC* [64], *CBMC* [65], and *Cseq* [66], primarily differ in their choices of program encoding and symbolic abstractions.

2.3.1.1 Bounded model checking tools

Over the past years, bounded model checking has been successfully applied to verify concurrent C programs [67]. There are several state-of-the-art bounded model checkers available in the field, such as *ESBMC* [30] and *CBMC* [31] that can handle both sequential and multi-threaded C programs and find concurrency vulnerabilities (e.g., data races, deadlocks, etc.) and other vulnerabilities (e.g., buffer overflows, dangling pointers, etc.). For instance, *ESBMC*, is a state-of-the-art bounded model checker [68]. It verifies safety properties in both sequential and multi-threaded C programs. *ESBMC* takes a C program as input and employs a symbolic execution engine to unroll the program up to a bound k and generate a bounded program trace. The program trace is limited by an interleaving threshold (i.e., context switch) for concurrency. Then, the program trace is converted into an SMT formula using the same approach described in the BMC background (Section 2.3.1). *ESBMC* can verify array bounds violations, divisions by zero, pointer safety, overflows, memory leaks, deadlocks, data races, and offers flexibility in selecting between fixed and floating-point arithmetic. It has also been used to verify the safety/security of digital control systems [69], digital filters [70], unmanned aerial vehicles [71], and telecommunication software [72]. In contrast, *CBMC* encodes each concurrent execution unit separately and combines them with partial order formulae [73]. Similarly, *TCBMC* [74] is an extension of *CBMC*, which introduces constraints on the number of allowed context switches between threads.

Several other BMC tools have efficiently verified concurrent C programs at the annual

SV-COMP software verification competition [75]. Specifically, over the past decade, *Lazy-CSeq* [34], [76] has consistently led the concurrency category at SV-COMP. Their approach converts a multi-threaded C program into a non-deterministic sequential program that considers all possible round-robin schedules up to a specified number of rounds. Subsequently, they verify the obtained sequential program using a bounded model checker for sequential programs (e.g., *CBMC*, *ESBMC*). Similarly, *Deagle*, the winner for two consecutive years in the concurrency category in SV-COMP 2022 [77] and SV-COMP 2023 [78], respectively, introduced a novel theory of ordering consistency for multi-threaded programs [33]. They have also implemented a more efficient solver for this theory using *CBMC* as the front-end and MathSAT [79] as the back-end.

2.3.2 Fuzzing

Fuzzing is an automated testing technique that identifies software vulnerabilities by repeatedly executing a program with randomly generated inputs and monitoring its behaviour [80]. Since most inputs generated in this way are invalid, state-of-the-art fuzzers allow users to specify a small set of valid program inputs (known as seeds) and then use a mutation-based strategy to generate new ones. It is important to note that these inputs should meet the requirement of the program under test (PUT), such as the input format, to maximize the chance of triggering a crash [49].

2.3.2.1 Fuzzing process

Figure 2.3 illustrates the general fuzzing process [81], comprising four main components: the monitor, test generator, bug detector, and bug filter. The fuzzer takes the program under test (PUT), which is the target program to be tested (either as a binary or the source code), along with the initial seed files (test cases) as inputs. While executing the PUT, the fuzzer monitors its execution state to detect crashes or abnormal behaviour. The monitor employs various techniques, such as code instrumentation, to collect code coverage or other runtime information from the PUT.

Typically, three types of fuzzers exist: black-box, white-box, and gray-box; the last two use their own type of monitoring. However, this monitoring is not necessarily integrated into a black-box fuzzer. During the Program Under Test (PUT) fuzzing process, the test case generator mutates and generates additional seeds or test cases in various formats, such as files or network packets. Fuzzers commonly employ two main techniques for generating test cases: mutation-based and generation-based methods [81].

In the mutation-based method, test cases are generated by randomly mutating well-formed seed files or applying predefined mutation strategies that can adapt based on runtime information. On the other hand, the generation-based method does not rely on existing seed files; instead, it generates test cases based on a specification of the PUT.

In general, fuzzing test cases or inputs are semi-valid inputs, designed to be valid enough to pass the initial parsing stage of the PUT and invalid enough to trigger a crash in the deeper paths of the PUT, as further detailed in Section 2.3.2.2.

When the PUT crashes or reports bugs, the bug detector collects and analyzes relevant information, such as stack traces, to confirm the presence of a bug. The bug detector is built inside a fuzzer to help users identify potential bugs within the PUT. Finally, filtering exploitable vulnerabilities from all the bug reports is crucial. This task is often performed manually, which can be time-consuming and challenging. Recent developments [82]–[84] have aimed to mitigate this issue by sorting the fuzzer's outputs (bug-inducing test cases) or prioritizing interesting test cases. The output of the fuzzer is a Unix core dump [85], which captures an image of the process memory at the time of the crash. While this dump can help identify local information about the crash, such as variable values, it cannot be used to determine the path conditions that must be satisfied during execution, such as variables that have been overwritten.

2.3.2.2 Types of fuzzers

Fuzzing technique can be classified from two perspectives. First, they can be categorized based on the method of input generation, namely mutation-based or generation-based. Second, they can be classified based on their understanding of the PUT [39], which includes white-box, black-box, or gray-box fuzzing.

Mutation-based and Generation-based

The generation of test cases in the fuzzer can be classified into generation-based and mutation-based. A significant challenge lies in generating test cases that meet the requirements of complex data structures and can potentially trigger deep and hard-to-reach paths [49].

Mutation-based fuzzers: start with a required set of valid initial inputs or test cases and then randomly mutate them to generate new test cases [86]. These mutations can involve bit flipping, byte changes, data addition or removal, or modifications to specific parts (e.g., in network packets) [87]. Most state-of-the-art fuzzers, including *OpenGBF*, use a mutation-based strategy because these fuzzers do not require knowledge of the PUT or the input format.

Generation-based fuzzers: start with the required knowledge of the inputs, typically provided through a configuration file that defines the specific file format. Leveraging this file format knowledge allows the fuzzer to generate test cases that can more efficiently pass the program's validation [49]. In particular, these generated test cases can navigate deeper paths

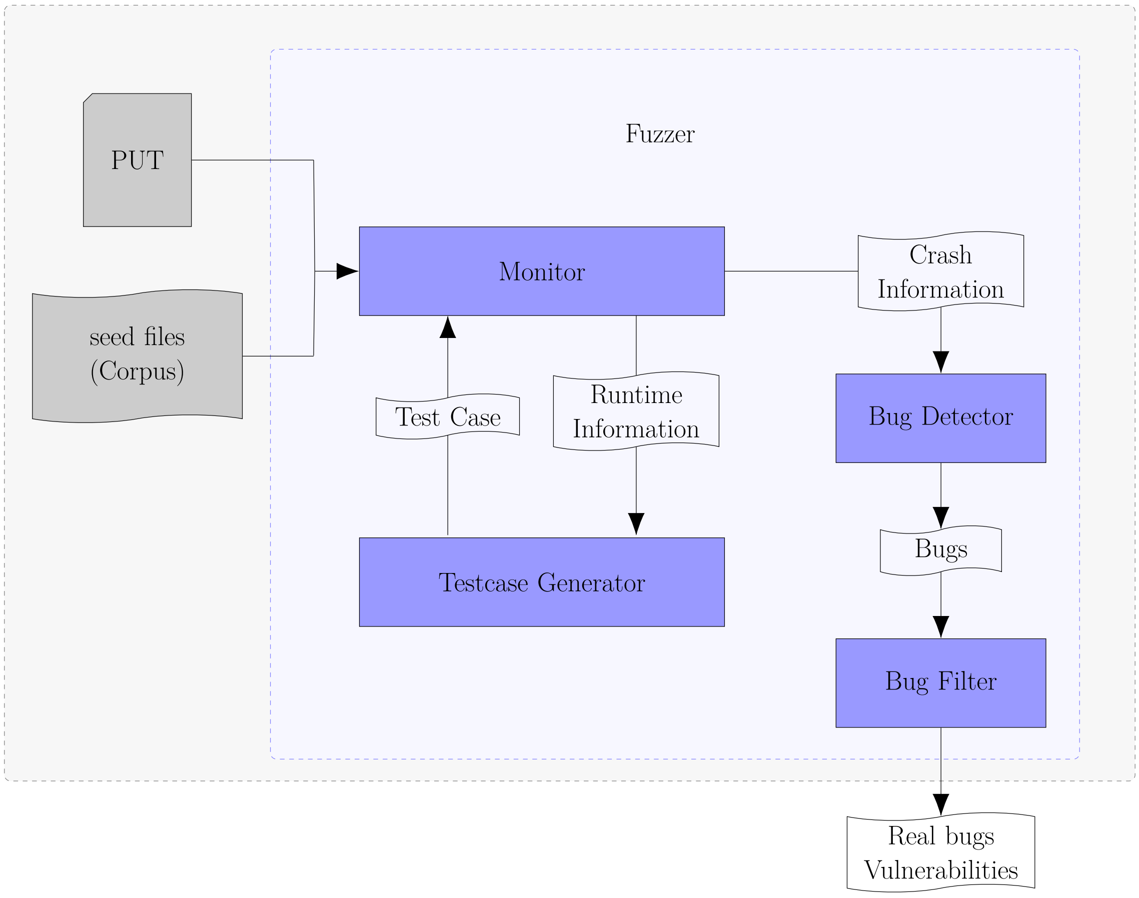


Figure 2.3. The general fuzzing process contains four main components (Monitor, Testcase generator, Bug detector, and Bug filter). It takes the PUT and seed files as input, executes the fuzzing process, and generates a report when it crashes.

within the PUT. However, analyzing the file format can become challenging without available documentation. As a result, mutation-based fuzzers are often considered more user-friendly, easier to start, more broadly applicable, and thus widely preferred by state-of-the-art fuzzers.

Black-box, White-box, and Gray-box

The fuzzer's dependence on the program source code and the depth of program analysis classify it into three categories: black-box, white-box, and gray-box fuzzers. Table 2.1 lists some common white-box, black-box, and gray-box fuzzers along with their test case generation strategies. The following subsection explains the differences between these fuzzers and highlights the advantages of each approach.

Black-box fuzzer: is often referred to as "black-box random testing". It operates by fuzzing the PUT without any knowledge of its internal logic. Instead of requiring information from the target program or input format, black-box random testing relies on predefined rules to randomly mutate well-formed input files, generating malformed inputs [49].

The effectiveness of black-box random fuzzers depends on the quality of their initial inputs,

	Black-box fuzzers	White-box fuzzers	Gray-box fuzzers
Generation based		SPIKE [88], Sulley [89], Peach [90]	
Mutation based	Libfuzzer [91]	Miller [92],SAGE [93]	AFL [84], AFL++ [94],Driller [47], VUzzer [95], MUZZ [41]

Table 2.1. Represents the common types of fuzzers (Black-box, Grey-box, and White-box Fuzzers) and the tools that use these types.

which are well-formed seeds used to start the fuzzing process. Well-formed initial inputs enhance the speed of the fuzzing process. In contrast, poorly formed inputs can result in inefficient resource consumption [96].

White-box fuzzer: was first proposed by Godefroid *et al.* [97] as a solution to address the limitations of black-box fuzzing [93]. A white-box fuzzer requires knowledge of the internal logic of the PUT by having access to its source code. It uses dynamic symbolic execution (called Concolic execution) and employs a coverage-maximizing guided search algorithm [81]. These features enable the white-box fuzzer to thoroughly and rapidly explore the PUT [81].

Gray-box fuzzer: denoted as GBF, stands between black-box and white-box fuzzers, effectively discovering software vulnerabilities with partial knowledge of the PUT. It can work without access to the source code, such as when dealing with binary files. It collects internal information about the PUT through program analysis. A commonly used method for this purpose is code instrumentation [98], [99]. Code instrumentation involves adding additional code to the PUT that tracks the required metrics, like code coverage during runtime. This information is then used to adjust its mutation algorithm, often through techniques like genetic algorithms [100], [101], to generate test cases that explore more execution paths or discover vulnerabilities more efficiently.

OpenGBF can be considered a gray-box fuzzer because it builds on top of *AFL++* (see Table 2.1) and employs code instrumentation on the PUT to gather information about the program's behaviour.

Algorithm 1 [41], [102] demonstrates the standard workflow of a gray-box fuzzer. It takes a target PUT and initial seeds M as inputs. It then instruments the PUT (line 1) by injecting additional code that enables the fuzzer to collect code coverage statistics in the PUT.

In each iteration of the main fuzzing loop (line 4), the fuzzer selects a seed t (line 5) and determines a random number N of mutations (line 6). Subsequently, the fuzzer repeatedly executes the instrumented program P_f (line 9) with a different mutated seed t' (line 8) as input and captures the execution statistics. If t' triggers a crash in the instrumented program P_f (line 10), it is added to the set of vulnerable inputs (line 11). However, if t' does not cause a crash but covers a new branch in the PUT (line 12), it is added to the seed queue Q_S (line 13). This approach helps the fuzzer discover more vulnerabilities in subsequent iterations. Finally, the execution of the main fuzzing loop continues until the predefined timeout is reached.

Algorithm 1 Gray-Box Fuzzing

```
Input: $PUT$ – program under test, $M$ – corpus of initial seeds.
Output: $Q_S$ – seed queue, $S_I$ – crash inputs found
 1: $P_f \leftarrow instrument(PUT)$                      {instrument the PUT}
 2: $Q_S \leftarrow M$                                    {initialize the seed queue}
 3: $S_I = \emptyset$
 4: while not $timeout$ do
 5:   $t \leftarrow select\_next\_seed(Q_S)$              {pick seed from queue}
 6:   $N \leftarrow get\_mutation\_chance(P_f, t)$
 7:   for all $i \in 1 \ldots N$ do
 8:     $t' \leftarrow mutate\_input(t)$                  {mutate the seed}
 9:     $rep \leftarrow run(P_f, t', M_c)$                {execute the PUT}
10:     if $is\_crash(rep)$ then
11:       $S_I \leftarrow S_I \cup t'$                    {new vulnerable input found!}
12:     else if $covers\_new\_trace(t', rep)$ then
13:       $Q_S \leftarrow Q_S \oplus t'$                  {add promising seeds to queue}
14:     end if
15:   end for
16: end while
```

2.3.2.3 Code instrumentation

Code instrumentation inserts additional code into a software program to collect specific runtime information or modify its behaviour. One example that provides code instrumentation is LLVM Pass, an essential part of the LLVM compiler infrastructure. LLVM Pass analyzes and transforms the PUT to monitor the behaviour of the PUT; it injects additional code into the LLVM intermediate representation (IR) program [103].

LLVM IR is a low-level representation of a program's source code, similar to assembly language. Listing 2.1 and 2.2 present a simple C code along with its corresponding IR, respectively. In Line 1, it defines the function `foo()`, which returns `void` (corresponding to Line 2 in the original C code). In Line 3, it calls the `printf()` function and stores its value in `%call` (corresponding to Line 3 in the original C code). Additionally, Line 8 calls the function `foo()` (corresponding to Line 6 in the original C code). This representation enables LLVM to perform several analyses and transformations on the source code, which can be categorized as follows:

Analysis passes: analyze the program's IR without actually changing the IR and collect information about its structure, behaviour, and properties, which other passes can use [104].

Transformation passes: modify the program's IR to perform optimization or transformation. It can use the `Analysis pass` to collect information about the program structure [104].

Utility Passes: provide various utilities that do not directly analyze or transform the source code but support other passes. For instance, they can include extracting functions to bitcode [104].

In this Ph.D. thesis, we will use the `Analysis` and `Transformation passes` in our *OpenGBF* design because it is compatible with any clang wrapper, such as *AFL++*, making it easy to integrate and analyze the original source code without modifying it.

Listing 2.1. A simple C code.

```c
#include <stdio.h>
void foo() {
    printf("Hello\n");
}
int main() {
    foo();
    return 0;
}
```

Listing 2.2. The corresponding IR to Listing 2.1

```
define dso_local void @foo() {
entry:
  %call = call i32 (i8*, ...) @printf(↩
      i8* getelementptr inbounds ([7 x ↩
      i8], [7 x i8]* @.str, i64 0, i64 ↩
      0))
  ret void
}
define dso_local i32 @main() {
entry:
  %call = call void @foo()
  ret i32 0
}
declare dso_local i32 @printf(i8*, ...)
@.str = private unnamed_addr constant ↩
    [7 x i8] c"Hello\0A\00", align 1
```

2.3.2.4 Sanitizers

Sanitizers play a crucial role in software security by providing valuable tools for detecting various types of bugs and vulnerabilities. They add runtime checks (code instrumentation) to the PUT, enabling the detection of bugs during program execution [105]. We can classify sanitizers into different types based on the software vulnerabilities they can detect, as presented below:

Thread Sanitizer: it adds code instrumentation to the PUT during compilation, tracking memory access, and synchronization operations. This sanitizer detects concurrency-related bugs, such as data races, deadlocks, and thread leaks [106].

Address Sanitizer: it adds code instrumentation to the PUT during the compilation process by allocating shadow memory, which maps to the original memory used by the program. This shadow memory keeps track of each load or store in the PUT to detect invalid memory access. This sanitizer detects memory-related bugs like out-of-bounds and use-after-free [107].

Memory Sanitizer: it adds code instrumentation to the PUT during the compilation process by tracking the state of memory locations and checking for any reads are performed on uninitialized memory. This sanitizer is used to detect uninitialized memory bugs [108].

Undefined Behaviour Sanitizer: it adds code instrumentation to the PUT during compilation by inserting runtime checks to detect undefined behaviour. This sanitizer can detect different types of undefined behaviour, such as out-of-bounds access, array access, and null pointers [109].

Using sanitizers with fuzzing improves bug detection's overall efficacy and efficiency, making it a powerful approach for improving software security. Furthermore, various types of sanitizers specialize in finding different vulnerabilities.

In this Ph.D. thesis, we use different sanitizers based on user preferences for bug detection (specifically, the types of bugs they aim to find).

2.3.2.5 Concurrency-aware gray-box fuzzers

Traditional fuzzing techniques are primarily designed for sequential programs and do not translate well to concurrent programs (e.g., *AFL++* [94]) due to their limitation of allowing the fuzzer to control only the program's input rather than the scheduling of its threads [41]. Various efforts have been made to address this limitation in concurrency-aware fuzzing [41] [12][110][111][112]. We classify these past efforts into five categories, as presented in the taxonomy in Table 2.2. The first three categories concern the usability of each fuzzer: whether they are used for verifying user programs or operating system code (*Scope*), the types of bugs they can detect (*Vulnerabilities*), and whether their code is openly accessible (*Open Source*). In this context, none of the existing state-of-the-art fuzzers meet our research requirements. Specifically, there is no fully open-source fuzzer capable of finding both concurrency and memory-related vulnerabilities in PUT. To address this gap, we introduce our own concurrency-aware fuzzer in Section 3.3.

The last two categories concern the fuzzing technique itself. In particular, the general gray-box fuzzing algorithm presented in Algorithm 1 requires some adaptations to achieve good results on concurrent programs. First and foremost, it is essential to have a mechanism that forces the execution of many different threads interleaving (*Interleaving Control*). Existing fuzzers, such as *MUZZ* [41] and *ConAFL* [12], modify thread priorities at the assembly level. Other tools like *Krace* [112] insert `sleep` instruction to force a context switch, while *AutoInter-fuzzing* [110] and *Conzzer* [113] instrument the PUT with precise synchronization barriers or thread locks. Alternatively, the exploration of different thread interleavings can be left to the natural non-determinism of the operating system, as demonstrated in tools like *ConFuzz* [111]. Lastly, some authors propose changing the feedback to the input mutation en-

Fuzzer Name	Scope	Vulnerabilities	Open Src.	Interleaving Control	Mutation Feedback
OpenGBF	User Space	Multiple	Yes	Delay Injection	Branch Coverage
MUZZ [41]	User Space	Multiple	No	Thread Priority	Thread-Aware
ConAFL [12]	User Space	Invalid Mem. Acc.	Partial	Thread Priority	Branch Coverage
ConFuzz [111]	User Space	Multiple	No	None	Thread-Aware
AutoInter-fuzzing [110]	User Space	Multiple	No	Barrier/Lock	Thread-Aware
Conzzer [113]	Kernel Space	Data Races	No	Barrier/Lock	Thread-Aware
Krace [112]	Kernel Space	Data Races	Yes	Delay Injection	Thread-Aware
SEGFUZZ [114]	Kernel Space	Multiple	Yes	Interleaving segmentation	Thread-Aware

Table 2.2. Taxonomy of existing state-of-the-art concurrency-aware gray-box fuzzers, including the scope of the tool, the vulnerabilities they can detect, the availability of the source code, and the strategies and mutations they use.

gine to guide the fuzzer towards more interesting thread interleavings (*Mutation Feedback*). We categorize such attempts as *Thread-Aware* instead of the default *Branch Coverage* metrics used in sequential fuzzing.

First, since we built our concurrency-aware fuzzer on top of American Fuzzy Lop Plus Plus (AFL++), we will explain how it operates. American Fuzzy Lop Plus Plus (AFL++), known as *AFL++*[94], is an open-source tool that incorporates state-of-the-art fuzzing techniques. *AFL++* is a mutation-based gray-box fuzzer that mutates initial test cases to explore different execution paths. It is an advanced version of the original *AFL* fuzzer[84], designed to enhance performance and effectiveness in detecting software vulnerabilities in sequential programs.

AFL++ improves the original *AFL* by introducing several new features. Firstly, it includes code instrumentation to analyze the program's execution and coverage information. This instrumentation enables *AFL++* to prioritize test cases that lead to previously unexplored paths within the PUT, thereby increasing the probability of finding hidden software vulnerabilities.

Moreover, *AFL++* introduces a wider range of mutation strategies and coverage-guided optimizations. One of its key advantages over *AFL* is that its instrumentation is thread-safe, as documented in [115]. This thread-safety feature makes *AFL++* more compatible with concurrent programs.

AFL++ also utilizes LLVM Pass as a code instrumentation technique to instrument the PUT for performance improvement (i.e., execution speed [116]). One key pass it employs is the `Edge Coverage Pass`, designed to monitor the coverage of control flow edges within the PUT. This pass helps identify unexplored paths and guides the fuzzer towards these new paths [94]. Additionally, *AFL++* incorporates the `Context Sensitive Coverage Pass`, which tracks the coverage of specific execution contexts within the tested program. Consequently, when compiling *AFL++*, we can include these LLVM Passes as integrated components within its clang wrapper during the compilation process.

Regarding concurrency-aware fuzzer, *MUZZ* [41] is an example of a gray-box fuzzer that uses static analysis to identify code blocks that are more likely to trigger a concurrency vulnerability [117]. When the code is instrumented, such blocks receive heavier instrumentation, helping the fuzzer dynamically track the execution of different schedules. To facilitate the exploration of a large number of thread interleavings, *MUZZ* controls the execution order by allocating random priorities to the threads at the assembly level. Despite the promising experimental results, *MUZZ* is not publicly available.

Similarly, *ConAFL* [12] is a gray-box fuzzer specialized for user-space multi-threaded programs. It uses static analysis to find sensitive concurrent operations that determine the execution order, focusing only on three types of invalid memory access vulnerabilities: *buffer-overflow*, *double-free*, and *use-after-free*. To control thread interleavings, *ConAFL* indirectly changes the execution priority of each thread at the assembly level. Alternatively, the authors mention the possibility of injecting a `sleep` instruction at the code level but do not test it. However, *ConAFL* relies on the default mutation feedback of the sequential fuzzer *AFL* [118], based on branch coverage. Due to its extensive thread-aware static and dynamic analysis, *ConAFL* struggles to scale to large programs. Additionally, the authors' static analysis tool is not publicly available [119].

A more straightforward approach is implemented in a tool called *ConFuzz* [111]. It lets the natural non-determinism of the operating system randomly guide the exploration of different thread interleavings. To compensate for this, *ConFuzz* modifies the standard branch coverage feedback of the mutation engine by calculating how far each code block is from a thread-related instruction. Seeds that execute these code blocks closer to such instructions have a higher probability of survival with each mutation. Unfortunately, the *ConFuzz* tool [111] is not publicly available.

Recently, another concurrency-aware gray-box fuzzer was proposed in [110]. This tool, called *AutoInter-fuzzing*, uses static analysis to identify instruction pairs that access the exact memory location but are executed by different threads. The source code (PUT) is then instrumented with synchronization barriers that control the execution order of the instructions in each pair. Every time one such pair is identified during fuzzing, the program is executed again, forcing the opposite execution order of the pair. Unfortunately, this approach for exploring thread interleavings causes *AutoInter-fuzzing* to suffer from low path coverage compared to other fuzzers. Like most of the fuzzers listed in the present section, *AutoInter-fuzzing* is not publicly available.

Conzzer [113] improves upon the ideas of *AutoInter-fuzzing*. Specifically, the instruction pairs are acquired at runtime and contain information about the execution trace. The authors claim that the fuzzer can be used to explore different thread interleavings for a critical region by being context-aware, and they implemented their own mutation algorithm, allowing the fuzzer to explore more thread interleavings than *AutoInter-fuzzing*.

On a different scope, *Krace* [112] is a fuzzer developed for kernel file systems specializing in detecting data races. It is relevant to mention it here because it also uses the thread interleaving control strategy of injecting delays in the source code. Additionally, it improves the standard branch coverage metrics by explicitly monitoring the order of execution of any pair of instructions that access the exact memory location. This feedback yields the mutation engine to explore more thread interleavings. While the source code of [112] is available, it cannot be used in our research as it targets data races in the kernel space.

Similarly, *SEGFUZZ* [114] is a new fuzzer implemented for the kernel space, which works by exploring the search space of thread interleavings. It decomposes the entire thread interleavings into segments, each representing the interleaving of a small number of instructions. Subsequently, it mutates these interleaving segments to generate new ones that have not been previously explored. Like *Krace*, it is publicly available; however, it is out of the scope of our evaluation because it is specialized in kernel space.

OpenGBF, as will be discussed in Section 3.3, implements many of these ideas.This includes instrumenting the source code (PUT) with the `sleep` function, which forces the exploration of random thread interleavings and allows the fuzzer to control the randomness through its mutation engine. In the future, if the aforementioned concurrency-aware fuzzers become open source [50], it will be possible to evaluate their efficacy when combined with BMC tools, as we do here with our GBF tool.

2.4 Cooperative verification approach

Cooperative verification is an approach in which multiple verifiers collaborate to solve verification problems by sharing artifacts related to the verification process [45]. Specifically, cooperative verifiers exchange information (verification artifacts) with each other or use information from other verifiers, aiming to improve the overall efficiency and effectiveness of the verification process. The cooperative approach can be structured based on the verifiers' communication interfaces. Verification artifacts play a central role in the cooperative approach, as they facilitate the exchange of information, and can be classified as follows:

Verification Result: verifiers produce a result of the evaluation statement "program satisfies specification". The result will be one of the following: {`Verification_failed`, `Verification_successful`, `Unknown`}.

Verification Witness: exchangeable witnesses work as envelopes for error paths, facilitating information exchange between several tools. The verification witnesses record the result of a verification process; it comes in the form of a violation witness and a correctness witness. A *violation witness* [120] defines the specification violation by representing a full program path that violates the specification. A *correctness witness* [121] defines why the program satisfies the specification by describing valuable invariants in a proof of correctness [122], [123]. The verification witness is formatted in an XML-based format [124], which is supported by the validators that validate these witnesses.

Test case: specifies a sequence of values for all external function calls in the program, providing inputs to the PUT.

Condition: specifies the part of the program's behaviour that requires no further exploration. In the case of verification, it represents the parts already verified; in the case of testing, it represents the parts already covered by an existing test suite.

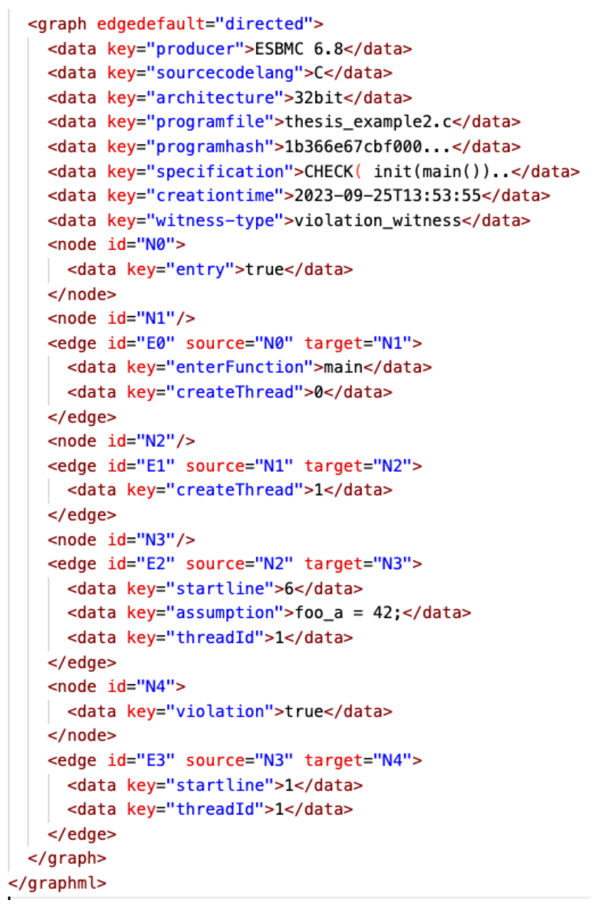
In this Ph.D. thesis, we incorporate the use of `Verification Result`, `Verification Witness`, and `Test case` in our cooperative framework.

2.4.1 Verification witness file format

A witness file plays an important role in software verification; it provides a bug trace or counterexample to be used to analyze and validate the verification results. In the context of Software Verification Competition (SV-COMP) [124], witness files are representations of execution traces that show the correctness or incorrectness of a given verification property. It is formally represented as witness automata. Since an automaton is a graph, they extend an existing exchange format (GraphML) for graphs and apply it to witness automata. The idea of the violation-witness automaton is that it guides the verifier through a finite number of program steps along an error path to find safety property violations [124].

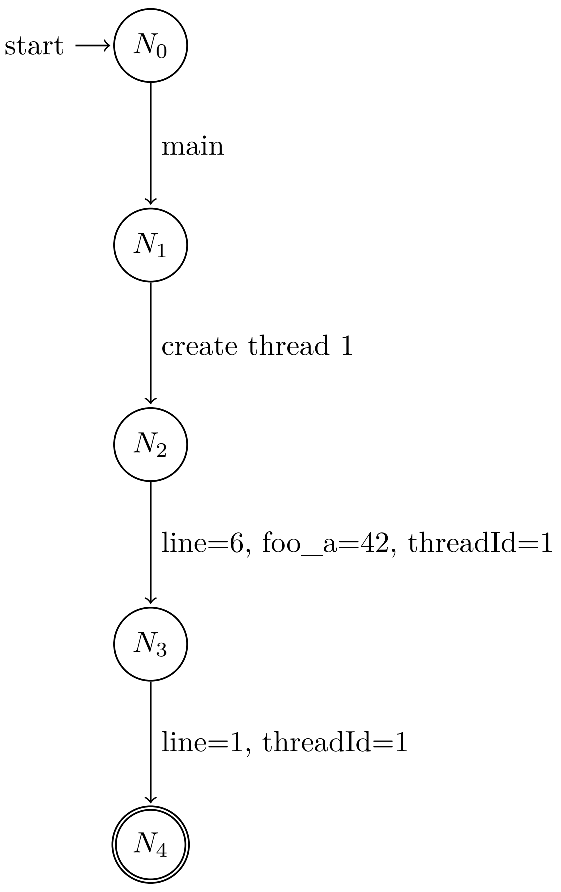
The annotations used in generating the GraphML graph are categorized into two sections: Graph Data for Witness Automata (see Table 2.3) and Edge Data for Automata Transitions (see Table 2.4). The former, Graph Data for Witness Automata, provides high-level information about the witness automaton as a whole. It outlines the properties of the automaton. It contains data such as the tool name, the programming language used, the creation time, and so on. The latter, Edge Data for Automata Transitions, provides information about the individual transitions within the witness automaton. It contains data specific to each transition, such as line number function name, and so on [124].

Example 2.3 illustrates a simple C code where a thread invokes the `foo()` function. The program crashes (reaches the error statement) if the thread is executed before the main thread finishes. Figure 2.4 shows the corresponding witness file in GraphML format and the graphical representation of the witness. Figure 2.4a shows the included key data described in Tables 2.3 and 2.4, while Figure 2.4b shows visually the path that leads to an assertion failure in the program. The execution sequence starts at state $N0$, where the main function starts at state $N1$; then thread 1 is created at line 10, leading to state $N2$, where the assignment of `foo_a`



```
  <graph edgedefault="directed">
    <data key="producer">ESBMC 6.8</data>
    <data key="sourcecodelang">C</data>
    <data key="architecture">32bit</data>
    <data key="programfile">thesis_example2.c</data>
    <data key="programhash">1b366e67cbf000...</data>
    <data key="specification">CHECK( init(main())..</data>
    <data key="creationtime">2023-09-25T13:53:55</data>
    <data key="witness-type">violation_witness</data>
    <node id="N0">
      <data key="entry">true</data>
    </node>
    <node id="N1"/>
    <edge id="E0" source="N0" target="N1">
      <data key="enterFunction">main</data>
      <data key="createThread">0</data>
    </edge>
    <node id="N2"/>
    <edge id="E1" source="N1" target="N2">
      <data key="createThread">1</data>
    </edge>
    <node id="N3"/>
    <edge id="E2" source="N2" target="N3">
      <data key="startline">6</data>
      <data key="assumption">foo_a = 42;</data>
      <data key="threadId">1</data>
    </edge>
    <node id="N4">
      <data key="violation">true</data>
    </node>
    <edge id="E3" source="N3" target="N4">
      <data key="startline">1</data>
      <data key="threadId">1</data>
    </edge>
  </graph>
</graphml>
```

(a) Witness file in GraphML format.



$$\begin{tikzcd}
\text{start} \arrow[r] & N_0 \arrow[d, "\text{main}"] \\
& N_1 \arrow[d, "\text{create thread 1}"] \\
& N_2 \arrow[d, "\text{line=6, foo\_a=42, threadId=1}"] \\
& N_3 \arrow[d, "\text{line=1, threadId=1}"] \\
& N_4
\end{tikzcd}$$

(b) Graphical representation.

Figure 2.4. The witness file in GraphML format Vs. graphical representation for Listing 2.3.

occurs during thread 1. This transition results in state $N3$, which leads to the invocation of `reach_error()`, resulting in the acceptance state $N4$.

Listing 2.3. A simple C code.

```c
void reach_error() { assert(0); }

void foo(void * arg){
    int foo_a = __VERIFIER_nondet_int();
    if(foo_a == 42)
        reach_error();}

int main() {
    pthread_t main_t1;
    pthread_create(&main_t1, 0, foo, 0);}
```

In this Ph.D. thesis, we use this format to generate the final witness file.

key	Meaning
producer.	Specifies the name of the tool that generates the witness automaton. For example, *EBF* 4.0.
sourcecodelang.	Specifies the programming language, for example, C.
architecture	Specifies the user-defined textual representation of the machine architecture assumed for the verification task, such as ”32-bit” or ”64-bit” systems.
programfile.	Specifies the program file path provided as input to the verifier tool. (e.g., thesis_example.c). Note that this key is only for documentation purposes, and it is not necessary for the validator to have direct access to the specified file location because the source code is explicitly provided as input to the validator.
programhash.	Stores the SHA-256 hash value of the verified program. For example, ”7e50ae1d6af13c623583d6a2f949e9597ecd4d402771d52399b9fc659c58a3d1”.
specification.	Provides a user-defined textual representation of the verification task's specification. For example, in SV-COMP, if the property specification is reachability, the text $CHECK(init(main()), LTL(G!call(reach\_error())))$ is used to denote the specification.
creationtime.	Specifies the creation date and time of the witness in ISO 8601 format. The date must include the year, month, and day, split by dashes ('-'). The date and time are separated by the capital letter 'T'. The time must include the hours, minutes, and seconds, separated by colons (”:”). If the timestamp is in UTC time, it concludes with a 'Z'. For example, '2023-05-22T12:16:59Z' represents the year 2023, May 22nd, at 12:16:59.
witness-type.	Specifies the type of witness (i.e., correctness_witness or violation_witness).

Table 2.3. A key data and its meaning for Graph Data for Witness Automata.

key	Meaning
enterFunction.	Specifies the function calls in the source code.
createThread.	Specifies the creation of a new thread within the witness.
assumption.	Specifies the non-deterministic values that lead to the violation.
threadId.	Represents the currently active thread for that particular transition. The value associated with this key must uniquely identify an active thread, meaning a thread that has been created but has not yet terminated.
startline.	Specifies the source code line number.

Table 2.4. A key data and its meaning for Edge Data for Automata Transitions.

2.4.2 Cooperative/Hybrid verification tools

Recently, several efforts have combined fuzzing with different forms of symbolic execution and static analysis [49]. To our knowledge, none of these methods have been combined to find vulnerabilities in concurrent programs. However, the rationale behind these efforts is that fuzzing alone struggles to find deep bugs that lie in complex path conditions (e.g., the branch `if(x*x -2*x +1 == 0)`) because the inputs introduced from the random mutations have a low probability of hitting such a complex path in the program. In contrast, when the fuzzer is provided with a set of input seeds close to the correct target, the evolutionary algorithm has a higher probability of exposing bugs and vulnerabilities. To bridge this gap, we combine our concurrency-aware fuzzer *OpenGBF* with the state-of-the-art BMC tools in Chapter 4.

To achieve this, Ognawala *et al.* [42] proposed a tool called *Munch* that increases the fuzzing coverage by augmenting the set of input seeds with an additional round of concolic execution. This approach leads to a significant increase in code coverage. There are other examples of tools using concolic execution, like *Driller* [47] and *QSYM* [48]. The former is a hybrid tool that combines fuzzing with concolic execution to detect deep vulnerabilities. It analyzes the program and generates interesting seed inputs using concolic execution that guides the fuzzer toward unexplored paths in the program. The latter works by obtaining the fuzzer's output as an initial seed input and using concolic execution to these seeds, particularly executing the program concretely (using actual values) and symbolically (using variables). This approach enables *QSYM* to explore different execution paths and generate new inputs that can potentially expose more vulnerabilities.

Similarly, *VeriFuzz* [44] is a hybrid tool that combines bounded model checking with fuzzing, achieving first place in Test-Comp 2020 [125]. It addresses the issue of off-the-shelf fuzzers being unable to find seed inputs that pass complex blocks of program logic. Their solution is using a BMC to solve the corresponding reachability statement and generate concrete input seeds that satisfy the complex conditions of the PUT. Then, the fuzzer is free to explore the search space beyond that.

FuSeBMC [43], which achieved first place in Test-Comp 2022 and 2023 [126], [127], respectively, is also a hybrid tool. It uses a selective fuzzer when the bounded model checker of their *FuSeBMC* tool fails to detect all vulnerabilities. Such a fuzzer employs the statistics the model checker collects to create a specific set of input seeds.

Map2check [128], [129] works by performing code instrumentation at the LLVM level. This instrumentation includes adding a verification framework and explicit assertions. The instrumented code is then subjected to symbolic execution (using KLEE [130]) or fuzzing (using Libfuzzer [131]).

On a different note, *SAGE* (Scalable Automated Guided Execution) is a hybrid white-box fuzzer developed at Microsoft Research by Godefroid *et al.* [93], [132]. It uses generational search to increase the number of new input seeds produced from dynamic symbolic execution. Given a path constraint, a constraint solver systematically negates and solves all the constraints.

LibKluzzer [133] is a tool that harnesses white-box and coverage-guided fuzzing strengths. It achieves this by employing the coverage-guided fuzzer to discover new execution paths and using white-box fuzzing to navigate complex branch conditions.

EBF is similar to these hybrid tools in the sense that it exploits the combined advantages of fuzzing and bounded model checking. Nevertheless, the aforementioned hybrid tools are built around a close integration between the two techniques, often requiring specific assumptions about the verification task at hand. In contrast, our cooperative framework is more flexible and allows virtually any existing tool to be combined together. Finally, none of the existing hybrid approaches can verify concurrent programs.

2.5 Other techniques for finding software vulnerabilities in concurrent programs

Other techniques (neither fuzzing nor bounded model checking) for detecting software vulnerabilities in concurrent programs have been proposed. Wen *et al.*[134] proposed a controlled concurrency testing technique called *Period*, which employs a periodical execution to model the execution of concurrent programs and systematically explores the space of possible thread interleavings. They provide the periodical executor with a key point slice of the program code and apply an analyzer to collect feedback on runtime information. On the contrary, *Peahen* [135] is a proposed approach called context reduction that combines context-sensitive and context-insensitive static techniques. This context reduction filters the vulnerabilities found by a context-insensitive technique with a path feasibility check. After that, a context-sensitive approach is employed to validate the vulnerability. *Peahen* is designed to detect only deadlock vulnerability in concurrent programs. Finally, Mukherjee *et al.* [136] from Microsoft research proposed a tool called *QL* that uses reinforcement learning to guide the exploration of thread interleavings. *QL* uses an explicit scheduler.

On a different note, some methods aim to improve classic verification techniques. For example, in dynamic analysis, certain works focus on improving soundness and completeness [137], [138]. Others create new value flow analyses for interprocedural data flow that detect concurrency issues. For example, *Canary* [139] is a value-flow analysis framework that analyzes data and interference dependencies by creating a value-flow graph (VFG). It annotates value flows with their constraints, checks for value flows connecting source to sink between different threads, and uses the Z3 SMT solver to ensure feasible interleaving executions, enabling the detection of concurrency use-after-free vulnerabilities. Similarly, *DCUAF* [140] aims to detect use-after-free errors in Linux device drivers by analyzing each driver's lock usage as local information. It combines local information on driver functions to perform a global statistical analysis.

At the same time, some techniques use a different flavor of Model Checking known as stateless model checking (SMC)[141]. The method emerged from the intuition that caching states in Model Checking was not as effective as a stateless approach. For example, *RCMC* [142] and *GenMC* [143] rely on having a code interpreter capable of computing a reachability graph over the program and using system calls during the analysis to provide more accurate results. On the other hand, *LAPOR* [144] uses a lock-aware Partial Order Reduction (POR) algorithm to handle programs with locks. Additionally, *ConVulPOE* [145] employs partial-order reduction techniques to generate execution traces that expose vulnerabilities in concurrent programs by recording thread operations and memory access events while extracting vulnerability-potential event pairs, improving vulnerability detection.

2.6 Summary

In this chapter, we have explored common software vulnerabilities, including memory-related vulnerabilities like invalid memory access and memory leaks and concurrency-related vulnerabilities such as data races and deadlocks. Additionally, we discussed user-defined properties, which enable users to specify conditions defining expected program states that should not be reached during execution.

The chapter introduced various software verification and testing techniques for concurrent programs, including bounded model checking and fuzzing. We provided detailed explanations of how these methods work. In the case of BMC, we also presented one of the state-of-the-art BMC tools (*ESBMC*). Regarding fuzzing, we explained the fuzzing process and the types of fuzzers that can be categorized based on the method of input generation, namely mutation-based or generation-based. Additionally, we discussed their understanding of the PUT, which includes white-box, black-box, or gray-box fuzzing. The chapter also includes an explanation of code instrumentation and the utilization of sanitizers integrated with the fuzzing process. Lastly, we introduced the state-of-the-art fuzzing tool (*AFL++*) upon which *OpenGBF* is built. Furthermore, we summarized the differences between BMC and the fuzzing technique.

Moreover, we clarified the cooperative verification approach in which tools communicate via verification artifacts. Furthermore, we explained these verification artifacts, including verification results, witnesses, test cases, and conditions.

To provide a comprehensive overview, we covered state-of-the-art BMC tools such as *CBMC*, *ESBMC*, *Lazy-CSeq* and *Deagle*, known for their efficiency in verifying concurrent C programs. Additionally, we covered fuzzing techniques adapted for concurrent programs. In particular, we discussed the limitations of traditional fuzzing techniques in the context of concurrency and reviewed several concurrency-aware fuzzing tools like *MUZZ*, *ConAFL*, *ConFuzz*, *AutoInter-fuzzing*, and *Conzzer*.

Additionally, we highlighted the potential of hybrid approaches that combine fuzzing with symbolic execution, static analysis, and bounded model checking to overcome the limitations of each technique individually. Tools like *VeriFuzz*, *FuSeBMC*, and our tool *EBF* demonstrate these hybrid approaches, achieving significant success in various verification competitions.

Lastly, we explored alternative techniques that go beyond bounded model checking and fuzzing. These include controlled concurrency testing, context reduction, and reinforcement learning-guided exploration, which offer unique perspectives on detecting vulnerabilities in concurrent programs.

Chapter 3

Concurrency-aware gray-box fuzzer

3.1 Chapter introduction

In this Ph.D. thesis, our main challenge is the lack of mature open-source tools for fuzzing concurrent programs. Consequently, our central contribution is to overcome this challenge through the proposal, implementation, and evaluation of an open-source concurrency-aware gray-box fuzzer. This fuzzer, referred to as *OpenGBF* here, will be comprehensively discussed in the coming sections. The content of this chapter is based on the research presented in [1], which presents detailed insights into our concurrency-aware gray-box fuzzer, *OpenGBF*. Firstly, we will outline the challenges of implementing a concurrency-aware gray-box fuzzer in Section 3.2. Secondly, the main framework of *OpenGBF* will be introduced in Section 3.3. The core framework design mainly consists of instrumenting the Program Under Test (PUT) by injecting function calls and implementing them using the runtime library, as explained in Sections 3.3.2 and 3.3.3 respectively. Finally, we will present an illustrative example to explain our instrumentation process in Section 3.3.4.

3.2 Challenges of fuzzing concurrent programs.

Fuzzing concurrent programs is challenging due to the complexity and non-deterministic behaviour of concurrent execution. Some of the key challenges include:

1. **Thread interleavings:** the scheduling of threads in concurrent programs can result in different thread interleavings, leading to multiple program behaviours. Therefore, fuzzing is required to explore different thread schedules to identify potential concurrency-related bugs. Furthermore, instrumenting the code can introduce new interleavings that must be considered during the instrumentation process; we address this challenge in Section 3.3.2.1.
2. **Data races and atomicity violations:** concurrent programs can be exposed to data races and atomicity violations when multiple threads simultaneously access shared data, lead-

ing to synchronization issues. Therefore, fuzzing should aim to identify these concurrency-related bugs by generating test inputs that can trigger such vulnerabilities, which we address in Section 3.3.1.

3. **Scalability:** fuzzing concurrent programs can be computationally exhausting and time-consuming. The vast number of thread interleavings and the complex nature of concurrent execution pose challenges in scaling fuzzing tools effectively; we address this challenge in Chapter 4, Section 3.3.2.2.
4. **Non-Deterministic behaviours:** reproducing a specific bug in a concurrent program is challenging due to the non-deterministic behaviour of concurrent programs. Thus, the fuzzer needs to have an approach for reproducing the exact input and interleavings that lead to the bug; we address this challenge in Section 3.3.2.3.

Addressing these challenges is essential for effectively identifying vulnerabilities in concurrent programs [146]. In the following sections, we will introduce our algorithm to tackle these four challenges.

3.3 Designing a state-of-the-art concurrency-aware gray-box fuzzer

Recall that our *OpenGBF* is based on existing techniques that control the fuzzer by introducing delays to explore different thread interleavings and use branch coverage to guide the fuzzer mutation engine. It is implemented and evaluated for two important reasons:

1. To the best of our knowledge, *OpenGBF* is the only fully open-source, user-space concurrency-aware fuzzer available, as reported in [1], [2]. Therefore, it is a valuable reference for users and researchers, enabling them to extend the algorithm and conduct further research.
2. Since it is impossible to confirm the claims made in the existing literature due to the absence of open-source codebases, our *OpenGBF* is a transparent effort to reproduce and confirm these claims.

3.3.1 *OpenGBF* framework

We have built our *OpenGBF* framework on top of the state-of-the-art gray-box fuzzer *AFL++* [115]. *AFL++* is primarily designed to detect vulnerabilities in sequential programs and is not inherently equipped to handle concurrent programs.

To address this challenge, we developed the Open-source Gray-Box Fuzzer (*OpenGBF*), a thread-aware concurrency fuzzer that extends *AFL++*. We achieved concurrency awareness

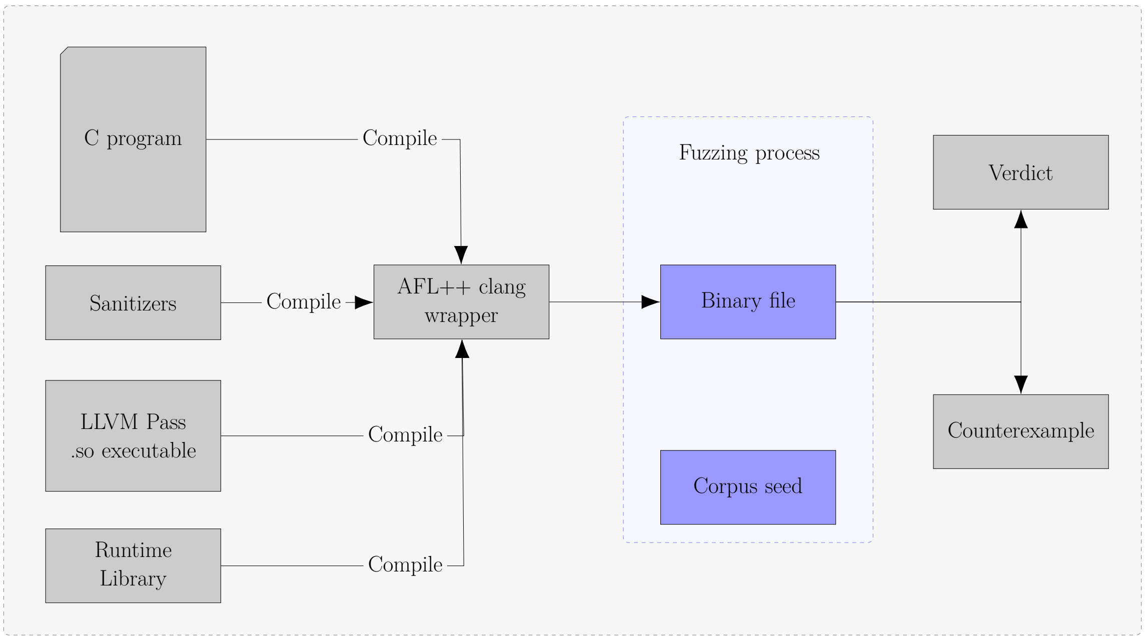


Figure 3.1. *OpenGBF* framework, which consists of eight components: C program, Sanitizer, LLVM Pass executable, Runtime library, AFL++ clang wrapper, fuzzing process, verdict, and counterexample.

by instrumenting the program using LLVM Pass. We chose LLVM Pass for code instrumentation due to its compatibility with *AFL++*, which also uses LLVM Pass to instrument the Program Under Test (PUT) for improved execution speed [116].

Figure 3.1 illustrates the high-level structure of the *OpenGBF* framework, consisting of eight components. The first four components (C program, sanitizer, LLVM Pass executable, and runtime library) are compiled using the fifth component (*AFL++ clang wrapper*). Once compiled, the resulting executable (binary file) can be fuzzed using the provided Corpus seed. Finally, the last two components (verdict and counterexample) represent the output of the fuzzing process. In the following section, we will explain each of these components.

1. **C program:** it refers to the program the user requires to test, written in C, C++, or a preprocessed C code (*.i) file. Throughout this thesis, we will consistently refer to this as the Program Under Test (PUT).

2. **Sanitizers:** *AFL++* allows the use of sanitizer instrumentation during the compilation process [147]. Depending on the specific sanitizer employed, we can identify the types of bugs *OpenGBF* can detect. For example, when compiling the program using:

   - *Thread Sanitizer*: *OpenGBF* can detect concurrency-related bugs, including data races, deadlocks, and thread leaks, as outlined in [106].

   - *Address Sanitizer*: *OpenGBF* can detect memory-related bugs, including out-of-bounds and use-after-free [107].

- *Memory Sanitizer*: *OpenGBF* can detect uninitialized memory bugs [108].
- *Undefined behaviour Sanitizer*: *OpenGBF* can detect undefined behaviour bugs such as Out-of-bounds Array Access and Null Pointers [109].

One of the challenges in designing a concurrency-aware fuzzer for concurrent programs is that concurrent programs can have concurrency bugs that are difficult for a general fuzzer to detect [148]. To address this issue, we employ *Thread Sanitizer* to detect concurrency-related bugs. We can enable multiple sanitizers simultaneously during compilation, which allows us to leverage their capabilities, find more bugs, and ensure a more comprehensive analysis of the program's behaviour [9], [147]. However, enabling multiple sanitizers will increase runtime overhead and slow down the fuzzer for bug detection [149]. Therefore, selecting a specific sanitizer that best suits the particular needs is more convenient, facilitates debugging, and enables precise reporting of the bugs detected by *OpenGBF*.

3. **LLVM Pass:** it contains our custom LLVM Pass instrumentation, which injects five function calls. Detailed explanations can be found in Section 3.3.2.
4. **Runtime Library:** it is a dynamic library that implements the functions we instrument in our LLVM Pass. Detailed explanations can be found in Section 3.3.3.
5. **AFL++ clang wrapper:** a clang replacement used to compile the program with *AFL++* instrumentation [150]. This component compiles the program with the required sanitizer flags, depending on the property we need to test, along with the original PUT, our custom LLVM Pass, and our runtime library.
6. **Fuzzing Process:**
   - **Binary File:** The executable is obtained once we compile the PUT with the necessary sanitizer (if any), LLVM Pass instrumentation, and runtime library. This executable is then used for fuzzing with *AFL-fuzz*, and we initiate the fuzzing process by providing the required initial seeds.
   - **Corpus seed:** The corpus directory contains the initial seeds provided to the fuzzer to initiate the fuzzing process. These initial seeds operate as the basis for creating new seeds through mutation, resulting in a more diverse generation of test cases.
7. **Verdict:** We examine the crashes detected by the fuzzer and incorporate them later into the decision matrix to produce the final verdict. By default, we set the property as a reachability check unless we use any of the sanitizers.
8. **Counterexample:** During the runtime of the fuzzing process, we generate a file containing all the necessary information for tracking the bug [151]. This file helps as a reference for generating the witness file at a later stage.

Algorithm 2 LLVM pass Instrumentation

```
Input: $PUT$ – program under test.
Output: $M$ – instrumented program.
Shorthands:
$\lambda_d - \_delay\_function()$;
$\lambda_a - pthread\_add()$;
$\lambda_j - pthread\_release()$;
$\lambda_e - EBF\_add\_store\_pointer()$;
$\lambda_l - EBF\_alloca()$;
 1: $M \leftarrow PUT$
 2: for all Function $F \in PUT$ do
 3:   for Instruction $I$ in $F$ do
 4:     $M \leftarrow$ instrument $(\lambda_d, I, M)$ {insert a call to _delay_function() (Algorithm 3) after each instruction
        to run a delay at runtime}
 5:     if $I == pthread\_create()$ then
 6:       $M \leftarrow$ instrument $(\lambda_a, I, M)$ {insert a call to $pthread\_add()$ (Algorithm 4) to increase the active
          threads counter at runtime}
 7:     else if $I == pthread\_join()$ then
 8:       $M \leftarrow$ instrument $(\lambda_j, I, M)$ {insert a call to $pthread\_release()$ (Algorithm 5) to decrease the
          active threads counter at runtime}
 9:     else if $I$ is DECLARATION then
10:       $M \leftarrow$ instrument $(\lambda_l, I, M)$ {insert a call to EBF_alloca() function (Algorithm 6) to record a pair
          of the name and address of the variable declaration.}
11:     else if $I$ is STORE then
12:       $M \leftarrow$ instrument $(\lambda_e, I, M)$ {insert a call to EBF_add_store_pointer() (Algorithm 7) function to
          record the assignment information for witness generation}
13:     end if
14:   end for
15: end for
16: return $M$
```

3.3.2 Custom LLVM pass instrumentation

We built *OpenGBF* on top of *AFL++* by combining the standard LLVM Pass used in the *AFL++* clang wrapper with our custom, independent LLVM Pass. Specifically, we instrument the PUT using our custom LLVM Pass by injecting five function calls. These calls include a delay function, two thread-monitoring functions, and two information-collecting functions. Algorithm 2 illustrates these instrumented functions, where the delay function (line 4) is injected to control thread interleavings. In contrast, the two thread-monitoring functions (lines 6 and 8) are injected to monitor the number of active threads, where the two information-collecting functions (lines 10 and 12) are injected to record the information needed to generate a witness file containing the execution trace (bug trace).

3.3.2.1 Delay function instrumentation

Concurrent programs can have different thread interleavings, leading to different program behaviours. To verify such programs, it is essential to ensure that the fuzzer explores these different interleavings. Therefore, to control thread interleavings, we insert a call to the (_-

delay_function()) after each instruction in the Intermediate Representation (IR) of a function. Specifically, we insert a call to this *_delay_function()* for each function in the PUT and each instruction within that function (see lines 2 to 4). Then, we implement this function in a runtime library by introducing delays, as explained in Section 3.3.3.1. However, The LLVM-IR requires that the first instruction in basic blocks be a PHI instruction [152]. For this reason, we ensure that no *_delay_function()* is injected before a PHI instruction.

3.3.2.2 Thread-monitoring functions instrumentation

Since concurrent programs can have many threads, it presents challenges in efficiently scaling fuzzing for their verification. Consequently, monitoring the number of active threads becomes crucial. To address this challenge, we introduce two thread-monitoring functions: *pthread_add()* and *pthread_release()*. We instrument the PUT by injecting these functions. Specifically, we insert the *pthread_add()* function after each *pthread_create()* call (lines 5 and 6), and we inject a function named *pthread_release()* after each *pthread_join()* call (lines 7 and 8). The implementation details of these functions are explained in Section 3.3.3.1.

3.3.2.3 Information-collecting functions instrumentation

The non-deterministic behaviour of concurrent programs makes reproducing a specific bug trace challenging. Consequently, collecting the necessary information for generating this bug trace is crucial. We achieve this by inserting function calls for *EBF_alloca()* and *EBF_-add_store_pointer()* respectively (lines 10 and 12). After each *declaration* instruction in the PUT (e.g., $int\ a$;), we inject a function call to *EBF_alloca()*. Similarly, after each *load-store* instruction in the PUT (e.g., $a = 42$), we inject a function call to *EBF_add_store_pointer()*. The implementation details of these functions are explained in Section 3.3.3.3.

3.3.3 Runtime library for the LLVM pass instrumentation functions

We bundle the instrumented functions into a runtime library. We utilize the runtime library to link the functions instrumented using LLVM Pass for several reasons. First, the runtime library enables dynamic behaviour; the functions can be executed during runtime, which is particularly helpful when monitoring the number of active threads. Second, the runtime library provides flexibility; it can be easily updated, changed, or replaced without affecting the LLVM Pass codebase. Lastly, integrating is straightforward, and users can modify the implementation without changing the LLVM Pass. In the following sections, we will provide a detailed explanation of implementing these five instrumented functions.

3.3.3.1 Controlling the thread interleaving

We explain the implementation of the delay function, which is instrumented using LLVM Pass. The purpose of this function is to control thread interleavings. Our main algorithmic idea is injecting random delays in the PUT to force different thread executions. Furthermore, *OpenGBF* will also address several significant corner cases, which we will discuss in the following sections.

Firstly, we limit the number of active threads during program execution to maximize the chances of finding bugs. Determining the limit of active threads is an unfortunate but necessary approximation of the PUT runtime behaviour. This is based on empirical experiments where we found that bugs can often be discovered in shallow paths or with fewer thread interleavings. Additionally, some PUTs may contain a very large or "infinite" number of active threads, which can significantly slow down execution and consume excessive computing resources during fuzzing. This may result in either undefined behaviour or an undecidable problem. To address this, we monitor the number of active threads and start a new run with different interleavings if it exceeds a predefined threshold.

Secondly, to prevent the PUT from getting stuck due to a deadlock during execution, we introduce a mechanism to terminate the run and initiate a new one non-deterministically. This mechanism introduces a probability factor p that determines the probability of exiting the run at each instruction. Introducing this probability factor p reduces the risk of the PUT becoming stuck during execution.

Thirdly, we have defined atomic functions, namely *EBF_atomic_begin()* and *EBF_atomic_-end()*. These functions are necessary for several reasons. Firstly, since not all versions of the standard C language define atomic instructions, we must implement these functions to ensure that all instructions are executed atomically within these function blocks. Secondly, within our implementation, we must guarantee the atomicity of certain functions to prevent data races. Lastly, these functions are required by the set of benchmarks on which we will evaluate *OpenGBF*.

In our delay function, we enforce that all other threads wait for the atomic block to finish. This is accomplished by initializing a global mutex (*EBF_mutex*) that an active thread can lock. If the global mutex is locked and the current thread does not own the mutex, the active thread will wait for the mutex to be released by its owner. Furthermore, we ensure no delay function is injected inside these atomic blocks. Since verification time is asymptotically dominated by the number of interleavings in the PUT, avoiding thread interleavings inside the atomic blocks optimizes the overall verification time and memory usage.

Finally, we introduce different interleavings by providing different amounts of delay (in milliseconds) after each instruction. These values are chosen uniformly from a preset range.

Algorithm 3 Function *_delay_function()*

Global: T_T – thread threshold, T_N – number of threads running, p – probability of exiting, T_C – current thread, EBF_mutex – global mutex.

```
 1: Function _delay_function()
 2: if $T_N > T_T$ or $Bernoulli(p) == 1$ then
 3:     exit                                          {exit this analysis normally}
 4: end if
 5: if $T_C == EBF\_mutex$ then
 6:     $run\_instruction$                            {run the current instruction}
 7:     return
 8: end if
 9: $\phi \leftarrow wait\_for\_timeout$                {wait until EBF_mutex is released}
10: if $\phi$ is timeout then
11:     exit                                          {exit this analysis normally}
12: end if
13: $sleep(*)$                                        {run a delay for * nanoseconds}
14: EndFunction
```

Precisely, we let *AFL++* generate the seed values for the random number generator, which provides the final delay values. We evaluate the effect of different delay values on the bug-finding capability of *OpenGBF* in Section 6.5.

We take care of these corner cases in the implementation of the *_delay_function()*, as defined in Algorithm 3. In lines 2-4, we handle the first two cases: limiting the number of active threads and introducing the probability factor p. If the active thread T_N exceeds the predefined threshold T_N, or if we extract a value of 1 from a Bernoulli distribution with a success probability of p, the fuzzer exits the current run normally and starts a new run with different delay values, resulting in different interleavings. The third corner case ensures atomicity; in line 5, we check if the current thread owns the global mutex. If it does, we allow the thread to finish its execution and release the mutex (lines 6 and 7). In lines 9-10, if the global mutex exceeds the timeout without being released by the thread, the fuzzer is permitted to exit the current run normally and start a new one. In this situation, if we encounter an infinite loop inside the critical section or a deadlock due to the global mutex never being released, we give the fuzzer the opportunity to exit normally without reporting a bug, allowing it to start a new fuzzing run (line 11). Finally, in line 13, we execute the delay by running a *sleep* function for the duration value provided by the fuzzing engine. These duration values are generated using the UNIX *rand* function, with its initial seeds provided by the fuzzer's non-deterministic functions.

3.3.3.2 Limiting the number of active threads:

We monitor the count of active threads using two functions: *pthread_add()* and *pthread_release()* to limit the number of active threads inside the delay function. Their definitions are provided in Algorithm 4 and Algorithm 5, respectively. In Algorithm 4, the active thread counter, denoted as T_N, is incremented by one (line 3). Before this increment, the counter is

Algorithm 4 Function $pthread\_add()$

Global: $Mutex\_lock, T_N$ - active threads counter.

```
1: Function $pthread\_add()$
2: lock thread $\leftarrow Mutex\_lock$
3: $T_N$++
4: unlock thread $\leftarrow Mutex\_lock$
5: EndFunction
```

Algorithm 5 Function $pthread\_release()$

Global: $Mutex\_lock, T_N$ - active thread counter.

```
1: Function pthread_release()
2: lock thread $\leftarrow Mutex\_lock$
3: $T_N$- -
4: unlock thread $\leftarrow Mutex\_lock$
5: EndFunction
```

locked for synchronization, and it is unlocked afterward (lines 2 and 4). Conversely, Algorithm 5 handles the decrement of T_N (line 3). In this case, T_N is also locked to prevent data races between different threads, and it is unlocked afterward (lines 2 and 4).

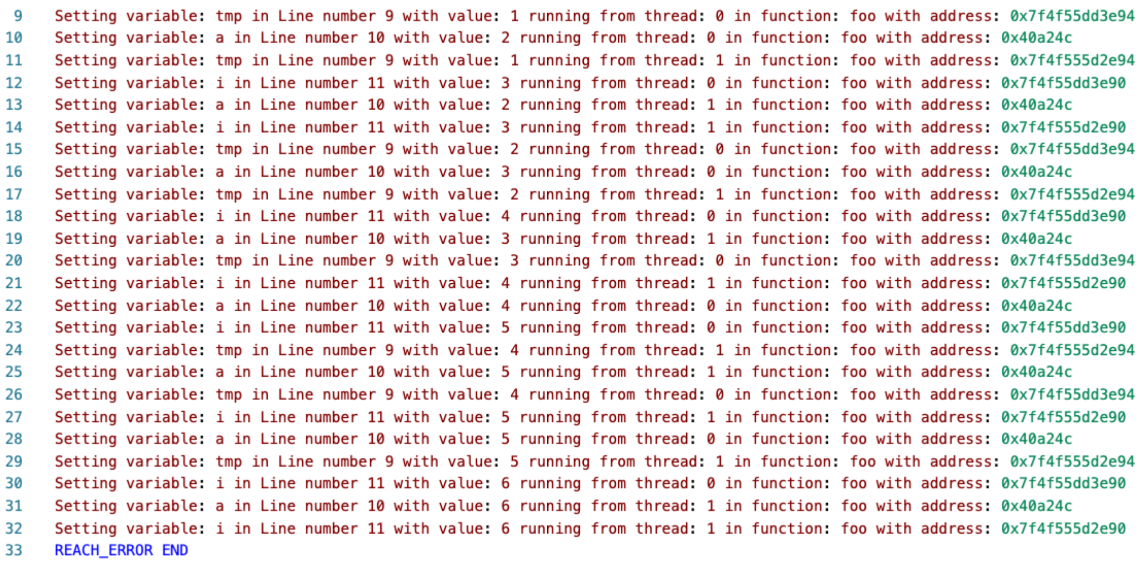
3.3.3.3 Counterexample

In concurrent programs, non-deterministic behaviour poses a significant challenge in detecting bugs and creating a trace that leads to the identified bug. Since the fuzzer outputs a core dump, it lacks the entire program trace, making it insufficient for generating the witness file [85]. To effectively address this challenge, we have defined two functions, namely *EBF_alloca()* and *EBF_add_store_pointer()*. These functions are crucial in collecting the necessary information to generate bug reports (counterexample). Such bug reports are valuable for users and tools, as they facilitate the tracing, reproduction, and confirmation of the detected violations.

Algorithm 6 and Algorithm 7 illustrate the definitions of the functions *EBF_alloca()* and *EBF_add_store_pointer()* respectively. The former function records the variable name, function name where the variable was declared, thread ID, and variable address (see line 3). The latter function records the variable's assigned value, the line of code where the assignment occurs in the PUT, the function name, thread ID, and address (see line 3). For example:

```
   int a; ⟶ EBF_Alloca("a", "function name", thread ID &a);
a = 42; ⟶ EBF_add_store_pointer(&a, line_dbg, "function name", thread
                                ID, 42);
```

To avoid data races while reading and writing information to the file, both functions record the information atomically. This means that the thread is locked before writing, as shown in line 2, and the thread is unlocked afterward, as shown in line 4.



```
Setting variable: tmp in Line number 9 with value: 1 running from thread: 0 in function: foo with address: 0x7f4f55dd3e94
Setting variable: a in Line number 10 with value: 2 running from thread: 0 in function: foo with address: 0x40a24c
Setting variable: tmp in Line number 9 with value: 1 running from thread: 1 in function: foo with address: 0x7f4f555d2e94
Setting variable: i in Line number 11 with value: 3 running from thread: 0 in function: foo with address: 0x7f4f55dd3e90
Setting variable: a in Line number 10 with value: 2 running from thread: 1 in function: foo with address: 0x40a24c
Setting variable: i in Line number 11 with value: 3 running from thread: 1 in function: foo with address: 0x7f4f555d2e90
Setting variable: tmp in Line number 9 with value: 2 running from thread: 0 in function: foo with address: 0x7f4f55dd3e94
Setting variable: a in Line number 10 with value: 3 running from thread: 0 in function: foo with address: 0x40a24c
Setting variable: tmp in Line number 9 with value: 2 running from thread: 1 in function: foo with address: 0x7f4f555d2e94
Setting variable: i in Line number 11 with value: 4 running from thread: 0 in function: foo with address: 0x7f4f55dd3e90
Setting variable: a in Line number 10 with value: 3 running from thread: 1 in function: foo with address: 0x40a24c
Setting variable: tmp in Line number 9 with value: 3 running from thread: 0 in function: foo with address: 0x7f4f55dd3e94
Setting variable: i in Line number 11 with value: 4 running from thread: 1 in function: foo with address: 0x7f4f555d2e90
Setting variable: a in Line number 10 with value: 4 running from thread: 0 in function: foo with address: 0x40a24c
Setting variable: i in Line number 11 with value: 5 running from thread: 0 in function: foo with address: 0x7f4f55dd3e90
Setting variable: tmp in Line number 9 with value: 4 running from thread: 1 in function: foo with address: 0x7f4f555d2e94
Setting variable: a in Line number 10 with value: 5 running from thread: 1 in function: foo with address: 0x40a24c
Setting variable: tmp in Line number 9 with value: 4 running from thread: 0 in function: foo with address: 0x7f4f55dd3e94
Setting variable: i in Line number 11 with value: 5 running from thread: 1 in function: foo with address: 0x7f4f555d2e90
Setting variable: a in Line number 10 with value: 5 running from thread: 0 in function: foo with address: 0x40a24c
Setting variable: tmp in Line number 9 with value: 5 running from thread: 1 in function: foo with address: 0x7f4f555d2e94
Setting variable: i in Line number 11 with value: 6 running from thread: 0 in function: foo with address: 0x7f4f55dd3e90
Setting variable: a in Line number 10 with value: 6 running from thread: 1 in function: foo with address: 0x40a24c
Setting variable: i in Line number 11 with value: 6 running from thread: 1 in function: foo with address: 0x7f4f555d2e90
REACH_ERROR END
```

Figure 3.2. A snippet of the counterexample ($witnessInfoAFL\_pid$) generated by our *OpenGBF* resulted from our example presented in 3.1.

Algorithm 6 Function *EBF_alloca()*

Inputs: a – variable name, f – function name, tid – thread id,&a – variable address.
Global: $Mutex\_lock$, $witnessInfoAFL_{pid}$ – witness file for the process with ID = pid.

```
1: Function EBF_alloca($a, f, tid, \&a$)
2: lock thread ← $Mutex\_lock$
3: $witnessInfoAFL_{pid} \leftarrow write(a, f, tid, \&a)$
4: unlock thread ← $Mutex\_lock$
5: EndFunction
```

Therefore, since we can have different threads from different processes accessing the same file simultaneously for writing, which can lead to a deadlock between different threads, we require a separate file for each process to write to. So, when the fuzzing process starts, we execute a constructor function that runs before the main function is called in the PUT. This function initializes the mutex *EB_mutex*, obtains the process ID (pid), and creates a file uniquely identified by the process ID (i.e., *witnessInfoAFL_pid*). This file will contain the information recorded by *EBF_alloca()* and *EBF_add_store_pointer()*.

On the one hand, when the fuzzing process completes (either due to reaching a timeout or normal termination with exit code 0), we delete all the created files using a destructor function [153]. On the other hand, if the fuzzer detects a crash in one of the PUT executions, we retrieve the ID of the crashed process, save the file associated with this process ID, and remove the others. This file should contain the information that caused the crash, such as the sequence of operations (i.e., memory accesses) that led to the crash of the PUT. Lastly, we convert the content of this file into a specific format (*GraphML format*) that is accepted by the witness validators. Figure 3.2 displays the content of the *witnessInfoAFL_pid* file generated by *OpenGBF* during the fuzzing process. This file contains all the information recorded by *EBF_alloca()* and *EBF_add_store_pointer()*.

Algorithm 7 Function *EBF_add_store_pointer()*

Inputs: $\&a$ - variable address, l - line number in the code, f - function name, tid – thread id, v - variable value.
Global: $Mutex\_lock$, $witnessInfoAFL_{pid}$ – witness file for the process with ID = pid.

```
1: Function EBF_add_store_pointer($\&a, l, f, tid, v$)
2: lock thread ← $Mutex\_lock$
3: $witnessInfoAFL_{pid} \leftarrow write(\&a, l, f, tid, v)$
4: unlock thread ← $Mutex\_lock$
5: EndFunction
```

3.3.3.4 Harnessing functions

This section aims to enhance the capabilities of our *OpenGBF* by incorporating essential functions specifically tailored to meet the requirements of our evaluation benchmarks (i.e., SV-COMP benchmarks[154]). We have implemented these functions within our runtime library, which can be invoked while executing the PUT. These functions are designed to ensure the successful compilation of the PUT in our *OpenGBF*.

Atomic functions. To guarantee the absence of interleavings within an atomic block of instructions, denoted by an initial call to *_VERIFIER_atomic_begin()* and ending with *_VERIFIER_atomic_end()*, *OpenGBF* employs a locking mechanism held by the current thread. This mechanism prevents other execution paths from resuming until the atomic block is finished. The lock is implemented through a global mutex, checked inside the delay function. Also, this delay function executes after each instruction, effectively halting the progress of other threads until the atomic block releases the lock. The delay function is not injected within the atomic block to avoid introducing unnecessary interleavings.

Non-deterministic functions. These are abstract functions that provide answers to specific problems within the PUT. In this case, they model non-deterministic values that can generate the values resulting in finding the property violation when the program is executed using the produced values. These functions are defined with the prefix *VERIFIER_-nondet<X>()*, where X can be any C type (e.g., integer, double, char, and so on). In *OpenGBF*, this is achieved by reading the fuzzer inputs (through `stdin`). These inputs can be read either in byte representation or string representation. For byte representation, we rely on the `fread()` function, and for string representation, we rely on `fscanf()`.

3.3.4 Full illustrative example

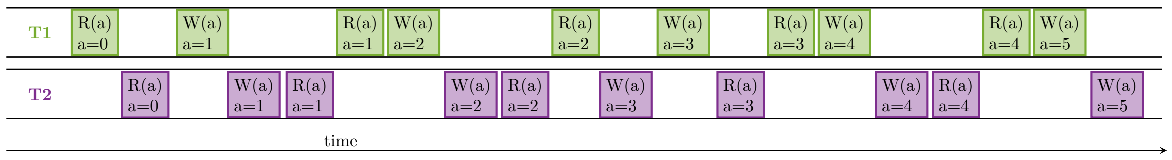
By connecting all the design choices, we illustrate using an example referred to as `thesis_-example.c` throughout this thesis. Suppose a multithreaded program contains a reachability bug (user-defined assertions), as shown in Listing 3.1. The program consists of two threads, $t1$ and $t2$ (see lines 17 and 18), both of which are mis-synchronized and call the same function *foo*. The function *foo* (see line 6) contains a while loop with 5 iterations, incrementing a variable a by 1 during each iteration (see line 10). Since both threads call this function, the value of the variable a at the end of the function execution should be 10. In line 21, a conditional statement checks whether this condition holds. If the value of a is not as expected (i.e., not equal to 10), we report a crash (property violation). However, this error statement can only be reached if the two threads are mis-synchronized when reading and writing to a.

Listing 3.1. Original multi-threaded C code (`thesis_example.c`).

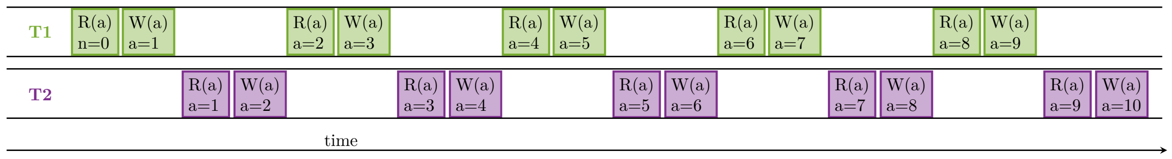
```
1  #include<pthread.h>
2  #include<stdlib.h>
3  #include<assert.h>
4  void reach_error(){ assert(0);}
5  int a=0; //shared variable
6  void* foo(void* arg) {
7   int tmp, i=1;
8   while (i<=5) {
9   tmp = a;
10  a = tmp + 1;
11  i++;
12  }
13  return 0;
14 }
15 int main () {
16  pthread_t t1, t2;
17  pthread_create(&t1, 0, foo, 0);
18  pthread_create(&t2, 0, foo, 0);
19  pthread_join(t1, 0);
20  pthread_join(t2, 0);
21  if ((a) != 10)  reach_error();
22 }
```

Figure 3.3 presents a visualization of the memory access of variable a in Listing 3.1. $T1$ refers to thread 1, and $T2$ refers to thread 2, with each thread capable of reading and writing to variable a. Figure 3.3a illustrates one of the scenarios that can lead to interleavings reaching the violation statement when the two threads are mis-synchronized. Thread 1 ($T1$) starts execution first and reads the initialized value of $a = 0$ in Line 5. Then thread 2 ($T2$) starts execution and also reads the value of $a = 0$ before $T1$ writes to $a = 1$ (see line 10). Since $T1$ reads the value as $a = 0$, it updates the value to $a = 1$, and $T2$ also reads $a = 0$ and updates the value to $a = 1$. This pattern repeats until $a = 5$ is reached rather than the expected value of $a = 10$.

The ideal scenario is when both threads read and write the value of a without interfering with each other (e.g., locking the current thread while reading and writing to a global vari-



(a) Mis-synchronized memory accesses.



(b) Synchronized memory accesses.

Figure 3.3. Visualization of the memory accesses to variable a in Listing 3.1 for two different thread interleavings. In Figure 3.3a, the accesses are mis-synchronized: both `T1` and `T2` read a before simultaneously updating the value to $a = 1$. This pattern continues until the end of the execution, where the final value will be $a = 5$. Conversely, Figure 3.3b shows synchronized accesses: `T1` reads a and updates the value to $a = 1$, then `T2` reads a and updates the value to $a = 2$. In the end, the final value will be $a = 10$, as expected.

able). Figure 3.3b presents this scenario when synchronizing the two threads. $T1$ reads the initialization value $a = 0$ and updates the value to $a = 1$, then $T2$ reads the value $a = 1$ and updates it to $a = 2$. This pattern ensures the property will hold, resulting in the final value of $a = 10$.

When verifying the code in Listing 3.1 using *EBF* (as explained in Chapter 4), we identified two different bugs. Our *OpenGBF* detected a data race caused by mis-synchronized threads between $T1$ and $T2$, while all the BMC tools identified a reachability bug at Line 21.

Furthermore, let us provide an example of our instrumentation at the LLVM-IR level. We will use the function *foo* from our code example in Listing 3.1.

Firstly, Listing 3.2 shows the original LLVM-IR for the *foo* function before any instrumentation is applied. Please note that each code statement corresponds to multiple LLVM instructions.

Secondly, Listing 3.3 shows the corresponding LLVM-IR for the *foo* function after our instrumentation. Specifically, Line 7 and Line 20 include calls to the *EBF_add_store_pointer* function. These calls are injected after each load instruction to record variable values, which are then saved in a file used for generating the witness file.

Additionally, Line 10, 13, and 16 demonstrate calls to the *EBF_alloca* function. These calls are injected after each allocation to record metadata about any variables declared in the PUT. This information is saved in a file to generate the witness file.

Finally, Line 11, Line 14, Line 17, and Line 22 introduce the *_delay_function*, which is injected after each instruction.

Listing 3.2. Snippet of the corresponding IR **before** instrumentation.

```
define dso_local i8* @foo(i8* %0) #0 {
  %2 = alloca i8*, align 8
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i8* %0, i8** %2, align 8
  store i32 1, i32* %4, align 4
  br label %5
  %6 = load i32, i32* %4, align 4
  %7 = icmp sle i32 %6, 5
  br i1 %7, label %8, label %14
  %9 = load i32, i32* @a, align 4
  store i32 %9, i32* %3, align 4
  %10 = load i32, i32* %3, align 4
  %11 = add nsw i32 %10, 1
  store i32 %11, i32* @a, align 4
  %12 = load i32, i32* %4, align 4
  %13 = add nsw i32 %12, 1
  store i32 %13, i32* %4, align 4
  br label %5
  ret i8* null
}
```

Listing 3.3. snippet of the corresponding IR **after** instrumentation.

```
define dso_local i8* @foo(i8* %0) #0 {
  %2 = alloca i8*, align 8
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = bitcast i8* %0 to i1*
  %6 = bitcast i8** %2 to i8*
  call void @EBF_add_store_pointer(i8* ↩
      %6, i64 0, i8* getelementptr ↩
      inbounds ([4 x i8], [4 x i8]* @0,↩
       i32 0, i32 0), i1* %5)
  store i8* %0, i8** %2, align 8
  %7 = bitcast i8** %2 to i8*
  call void @EBF_alloca(i8* ↩
      getelementptr inbounds ([4 x i8],↩
       [4 x i8]* @1, i32 0, i32 0), i8*↩
       getelementptr inbounds ([4 x i8↩
      ], [4 x i8]* @2, i32 0, i32 0), ↩
      i8* %7),
  call void @_delay_function()
  %8 = bitcast i32* %3 to i8*
  call void @EBF_alloca(i8* ↩
      getelementptr inbounds ([4 x i8],↩
       [4 x i8]* @3, i32 0, i32 0), i8*↩
       getelementptr inbounds ([4 x i8↩
      ], [4 x i8]* @4, i32 0, i32 0), ↩
      i8* %8),
  call void @_delay_function()
  %9 = bitcast i32* %4 to i8*
  call void @EBF_alloca(i8* ↩
      getelementptr inbounds ([2 x i8],↩
       [2 x i8]* @5, i32 0, i32 0), i8*↩
       getelementptr inbounds ([4 x i8↩
      ], [4 x i8]* @6, i32 0, i32 0), ↩
      i8* %9),
  call void @_delay_function()
  %10 = sext i32 1 to i64
  %11 = bitcast i32* %4 to i8*
  call void @EBF_add_store_pointer(i8* ↩
      %11, i64 6, i8* getelementptr ↩
      inbounds ([4 x i8], [4 x i8]* @7,↩
       i32 0, i32 0), i64 %10),
  store i32 1, i32* %4, align 4,
  call void @_delay_function(),
```

3.4 Summary

In this chapter, we discussed the primary challenge addressed in this thesis: the absence of open-source tools for fuzzing concurrent programs. We tackled this challenge by developing our open-source fuzzer, *OpenGBF*. Implementing a concurrency-aware fuzzer involves addressing four sub-challenges, including thread interleaving behaviours, detecting concurrency-related bugs, scaling the fuzzer, and dealing with the non-deterministic behaviour of concurrent programs.

The chapter introduces the design of the *OpenGBF* framework. Firstly, we present the high-level structure of *OpenGBF*, which comprises eight components (C program, sanitizer, LLVM Pass executable, runtime library, fuzzing process, verdict, and counterexample). Then, we present the detailed framework, where we instrument the PUT using the LLVM Pass and inject five functions to control thread interleavings (*_delay_function()*), two thread-monitoring functions (*pthread_add()* and *pthread_release()*), and record relevant information (*EBF_alloca()* and *EBF_add_store_pointer()*). We bundle these five instrumentation functions into a runtime library.

Specifically, we address the first sub-challenge, in which the scheduling of threads in concurrent programs can result in different thread interleaving behaviours. To explore these different thread interleavings and mitigate the introduction of unnecessary interleavings during code instrumentation, we inject the *_delay_function()* after each instruction. This introduces random delays imposed by the fuzzer to force various thread interleavings.

To address the second sub-challenge related to concurrency bugs that occur due to different thread interleavings in concurrent programs, we used the Thread Sanitizer with the fuzzer. This approach allowed us to add the necessary runtime checks to detect such vulnerabilities.

The third sub-challenge is that fuzzing concurrent programs can be computationally exhausting and time-consuming, especially with a huge number of active threads. We address this sub-challenge by injecting *pthread_add()* and *pthread_release()* functions. These functions work as counters, keeping track of the number of active threads. We control the number of active threads by setting a threshold inside the *_delay_function()*.

The last sub-challenge we address is the non-deterministic behaviour of concurrent programs, which makes it challenging to reproduce the exact input and interleavings that lead to the bug. To tackle this, we inject *EBF_alloca()* and *EBF_add_store_pointer()*. These functions record essential information needed to trace the bug, and later, we use this information to generate a witness file that can be confirmed using validators.

The chapter concluded with an illustrative example demonstrating how all the design choices and concepts connect together. The example analyzed a multithreaded program with a reachability bug, illustrating how our framework can detect data races and reachability bugs. We also showed the role of instrumentation at the LLVM-IR level.

3.5 Future work

In future work, several interesting ideas for enhancing the bug detection capabilities of our *OpenGBF* framework. Firstly, we could explore strategies to make the delay function more efficient by reducing the number of delays injected after each instruction. This optimization could dynamically adjust the delay intervals based on the specific program behaviour, potentially leading to more effective interleaving exploration. Secondly, we could incorporate advanced mutation algorithms based on machine learning techniques to introduce better-adapted inputs to find concurrency-related vulnerabilities. By extracting relevant features from program executions, machine learning models could generate inputs optimized for revealing complex concurrency bugs. These future endeavors can enhance *OpenGBF's* bug detection capabilities for both sequential and concurrent programs.

Chapter 4

EBF: A black-box cooperative verification for concurrent programs

4.1 Chapter introduction

Finding software vulnerabilities in concurrent programs is difficult due to the complex nature of concurrent programs. These vulnerabilities can have serious consequences, such as financial losses and threats to people's well-being. Therefore, various techniques are available to verify and test concurrent programs, including BMC and fuzzing, each with its strengths and weaknesses. Therefore, combining the strengths of BMC in resolving complex conditional guards with the flexibility of *OpenGBF* allows us, at least in theory, to improve performance by solving more problems (discovering more bugs). However, despite these techniques' availability, no approach is currently effectively combining BMC and fuzzing for the verification of concurrent programs [155].

In this context, there are several ways to combine different verification techniques, and the choice depends on the specific goals and challenges of the verification process. For example, cooperative verification [45] simplifies the communication interface between these tools by exchanging certain information (verification artifacts) to enhance the overall effectiveness of the verification process. Cooperative verification can be classified into black-box or white-box combinations based on analyzing the exchanged information [45], [156]. In the black-box combination, the combined tools are not modified. In contrast, the combined tools are modified based on the exchanged information required in the white-box combination.

Another example is the portfolio approach [156], where tools are independent of each other and can be distributed using various combinations, including sequential portfolios, parallel portfolios, and algorithm selections. Sequential portfolios execute multiple verification tools sequentially. In contrast, parallel portfolios execute all verification tools in parallel, sharing system resources like CPU time and memory. Algorithm selection chooses the most appropriate verification tool for a specific verification task.

Accordingly, our primary research question is whether combining BMC and fuzzing can improve bug detection performance in concurrent programs compared to using each technique individually. This question has a twofold answer. First, considering the existing state-of-the-art BMC tools for verifying concurrent programs, we recognized the need for an open-source, concurrency-aware fuzzer as the fuzzing engine. Therefore, we developed our *OpenGBF* in Chapter 3. Second, to effectively address our research question, we needed to develop and evaluate a cooperative framework. Hence, in this Ph.D. thesis, we adopt the philosophy of a black-box cooperative design, allowing us to share the verification results by instrumenting the PUT without modifying the tools employed in the framework. This flexibility in our framework allows virtually any existing BMC or GBF tools to be combined.

This chapter, based on materials from our recent publications [1] and [2], aims to provide a comprehensive explanation of our black-box cooperative framework called Ensemble Bounded Model Checking Fuzzing framework *EBF*, where *OpenGBF* plays a crucial role. Unlike the portfolio approach in previous work [156], [157], the execution of these tools in *EBF* is not entirely independent. Specifically, we use the verification witness file generated by BMC to seed *OpenGBF* as information exchange between the cooperative tools.

In the following sections, we will explain in more detail how we integrated our *OpenGBF* with the BMC engine within the *EBF* framework to enhance vulnerability detection in concurrent programs, where this integration leverages the combined capabilities of *OpenGBF* and the BMC engine. First, we will outline the challenges posed by combining both techniques in Section 4.2. Then, we will explain the main framework, which comprises four main stages elaborated in Sections 4.3.1.1, 4.3.1.2, 4.3.1.3, and 4.3.1.4. Section 4.3.1.5 provides details about the witness file generated by *EBF*, while Section 4.3.2 discusses the CPU time allocation choice within the *EBF* framework.

4.2 Challenges in designing black-box cooperative verification tool

BMC and fuzzing are fundamentally different approaches; therefore, their collaboration within the framework requires careful coordination. This includes addressing the following challenges:

1. **Input seed generation:** complex path conditions, such as (`if(x*x -2*x +1 == 0)`) are challenging for the fuzzer to explore, so providing effective initial seeds to cover most of the program's search space is essential to improve the fuzzer bug-finding capabilities (we address this challenge in Section 4.3.1.2).

2. **Result aggregation:** when implementing a cooperative framework of different tools, they are likely to return conflicting results. The main reason is that BMC relies on ab-

stractions of program execution states and symbolic execution, while the fuzzer tests concrete inputs and execution schedules. Therefore, when these two techniques disagree, we can make an informed decision regarding the final verification result and generate a witness file for both results. These witness files represent error paths that lead to the violation and can be used to confirm and reproduce the violation [158]. Our proposed approach to address this challenge is using a decision matrix (we address this challenge in Section 4.3.1.4).

3. **Resource allocation trade-off:** the main drawback of using a cooperative framework that combines different tools sequentially is that they all share the same computational resources. It is necessary to make resource allocation decisions for each tool, especially in programs with limited time, memory, computational power, or combined sequentially. Generally, these decisions depend on the problem at hand and the partial results obtained from the tools within the cooperative framework (we address this challenge in Section 4.3.2).

4.3 Designing cooperative black-box verification tool

Thanks to our *OpenGBF*, we now have both engines for effectively combining state-of-the-art BMC and GBF tools. We combine them sequentially, simplifying the communication interface between these tools. In this cooperative framework, we execute the BMC engine as a black box, obtaining the counterexample, extracting the seed values, and then executing *OpenGBF* as a black-box with these seed values. This black-box design offers several advantages. Firstly, it is universal; we can employ any BMC or GBF tool that accepts the C program as input. Secondly, the types of software vulnerabilities that can be detected using this design solely depend on the capabilities of each tool. For instance, most BMC tools can detect different memory-related vulnerability types, such as buffer overflows, invalid pointer dereferences, double frees, and use-after-free issues [159]. Certain BMC tools can also detect concurrency-related vulnerabilities like deadlocks and data races [17], [160]–[162].

As for *OpenGBF*, its primary role involves exploring different executions of the PUT by sampling various thread interleavings and program inputs. To evaluate whether such executions lead to a bug, we depend on the type of property we need to verify in a program, such as concurrency-related, memory-related, or reachability bugs (user-defines assertions). For detecting concurrency and memory-related bugs, *OpenGBF* relies on the capabilities of sanitizers [147]. These sanitizers introduce additional instrumentation to the PUT at runtime for bug detection [108]. Specifically, *ThreadSanitizer*[106] detects concurrency-related bugs, while *AddressSanitizer* [107] detects memory-related bugs.

4.3.1 *EBF* framework

Figure 4.1 provides an overview of the *EBF* framework, which consists of four main stages: *safety proving*, *seed generation*, *falsification*, and *results aggregation*. The first three stages take the concurrent C program and safety property as input.

In the safety proving stage, the primary focus is verifying program safety using the BMC engine. If BMC confirms the program is safe, the verdict is recorded in the decision matrix during the results aggregation stage. However, if BMC identifies the program as unsafe, this verdict is also recorded, and seed values are extracted from its counterexample (witness file) and stored in a seed corpus directory.

Following this initial round, the seed generation stage starts. Here, multiple error statements are injected after each branch in the program. Subsequently, one error statement is randomly chosen, and BMC is executed on that specific file. This process continues until the allocated time is reached or BMC has been run on all files containing error statements. For each successful run where BMC identifies a bug, seed values are extracted and stored in the seed corpus.

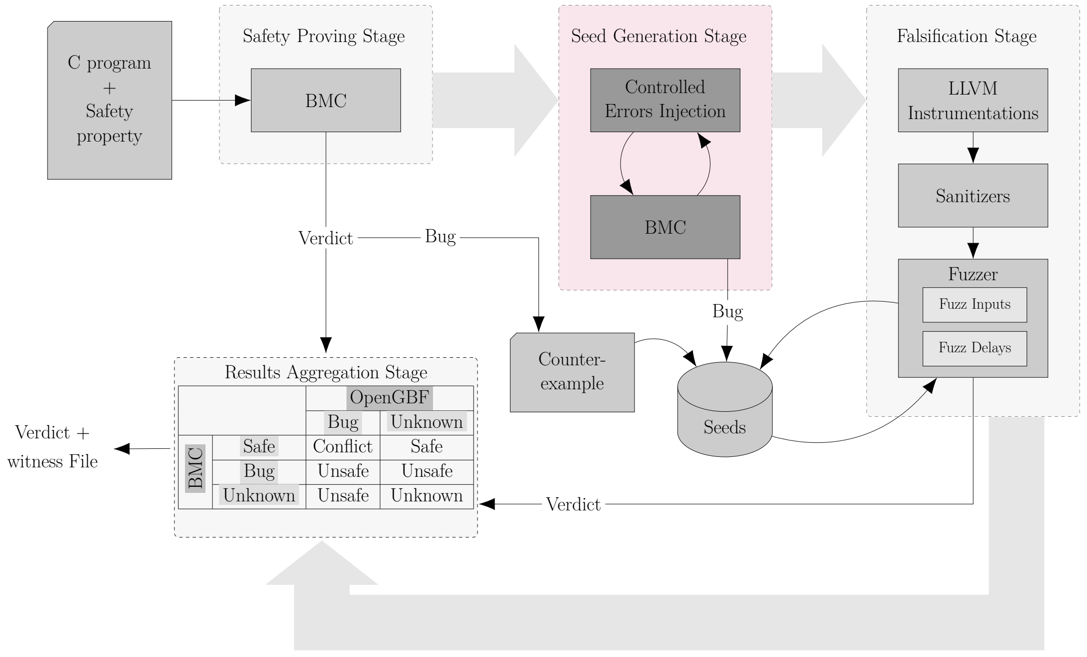
Next is the falsification stage, where *OpenGBF* is executed with the seed values stored in the seed corpus. During the allocated time for the fuzzing process, if *OpenGBF* detects a bug, it records the verdict in the decision matrix. Finally, in the results aggregation stage, the results are aggregated to generate the final verdict and its associated witness file.

In the following sections, we provide a detailed explanation of the *EBF* framework and its stages:

4.3.1.1 Safety proving stage

This stage is essential because the fuzzer cannot guarantee the exploration of all thread interleavings; it can only maximize code coverage since we do not track them as coverage metrics. Therefore, in this stage, we rely on the BMC to prove that a program is safe for a given bound and limited context switch (default is two context switches). Algorithm 8 demonstrates how this is achieved. First, we execute the BMC tool on the PUT and the safety property (see line 2). The BMC tool's output can be one of three verdicts: *Verification successful* if it confirms the PUT's safety concerning the given property, *Verification Failed* if it finds a violation, or *Unknown*, which can occur for various reasons, including reaching timeouts, running out of memory, or unexpected crashes (see lines 3 and 5).

On the one hand, if the BMC tool detects a vulnerability and generates a counterexample, a sequence of program inputs and thread interleavings leading to the vulnerability, we store



		OpenGBF	
		Bug	Unknown
BMC	Safe	Conflict	Safe
	Bug	Unsafe	Unsafe
	Unknown	Unsafe	Unknown

Figure 4.1. *EBF* framework.

these input values as initial seeds to be used in the Falsification Stage, where we employ *OpenGBF* (see lines 6 and 7). Extracting the seeds is only possible when the BMC tool reports a failed verification result. Notably, these seeds are concrete values that cause an assertion to fail. However, despite BMC tools providing thread scheduling information in their bug reports, it is not always straightforward to determine the necessary delay values for replicating these bug-inducing schedules.

On the other hand, if BMC proves the program is safe, we record this verdict in the results aggregation stage, where we use a decision matrix to determine the final verdict (see line 4).

4.3.1.2 Seed generation stage

This stage was introduced in *EBF* version 4.2 [2] to address the challenge in designing our cooperative framework, where the Gray-Box Fuzzer (GBF) requires initial seed values to initiate the fuzzing process, especially for exploring complex path conditions, such as `if(x*x - 3*x + 2 == 0)`. When the BMC could not generate a counterexample for reasons like exceeding the time limit or producing an *Unknown* result, we still need to seed the fuzzer. To overcome this challenge, we analyze the PUT and adopt an approach similar to that proposed by *FuseBMC* and *LibKluzzer* [43], [133] by repeatedly inserting the error statement `assert(0)` into each conditional branch of the PUT

Specifically, after an *if* statement, a *while* loop, or a *do-while* loop (see Algorithm 8, line 10), we insert a numerical label in the form of $LABEL\_i$ (e.g., LABEL_1, LABEL_-2, and so on), as demonstrated in Listing 4.1. In this particular case, there are six labels.

Algorithm 8 *EBF* Overall Framework

```
Input: $PUT$ – program under test, $P$ safety property .
Output: $V$ – verdict, $W$ – witness file.
Shorthands:
$\lambda_a ss - assert(0)$;
$S$ - Seed values
$B$ - The output from Algorithm 2
 1: $M \leftarrow PUT$
 2: $V \leftarrow Run\_BMC\ (PUT, P)$ {Run BMC tool for safety proving}
 3: if $V == Verification\_successful$ then
 4:     $V \leftarrow Aggregation\_table$ {Save the verdict in the aggregation table}
 5: else if $V == Verification\_Failed$ then
 6:     $S \leftarrow Extract\_seed\_Values$ {Extract the non-deterministic values from the BMC counterexample}
 7:     $Save(S) \leftarrow seed\_directory$
 8: end if
 9: for all $Conditional\_Branches\_in\_M$ do
10:     $E_i \leftarrow inject(\lambda_a ss(0))$ {inject error label in every branch in the program}
11: end for
12: $i \leftarrow number\_of\_error\_label$ {save how many error label we inject}
13: for all $i$ and $time\ ! = 0$ do
14:     $Run\_BMC(E_i)${For each error label $i$, and it is not a timeout, we run BMC tool}
15:     if $V == Verification\_Failed$ then
16:         $Save(S) \leftarrow seed\_directory$ {if BMC reaches this label, then we extract the values and save them in
            a seed directory}
17:     end if
18: end for
19: $V \leftarrow Run\_Fuzzer\ (M, S, P, B)$ {we run OpenGBF with the original PUT and the seed, property(Sani-
    tizer) and LLVM pass}
20: if $V == unknwon$ then
21:     $V \leftarrow Aggregationtable$ {if the fuzzer could not decide, we save the verdict in the aggregation table as
        unknown}
22: else
23:     $V \leftarrow Aggregationtable$ {if the fuzzer found the bug, we save the verdict in the aggregation table as
        Verification_failed}
24:     $Generate\_witness\_file$ {the file $witnessInfoAFL_pid$ will be generated}
25: end if
26: $V \leftarrow Aggregation\_table$ {generate the final verdict and the witness file (.graphml) corresponding to the
    tool we aggregate the verdict from}
27: return $V$ and $Witnessfile$
```

LABEL_1 is added before closing the function, LABEL_2 and LABEL_5 are added after the *while* and *if* statements' branch conditions, respectively, LABEL_3 and LABEL_6 are added after exiting the branch conditions, and finally, LABEL_4 is added before the *return* statement [43].

The difference between *EBF*, *FuseBMC* and *LibKluzzer* [43], [133] in this regard lies in their label selection methods. In *FuseBMC*, they use a tracer to select seed values with the highest impact considering different metrics (e.g., input size) [43], while *LibKluzzer*, runs multiple instances of coverage-guided and whitebox fuzzing simultaneously, sharing a corpus of seed values and tracking their progress. In contrast, *EBF* uses a more straightforward strategy. After obtaining the code containing all the labels, *EBF* randomly selects a label and replaces it with an error statement. For example, if LABEL_1 is chosen, *EBF* randomly replaces LABEL_1 with the error statement. This process is repeated for each insertion, resulting in an instrumented program with a specific error statement. However, it is important to note that the effectiveness of the seed generation stage can be affected by the strategy used to prioritize label replacement.

Once *EBF* replaces the labels, the BMC tool is used to verify each instrumented program individually. If the BMC tool reaches the intended error statement within the specified timeout, *EBF* generates a witness file for each label. Each witness file is given a name such as $thesis\_example\_1\_reach.c.graphml$, corresponding to the label it represents. The non-deterministic values in the generated $graphml$ file are then extracted and converted into seed values for the fuzzer. This extraction of seed values from witness files continues until the BMC tool reaches all the injected error statements or the predefined timeout is reached. Then, these seed values are stored in a dedicated directory for seeds. It is important to note that *EBF* only preserves unique and non-duplicated seed values. Extracting and storing the seed values to provide them to the GBF can help in reaching deeper paths, thus enhancing the bug-finding capability of the fuzzer during the Falsification stage (see Section 4.3.1.3).

However, in the case where the BMC tool cannot reach any of the error statements, either due to a timeout or producing an *Unknown* result, we still require to provide the fuzzer with initial seeds to start the fuzzing process. Therefore, *EBF* generates random strings and saves them in the seed corpus directory. Note that these values serve as an initial seed for the fuzzer and are unrelated to the delay values within the delay function. The delay values, conversely, are generated by a separate random number generator that derives its seed from one of the harnessing functions presented in Section 3.3.3.4.

Listing 4.1. An instrumented code with adding labels correspond to the code in Listing 3.1.

```
   #include<pthread.h>
#include<stdlib.h>
#include<assert.h>
void reach_error (){ assert(0);
LABEL_1:; //label 1
}
 int a=0;
 void* foo(void* arg) {
 int tmp , i=1;
 while (i <=5) {
LABEL_2:; //label 2
 tmp = a;
 a = tmp + 1;
 i++;
 }
LABEL_3:; //label 3
LABEL_4:; //label 4
return 0;
```

```
 }
int main () {
pthread_t t1 , t2;
pthread_create (&t1 , 0, foo , 0);
pthread_create (&t2 , 0, foo , 0);
pthread_join(t1 , 0);
pthread_join(t2 , 0);
if ((a) != 10)
{
LABEL_5:; //label 5
 reach_error ();
}
LABEL_6:; //label 6
}
```

4.3.1.3 Falsification stage

This stage is built on top of *OpenGBF*, which was introduced previously in Chapter 3. In this stage, we compile the original PUT without error statements using the LLVM Pass and the runtime library (we use sanitizer flags depending on which property we intend to verify). Then, we fuzz the binary with the initial seeds generated in the seed generation stage. Based on the verdict (either *Verification Failed* or *Unknown*), we save the output for the results aggregation stage.

Recall that GBF does not guarantee that we have exhaustively explored the entire search space because it only tracks code coverage, not thread interleavings, making it difficult to confirm the program's safety. Therefore, to prove that a program is safe up to a context switch, we rely on BMC alone (see Table 4.1). Additionally, achieving a specific thread order by injecting delays without a scheduling algorithm is nearly impossible because the impact of these introduced delays depends on the multi-threaded program's implementation within the corresponding operating system (e.g., sometimes, the same delay values may lead to the execution of different thread schedules). Hence, *OpenGBF* uses only the bug-inducing inputs and not the thread schedule (thread interleavings) information as seed values. This means it may be challenging for *OpenGBF* to replicate every bug detected by the BMC tool since it might not be able to sample the sequence of delay values that reproduce the bug-inducing thread interleavings. However, this approach allows *OpenGBF* to explore different randomly generated interleavings that may lead to the discovery of other bugs.

4.3.1.4 Results aggregation stage

One of the challenges of developing a cooperative framework is the conflict outcome, when BMC and GBF may disagree on proving the program's safety. Therefore, after running all the cooperative tools, *EBF* needs to aggregate their outcomes and generate a witness file. The decision matrix in Table 4.1 outlines our aggregation rules.

The first rule is straightforward: If one of the tools cannot conclude the outcome (e.g., *unknown*), we trust the other tool. Specifically, when GBF could not find the bug and reports *unknown*, our decision matrix aligns with the result from the BMC tool. Vice versa, when the BMC tool cannot find the bug or prove the program's safety, we rely on the bug found by the GBF tool.

The second rule applies when there is a disagreement between the cooperative tools. For example, when the BMC tool proves the program's safety and produces *Verification Successful*, while GBF may find a bug and produce *Verification Failed*, in this interesting scenario, *EBF* reports *Conflict* and generate witness file for both bugs to provide further confirmation. This can happen because of the over-approximation in the computational models of the BMC tool or because our *OpenGBF* introduces a bug in our instrumented code. We can resolve such *Conflict* by analyzing the witness file generated by *EBF*.

To provide a more detailed explanation of such a case, let us consider the example in Listing 4.2. This example demonstrates a simple code in which the created threads call the functions `t1()` and `t2()` respectively (see lines 6 and 10). These two functions implement the Fibonacci Sequence [163], where the variable i adds the value of variable j, and vice versa until the loop condition is reached. Then, we retrieve the value of $correct$ in line 25, which is the value returned by the function `foo()` (see line 14). The function `foo()` also follows the Fibonacci Sequence rule. In line 26, a condition checks if both i and j are greater than $correct$, leading to an error. Verifying this example using *EBF* produces a *Conflict* verdict. *OpenGBF* detects the bugs, while *ESBMC* produces a *Verification Successful* result. This discrepancy occurs because *ESBMC* uses two context switches by default, as it is fast at verifying programs. Additionally, many concurrency bugs in real applications are shallow, requiring only a few context switches to expose them [63]. However, *ESBMC* can detect the bug and produce a *Verification Failed* verdict if we increase context switches by more than two.

Listing 4.2. A simple code where *ESBMC* report *Verification_true* and *OpenGBF* report *Verification_failed*.

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  void reach_error() {assert(0);}
5  int i=1, j=1;
6  void *t1(void *arg) {
7    for (int p = 0; p < 5; p++) {
8      i = i + j;}
9    return (0);}
10 void *t2(void *arg) {
```

EBF		GBF	
		Bug	Unknown
BMC	Safe	Conflict	Verification Successful
	Bug	Verification Failed	Verification Failed
	Unknown	Verification Failed	Unknown

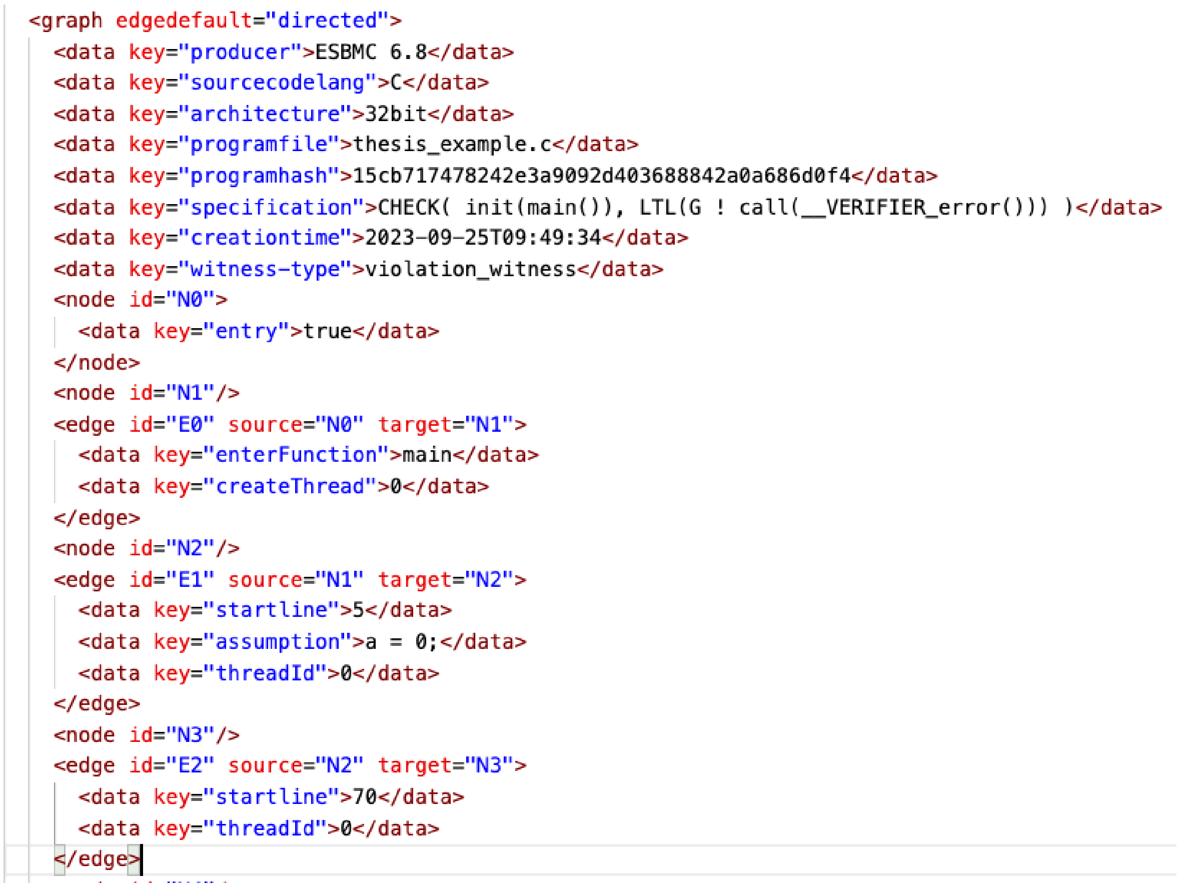
Table 4.1. *EBF* reports a program Safe (*Verification Successful*), Unknown, Unsafe (*Verification Failed*) or reports a Conflict by aggregating the outputs of *BMC* and *GBF*.

```
  for (int q = 0; q < 5; q++) {
    j = j + i;}
  return (0);}
int foo() {
  int cur = 1, prev = 0, next = 0;
  for (int x = 0; x < 10; x++) {
    next = prev + cur;
    prev = cur;
    cur = next;}
  return prev;}
  int main() {
  pthread_t id1, id2;
  pthread_create(&id1, 0, t1,0);
  pthread_create(&id2, 0, t2,0);
  int correct =  foo();
  if(i > correct && j > correct) reach_error();
  return 0;}
```

Note that the rules established in the decision matrix, as proposed in Table 4.1, were motivated by the rules of the Software Verification competition (i.e., SV-COMP [164]), where interactive verification is not available [77]. At the same time, wrong verdicts are penalized through deductions of competition points (for more details, refer to Section 6.3). Nevertheless, verifying complex software systems can benefit from a more descriptive decision matrix. For instance, it can differentiate between various causes leading to *Verification Failed* outcomes in Table 4.1, such as identifying specific bugs like reachability or data races (see Section 2.2). Ultimately, based on the decision matrix, the final verdict will be presented as the final outcome. If the final decision (i.e., verdict) is *Verification Failed*, *EBF* proceeds to generate a final witness.

4.3.1.5 Final witness generation

In *EBF*, a final witness file is generated only when the outcome is determined to be *Verification Failed*. This decision is based on the fact that no witness validator is available to validate a correct witness for concurrent programs when writing this Ph.D. thesis [165]. Validating a witness is as challenging as the verification problem itself. Therefore, the witness



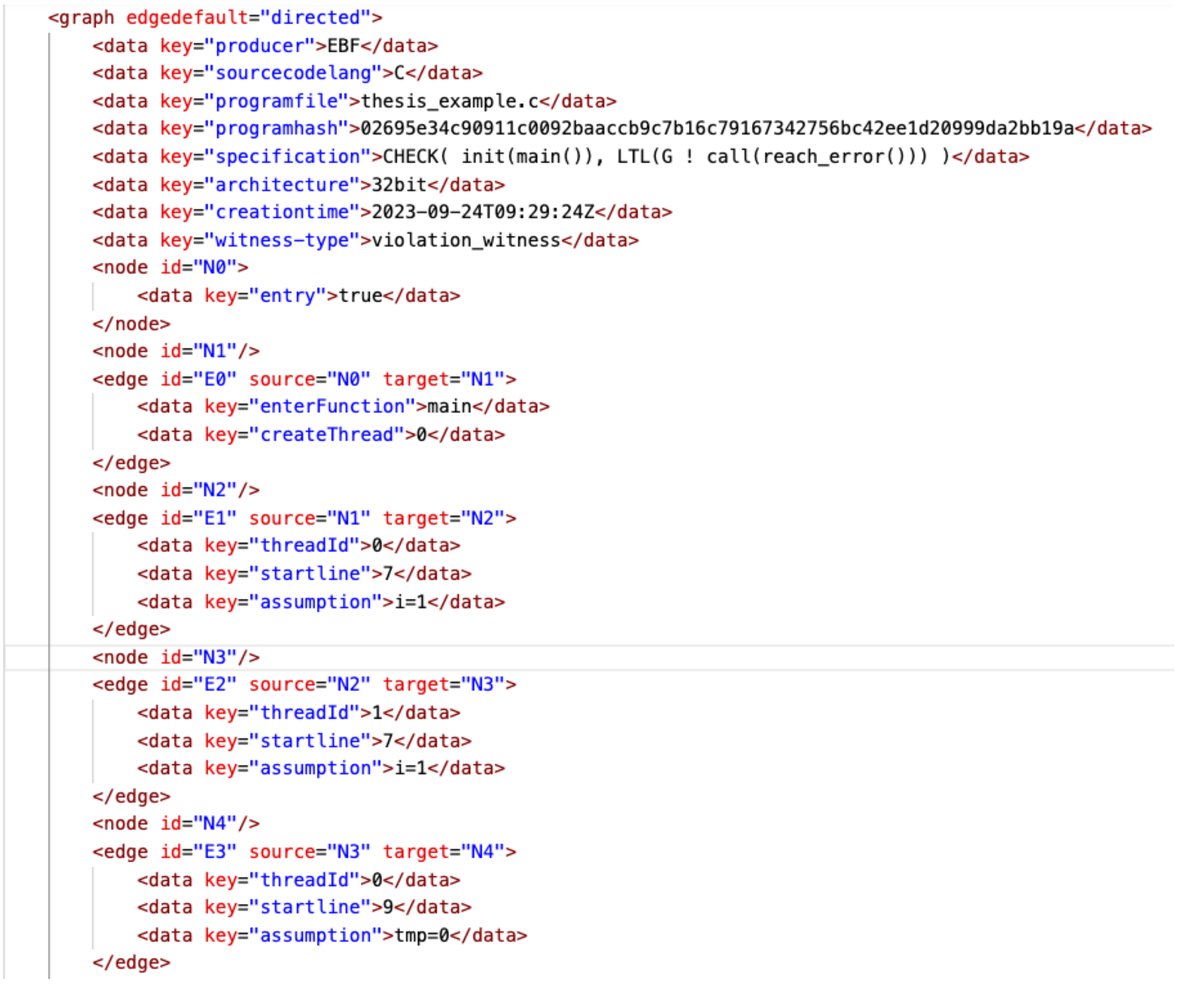
```
<graph edgedefault="directed">
  <data key="producer">ESBMC 6.8</data>
  <data key="sourcecodelang">C</data>
  <data key="architecture">32bit</data>
  <data key="programfile">thesis_example.c</data>
  <data key="programhash">15cb717478242e3a9092d403688842a0a686d0f4</data>
  <data key="specification">CHECK( init(main()), LTL(G ! call(__VERIFIER_error())) )</data>
  <data key="creationtime">2023-09-25T09:49:34</data>
  <data key="witness-type">violation_witness</data>
  <node id="N0">
    <data key="entry">true</data>
  </node>
  <node id="N1"/>
  <edge id="E0" source="N0" target="N1">
    <data key="enterFunction">main</data>
    <data key="createThread">0</data>
  </edge>
  <node id="N2"/>
  <edge id="E1" source="N1" target="N2">
    <data key="startline">5</data>
    <data key="assumption">a = 0;</data>
    <data key="threadId">0</data>
  </edge>
  <node id="N3"/>
  <edge id="E2" source="N2" target="N3">
    <data key="startline">70</data>
    <data key="threadId">0</data>
  </edge>
```

Figure 4.2. A snippet of the witness file generated from *ESBMC* corresponds to our example presented in 3.1.

validator would need to rely on verification techniques that work for concurrency problems (e.g., sequentialization and bounded context-switch). As explained in Section 3.3.3.3, all the necessary information required for generating the witness is recorded during fuzzing. However, since the final outcome is affected by the decision matrix discussed in Section 4.3.1.4, the content of the witness file also relies on the specific tool from which the final verdict is obtained.

When BMC produces the verdict: we extract the variable names, the non-deterministic values, and the thread ID from their witness file. Note that all the BMC tools participating in SV-COMP generate such a witness [77]. Various tools produce witness files with different levels of verbosity. Therefore, we apply a filter to retain only non-deterministic assignments by referring to the original source code using the line numbers provided by the witness. Figure 4.2 shows a snippet of such a witness generated by *ESBMC*, corresponding to the example in Listing 3.1. It includes all the key nodes explained in the background (see Section 2.4.1).

When *OpenGBF* produces the verdict: we obtain the variable names, variable values, and thread ID from the $witnessInfoAFL\_pid$ file generated by *OpenGBF* during runtime (see Section 3.3.3.3) using regular expressions (RegEx). These RegEx patterns, em-



```xml
<graph edgedefault="directed">
    <data key="producer">EBF</data>
    <data key="sourcecodelang">C</data>
    <data key="programfile">thesis_example.c</data>
    <data key="programhash">02695e34c90911c0092baaccb9c7b16c79167342756bc42ee1d20999da2bb19a</data>
    <data key="specification">CHECK( init(main()), LTL(G ! call(reach_error())) )</data>
    <data key="architecture">32bit</data>
    <data key="creationtime">2023-09-24T09:29:24Z</data>
    <data key="witness-type">violation_witness</data>
    <node id="N0">
        <data key="entry">true</data>
    </node>
    <node id="N1"/>
    <edge id="E0" source="N0" target="N1">
        <data key="enterFunction">main</data>
        <data key="createThread">0</data>
    </edge>
    <node id="N2"/>
    <edge id="E1" source="N1" target="N2">
        <data key="threadId">0</data>
        <data key="startline">7</data>
        <data key="assumption">i=1</data>
    </edge>
    <node id="N3"/>
    <edge id="E2" source="N2" target="N3">
        <data key="threadId">1</data>
        <data key="startline">7</data>
        <data key="assumption">i=1</data>
    </edge>
    <node id="N4"/>
    <edge id="E3" source="N3" target="N4">
        <data key="threadId">0</data>
        <data key="startline">9</data>
        <data key="assumption">tmp=0</data>
    </edge>
```

Figure 4.3. A snippet of the Final witness file (*EBF_thesis_example.c.graphml*) generated from our *EBF* after aggregating the result.

ployed to match specific text patterns, simplify extracting relevant information from extensive text sources. Listing 4.3 illustrates the syntax of such expressions, represented by (.*), which captures any sequence of characters (excluding newline characters). The content captured by these patterns typically corresponds to the variable name, line number, variable value, thread ID, function name, and memory, all the information required to generate the final witness as shown in Figure 4.3. Subsequently, we format the extracted information in a *GraphML*-based format and name it as *EBF_ProgramName.programExtension.graphml* (e.g., *EBF_thesis_example.c.graphml*).

Listing 4.3. The RegEx used to extract the information from $witnessInfoAFL\_pid$.

```
Setting variable: (.*) in Line number (.*) with value: (.*) running from thread: ↩
    (.*) in function: (.*) with address:(.*)
```

Figure 4.3 presents the final witness generated by *EBF* in the *GraphML*-based format. Both examples correspond to the one in Listing 3.1. It is worth noting that the assumption values in the final witness file are presented in string format to enhance readability for the user.

In the last phase of the verification process, this witness file will be validated to confirm the bug using a witness validator. This validation is essential according to SV-COMP rules, and we will include it as part of our evaluation (see Section 6.3.2).

4.3.2 CPU time allocation

Designing a cooperative framework with engines executing sequentially poses a challenge due to resource sharing between them. The time limit is necessary for *EBF* because BMC is required to execute before *OpenGBF* to generate the seeds needed to start the fuzzing process. Therefore, efficiently distributing computational resources between verification tools is important for improving *EBF*'s bug detection capabilities. This ensures that each engine has an appropriate amount of time to discover bugs, especially for time-limited and memory-limited programs. Choosing the right balance for these resources can significantly impact the success of cooperative verification, ensuring that each tool contributes optimally to the verification process. Achieving this trade-off requires careful consideration and fine-tuning to harness the strengths of both engines, ultimately leading to more robust and reliable verification outcomes for concurrent programs. We will demonstrate the effects of distributing the available CPU time between BMC and GBF tools to maximize the search space coverage for both tools and improve the overall performance of *EBF*.

4.4 Summary

This chapter explores the challenges of combining cooperative verification tools for concurrent programs. These challenges include effective input seed generation, result aggregation, and resource allocation trade-offs.

Firstly, we introduce the design of *EBF*, a cooperative black-box verification framework that combines BMC and fuzzing techniques. It consists of four main stages. Beginning with the safety proving stage, where we execute the BMC engine to prove program safety up to a context switch (since the fuzzer can not comprehensively explore all the execution of thread interleavings), our reliance is solely on BMC.

Secondly, we introduce the seed generation stage, which addresses the challenge of the fuzzer encountering difficulties when exploring complex path conditions. Therefore, providing good initial seeds that cover most of the program's search space is essential to enhance the fuzzer's bug-finding capabilities. We achieve this by executing the BMC engine with an error statement in a specific branch and extracting the values generated by BMC.

Thirdly, we introduce the falsification stage, where we employ our fuzzer *OpenGBF*. The outcomes from this stage will be saved for the aggregation stage, where we emphasize the importance of a well-structured decision matrix in cooperative verification frameworks, ensuring that conflicting outcomes between verification tools are appropriately handled. However, addressing disagreements between the engines is one of the challenges we tackled by setting a conflict when the two engines disagree.

After generating the final outcome, *EBF* generates the final witness file based on the engine from which we aggregate the result. If it is BMC, we convert their witness file; if it is *OpenGBF*, we use regular expressions to extract the required information, such as variable names, values, thread IDs, function names, and memory addresses.

The chapter concluded by introducing the resource allocation trade-off challenge in the cooperative verification framework, which affects the overall performance of such a system. It is crucial to make the design choice to allocate the appropriate CPU time to improve *EBF*'s bug detection capabilities. So, we split the available CPU time between the two tools, BMC and GBF, to maximize the search space coverage for both tools as much as possible.

4.5 Future work

While this thesis has made significant progress in addressing the challenges of cooperative verification for concurrent programs, there are several ideas for future research and development to enhance further the effectiveness and efficiency of the *EBF* framework.

Firstly, the seed generation stage of *EBF* heavily relies on selecting suitable labels to guide the BMC engine in efficiently finding error statements. Investigating and developing strategies for determining which labels to prioritize can have a deep impact on the quality and efficiency of the seed generation stage. Future research should explore advanced label selection algorithms that can dynamically adapt to the program's complexity and structure, ultimately enhancing the quality of initial seeds.

Secondly, allocating computational resources between verification tools, such as BMC and fuzzing, plays a vital role in maximizing bug detection capabilities while minimizing resource wastage. Developing heuristic methods for dynamically dividing the available CPU time between these tools can further improve the cooperative verification process. These heuristics should consider factors such as program complexity and verification tool performance. Implementing such heuristics can lead to more effective resource allocation and improved overall *EBF* performance.

Thirdly, while *EBF*'s decision matrix provides a foundation for aggregating results from different verification tools, future work can focus on refining and expanding this matrix. A more detailed decision matrix can distinguish between various types of vulnerabilities, such as reachability or data races, providing in-depth insights into the detected bugs. This improvement can facilitate a more fine-grained analysis of verification results, helping users better understand the nature of the identified bugs.

Another promising area for future work is developing an adaptive delay function within the *EBF* framework. Unlike the fixed delay approach used in this thesis, an adaptive delay function can dynamically adjust delay values based on the program's behaviour and execution patterns. This adaptive mechanism can lead to more efficient bug detection, especially for programs with varying execution speeds or delay sensitivity.

Furthermore, while *EBF* primarily targets concurrent programs, expanding its applicability to sequential programs is worth exploring. Disabling the delay function for sequential programs could be a practical direction, allowing the framework to accommodate a wider range of software testing scenarios and potentially improving its efficiency in detecting bugs in sequential programs.

Lastly, addressing challenges related to the sequencing of seeds is crucial, mainly when using BMC to seed the fuzzer. Mismatches between the order of input extracted from BMC and the input order used by the program can impact the overall effectiveness of fuzzing. Additionally, input values produced by BMC may become "Lost" in the mixture of the inputs and delay values generated using the nondeterministic functions during the fuzzing process. Investigating strategies to mitigate these challenges and enhance the synchronization of seed sequences can further improve *EBF*'s bug detection capabilities.

Chapter 5

Implementation of *EBF*

5.1 Chapter introduction

This chapter aims to extend Chapters 3 and 4 to describe implementation details. It provides a guide through the compilation and execution processes of *EBF* and *OpenGBF*, along with a running example that illustrates the output of each stage of *EBF* in practice.

5.2 *OpenGBF* design choices

In this section, we will elaborate on the design choices we made to align with the methodology outlined in Chapters 3 and 4. Specifically, we will discuss the following aspects:

5.2.1 The fuzzer choice

We have developed our *OpenGBF* by leveraging the capabilities of the state-of-the-art gray-box fuzzer called *AFL*++[115]. Our choice was motivated by the widespread usage, easy integration, and proven effectiveness of *AFL*++. Additionally, *AFL*++ strongly emphasizes requiring high-quality initial seeds to initiate the fuzzing process. This characteristic aligns well with our methodology and makes it suitable for our framework.

5.2.2 LLVM pass choice

Two key factors motivated the selection of the LLVM pass. Firstly, it was chosen because *AFL*++ utilizes LLVM pass for program instrumentation, ensuring compatibility with *OpenGBF*. Secondly, LLVM pass compatibility extends to any fuzzer that relies on the Clang compiler, providing flexibility and compatibility with various fuzzing tools.

5.2.3 The benchmarks choice

For our evaluation, we employed SV-COMP benchmarks (as described in Section 6.3) for several reasons. Firstly, this benchmark collection provides a wide range of concurrent programs, facilitating thorough testing and analysis of our framework. Secondly, including concurrent programs in the benchmark set allows us to evaluate *EBF*'s performance in effectively handling concurrent code. Lastly, the automation capabilities offered by *benchexec* [166] simplify the execution of all benchmarks, providing a convenient and efficient means of evaluation (see Section 6.3.1 for more details about *benchexec*).

5.3 *EBF* implementation details

We used three different programming languages for the development of *EBF*, each chosen for its suitability to specific components:

- The `C++` programming language was used for implementing the **LLVM Pass**, leveraging its robust capabilities and efficiency in handling low-level operations [167].
- The **run-time library**, a critical component of *EBF*, is written in `C` to align with *EBF*'s primary focus on verifying C programs, ensuring smooth integration with the language.
- The ***EBF* wrapper**, responsible for controlling all the components and processes, was developed using the `Python` programming language. This choice derived from Python's flexibility in supporting a wide range of libraries and its reputation for facilitating rapid development [168].

5.3.1 *EBF* usage

EBF versions 4.0 and 4.2 are released under the MIT License and can be accessed on GitHub [169], [170]. To install *EBF*, a shell script (i.e., *bootstrap.sh*) is provided to install all the required dependencies and compile the LLVM Pass and runtime library. Users are required to specify the Clang version during installation. Once the compilation is complete, users can run the tool. Users are expected to specify certain flags, as listed in Table 5.1, to customize their preferences and requirements when using *EBF*.

These flags include setting the paths for the benchmark (PUT) and the property file ($-p$). Additionally, optional flags can be adjusted, such as modifying the default time ($-t$) and memory ($-vm$) limits, changing the architecture ($-a$), or selecting the BMC engine ($-bm$). Moreover, an option enables parallel fuzzing ($-m$), activating the concurrency flag to detect concurrency bugs or including specific paths necessary for benchmark compilation using Clang ($-i$).

Flag	Description	Required	Default option
$-v$	check the version	No	No
benchmark	path to the benchmark	Yes	No
$-i$	To include the required paths for compiling the benchmark	No	No
$-p$	path to the property file	Yes	No
$-a$	set the system architecture	Yes	Yes (32 bit)
$-t$	set the time limit for BMC and *OpenGBF* respectively	No	Yes (6 and 5 min)
$-vm$	set Maximum memory for BMC and *OpenGBF* respectively	No	Yes (10 and 5 G)
$-c$	set concurrency flag (to set thread sanitizer)	No	No
$-m$	set *OpenGBF* to run parallel instances	No	No
$-bmc$	choose the BMC engine (*ESBMC*, *CBMC*, *Deagle*, *Cseq*)	Yes	Yes (*ESBMC*)

Table 5.1. The flag set supported in *EBF*.

It is worth mentioning that some of the required flag options are necessary because of the SV-COMP rules. For instance, the $-p$ flag specifies a specific file required to check for the property. Additionally, the $-a$ flag, specifying the architecture, is required by the SV-COMP rules. We will explain these benchmark requirements in more detail in Chapter 6. To run *EBF*, the user can execute the following command from the *EBF* root directory:

```
./scripts/RunEBF.py [-h] [-a {32,64}] [-p PROPERTY_FILE]
                                       [benchmark]
```

5.3.1.1 LLVM Pass and runtime library individual usage

To compile the LLVM pass and runtime library independently, without including *EBF*. First, the user executes the provided shell script (i.e., *bootstrap.sh*). After the shell script compiles the required files, it will generate a directory named *lib*. This directory contains the dynamically loadable plugin (*.so) and the compiled runtime libraries (*.a). To use the LLVM pass, users should load the pass while compiling the PUT using Clang (see Listing 5.6). This approach allows users to incorporate delay injection with their chosen fuzzer, offering flexibility in adjusting and configuring settings independently of the *EBF* script.

5.3.2 Running example

Let us examine how *EBF* verifies the example presented in Listing 5.1. In this example, two threads, *t1* and *t2*, access the same function, *foo()*. Within *foo()*, we have two variables, *foo_a* and *foo_b*, which receive non-deterministic integer values (see lines 6 and 7). A condition in line 8 checks whether the sum of these two variables equals 42. When running *EBF* with *ESBMC* as the default BMC engine and reachability as the property, it generates the output shown in Figure 5.1, which illustrates the *EBF* process.

Listing 5.1. C program with non-deterministic function.

```
#include <assert.h>
#include <pthread.h>
void reach_error() { assert(0); }
extern int __VERIFIER_nondet_int();
void foo(void * arg){
int foo_a = __VERIFIER_nondet_int();
int foo_b = __VERIFIER_nondet_int();
if((foo_a + foo_b) == 42) reach_error()↩
    ;}
int main() {
pthread_t t1, t2;
pthread_create(&t1, 0, foo, 0);
pthread_create(&t2, 0, foo, 0);
pthread_join(t1, 0);
pthread_join(t2, 0);
}
```

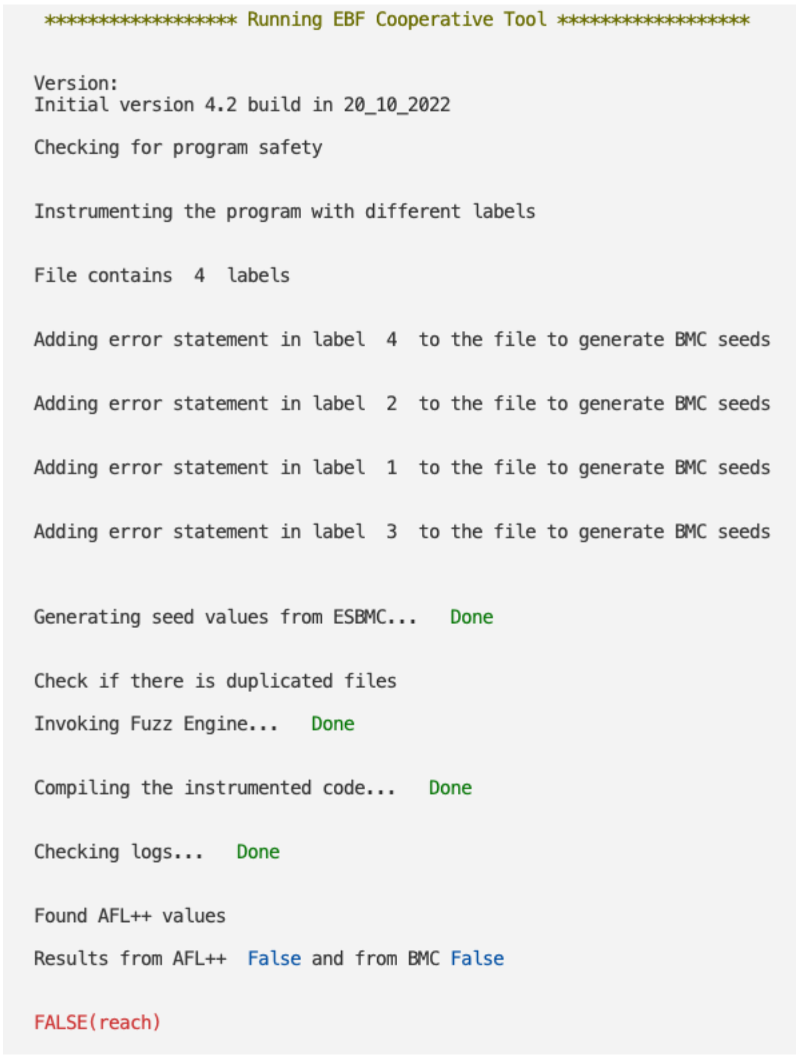
Listing 5.2. Correspond program with an injected labels.

```
#include <assert.h>
#include <pthread.h>
void reach_error() { assert(0);
LABEL_1:;}
extern int __VERIFIER_nondet_int();
void foo(void * arg){
int foo_a = __VERIFIER_nondet_int();
int foo_b = __VERIFIER_nondet_int();
if((foo_a + foo_b) == 42){
LABEL_3:;
 reach_error();}
LABEL_2:;}
int main() {
pthread_t t1, t2;
pthread_create(&t1, 0, foo, 0);
pthread_create(&t2, 0, foo, 0);
pthread_join(t1, 0);
pthread_join(t2, 0);
LABEL_4:;}
```

Safety proving stage. First, *EBF* checks program safety using the *ESBMC* engine. Therefore, we will run the PUT with the BMC engine. If the BMC engine produces a result (either *Verification successful* or *Verification failed*), we store the result in a *log* file. If BMC could not produce any result, we store *Unknown* in the *log* file. This *log* file will be used in the aggregation result stage to determine the final verdict.

Seed generation stage. Then, the seed generation stage starts, where we instrument the code to inject labels. In this example, *EBF* injects 4 labels, as shown in Listing 5.2. Each label is replaced with an error statement, resulting in one error statement per PUT file. So, in this example, we will store four files that contain an error statement, excluding the original program error statement. Then, *ESBMC* is executed on each file for 30 seconds per file, with a total time limit of 150 seconds for all files, or until it runs all the files. We analyze every witness file generated by *ESBMC* and extract seed values, ensuring no duplicate values are extracted. In this case, for instance, BMC generates identical assumption values for separate witness files with different error statement locations.

Listing 5.3 presents a snippet from one of the witness files generated by *ESBMC*. In this witness file, the assumption values for *foo_a* and *foo_b* that violate the assertion are 42 and 0, respectively. We extract these values in *EBF* by analyzing the *ESBMC* witness file. Specifically, we check the <edge> tag, check the "assumption" key, retrieve the assumption value for


```
****************** Running EBF Cooperative Tool ******************

Version:
Initial version 4.2 build in 20_10_2022

Checking for program safety


Instrumenting the program with different labels


File contains  4  labels


Adding error statement in label  4  to the file to generate BMC seeds


Adding error statement in label  2  to the file to generate BMC seeds


Adding error statement in label  1  to the file to generate BMC seeds


Adding error statement in label  3  to the file to generate BMC seeds



Generating seed values from ESBMC...   Done


Check if there is duplicated files

Invoking Fuzz Engine...   Done


Compiling the instrumented code...   Done


Checking logs...   Done


Found AFL++ values

Results from AFL++  False and from BMC False


FALSE(reach)
```

Figure 5.1. The *EBF* output for example 5.1 shows that both engines successfully reach the assert statement.

each key node, store these values in a file, and save the file in a directory named *CORPUS*. The content of this directory will operate as initial seed values for the fuzzer.

Listing 5.3. A snipt of witness file generated from *ESBMC* based on the results of Listing 5.1.

```
<node id="N3"/>
<edge id="E2" source="N2" target="N3">
  <data key="startline">6</data>
  <data key="assumption">foo_a = 42;</data>
  <data key="threadId">1</data>
</edge>
<node id="N4"/>
<edge id="E3" source="N3" target="N4">
  <data key="startline">7</data>
  <data key="assumption">foo_b = 0;</data>
  <data key="threadId">1</data>
</edge>
```

Falsification stage. Next, we invoke *OpenGBF* to start the fuzzing process. The process begins with the compilation of the original PUT, the LLVM Pass, and the runtime library using an *afl-clang* wrapper. After the LLVM Pass instrumentation, the PUT includes additional functions injected as described in Chapter 3, such as the delay function, information collection, and monitoring functions. Listing 5.4 shows a snippet of the instrumented IR of the PUT, which includes these functions.

Then, we initiate the fuzzing process on the generated binary. This process utilizes the *CORPUS* directory, which contains the seed values, and creates an output directory, *AFL_results*, where the fuzzer stores its results, including both the seeds that cause the PUT to crash and the mutated seeds. However, as briefly mentioned in Section 3.3.3.4, we can represent non-deterministic values either as bytes or in string format. After empirical evaluation, we found that the results are comparable, and neither method is superior as long as we ensure consistency in the initial seed format for the fuzzer. We used string representation to generate non-deterministic values in implementing *OpenGBF*. Consequently, in *ESBMC*, the counterexample (witness file) produced by BMC represents non-deterministic values as strings. Therefore, we do not need to convert these string values to byte representation format.

Listing 5.4. A snipt of LLVM-IR corresponds to function *foo()* in Listing 5.1.

```
define dso_local void @foo(i8* %0){
 call void @_delay_function()}
 %2 = alloca i8*, align 8
 call void @_delay_function()
 %3 = alloca i32, align 4
 call void @_delay_function()
 %4 = alloca i32, align 4
 call void @_delay_function()
 %typecast_store_double = bitcast i8* %0 to i1*
 call void @_delay_function()
 %bitcast_EBF_ptr = bitcast i8** %2 to i8*
 call void @_delay_function()
 call void @EBF_add_store_pointer(i8*
 call void @_delay_function()
 store i8* %0, i8** %2, align 8
 call void @_delay_function()
 %5 = call i32 (...) @__VERIFIER_nondet_int()
 call void @_delay_function()
```

Throughout the fuzzing process, we continuously record information about the variable name, value, thread ID, address, and line number in the file named $witnessInfoAFL\_pid$. This recording continues until the fuzzing process either identifies a crash or reaches the specified time limit. Listing 5.5 illustrates a counterexample generated by *OpenGBF* when the fuzzer finds the crash. Hence, we check the seed values that trigger the crash when the fuzzer detects a crash or reaches the time limit. Then, we store the result in *log* file.

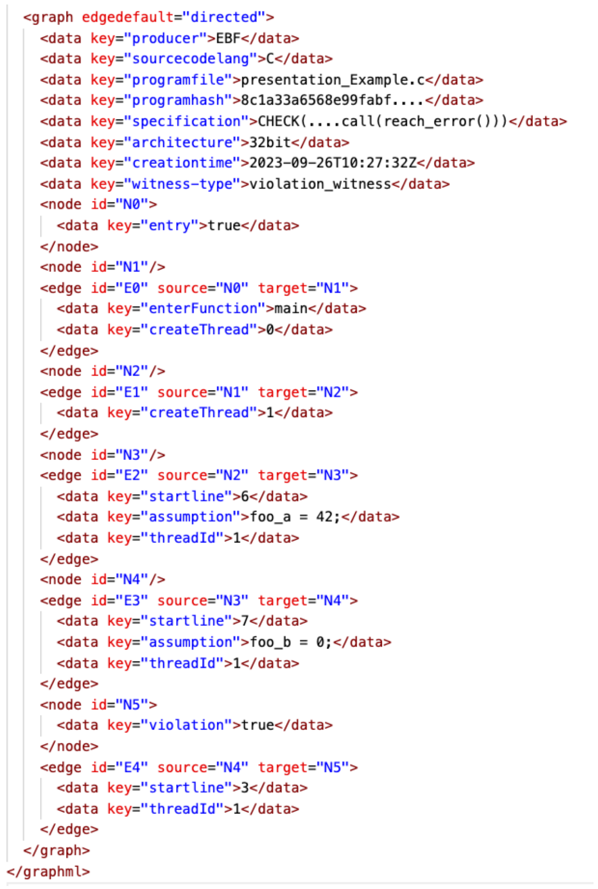
Listing 5.5. The content of $witnessInfoAFL_{pid}$ is generated by *OpenGBF* based on the code in Listing 5.1.

```
Setting variable: foo_a in Line number 6 with value: 42 running from thread: 0 in ↩
    function: foo with address: 0x7fc9c3a90e94
Setting variable: foo_a in Line number 6 with value: 0 running from thread: 1 in function↩
    : foo with address: 0x7fc9c328fe94
Setting variable: foo_b in Line number 7 with value: 0 running from thread: 0 in function↩
    : foo with address: 0x7fc9c3a90e90
Setting variable: foo_b in Line number 7 with value: 0 running from thread: 1 in function↩
    : foo with address: 0x7fc9c328fe90
REACH_ERROR END
```

Results aggregation stage. Next, *EBF* analyzes the *log* files to determine the final verdict and generate the final witness file, as shown in Listing 5.2a by applying the rules in the decision matrix outlined in Section 4.3.1.4 to determine the outcome. While 5.2b shows the graphical representation of the witness. In this example, both engines produced *Verification Failed* verdict, leading to the final result of *Verification Failed*.

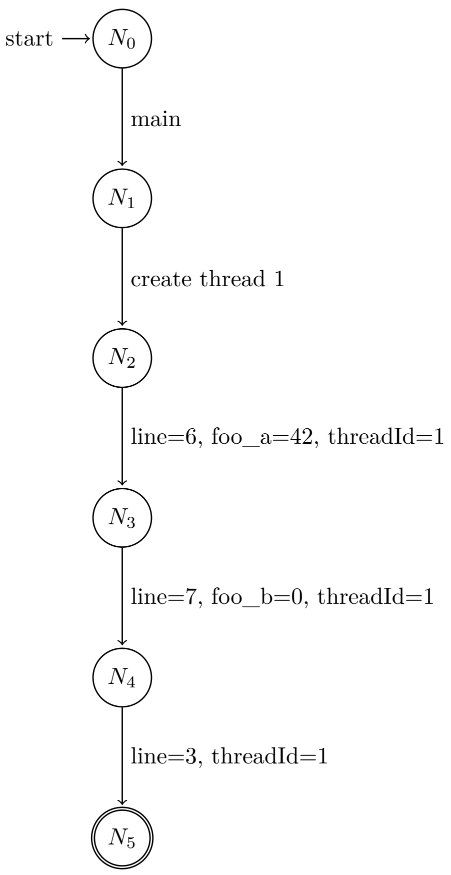
Lastly, during the *EBF* program verification process, it creates a directory called *EBF_results*. A subfolder is generated inside this directory with a unique ID appended to the benchmark name. This ID helps differentiate benchmarks with the same name but from different runs. For example, in the provided Listing 5.1, the directory name would be *EBF_results_thesis_example.c_216*, where 216 represents the unique ID. Within this folder, users will find five sub-directories:

- **AFL_results:** this directory contains the output from our *OpenGBF*. This output is the result of the *OpenGBF* and can be used for debugging and checking the queue seeds.
- **Executable-Dir:** this directory contains a copy of the original PUT and the instrumented versions, including all the additional error statements. It allows for easy examination of the number of goals and provides information about the locations where these goals are inserted within the instrumented program.
- **Log-Files:** in *EBF*, we store the output generated by BMC and *OpenGBF* tools, in *log* files. These *log* files serve as a source for reading the verdicts, enabling *EBF* to gather the necessary information for the decision matrix.
- **Witness-Files:** this directory contains the $witnessInfoAFL_{pid}$ file, which contains the witness information generated by *OpenGBF*. The directory includes witness files (*graphml* files) from BMC and witnesses files for all labels instrumented programs. However, the final witness file produced by *EBF* will be saved in the root directory of *EBF* itself, making it available for validation using the witness validator.



```
  <graph edgedefault="directed">
    <data key="producer">EBF</data>
    <data key="sourcecodelang">C</data>
    <data key="programfile">presentation_Example.c</data>
    <data key="programhash">8c1a33a6568e99fabf....</data>
    <data key="specification">CHECK(....call(reach_error()))</data>
    <data key="architecture">32bit</data>
    <data key="creationtime">2023-09-26T10:27:32Z</data>
    <data key="witness-type">violation_witness</data>
    <node id="N0">
      <data key="entry">true</data>
    </node>
    <node id="N1"/>
    <edge id="E0" source="N0" target="N1">
      <data key="enterFunction">main</data>
      <data key="createThread">0</data>
    </edge>
    <node id="N2"/>
    <edge id="E1" source="N1" target="N2">
      <data key="createThread">1</data>
    </edge>
    <node id="N3"/>
    <edge id="E2" source="N2" target="N3">
      <data key="startline">6</data>
      <data key="assumption">foo_a = 42;</data>
      <data key="threadId">1</data>
    </edge>
    <node id="N4"/>
    <edge id="E3" source="N3" target="N4">
      <data key="startline">7</data>
      <data key="assumption">foo_b = 0;</data>
      <data key="threadId">1</data>
    </edge>
    <node id="N5">
      <data key="violation">true</data>
    </node>
    <edge id="E4" source="N4" target="N5">
      <data key="startline">3</data>
      <data key="threadId">1</data>
    </edge>
  </graph>
</graphml>
```

(a) Witness file in GraphML format.



(b) Graphical representation.

Figure 5.2. The witness file in GraphML format Vs. graphical representation of the witness for Listing 5.1.

- **CORPUS:** this directory contains all the seeds generated by *EBF* in Section 4.3.1.2, which will subsequently be used by the *OpenGBF*.

Listing 5.6. Compiling AFL++ clang wrapper with the Sanitizer, our LLVM Pass instrumentation and Runtime library.

```
./afl-clang-fast -fsanitize=the required sanitizer -Xclang -load -Xclang LLVMPASS↩
    .so test.c -L lmyRunTimeLibrarys -o BinaryFile
```

5.4 Summary

This chapter is a comprehensive guide to implementing *EBF*. It delves into the rationale behind selecting key components, explaining the reasoning behind the choice of the fuzzer, the LLVM Pass, and the specific benchmark suite employed for the evaluation process. These insights provide a deeper understanding of the decision-making process behind the tool's design and usage.

Furthermore, the chapter provides information about the programming languages used for developing both *EBF* and *OpenGBF*, which shows the diversity in the languages used for the implementation. Additionally, it presents a detailed explanation regarding how to compile and use the tool.

Lastly, a practical running example has been included in this chapter. This example demonstrates the output of each stage of *EBF*, allowing us to gain a clear and practical understanding of its functionality.

Chapter 6

EBF evaluation

6.1 Chapter introduction

In this chapter, we aim to demonstrate the effectiveness of *EBF* across a diverse set of scenarios. We will present our experimental goals before detailing the deployed benchmarks and our findings (see Section 6.2).

Our evaluation of *EBF* spans a significant time frame during which we continuously refined and enhanced its design. We present our results separately for three versions of *EBF*. Specifically, we discuss the participation of *EBF* 2.3 in the *Concurrency Safety* category of SV-COMP 2022 (see Section 6.3.2). This version used *CBMC* v5.43 as the BMC engine and represented the initial implementation of our concurrency-aware fuzzer. Furthermore, we evaluate *EBF* versions 4.0 (see Section 6.3.3). *EBF* 4.0 includes the full implementation of *OpenGBF*, as explained in Section 3.3, and incorporates a variety of BMC engines (i.e., *ESBMC*, *CBMC*, *Cseq* and *Deagle*). Also, we discuss the participation of *EBF* 4.2 in the *Concurrency Safety-main* category of SV-COMP 2023. *EBF* 4.2 integrates all the features of *OpenGBF* along with the seed generation method detailed in Section 4.3.1.2. Additionally, we demonstrate our fuzzer's capability to detect data races in the open-source *wolffMQTT* cryptographic library (see Section 6.4). We initially discovered this bug using an earlier version of our fuzzer and replicated the experiment using the version of *OpenGBF* included in *EBF* 4.0. Furthermore, we evaluate *EBF* version 4.0 on several real-world concurrent programs in Section 6.4.2 to highlight the capabilities of our *OpenGBF*. Lastly, we conduct a comprehensive performance comparison of *EBF* 4.0 under various parameter configurations (see Section 6.5).

6.2 Evaluation goals

Our experimental evaluation has been designed with the following goals:

EG1 - Detection of violations in concurrent programs

Illustrate the capabilities of *EBF* in finding more violations in concurrent programs than state-of-the-art BMC tools alone.

EG2 - Real-world performance of *OpenGBF*
Illustrate the effectiveness of the *OpenGBF* we implement in *EBF* in finding violations in real-world concurrent programs.

EG3 - Scalability and robustness of *OpenGBF*
Illustrates the scalability and robustness of *EBF* performance in finding similar or different bugs in real-world programs using both verification engines.

EG4 - The effectiveness of good initial seed values
Illustrate how the seed can impact the outcome of the GBF.

EG5 - Parameter trade-offs in our concurrency-aware fuzzer
Illustrate the consistent performance of *EBF* across a wide range of parameter settings.

Note that the two objectives, **EG2** and **EG5**, aim to demonstrate that *OpenGBF* represents state-of-the-art gray-box fuzzing techniques.

6.3 Evaluating *EBF* on SV-COMP benchmarks

SV-COMP stands for Software Verification Competition, an annual competition that evaluates the effectiveness and performance of software verification tools in detecting software vulnerabilities. SV-COMP provides an extensive collection of benchmark suites covering various software verification aspects, including concurrency, memory safety, and reachability categories. These benchmarks consist of software programs designed to evaluate the performance and efficiency of software verification tools selected to represent real-world scenarios and challenges faced in the field. Participants in SV-COMP submit their verification tools, which are then evaluated against the provided benchmarks based on criteria such as correctness, efficiency, and scalability. The primary goal of SV-COMP is to facilitate the advancement of software verification techniques and encourage the development of more robust and reliable verification tools [77].

The competition aims to evaluate the participating tools based on the following outcomes:

- **Correct True.** The tool accurately verifies the program as safe and correctly confirms the generated witness file using the competition's *witness validator* tools.
- **Correct False.** The tool correctly identifies the presence of a bug and correctly confirms the generated witness file using the competition's *witness validator* tools.

Verification outcome	**Score per benchmark**
Correct True	2
Correct False	1
Correct False Unconfirmed	0
Incorrect True	-32
Incorrect False	-16
Unknown	0

Table 6.1. SV-COMP scoring system.

- **Correct False Unconfirmed.** The tool correctly identifies the presence of a bug, but the associated witness file cannot be confirmed using the competition's *witness validator* tools.
- **Incorrect True.** The tool incorrectly confirmed the program was safe despite the presence of a bug.
- **Incorrect False.** The tool incorrectly confirms the program contains a bug when it is actually safe.
- **Unknown.** The tool cannot determine the result within CPU time and memory constraints.

Table 6.1 presents the scores assigned to each verification outcome. A significant penalty is imposed on incorrect results, and the overall score for each tool comprises the sum of scores obtained across all benchmarks.

We evaluated *EBF* 2.3 and 4.0 in the *Concurrency Safety* category for SV-COMP 2022 [171], and *EBF* 4.2 in the *Concurrency Safety* category for SV-COMP 2023 (based on the year of participation). However, the *Concurrency Safety* category for SV-COMP 2022 comprises a collection of 763 concurrent C programs, with 398 of them considered Safe. The remaining 365 programs are designed with bugs defined in terms of reachability conditions. If a predetermined error function is reachable within the given program, it is labeled as Unsafe; otherwise, it is labeled as Safe [164].

It is important to note that the benchmarks used in SV-COMP can contain *Undefined behaviour*, where reachability errors may not pose a problem for static analysis tools like BMCs. However, since the fuzzer relies on compilation (dynamic analysis), it may find other bugs with respect to the given property; this scenario makes it difficult to control the fuzzer. Therefore, according to the competition rules, we would miss the point if we produce *Unknown*. Producing incorrect results regarding the specified property would result in a penalty score. However, SV-COMP organizers continuously address and update these benchmarks to ensure accuracy. Therefore, in the *Concurrency Safety-main* category for SV-COMP, 2023, the number of concurrent C programs with reachability conditions decreased to 665 benchmarks.

6.3.1 Running SV-COMP benchmark using *BenchExec*

Before presenting the evaluation of *EBF* on SV-COMP 2022 benchmarks, we will introduce the tool used to run around 700 benchmarks automatically. *BenchExec* is an automated open-source tool designed for reliable benchmarking and resource measurement [166]. It provides an automated environment for executing and evaluating software verification tools, simplifying the benchmarking process by providing features such as result comparison, resource measurement, and statistical analysis. In order to run *BenchExec*, two main files are required: `ebf.py` and `ebf.xml`. The `ebf.py` file is a program used to configure the main file *BenchExec* needs to execute, along with the necessary flags. The `ebf.xml` file is used to specify all the categories that need to be executed, along with the required CPU time and memory.

6.3.2 *EBF* 2.3 participation in SV-COMP 2022

In SV-COMP 2022 [77], we participated with *EBF* version 2.3. This version was built on top of *CBMC* v5.43 as the BMC engine and included the initial implementation of *OpenGBF*. We chose *CBMC* because it is one of the state-of-the-art BMC tools that consistently achieved high rankings in the concurrency category of SV-COMP over the past decade. Additionally, in *EBF* 2.3, the implementation of *OpenGBF* was the initial version, featuring no limitations on the number of threads, no early termination probability, and no mechanism to avoid injecting delays inside atomic blocks.

During the competition, the SV-COMP servers were equipped with 8 CPUs (Intel Xeon E3-1230 v5 @ 3.40 GHz) and a total of 33 GB of RAM. Each benchmark had a maximum CPU usage limit of 15 minutes and a maximum RAM usage limit of 15 GB.

Figure 6.1 illustrates that *EBF* achieved the 7th place out of 20 participants in SV-COMP 2022, scoring a total of 496 points. Notably, *EBF* 2.3 outperformed *CBMC* 5.43, which achieved the 10th place with a score of 460 points. Table 6.2 reports these two tools' official SV-COMP 2022. It is worth noting that *CBMC* obtained a higher score than *EBF* in predicting program safety, with scores of 148 and 139, respectively. This outcome was expected since *EBF* allocated only 6 out of 15 minutes to BMC, with the remaining time dedicated to *OpenGBF*, which cannot prove program safety. However, *EBF* outperformed *CBMC* in detecting bugs that could be confirmed by the *witness validator*, scoring 234 points compared to *CBMC's* 212 points, thus scoring additional points.

Furthermore, *EBF* reported only one verdict, which was *Incorrect False*, while *CBMC* reported three incorrect verdicts, resulting in 48 penalty points. Interestingly, *EBF* managed to avoid reproducing the latter three incorrect verdicts by returning *Unknown* instead. This

Verification outcome	**Tool**		**Score per benchmark**
	EBF 2.3	*CBMC*	
Correct True	139	**148**	× 2
Correct False	**234**	212	× 1
Correct False Unconfirmed	55	**90**	× 0
Incorrect True	0	0	× -32
Incorrect False	**1**	3	× -16
Unknown	334	310	× 0
Overall SV-COMP 2022 score	***496***	*460*	

Table 6.2. The results presented by *EBF* 2.3 and *CBMC* 5.43 in the *Concurrency Safety* category of SV-COMP 2022.

happened because *CBMC* did not have enough time to wrongly detect these bugs while running as part of the cooperative framework, hence reporting *Unknown* as a result. Similarly, *OpenGBF* also could not find any bugs in these benchmarks within the remaining time, resulting in another set of *Unknown* verdicts. In contrast, the only incorrect verdict obtained by *EBF*, which differed from the three false positives produced by *CBMC*, was caused by a bug within *OpenGBF*. This bug led to the generation of a spurious counterexample. It is worth noting that this issue has been resolved in the latest version of *EBF* by introducing the probability p for exiting the current run at each instruction (see Section 3.3.3.1).

Overall, *EBF* 2.3 improved results by approximately $\sim 7.8\%$ compared to *CBMC* 5.43. Additionally, *EBF* successfully fined and confirmed a property violation in a specific benchmark that no other dynamic tool in the competition detected. These outcomes fulfill the first experimental goal **EG1**, and we will present further supporting experimental evidence in Section 6.3.3.

6.3.3 *EBF* 4.0 with different state-of-the-art BMC tools

Following *EBF* participation in SV-COMP 2022, we improved *OpenGBF* using the algorithmic concepts discussed in Section 3.3. In this context, we present the outcomes of additional experiments aimed at evaluating the improvement of any BMC tool through the integration of our latest version of *OpenGBF* (evaluation goal **EG1**). For more clarification, we refer to this version of our cooperative framework as *EBF* 4.0, presented in Chapters 3 and Chapter 4. Except for Section 4.3.1.2, we refer to *EBF* with this optimization feature as *EBF* 4.2.

As mentioned in Section 6.3, we evaluated *EBF* 4.0 using the same benchmarks as those from the SV-COMP 2022 *Concurrency Safety* category. However, we omit the SV-COMP aggregate scoring system (refer to Table 6.1), as its different weights could obscure the advantages of each verification technique. Instead, our focus is on analyzing the trade-off between the capability to prove safety (solely relying on BMC, which involves BMC's capability to verify all reachable states but cannot detect an execution path that violates the safety property.) and bug-finding abilities (using both BMC and GBF) from the raw results.

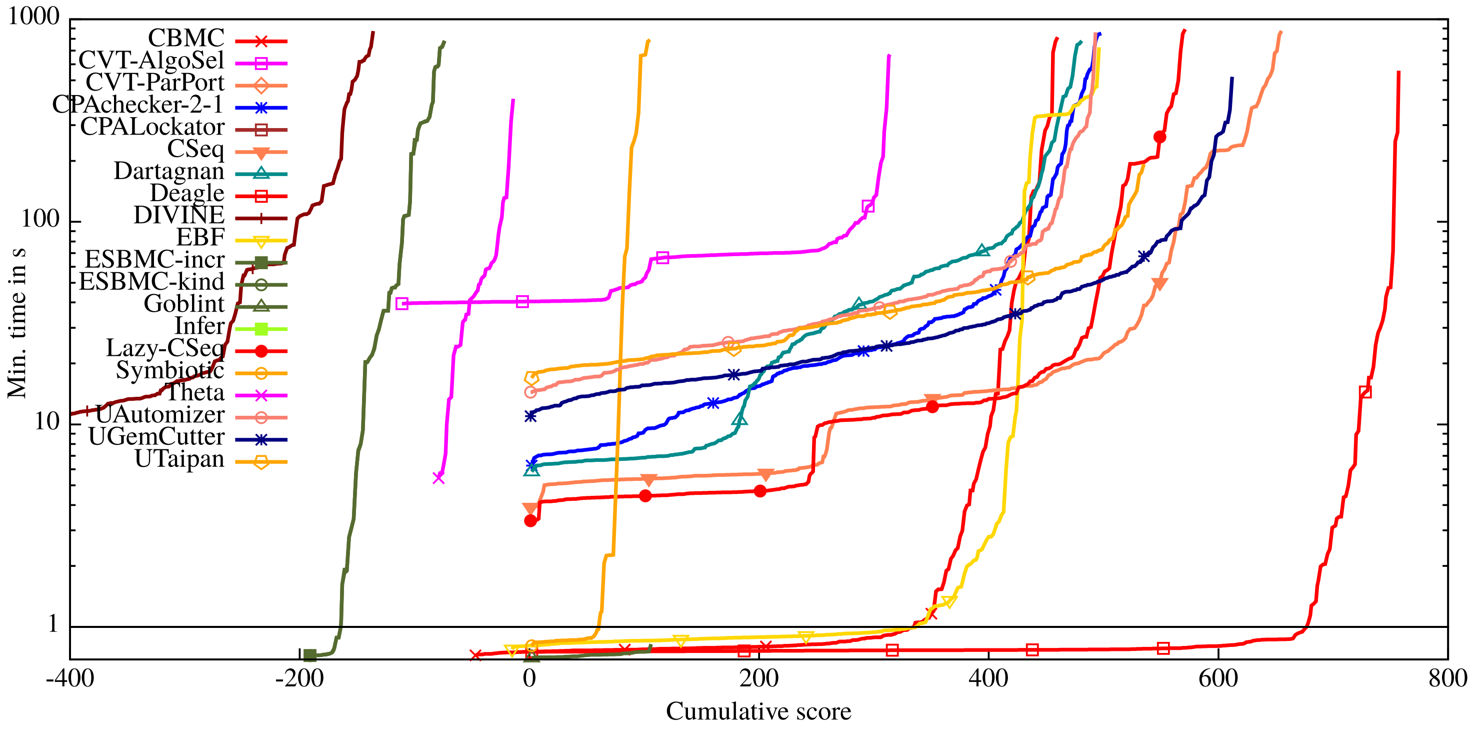


Figure 6.1. Quantile plot for *ConcurrencySafety* category in SV-COMP 2022.

Additionally, we consider three more BMC tools in our experiments, as listed in Table 6.3, instead of only *CBMC* [65]. These tools comprise *ESBMC* [64], a robust BMC tool with a consistent track record of high performance in SV-COMP over the past decade, along with *Deagle* [172] and *Cseq* [66]. *Deagle* achieved the 1st place, and *Cseq* the 2nd place in the *Concurrency Safety* category at SV-COMP 2022.

Our experiments were conducted on a virtual machine running Ubuntu 20.04 LTS, equipped with an Intel Core Processor (Broadwell, IBRS) operating at a frequency of 2.1 GHz. The virtual machine has 160 GB of RAM and 25 CPU cores. For *EBF* 4.0, we employed the following parameters: a maximum thread threshold of 5 and a delay range from $0\,[\mu s]$ to $10^5\,[\mu s]$. In terms of runtime allocation, we allocated the available time as follows: 6 minutes for the BMC engine, 5 minutes for *OpenGBF*, and 4 minutes for the remaining stages, including extracting initial seeds from BMC (or seeding the fuzzer with random values if no seeds are extracted from BMC), aggregating results, and generating the final witness file. These parameter settings are optimal for the SV-COMP 2022 benchmarks used in our evaluation (for a more detailed explanation, refer to Section 6.5). Furthermore, users have the flexibility to specify the time distribution among the tools in *EBF* using command-line arguments, as shown in Table 5.1.

Table 6.3 presents a comparative analysis between four individual BMC tools and the same BMC tools as part of the *EBF* 4.0 BMC engine. The results highlight that *EBF* outperforms all four BMC engines in terms of bug detection while reducing the number of instances cate-

Verification outcome	**Tool**							
	EBF	*Deagle*	*EBF*	*Cseq*	*EBF*	*ESBMC*	*EBF*	*CBMC*
Correct True	240	240	172	**177**	65	**70**	139	**146**
Correct False	**336**	319	**333**	313	**308**	268	**320**	303
Incorrect True	0	0	0	0	0	0	0	0
Incorrect False	0	0	0	0	0	1	0	3
Unknown	**187**	204	**258**	273	**390**	424	**304**	311

Table 6.3. Comparative analysis of the verification outcomes for *EBF* 4.0 with different BMC tools "plugged in" against their individual performance on the benchmarks from the *Concurrency Safety* category of SV-COMP 2022.

gorized as *Unknown*. In more detail, *EBF* achieves the best improvement when compared to *ESBMC*, detecting $\sim 14.9\%$ more bugs and correcting one incorrect outcome while only decreasing $\sim 7.6\%$ in the number of safety proofs. Similarly, the ability to double-check BMC counterexamples enables *EBF* to correct all three incorrect outcomes from *CBMC*, with a marginal difference between the improvement in bug detection ($\sim 5.6\%$) and the decrease in safety proofs ($\sim 5\%$). In contrast, when compared to *Deagle*, *EBF* shows the same number of safety proofs, indicating that *Deagle* is quick in proving safety (within 6 minutes). However, *EBF* detects more bugs than *Deagle* by $\sim 5.3\%$. Regarding *Cseq*, the number of safety proofs produced by *EBF* decreased by only $\sim 2.9\%$, while the number of *Correct False* outcomes increased by $\sim 6.3\%$.

In general, *EBF* provides a better trade-off between bug detection and safety proving compared to individual BMC engines. On average, *EBF* detects more than 8% concurrency bugs while only reducing the number of programs declared safe by 3.8%.

Therefore, this evaluation successfully achieves our first evaluation goal (**EG1**).

6.3.4 *EBF* 4.2 participation in SV-COMP 2023

In SV-COMP 2023 [78], we participated in the *Concurrency Safety-main* category with *EBF* version 4.2, built on top of *ESBMC* $v6.8$. This version introduces an additional feature not found in *EBF* 4.0, as explained in Section 4.3.1.2. This feature generates initial seeds to enhance the fuzzer's ability to explore deep paths, thereby improving its effectiveness. To achieve this, we instrument the PUT, inject an error statement, and execute *ESBMC* to generate witness files, each containing an error statement (counterexample). These seeds are then extracted and used to initiate the fuzzing process in *OpenGBF* (referred to as *OpenGBF* 4.2 for future reference).

We evaluated *EBF* 4.2 on the competition servers, using the same machines discussed in Section 6.3.2 for SV-COMP 2022. However, this year, the total number of verification tasks in the *Concurrency Safety* category was reduced to 665 due to continuous benchmark updates by the organizers, as mentioned in Section 6.3. In this context, *EBF* 4.2 achieved a total score of 369, while *ESBMC* scored 346, representing an increase of $\sim 6.65\%$. It is worth noting that this version of the evaluation primarily aims to evaluate the impact of generated good seed values on bug detection within *OpenGBF*. We will further evaluate *OpenGBF* 4.2 against the previous version in Section 6.5.4.

6.4 Evaluating *EBF* on real-world concurrent programs

Evaluating *EBF* on real-world concurrent programs provides valuable insights into its practical applicability, scalability, and effectiveness. It allows us to measure its ability to handle the complexities of concurrent programs, detect concurrency bugs, ensure safety, and provide reliable results in various contexts.

6.4.1 Detecting a data race in *wolfMQTT*

The *wolfMQTT* library is an open-source implementation of the *MQTT* (Message Queuing Telemetry Transport) protocol. It is a lightweight messaging protocol designed for resource-constrained environments like the Internet of Things (IoT). It operates on a *publish-subscribe* messaging pattern, where clients publish messages to particular topics (e.g., temperature), and other clients subscribe to those topics in order to receive the messages. The *wolfMQTT* library provides a client implementation of the *MQTT* protocol written in C programming language for the constraint devices [173]. We used the library's API to verify the concurrent aspect of the protocol's implementation.

In standard network communication, the client directly communicates with the server. The clients initiate a request to the server to use resources or data, then the server handles the request and sends a response back to the clients. However, *MQTT* uses a publish/subscribe pattern to separate the message sender (publisher) from the message receiver (subscriber) by an intermediary component known as a message broker. The broker is responsible for handling the communication between publishers and subscribers. Its primary role is filtering incoming messages from publishers and ensuring their proper distribution to subscribers [174].

An overview of how *MQTT* operates is as follows: First, clients establish a connection with the broker. Then, clients have the option to publish messages, subscribe to specific messages, or do both. Lastly, when the broker receives a message, it transmits it to the intended subscriber. The packet types in *wolfMQTT* include *Connect*, *Publish*, *Subscribe*, and *Unsubscribe*. The *Connect* packet is used when a client requests to establish a connection with

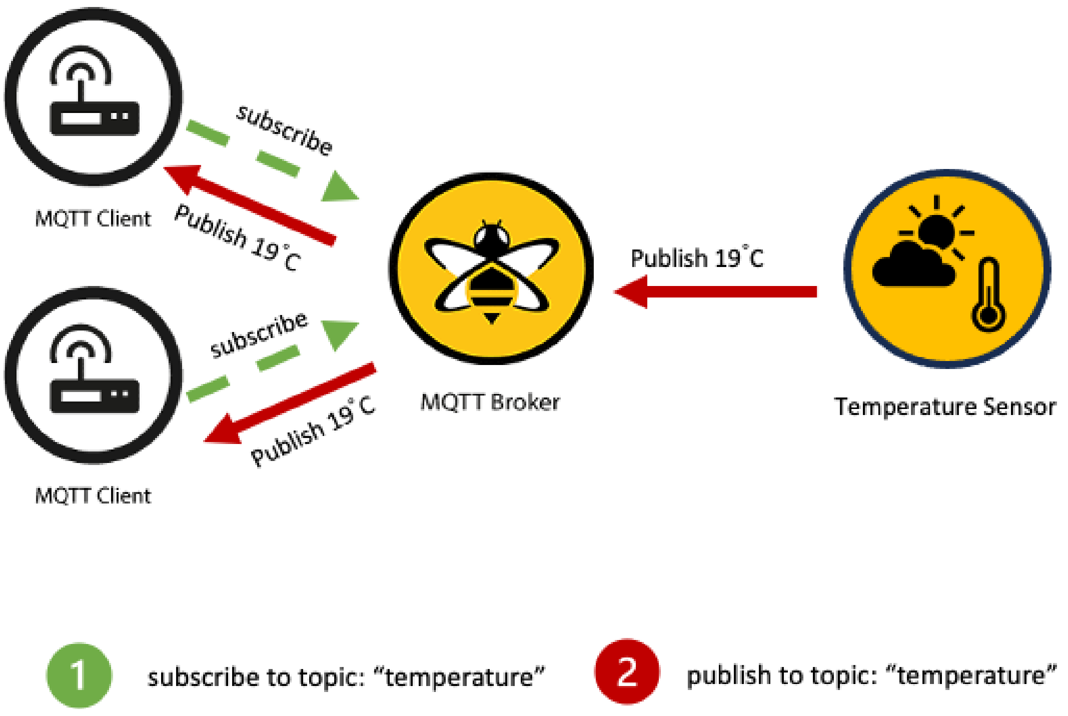


Figure 6.2. Client/broker message passing, clients subscribe to topics, and brokers publish messages to those topics (derived from [175]).

the broker. The *Publish* packet allows clients to send messages to the broker. The *Subscribe* packet enables clients to subscribe to particular topics, and the *Unsubscribe* packet allows clients to unsubscribe from specific topics. Figure 6.2 provides an overview of the publish and subscribe structure, illustrating two clients subscribing to a topic (e.g., temperature) while the broker publishes the results (19°C).

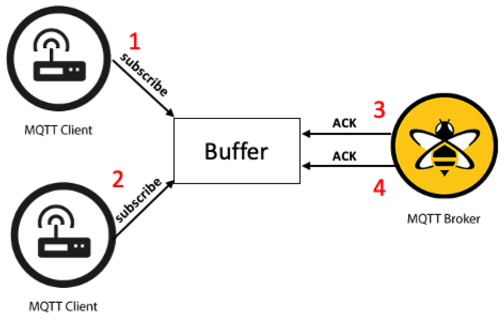
We evaluated our *EBF* 4.0 on the *wolfMQTT* client library [174]. Specifically, our *OpenGBF* detected a data race [1] in *wolfMQTT* after running for 15 minutes and using 24 MB of RAM. Figure 6.3a illustrates the issue: when more than two clients send a subscribe packet to the broker, the broker sends an Acknowledgment (ACK) to the *Subscribe* function. This ACK was received by an unprotected pointer for the status code (the return code associated with the ACK message), resulting in the data race, as they share the same buffer between clients. The data race was found in the function *MqttClient_WaitType*, which could potentially lead to information leakage or data corruption. We reported the issue to the *wolfMQTT* developers, and it was successfully replicated and subsequently fixed[2]. Figure 6.3b presents the fixed version of the bug, where they copied the status code into different buffers. In Appendix A, Figure A.2 shows the release note expressing gratitude for the bug discovery, and Figure A.1 shows the complimentary gift sent to my address as a token of appreciation for finding the bug.

[1]`https://github.com/wolfSSL/wolfMQTT/issues/198`
[2]`https://github.com/wolfSSL/wolfMQTT/pull/209`

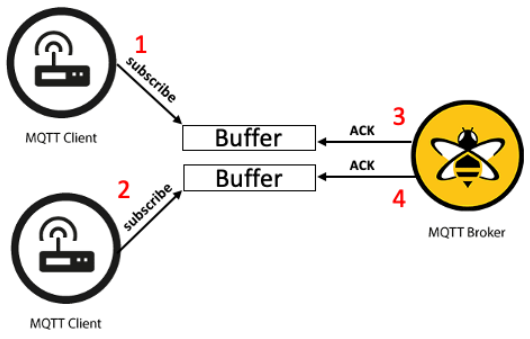
Our experimental setup is as follows: we executed *EBF* 4.0 on a machine with an Intel Core i7 2.7 GHz processor and 8 GB of RAM, running Ubuntu 18.04.5 LTS as the operating system. We used a Mosquitto server for communicating with the *wolfMQTT* client [176]. To identify concurrency bugs that are not explicitly defined in terms of reaching a predetermined error function (such as in the SV-COMP 2022 concurrency benchmarks or violating a safety assertion), we enabled *ThreadSanitizer* in *OpenGBF*. Finally, we configured *OpenGBF* with a thread threshold of $Th = 5$, a delay range from $0\,[\mu s]$ to $10^5\,[\mu s]$ and a probability of $p = 0.01\%$.

We evaluated all the tools mentioned in Section 6.3.3 on the *wolfMQTT* source code to evaluate their effectiveness in analyzing and detecting potential vulnerabilities. However, none of the tools successfully detected the same vulnerability. Specifically, neither the bounded model checker *CBMC* $v5.43$ nor *ESBMC* $v6.8$ could identify the data race within the given time limit of 15 minutes. Additionally, due to its use of an outdated version of the C parser, the BMC tool *Deagle* v1.3 encountered difficulties parsing the program correctly. Similarly, *Cseq* $v3.0$ does not support programs consisting of multiple source files. Finally, both the *AFL* fuzzer and *AFL++* failed to identify this bug in the *wolfMQTT* source code.

As a result of this experiment, we can conclude that our second evaluation goal (**EG2**) has been achieved



(a) Example of the bug found by *OpenGBF*



(b) The fixed version of the bug

Figure 6.3. Overview on the bug found by *OpenGBF* and how *wolfMQTT* developer fixed it.

6.4.2 Detecting memory-related vulnerabilities in real-world concurrent programs

Detecting memory violations in real-world concurrent programs is important to ensure software reliability and security. Memory violations, such as memory leaks or buffer overflows, can lead to system crashes and expose security vulnerabilities. However, detecting such violations in concurrent programs is challenging because of the complexities of concurrent execution. Therefore, to show the scalability and robustness of *EBF*, we evaluated the per-

Real-world programs	LOC	N_N	N_T	Median Time	ESBMC	OpenGBF
wolfMQTT	9.3k	1	Data Race	$361.7s$		✓
pfscan	1.1k	1	Invalid pointer dereference	$3.98s$	✓	✓
bzip2smp	5.3k	2	Invalid pointer dereference	$10.6s$	✓	
			Memory leak			✓
swarm 1.1	2.8k	1	Invalid pointer dereference	$339.6s$		✓

Table 6.4. Evaluation of *EBF* on real-world concurrent programs: For each program, we provide information about its size in terms of the number of lines of code (LOC), the number of vulnerabilities detected by *EBF* (N_N), the types of corresponding vulnerabilities (N_T), the median time (in seconds) from 20 *EBF* re-runs, and the *EBF* engine (i.e., ESBMC or *OpenGBF*) that detected the corresponding vulnerability.

formance of *EBF* 4.0 on three multi-threaded real-world programs using the same machine described in Section 6.3.3 (a virtual machine running Ubuntu 20.04 LTS with 160 GB of RAM and 25 CPU cores). In this evaluation, we used *ESBMC* as the BMC engine due to our close collaboration with its developers, which facilitated addressing any issues encountered during the evaluation of real-world concurrent programs.

Table 6.4 presents the number of lines of code (LOC) for each real-world program, along with the number of bugs detected (N_N) and types of vulnerabilities detected (N_T) by *EBF*, as well as the median time it takes *EBF* to find these bugs. Further, more detailed information can be found in Sections 6.4.2.1, 6.4.2.2, and 6.4.2.3.

6.4.2.1 *pfscan*

pfscan [51] is a multi-threaded file scanner designed for protein or DNA sequences written in the C programming language. In our evaluation of *EBF* 4.0, we used optimal settings, including a thread threshold of $Th = 5$, a delay range from 0 $[\mu s]$ to 10^5 $[\mu s]$, and a probability of $p = 0.01\%$ (for more details, see Section 6.5). In this experiment, we added extra initial seed values in the corpus directory with random DNA formats (e.g., *ATGCGTACAGTCGA*). This was accomplished to help the fuzzer in finding the required input format, as the program expects input in this specific format.

Both engines within *EBF* successfully identify a NULL pointer dereference in *pfscan* (see Table 6.4). This bug is caused by initializing a pointer by allocating a dynamic memory address for that pointer using the `malloc` instruction. The `malloc` allocation needs to be checked for success (i.e., the return address is not null). The program did not check for this pointer, resulting in a crash due to writing to a NULL pointer.

Evaluating the three tools *CBMC* $v5.43$, *Deagle* $v1.3$ and *Cseq* $v3.0$ on *pfscan* failed to identify any bug within the program. More specifically, *CBMC* reported *Verification Successful*, *Deagle* reported unsupported library function, while *Cseq* reported *Unknown*.

6.4.2.2 *bzip2smp*

bzip2smp [52] is a parallel implementation of the *bzip2* compressor, written in C programming language and using the Pthread library, and it accepts files as input. In our evaluation of *EBF*, we employed optimal settings for our fuzzer, which were similar to the settings used in Section 6.4.2.1. For this experiment, we used seed values generated from *ESBMC* counterexamples as initial seeds, which were saved in files and provided to the fuzzer.

As presented in Table 6.4, *EBF* detected two bugs, one from the *ESBMC* engine and the other from *OpenGBF*. *ESBMC* detects a vulnerability in the `BZ2_bzclose()` function because they dereference a pointer that might be NULL. Meanwhile, *OpenGBF* detected a memory leak in the `writerThread()` function where they allocated memory for the `buf` and never freed this allocated memory.

Unfortunately, we were unable to evaluate the other tools mentioned in Section 6.4.2.1 because *CBMC* $v5.43$ and *Deagle* $v1.3$ has a parsing error and *Cseq* $v3.0$ crashes with a python error.

6.4.2.3 *swarm1.1*

swarm1.1 [53], a library that provides a framework for concurrent programming on multi-core systems. We evaluated *EBF* using the optimal settings for our fuzzer (same settings used in Section 6.4.2.1 and 6.4.2.2).

Regarding *swarm* 1.1, *EBF* 4.0 detected an invalid pointer dereference resulting from incorrect thread initialization (i.e., calling the `pthread_create` function with a NULL pointer as an argument). Specifically, this bug was detected by *OpenGBF* as stated in Table 6.4.

We also evaluated *swarm* 1.1 with *CBMC* v5.43, and it could not detect any bugs; it returned *Unknown* within the specified time limit of 15 minutes. Similarly, *Cseq* $v3.0$ faced the same issue of verifying multiple files. A parsing error was encountered in the case of *Deagle* $v1.3$.

Based on the results of this experiment, we have achieved our third evaluation goal (**EG3**).

6.5 Optimizing *EBF*'s settings

In the forthcoming experiments, we aim to explore and analyze the effects of varying settings on the performance and overall results of *EBF* 4.0 and *OpenGBF* on *Concurrency Safety* category of SV-COMP 2022. This exploration of settings will provide valuable information for optimizing the usage and effectiveness of *EBF*. For the first four experiments in Sections 6.5.1,6.5.2, 6.5.3, and 6.5.4, we executed *EBF* with the BMC engine switched off, allowing the fuzzer to run for 11 minutes. For the fifth experiment presented in Section 6.5.5, we executed *EBF* with both engines enabled but with a different amount of time allocated (out of a total of 11 minutes) to each of them. Finally, we evaluate the noise effects on *OpenGBF* in Section 6.6.

6.5.1 Maximum number of threads in *OpenGBF*

In this experiment, we will evaluate the impact of limiting the number of active threads. Figure 6.4 shows the results of setting different values for the thread threshold (Th) on the number of bugs (i.e., the number of *Correct False* verdicts) found by *OpenGBF*. We conducted this experiment by disabling the BMC engine, setting the delay range from 0 $[\mu s]$ to 10^5 $[\mu s]$, and the probability of exiting $p = 0.01\%$. For the thread threshold (Th), we considered different values for comparison: $Th = 0$, meaning there are no limits on the number of active threads; $Th = 5$, $Th = 10$, $Th = 50$, $Th = 100$, $Th = 500$, and $Th = 1000$, where we limit the maximum number of concurrently active threads to 5, 10, 50, 100, 500, and 1000 threads, respectively.

It can be seen that the most optimal value lies in the region around $Th = 5$, and increasing the threshold value leads to fewer bugs being detected because of the increase in the number of computer resources required to maintain a more significant number of active threads. We can claim that many bugs can be identified without considering a large number of threads, as demonstrated by the *wolfMQTT* data race that was detected with $Th = 5$. However, drawing a more robust conclusion applicable to any concurrent program requires a more extensive evaluation of our GBF on a larger set of benchmarks.

6.5.2 Maximum amount of delay in *OpenGBF*

In this experiment, we will evaluate the impact of the amount of delay we inject to force scheduling in *OpenGBF* as described in Section 3.3.3.1. We used a logarithmic scale, as illustrated in Figure 6.5, to compare different delay ranges in *OpenGBF*. Similar to the evaluation of different thread thresholds, we use the number of *Correct False* to evaluate the effectiveness of a given delay bound. For this experiment, we set the thread threshold to 5 active threads and the probability of exiting $p = 0.01\%$. For the range of delay values, we changed the upper bound of the delay's range from 0 $[\mu s]$ (i.e., no delay) to 10^7 $[\mu s]$ (i.e., 10 seconds).

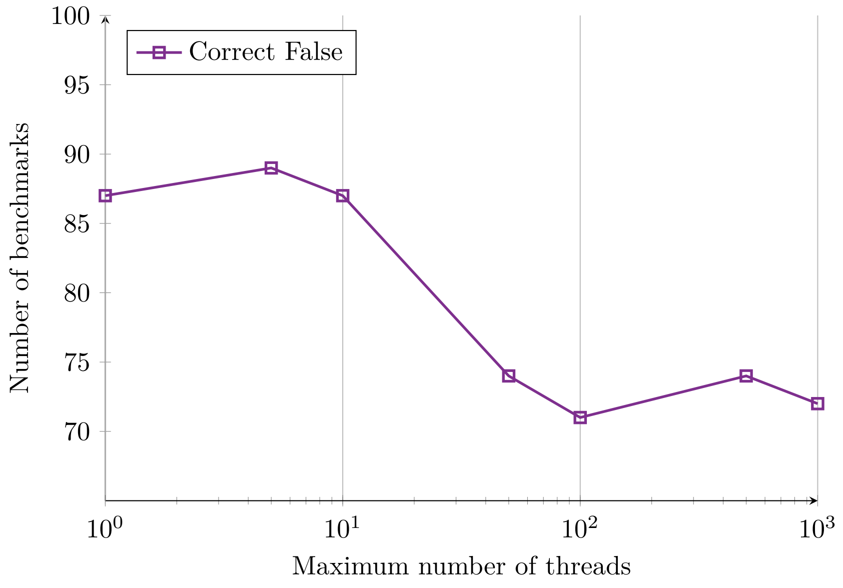


Figure 6.4. The number of bugs (i.e., *Correct False* verdicts) discovered by *OpenGBF* in *EBF* 4.0 for different values of the threshold on the maximum number of active threads.

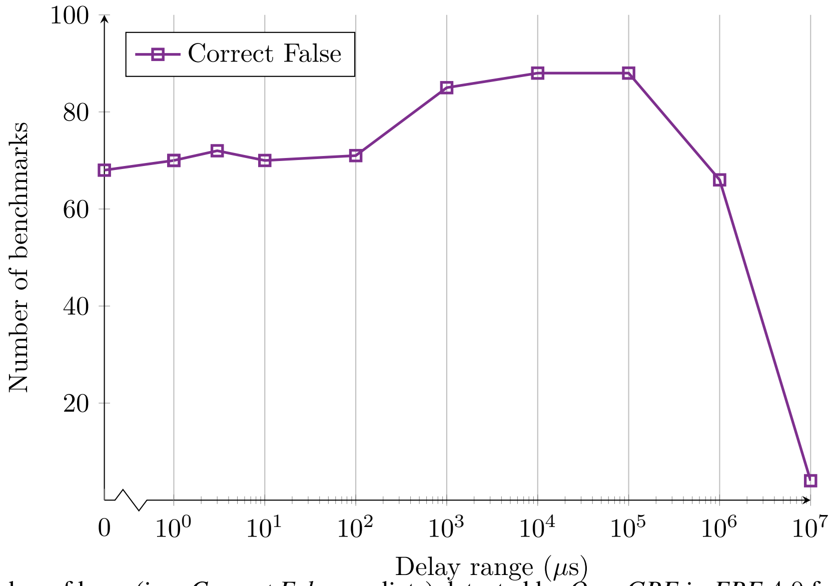


Figure 6.5. Number of bugs (i.e., *Correct False* verdicts) detected by *OpenGBF* in *EBF* 4.0 for different upper bounds of the random delay distributions.

The results show that raising the upper bound of the delay range from 0 to 10^5, $[\mu s]$ gradually enhances *OpenGBF*'s bug-finding capabilities from 68 to 88 benchmarks. Therefore, by setting a larger upper bound on the delay value, we increase the time range for a thread to stay inactive before it is rescheduled again, which increases the number of thread interleavings that our fuzzer explores. However, choosing a larger upper bound (e.g., 10^6 or 10^7) leads to fewer bugs found by the fuzzer due to a higher number of timeouts. This outcome is expected since, with larger delays, the fuzzer spends the majority of the time waiting rather than executing the code. In general, we believe that finding the correct trade-off in the delay range is benchmark-dependent.

6.5.3 Early thread termination in *OpenGBF*

In *OpenGBF*, we terminate the execution of each thread with a probability of p. This specific implementation detail is crucial for avoiding potential deadlocks in the PUT. Additionally, it is useful in helping the fuzzer terminate execution when it hangs, for example, when stuck in an infinite loop.

In this experiment, we aim to demonstrate the impact of different values of p on the bug-finding performance of our *OpenGBF* using the *Concurrency Safety* category of the SV-COMP 2022 benchmark suite. For comparison, we implement an alternative mechanism where each thread's execution is deterministically terminated after a fixed number of instructions (e.g., 10, 100, 1000, 10000 instructions). It is important to note that both termination mechanisms are local to each thread and do not introduce any synchronization overhead. Additionally, we align the plots based on each thread's average number of instructions, which corresponds to the mean $1/p$ of an exponential distribution.

The results in Figure 6.6 demonstrate the consistent performance of our *OpenGBF* across a wide range of p values. Interestingly, removing the termination mechanism altogether only results in minimal degradation in the fuzzer's performance. Furthermore, as the average number of instructions per thread increases, the performance difference between the probabilistic and deterministic termination methods is not significant. However, the probabilistic mechanism shows slower degradation in performance when the average number of instructions decreases. We hypothesize that the probabilistic termination mechanism allows our *OpenGBF* to explore numerous shallow paths along with a few deeper ones, thereby slightly increasing the chance of detecting bugs when the average number of instructions is low. Finally, based on these findings, we used the best parameter setting, $p = 0.01\%$, for all our experiments.

6.5.4 Impact of GBF design choices

To further validate the bug-detection capabilities of *OpenGBF*, we compare our GBF implementation and a "non-instrumented" version of the fuzzer (only *AFL++*). This non-instrumented version does not include the PUT instrumentation described in Algorithm 2. We perform this comparison using the optimal parameter settings mentioned earlier: $Th = 5$, $p = 0.01\%$, and a random delay upper bound of 10^5, $[\mu s]$.

Figure 6.7 illustrates the results of the number of detected bugs in the *Concurrency Safety* category of SV-COMP 2022 using our *OpenGBF*, *OpenGBF* 4.2 (explained in Section 6.3.4) and the non-instrumented version of the GBF. The comparison shows a nearly 75-fold increase in the number of detected bugs between *OpenGBF* 4.2 and the "non-instrumented" GBF. To be more precise, *OpenGBF* detects 88 out of 365 vulnerabilities, which is equivalent to 24.2%

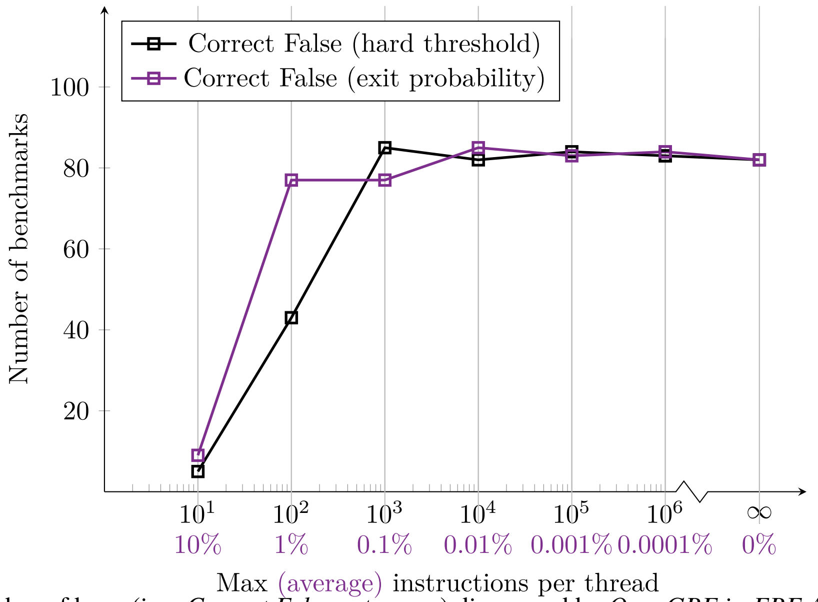


Figure 6.6. Number of bugs (i.e., *Correct False* outcomes) discovered by *OpenGBF* in *EBF* 4.0 for different early thread termination strategies.

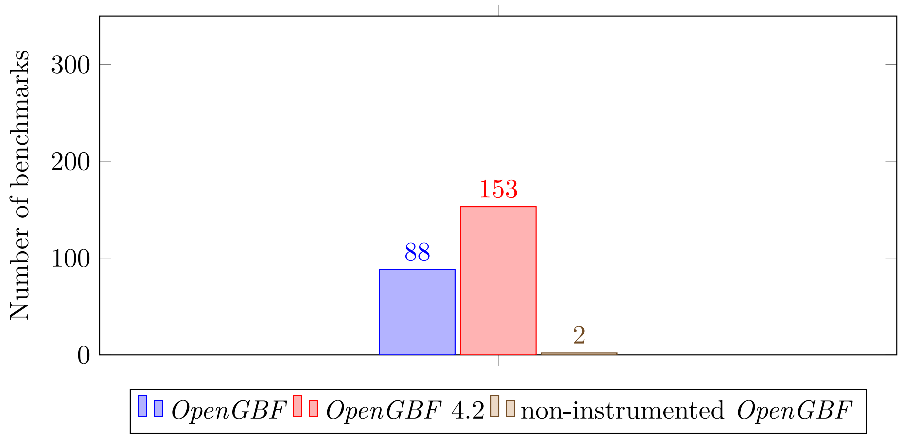


Figure 6.7. The difference between *OpenGBF*, *OpenGBF* 4.2, and the non-instrumented GBF in bug detection capabilities for concurrent programs.

of the total, while *OpenGBF* 4.2 detects 153 out of 365 vulnerabilities, which is equivalent to 41.9% of the total, while the "non-instrumented" GBF detects only 2 out of 365, which is equivalent to 0.55% of the total. This substantial difference in bug detection rates between *OpenGBF* and *OpenGBF* 4.2 and the "non-instrumented" GBF highlights the need to use a concurrency-aware fuzzer in our *EBF*.

Comparing *OpenGBF* to *OpenGBF* 4.2, the latter achieved nearly double the results. This allows us to demonstrate the significant impact of good initial seeds on enhancing the bug-finding capabilities of the GBF.

This evaluation demonstrates that we have successfully achieved our fourth evaluation goal (**EG4**).

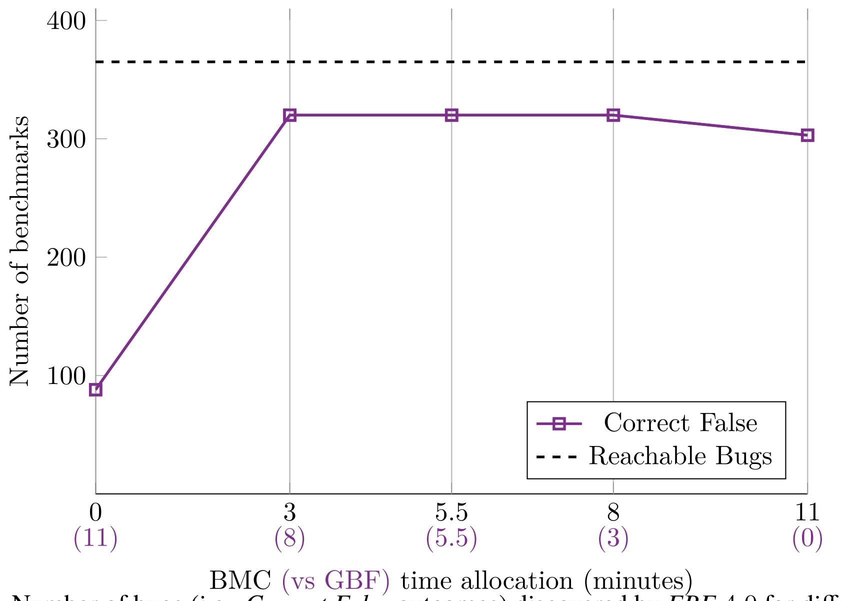


Figure 6.8. Number of bugs (i.e., *Correct False* outcomes) discovered by *EBF* 4.0 for different time allocations between the fuzzer and the BMC.

6.5.5 CPU time allocation inside *EBF*

In this experiment, we explore different ways of distributing the total verification time, which is 15 minutes in total, including 11 minutes allocated between the fuzzer and the BMC (i.e., *ESBMC* $v6.8$) verification engines inside *EBF*. Figure 6.8 demonstrates the results of a relatively wide range of values varying between 3 and 8 minutes per engine, where *EBF* 4.0 produces similar results finding 320 bugs out of 365. At the same time, when the entire 11 minutes are allocated to the BMC engine, the number of detected bugs decrease by approximately $\sim$ 5% to 303 out of 365. Conversely, allocating the entire time to *OpenGBF* leads to a substantial decrease in the overall bug-finding performance of *EBF* 4.0 by over 72.5%. In summary, these results confirm that BMC tools perform better than our *OpenGBF* tool when used in isolation to verify concurrent programs. However, combining both *OpenGBF* and BMC engines in a cooperative framework will achieve better results across very different time allocation choices.

Based on the results of this Section 6.5, we have achieved our fifth evaluation goal (**EG5**).

6.6 Analyzing the non-determinism of *OpenGBF*

Fuzzers, including *OpenGBF*, are fundamentally non-deterministic programs, resulting in performance variations across different runs. To evaluate the effect of non-determinism, we re-run our GBF 20 times on the benchmarks of the present experimental section.

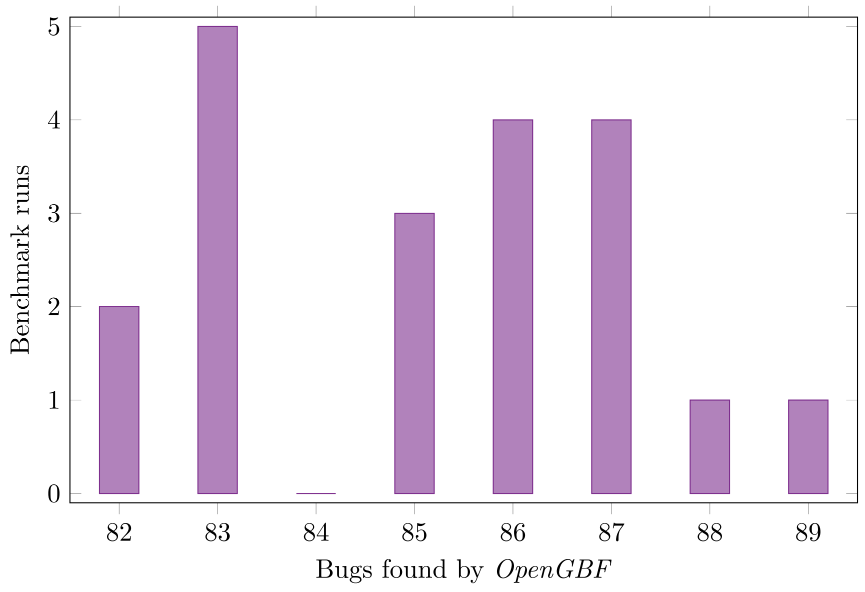


Figure 6.9. The results re-run *OpenGBF* 20 times on SV-COMP 2022.

6.6.1 Non-determinism on SV-COMP 2022 benchmark suite

We executed *OpenGBF* 20 times on SV-COMP 2022 using the same optimal settings as presented in Section 6.5. Figure 6.9 shows the results of the number of bugs *OpenGBF* found in each run, in which the results of 82 bugs occurred twice, 83 occurred five times, 85 occurred three times, 86 and 87 occurred four times, and both 88 and 89 occurred once.

In the worst-case scenario, our fuzzer detected only 82 bugs, while in the best-case scenario, it found 89. Considering that the SV-COMP 2022 suite contains 365 bugs, we expect the distribution to be approximately Gaussian. The empirical mean is 85.2, the variance is 4.3, and the standard deviation is 2.0. Given the small variance relative to the total number of bugs, we can trust the results from Figures 6.4, 6.5, 6.8, and 6.6 to provide robust values for the optimal *EBF* settings.

Additionally, we evaluate the impact of fuzzer non-determinism on each individual file in the SV-COMP 2022 benchmark suite. Specifically, after filtering the files, we found 74 files across the 20 independent runs in which *OpenGBF* always finds a bug. Among those, we select the file with the smallest and largest variance. In addition, we select six samples in between the smallest and the largest variance (i.e., between each 12.5 % we select a sample). Figure 6.10 shows the performance plot of *OpenGBF* in these 9 representative cases. The violin plot shows the extremes of the distributions, together with their median and kernel density estimation. We omit the mean because these distributions are highly non-Gaussian. For example, in the SV-COMP maximum sample, we can notice that the time distribution is highly concentrated around the median.

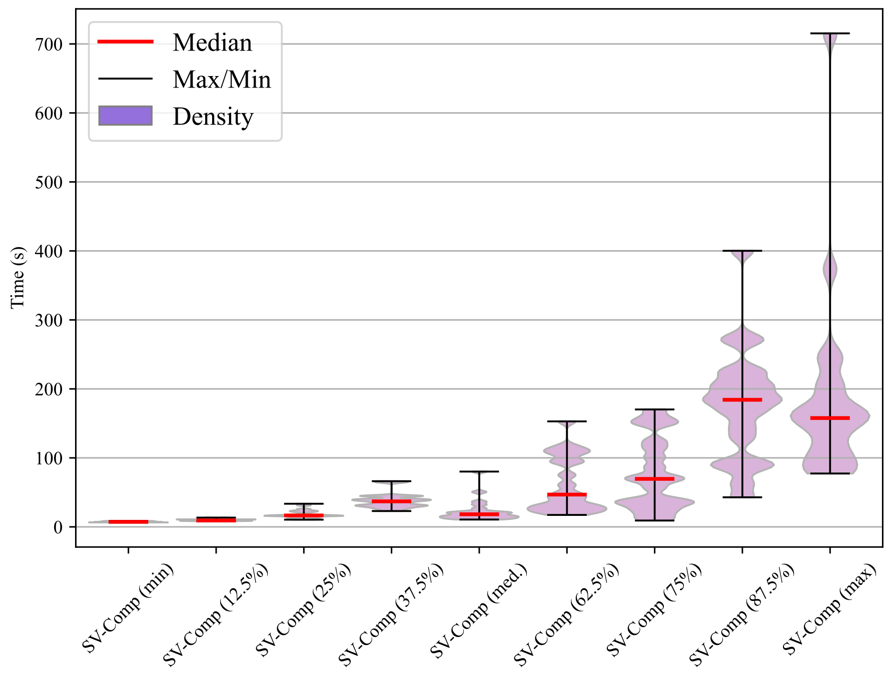


Figure 6.10. Non-determinism of *OpenGBF* across 20 re-runs of the SV-COMP'22 benchmark suite.

6.6.2 Non-determinism on *wolfMQTT* and real-world programs

We executed *OpenGBF* 20 times on both the *wolfMQTT* library and the three real-world programs presented in Table 6.4. Figure 6.11 shows the results as a violin plot. In the case of *pfscan* and *bzip2smp*, *OpenGBF* is capable of finding the bugs almost immediately (see also Table 6.4). In contrast, we can observe more variance for *wolfMQTT* and *swarm* 1.1. In the former case, the distribution is fairly compact in the range between $[10.6s, 66.8s]$. In the latter case, the distribution has a long tail. More precisely, the median time is $9.5s$, and 75% (15 out of the 20) of the runs find a bug in less than $45s$, but there are also occasional outliers where the first bug is reported between $200s$ and $320s$.

6.7 Limitations

We have identified several potential limitations in our current work, which are presented as follows:

6.7.1 Incompleteness of fuzzing for proving safety.

Fuzzing operates by executing a program through various concrete paths, aiming to identify the one that leads to the vulnerability. Accordingly, it cannot provide a formal guarantee of fully exploring the entire state space of the program. Therefore, *EBF* is designed with a

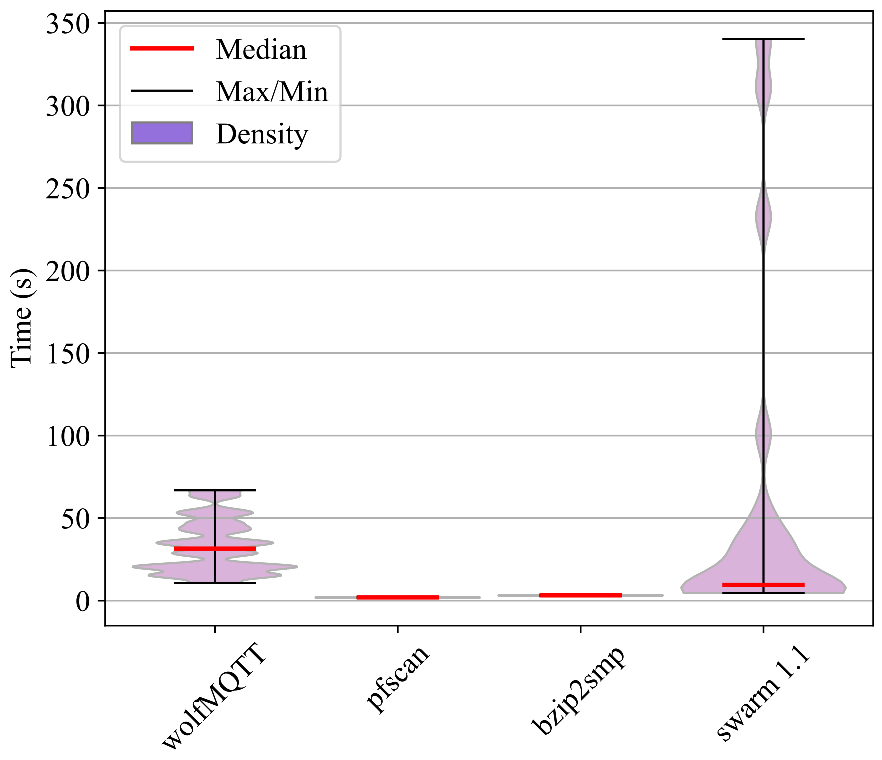


Figure 6.11. Non-determinism in *wolfMQTT* and the real-world programs from Table 6.4 across 20 re-runs.

primary focus on bug-finding rather than proving a program's safety. For safety proofs, we rely solely on the BMC engine up to a given bound k and context switch.

6.7.2 Correct seed sequence for the fuzzer.

It is crucial to establish a mechanism that ensures the seeds generated by BMC for the fuzzer match the input order of the program. This task becomes particularly challenging in concurrent programs, as the program input order also depends on thread interleavings. Additionally, input values generated by BMC can become "lost" within the mix of inputs and delay values generated using nondeterministic functions during the fuzzing process.

6.7.3 Sources of incorrect verdicts in *EBF*

While *EBF* does not encounter conflicting verdicts using the aggregation matrix presented in Table 4.1, the soundness of *EBF*'s verification outcomes depends significantly on the tools used within the cooperative framework.

For instance, if the BMC engine were to report an incorrect *Verification Successful* verdict while the GBF fails to detect any violations within the given time limit (resulting in an *Unknown* verdict), the final verification result would be marked as *Verification Successful*.

Similarly, *EBF* can potentially produce an incorrect false verdict if BMC reports *Unknown*, and the GBF crashes due to an internal bug within the GBF's implementation or unintended *Undefined behaviour* in the benchmarks, rather than detecting an actual vulnerability inside the PUT (with respect to the property violation).

Fortunately, this issue is not critical because *EBF* generates a witness file that leads to the bug. This file can be further evaluated using witness validators (for more details, see Section 4.3.1.5). However, *EBF* can produce incorrect true when the BMC engine reports wrong *Verification Successful*, and our GBF cannot confirm otherwise.

6.7.4 Choice of parameter settings in *EBF*

While we conducted our evaluations on a set of over 700 multi-threaded C programs (as described in Section 6.3), it is important to note that these benchmarks may not fully represent the real-world picture of concurrent software. As a result, the optimal parameter settings for our *OpenGBF* could differ when applied to a different set of multi-threaded benchmarks. Nevertheless, we anticipate that the parameter tuning process for an alternative benchmark set will follow similar patterns to those shown in Figures 6.4, 6.5, and 6.6.

6.8 Summary

In this chapter, we conducted a comprehensive evaluation of *EBF*, with a primary goal of its ability to detect vulnerabilities in concurrent programs. Our evaluations encompassed different aspects, including effectiveness, scalability, robustness, and parameter tuning.

First, to evaluate *EBF's* ability to detect vulnerabilities in concurrent programs, we conducted evaluations within the *Concurrency Safety* category of SV-COMP 2022. In the initial version, *EBF* 2.3, we participated in SV-COMP 2022 and demonstrated a significant improvement of approximately $\sim 7.8\%$ over *CBMC* 5.43, the BMC engine used in that version. Subsequently, we enhanced *EBF* by incorporating algorithmic concepts detailed in Chapter 3, resulting in *EBF* version 4.0. This new version was evaluated on the same benchmark suite, with the distinction that it was tested against various BMC engines, including *CBMC*, *ESBMC*, *Cseq*, and *Deagle*. The results showed that *EBF* 4.0 consistently outperformed all BMC engines. To continue enhancing *OpenGBF*, we introduced a mechanism to generate good initial seeds by extracting them from BMC using error statements. In this latest version, we participated in SV-COMP 2023, building our BMC engine on top of *ESBMC*. Despite this addition, *EBF* maintained its competitive advantage, outperforming *ESBMC* by approximately $\sim 6.65\%$.

Furthermore, we evaluated *EBF* 4.0 on real-world concurrent programs: *wolfMQTT*, *pfscan*, *bzip2smp*, and *swarm1.1*. *EBF* demonstrated its effectiveness by successfully identifying multiple vulnerabilities in these real-world programs, including newly discovered data races, null pointer dereferences, and memory leaks. The vulnerabilities were detected using the *ESBMC* and *OpenGBF* engines. While some BMC tools like *CBMC* and *Deagle* encountered issues when analyzing such programs, *EBF* proved robust and scalable in detecting vulnerabilities in real-world programs where other tools failed.

Additionally, we investigated the impact of varying parameters of *OpenGBF*, such as thread threshold, delay range, and the probability of exiting, on *EBF's* bug-finding capabilities. These experiments helped identify optimal settings, providing valuable insights into parameter tuning.

We also emphasize the importance of using a concurrency-aware fuzzer by comparing *OpenGBF* and *OpenGBF* 4.2, which incorporates initial seed values generated from BMC using the injecting of error statements, against the "non-instrumented" GBF. The results demonstrated that *OpenGBF* 4.2 outperformed the "non-instrumented" GBF, achieving a nearly 75-fold increase in the number of detected bugs.

Lastly, using the optimal settings, we executed *OpenGBF* 20 times on the SV-COMP 2022 benchmark suite and the real-world concurrent programs. We calculated the mean, variance, and standard deviation. The small variance observed in comparison to the total number of bugs detected demonstrates the reliability of *EBF*.

6.9 Future work

In future work, we can evaluate *EBF* on sequential programs by disabling the delay and comparing the results with individual BMC engines in the context of memory-related bugs. Additionally, the new feature of introducing input values from BMC could be valuable in generating effective test cases for sequential programs, which could consider our tool's participation in testing competitions.

Chapter 7

Conclusions

Finding vulnerabilities in concurrent programs remains a challenging problem due to the extreme explosion of the search space in the number of possible interleavings. This Ph.D. thesis focuses on two existing techniques to address this problem: Bounded Model Checking (BMC) and Gray-Box Fuzzing (GBF). However, each technique has its strength in software verification and testing. Consequently, when used independently, each technique can only find a subset of vulnerabilities within concurrent benchmarks. Given the current knowledge in software verification, no techniques effectively leverage both BMC and fuzzing for concurrent program verification. Therefore, the fundamental research question we addressed is *whether combining BMC and fuzzing enhances bug finding in concurrent programs*.

The challenge in answering this fundamental question is threefold. First, while many open-source BMC tools are available in the literature, all the existing concurrency-aware fuzzers are closed-source (at least partially). Consequently, using any of these concurrency-aware fuzzers requires a major reproducibility effort. Second, the challenge of combining BMC and fuzzing techniques for concurrency programs appears from the lack of existing baselines. Third, BMC and fuzzing are dissimilar techniques; their cooperation within the cooperative framework must be carefully coordinated. This specifically involves coordinating the exchange of verification artifacts (i.e., seed values), confirming the final verdict, generating the witness file, and allocating the appropriate time between engines.

In this Ph.D. thesis, we tackled these challenges and made original contributions. Firstly, we developed a novel, fully open-source, concurrency-aware gray-box fuzzer named *OpenGBF*. Secondly, we established a cooperative framework comprising BMC and our concurrency-aware fuzzer, referred to as *EBF*. Lastly, we evaluated *EBF*, focusing on its bug-finding capabilities, scalability, and correctness.

When developing a concurrency-aware fuzzer, specific challenges must be addressed. Since concurrent programs can have different thread interleavings that result in different behaviours, our concurrency-aware fuzzer must explore various thread interleavings and be thread-aware. Also, it is essential to detect not only memory-related errors like memory leaks or buffer overflows but also concurrency-related bugs like data races and deadlocks. Additionally, the fuzzer can become computationally exhausted due to the large number of thread interleavings, so limiting the number of active threads is necessary. Finally, since concurrent programs have non-deterministic behaviour, reproducing the exact input and interleavings that led to a bug is crucial once it has been found. Therefore, we need a mechanism to reproduce the counterexample, specifying the exact input and thread interleavings that resulted in this bug.

In Chapter 3, we provided a detailed explanation of *OpenGBF* and how it addresses these specific challenges. *OpenGBF* is built on top of the state-of-the-art *AFL*++, primarily designed for finding software vulnerabilities in sequential programs [94]. We extended *AFL*++ by instrumenting the Program Under Test (PUT) using an LLVM Pass to make it thread-aware. This extension involved injecting five function calls and using a runtime library for their implementation.

The first function is a delay function, which is inserted after each instruction. This function controls thread interleavings by introducing random delays. After each thread creation and joining function, the second and third functions were injected to monitor the number of thread interleavings by counting the number of active threads. Within the delay function, we determine the limit of active threads and start a new run with different interleavings if it exceeds a predefined threshold. The last two functions were injected after each load and store instruction and after each allocation to collect information about variable names, values, function names, and thread IDs. This information is used to generate a witness file that contains the trace leading to the discovered bug.

In Chapter 4, we introduced our cooperative framework, *EBF*, which combines a BMC engine with our concurrency-aware fuzzer, *OpenGBF*. This combination presents several challenges, particularly when the fuzzer suffers from reaching complex path conditions. Addressing this requires good initial seed values to explore deep branches effectively. Furthermore, since our cooperative framework works sequentially, requiring the BMC to run first to generate initial seeds, we face the challenge of coordinating computational resources effectively. Finally, when multiple tools run within a cooperative framework, the possibility of conflicting results arises. Specifically, in the case of BMC and fuzzing, BMC relies on abstractions of program execution states and symbolic execution, whereas the fuzzer tests concrete inputs and execution schedules. Therefore, we must aggregate these results and generate a witness file illustrating the path to the identified bugs.

EBF mainly consists of four stages. First the safety proving stage, in which we check the PUT for safety using the BMC engine. Second is the seed generation stage, where we instrument the PUT by injecting some error statements after each conditional branch in the program. We then randomly verify PUT containing one error statement using BMC and extract any seed values from the BMC witness file. Third, the falsification stage, where we employ *OpenGBF* and provide it with the extracted seed values. Lastly, the result aggregation stage, where we store the results of both engines and determine the final verdict. We also provide the witness file generated by the engine from which we obtained the final results.

In Chapter 5, we provide additional implementation details, including the programming languages used, the availability of *EBF*, instructions on compiling and running the tool, along with an example that demonstrates the output of each stage of *EBF*, and the available flags for its usage. Finally, it provides an overview of the tool's output.

In Chapter 6, we conduct experiments to show *EBF* performance in bug detection. First, we evaluate *EBF* against SV-COMP benchmark suits, where *EBF* participated in 2022 and 2023. We also evaluate *EBF* against state-of-the-art BMC tools, showing an improvement up to $\sim 14.9\%$ in detecting more bugs than individual BMC tools. Second, we evaluate *EBF* on different real-world programs, including *wolfMQTT* library, where we detected a data race bug and reported it to the developer. After they confirmed it, the bug was successfully fixed. Furthermore, *EBF* successfully reproduces known bugs in several other real-world programs such as *pfscan* [51], *bzip2smp* [52] and *swarm 1.1* [53].

Lastly, we explore and analyze the effects of various settings on the performance of *OpenGBF*. We evaluated the impact of limiting the maximum number of active threads, different delay ranges, and the probability of early termination. Furthermore, we compared the non-instrumented Gray Box Fuzzer (GBF) with two versions of *OpenGBF*, demonstrating significant improvements of up to 75-fold compared to the non-instrumented GBF. This illustrates the effect of injecting delays and generating good seed values from BMC, which enhances the bug-finding capabilities of *OpenGBF*. Additionally, we explore resource allocation within *EBF*. Finally, we analyze the non-deterministic aspects of *OpenGBF*.

7.1 Future work

This Ph.D. thesis opens up multiple research directions because combining two techniques allows us to delve deeper into each individually and collaboratively to make improvements. First, we can generalize our cooperative framework to detect software vulnerabilities in both sequential and concurrent programs. Second, we can expand our cooperative framework by incorporating different techniques and switching between them based on the program's characteristics.

As for the concurrency-aware fuzzer, several improvement directions exist. Firstly, we can enhance the seed generation for GBF by implementing a custom mutator to ensure that the values generated by the fuzzer differ from those seeded by the BMC engine. Another promising approach is applying machine learning techniques for generating initial seeds. Additionally, it is worth investigating the potential impact of reducing delay injection in the PUT, such as injecting delays only after each function call. Furthermore, we can consider re-implementing various concepts from the literature on concurrency-aware fuzzers and evaluating their effectiveness in bug detection.

Lastly, in a different direction for future work, it can be considered to develop a validator that can confirm correctness witnesses for concurrent programs in order to prove program safety. While correctness witnesses for sequential programs are typically the program invariants, ensuring correctness in concurrent programs is more challenging due to the need to consider all possible interleavings, which can lead to a space explosion problem. Therefore, the research question arises: Can the development of a correctness witness validator accurately confirm correctness witnesses and ensure program safety?

Answering this question can be achieved by developing an algorithm that extends existing techniques, used for validating violation witnesses in concurrent programs [177]. This extension can involve instrumenting the invariant interleavings to demonstrate that these invariants hold true in respect to the program and its properties.

References

[1] F. Aljaafari, R. Menezes, E. Manino, F. Shmarov, M. A. Mustafa, and L. Cordeiro, "Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs," *IEEE Access*, vol. 10, pp. 121 365–121 384, 2022. DOI: `10.1109/ACCESS.2022.3223359` (cited on pp. 9, 16, 19, 23, 24, 48, 49, 65).

[2] F. Aljaafari, F. Shmarov, E. Manino, R. Menezes, and L. Cordeiro, "EBF 4.2: Black-Box cooperative verification for concurrent programs (competition contribution)," in *Proc. TACAS (2)*, ser. LNCS, Springer, 2023 (cited on pp. 9, 23, 24, 49, 65, 68).

[3] K. Alshmrany, M. Aldughaim, A. Bhayat, F. Shmarov, F. Aljaafari, and L. Cordeiro, "FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis," *The Formal Aspects of Computing Journal (FAC)*, (cited on p. 9).

[4] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, "Parallelism via multithreaded and multicore cpus," *Computer*, vol. 43, no. 3, pp. 24–32, 2010 (cited on p. 16).

[5] P. A. Pereira, H. F. Albuquerque, I. da Silva, *et al.*, *Concurr. Comput. Pract. Exp.*, vol. 29, no. 22, 2017. DOI: `10.1002/cpe.3934` (cited on pp. 16, 17).

[6] vinod, *Multithreading realtime examples*, `https://androidmaniacom.wordpress.com/2016/12/16/multithreading-realtime-examples/`, 2022 (cited on p. 16).

[7] B. Allington, "Concurrency issues in online banking," *IEEE Concurrency*, May 2015 (cited on p. 16).

[8] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103 009, 2021 (cited on p. 16).

[9] K. M. Alshmrany, A. Bhayat, F. Brauße, *et al.*, "Position paper: Towards a hybrid approach to protect against memory safety vulnerabilities," in *IEEE Secure Development Conference, SecDev 2022, Atlanta, GA, USA, October 18-20, 2022*, IEEE,

2022, pp. 52–58. doi: 10.1109/SecDev53368.2022.00020. [Online]. Available: https://doi.org/10.1109/SecDev53368.2022.00020 (cited on pp. 16, 51).

[10] P. A. Pereira, H. F. Albuquerque, H. Marques, *et al.*, "Verifying CUDA programs using smt-based context-bounded model checking," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, S. Ossowski, Ed., 2016, pp. 1648–1653. doi: 10.1145/2851613.2851830 (cited on p. 16).

[11] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*. Pearson Education, 2006 (cited on p. 16).

[12] C. Liu, D. Zou, P. Luo, B. B. Zhu, and H. Jin, "A heuristic framework to detect concurrency vulnerabilities," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 529–541 (cited on pp. 16, 37–39).

[13] M. Y. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Smt-based refutation of spurious bug reports in the clang static analyzer," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds., IEEE / ACM, 2019, pp. 11–14. doi: 10.1109/ICSE-Companion.2019.00026. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00026 (cited on p. 16).

[14] J. Pan, "Software testing," *Dependable Embedded Systems*, vol. 5, no. 2006, p. 1, 1999 (cited on p. 17).

[15] M. R. Gadelha, R. S. Menezes, and L. C. Cordeiro, "ESBMC 6.1: Automated test case generation using bounded model checking," *Int. J. Softw. Tools Technol. Transf.*, vol. 23, no. 6, pp. 857–861, 2021. doi: 10.1007/s10009-020-00571-2. [Online]. Available: https://doi.org/10.1007/s10009-020-00571-2 (cited on p. 17).

[16] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339 (cited on pp. 17, 25).

[17] L. C. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., ACM, 2011, pp. 331–340.

DOI: 10.1145/1985793.1985839. [Online]. Available: https://doi.org/10.1145/1985793.1985839 (cited on pp. 17, 66).

[18] K. R. Apt, E.-R. Olderog, and K. Apt, *Verification of sequential and concurrent programs*. Springer, 2009, vol. 2 (cited on p. 17).

[19] Q. Stievenart, J. Nicolay, W. De Meuter, and C. De Roover, "Detecting concurrency bugs in higher-order programs through abstract interpretation," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, 2015, pp. 232–243 (cited on p. 17).

[20] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "Fusebmc v4: Smart seed generation for hybrid fuzzing - (competition contribution)," in *Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, E. B. Johnsen and M. Wimmer, Eds., ser. Lecture Notes in Computer Science, vol. 13241, Springer, 2022, pp. 336–340. DOI: 10.1007/978-3-030-99429-7_19. [Online]. Available: https://doi.org/10.1007/978-3-030-99429-7%5C_19 (cited on p. 17).

[21] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013 (cited on p. 17).

[22] Y. Li, S. Ji, C. Lyu, *et al.*, "V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3745–3756, 2022. DOI: 10.1109/TCYB.2020.3013675 (cited on pp. 17, 18).

[23] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1973, pp. 194–206 (cited on p. 17).

[24] M. B. Dwyer and L. A. Clarke, "Data flow analysis for verifying properties of concurrent programs," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 5, pp. 62–75, 1994 (cited on p. 17).

[25] A. Gosain and G. Sharma, "Static analysis: A survey of techniques and tools," in *Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014*, Springer, 2015, pp. 581–591 (cited on p. 17).

[26] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 158–171, 1976 (cited on p. 17).

[27] A. Alfred V, L. Monica S, S. Ravi, U. Jeffrey D, *et al.*, *Compilers-principles, techniques, and tools*. pearson Education, 2007 (cited on p. 17).

[28] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from c protocol code by symbolic execution," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 331–340 (cited on p. 17).

[29] A. Biere, "Bounded model checking," in *Handbook of Satisfiability*, IOS Press, 2009, pp. 457–481 (cited on pp. 17, 28).

[30] M. R. Gadelha, R. Menezes, F. R. Monteiro, L. C. Cordeiro, and D. Nicole, "Esbmc: Scalable and precise test generation based on the floating-point theory: (competition contribution)," in *International Conference on Fundamental Approaches to Software Engineering*, Springer International Publishing Cham, 2020, pp. 525–529 (cited on pp. 17, 30).

[31] D. Kroening and M. Tautschnig, "CBMC–c bounded model checker," in *TACAS*, Springer, 2014, pp. 389–391 (cited on pp. 17, 30).

[32] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, 2011, pp. 184–190 (cited on p. 17).

[33] F. He, Z. Sun, and H. Fan, "Satisfiability modulo ordering consistency theory for multi-threaded program verification," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1264–1279 (cited on pp. 17, 31).

[34] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Lazy-cseq: A lazy sequentialization tool for c: (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, Springer, 2014, pp. 398–401 (cited on pp. 17, 31).

[35] M. Vanhoef and F. Piessens, "Symbolic execution of security protocol implementations: Handling cryptographic primitives," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018 (cited on p. 17).

[36] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, “Chopped symbolic execution,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 350–360 (cited on p. 17).

[37] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990 (cited on p. 17).

[38] P. Tsankov, M. T. Dashti, and D. Basin, “Secfuzz: Fuzz-testing security protocols,” in *2012 7th International Workshop on Automation of Software Test (AST)*, IEEE, 2012, pp. 1–7 (cited on p. 18).

[39] V. J. Manès, H. Han, C. Han, *et al.*, “The art, science, and engineering of fuzzing: A survey,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019 (cited on pp. 18, 32).

[40] B. S. Pak, “Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution,” *School of Computer Science Carnegie Mellon University*, 2012 (cited on p. 18).

[41] H. Chen, S. Guo, Y. Xue, *et al.*, “{Muzz}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2325–2342 (cited on pp. 18, 34, 37, 38).

[42] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, “Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach,” in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1475–1482 (cited on pp. 18, 44).

[43] K. M. Alshmrany, R. S. Menezes, M. R. Gadelha, and L. C. Cordeiro, “Fusebmc: A white-box fuzzer for finding security vulnerabilities in c programs (competition contribution),” in *Fundamental Approaches to Software Engineering: 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings 24*, Springer International Publishing, 2021, pp. 363–367 (cited on pp. 18, 44, 68, 69).

[44] A. Basak Chowdhury and R. K. Medicherla, “Verifuzz: Program aware fuzzing: (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague,*

Czech Republic, April 6–11, 2019, Proceedings, Part III 25, Springer, 2019, pp. 244–249 (cited on pp. 18, 44).

[45] D. Beyer and H. Wehrheim, “Verification artifacts in cooperative verification: Survey and unifying component framework,” in *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2020, pp. 143–167 (cited on pp. 18, 40, 64).

[46] D. Beyer, M. Spiessl, and S. Umbricht, “Cooperation between automatic and interactive software verifiers,” in *Software Engineering and Formal Methods*, B.-H. Schlingloff and M. Chai, Eds., Cham: Springer International Publishing, 2022, pp. 111–128, ISBN: 978-3-031-17108-6 (cited on p. 18).

[47] N. Stephens, J. Grosen, C. Salls, *et al.*, “Driller: Augmenting fuzzing through selective symbolic execution.,” in *NDSS*, vol. 16, 2016, pp. 1–16 (cited on pp. 18, 34, 44).

[48] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “{Qsym}: A practical concolic execution engine tailored for hybrid fuzzing,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761 (cited on pp. 18, 44).

[49] J. Li, B. Zhao, and C. Zhang, “Fuzzing: A survey,” *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018 (cited on pp. 18, 25, 31–33, 43).

[50] `https://github.com/fatimahkj/EBF`, 2022 (cited on pp. 21, 40).

[51] `https://manpages.ubuntu.com/manpages/focal/man1/pfscan.1.html`, 2022 (cited on pp. 22, 98, 112).

[52] `http://bzip2smp.sourceforge.net/`, 2022 (cited on pp. 22, 99, 112).

[53] D. Bader, V. Kanade, and K. Madduri, “Swarm: A parallel programming framework fro multicore processors,” in *Proc. 21st Intl. Parallel and Distr. Processing Symp (IPDPS 2007), Long Beach, CA (March 2007)*, 2007 (cited on pp. 22, 99, 112).

[54] F. R. Monteiro, E. H. d. S. Alves, I. S. Silva, H. I. Ismail, L. C. Cordeiro, and E. B. de Lima Filho, “Esbmc-gpu a context-bounded model checking tool to verify cuda programs,” *Science of Computer Programming*, vol. 152, pp. 63–69, 2018 (cited on p. 25).

[55] F. R. Monteiro, M. Garcia, L. C. Cordeiro, and E. B. de Lima Filho, “Bounded model checking of C++ programs based on the qt cross-platform framework,” *Softw. Test. Verification Reliab.*, vol. 27, no. 3, 2017. DOI: `10.1002/stvr.1632`. [Online]. Available: `https://doi.org/10.1002/stvr.1632` (cited on p. 25).

[56] `https://www.intel.com/content/www/us/en/docs/inspector/user-guide-linux/2022/invalid-memory-access.html`, 2023 (cited on p. 26).

[57] T. V. N. Nguyen, F. Irigoin, C. Ancourt, and F. Coelho, "Automatic detection of uninitialized variables," in *Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 12*, Springer, 2003, pp. 217–231 (cited on p. 26).

[58] Y. Xie and A. Aiken, "Context-and path-sensitive memory leak detection," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005, pp. 115–125 (cited on p. 26).

[59] Y. Cai, B. Zhu, R. Meng, *et al.*, "Detecting concurrency memory corruption vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 706–717 (cited on p. 26).

[60] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t)," *Journal of the ACM (JACM)*, vol. 53, no. 6, pp. 937–977, 2006 (cited on p. 28).

[61] L. C. Chaves, I. Bessa, L. C. Cordeiro, and D. Kroening, "Dsvalidator: An automated counterexample reproducibility tool for digital systems," in *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC 2018, Porto, Portugal, April 11-13, 2018*, M. Prandini and J. V. Deshmukh, Eds., ACM, 2018, pp. 253–258. DOI: `10.1145/3178126.3178151`. [Online]. Available: `https://doi.org/10.1145/3178126.3178151` (cited on p. 28).

[62] L. Cordeiro, "Smt-based bounded model checking for multi-threaded software in embedded systems," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, pp. 373–376 (cited on p. 28).

[63] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," in *International conference on tools and algorithms for the construction and analysis of systems*, Springer, 2005, pp. 93–107 (cited on pp. 29, 30, 72).

[64] `https://github.com/esbmc/esbmc`, 2021 (cited on pp. 30, 93).

[65] *Cbmc*, `https://github.com/diffblue/cbmc`, 2022 (cited on pp. 30, 93).

[66] *Cseq*, `https://www.southampton.ac.uk/~gp1y10/cseq/cseq.html`, 2022 (cited on pp. 30, 93).

[67] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*, K. Etessami and S. K. Rajamani, Eds., Springer, 2005, pp. 82–97, ISBN: 978-3-540-31686-2 (cited on p. 30).

[68] M. R. Gadelha, R. S. Menezes, and L. C. Cordeiro, "Esbmc 6.1: Automated test case generation using bounded model checking," *International Journal on Software Tools for Technology Transfer*, vol. 23, pp. 857–861, 2021 (cited on p. 30).

[69] I. Bessa, H. Ismail, R. M. Palhares, L. C. Cordeiro, and J. E. C. Filho, "Formal non-fragile stability verification of digital control systems with uncertainty," *IEEE Trans. Computers*, vol. 66, no. 3, pp. 545–552, 2017. DOI: `10.1109/TC.2016.2601328`. [Online]. Available: `https://doi.org/10.1109/TC.2016.2601328` (cited on p. 30).

[70] R. B. Abreu, M. Y. R. Gadelha, L. C. Cordeiro, E. B. de Lima Filho, and W. S. da Silva Jr., "Bounded model checking for fixed-point digital filters," *J. Braz. Comput. Soc.*, vol. 22, no. 1, 1:1–1:20, 2016. DOI: `10.1186/s13173-016-0041-8`. [Online]. Available: `https://doi.org/10.1186/s13173-016-0041-8` (cited on p. 30).

[71] L. C. Chaves, I. Bessa, H. Ismail, A. B. dos Santos Frutuoso, L. C. Cordeiro, and E. B. de Lima Filho, "Dsverifier-aided verification applied to attitude control software in unmanned aerial vehicles," *IEEE Trans. Reliab.*, vol. 67, no. 4, pp. 1420–1441, 2018. DOI: `10.1109/TR.2018.2873260`. [Online]. Available: `https://doi.org/10.1109/TR.2018.2873260` (cited on p. 30).

[72] F. R. Monteiro, M. R. Gadelha, and L. C. Cordeiro, "Model checking C++ programs," *Softw. Test. Verification Reliab.*, vol. 32, no. 1, 2022. DOI: `10.1002/stvr.1793`. [Online]. Available: `https://doi.org/10.1002/stvr.1793` (cited on p. 30).

[73] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," in *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, Springer, 2013, pp. 141–157 (cited on p. 30).

[74] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *Computer Aided Verification: 17th International Conference, CAV 2005, Edin-*

burgh, Scotland, UK, July 6-10, 2005. Proceedings 17, Springer, 2005, pp. 82–97 (cited on p. 30).

[75] D. Beyer, "Software verification: 10th comparative evaluation (sv-comp 2021)," in *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27*, Springer, 2021, pp. 401–422. DOI: `10.1007/978-3-030-72013-1_24` (cited on p. 31).

[76] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded model checking of multi-threaded c programs via lazy sequentialization," in *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, A. Biere and R. Bloem, Eds., Springer, 2014, pp. 585–602, ISBN: 978-3-319-08867-9 (cited on p. 31).

[77] D. Beyer, "Progress on software verification: Sv-comp 2022," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2022, pp. 375–402 (cited on pp. 31, 73, 74, 89, 91).

[78] D. Beyer, "Competition on software verification and witness validation: Sv-comp 2023," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds., Cham: Springer Nature Switzerland, 2023, pp. 495–522, ISBN: 978-3-031-30820-8 (cited on pp. 31, 94).

[79] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The mathsat 4 smt solver," in *Computer Aided Verification*, 2008, pp. 299–303, ISBN: 9783540705437. DOI: `10.1007/978-3-540-70545-1_28` (cited on p. 31).

[80] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "Fusebmc: An energy-efficient test generator for finding security vulnerabilities in C programs," in *International Conference On Tests And Proofs*, F. Loulergue and F. Wotawa, Eds., vol. 12740, Sprnger, 2021, pp. 85–105. DOI: `10.1007/978-3-030-79379-1\_6` (cited on p. 31).

[81] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018 (cited on pp. 31, 34).

[82] Y. Chen, A. Groce, C. Zhang, *et al.*, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 197–208 (cited on p. 32).

[83] P. Francis, D. Leon, M. Minch, and A. Podgurski, "Tree-based methods for classifying software failures," in *15th International Symposium on Software Reliability Engineering*, IEEE, 2004, pp. 451–462 (cited on p. 32).

[84] M. Zalewski, "American fuzzy lop," 2015. [Online]. Available: `https://lcamtuf.coredump.cx/afl/` (cited on pp. 32, 34, 38).

[85] `https://linux.die.net/man/5/core`, 2023 (cited on pp. 32, 56).

[86] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007 (cited on p. 32).

[87] C. Miller, Z. Peterson, *et al.*, "Analysis of mutation and generation-based fuzzing. independent security evaluators," Tech. Rep, Tech. Rep., 2007 (cited on p. 32).

[88] A. Biyani, G. Sharma, J. Aghav, P. Waradpande, P. Savaji, and M. Gautam, "Extension of SPIKE for encrypted protocol fuzzing," in *MINES 2011*, 2011, pp. 343–347 (cited on p. 34).

[89] "Sulley fuzzing framework," 2017. [Online]. Available: `https://github.com/OpenRCE/sulley` (cited on p. 34).

[90] "Peach fuzzing framework," 2017. [Online]. Available: `https://www.peach.tech/` (cited on p. 34).

[91] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*, IEEE, 2016, pp. 157–157 (cited on p. 34).

[92] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018 (cited on p. 34).

[93] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012 (cited on pp. 34, 44).

[94] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020, pp. 10–10 (cited on pp. 34, 37, 38, 111).

[95] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," in *NDSS*, vol. 17, 2017, pp. 1–14 (cited on p. 34).

[96] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020 (cited on p. 34).

[97] P. 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 (cited on p. 34).

[98] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007 (cited on p. 34).

[99] C.-K. Luk, R. Cohn, R. Muth, *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005 (cited on p. 34).

[100] E. Jääskelä, "Genetic algorithm in code coverage guided fuzz testing," M.S. thesis, E. Jääskelä, 2016 (cited on p. 34).

[101] R. L. Seagle Jr, "A framework for file format fuzzing with genetic algorithms," 2012 (cited on p. 34).

[102] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 475–485 (cited on p. 34).

[103] T. Kempf, K. Karuri, and L. Gao, "Software instrumentation," in Wiley Online Library, Sep. 2008, pp. 1–11, ISBN: 9780470050118. DOI: `10.1002/9780470050118.ecse386` (cited on p. 35).

[104] `https://llvm.org/docs/Passes.html`, 2023 (cited on pp. 35, 36).

[105] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with {bek}," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011 (cited on p. 36).

[106] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71, ISBN: 9781605587936. DOI: `10.1145/1791194.1791203` (cited on pp. 36, 50, 66).

[107] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker,” in *2012 USENIX annual technical conference (USENIX ATC 12)*, USA, 2012, p. 28 (cited on pp. 36, 50, 66).

[108] E. Stepanov and K. Serebryany, “Memorysanitizer: Fast detector of uninitialized memory use in c++,” in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2015, pp. 46–55 (cited on pp. 37, 51, 66).

[109] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, 2023 (cited on pp. 37, 51).

[110] Y. Ko, B. Zhu, and J. Kim, “Fuzzing with automatically controlled interleavings to detect concurrency bugs,” *Journal of Systems and Software*, p. 111 379, 2022 (cited on pp. 37–39).

[111] N. Vinesh and M. Sethumadhavan, “Confuzz—a concurrency fuzzer,” in *ICTSCI e*, A. K. Luhach, J. A. Kosa, R. C. Poonia, X.-Z. Gao, and D. Singh, Eds., Springer, 2020, pp. 667–691, ISBN: 978-981-15-0029-9 (cited on pp. 37–39).

[112] M. Xu, S. Kashyap, H. Zhao, and T. Kim, “Krace: Data race fuzzing for kernel file systems,” in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1643–1660 (cited on pp. 37–40).

[113] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, “Context-sensitive and directional concurrency fuzzing for data-race detection,” 2022 (cited on pp. 37–39).

[114] D. R. Jeong, B. Lee, I. Shin, and Y. Kwon, “Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing,” in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023, pp. 2104–2121 (cited on pp. 38, 40).

[115] `https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.llvm.md/`, 2022 (cited on pp. 38, 49, 79).

[116] `https://aflplus.plus/docs/best_practices/`, 2023 (cited on pp. 38, 50).

[117] O. M. Alhawi, M. A. Mustafa, and L. C. Cordeiro, “Finding security vulnerabilities in unmanned aerial vehicles using software verification,” in *2019 International Workshop on Secure Internet of Things, SIoT 2019, Luxembourg, Luxembourg, September 26, 2019*, IEEE, 2019, pp. 1–9. DOI: `10.1109/SIOT48044.2019.9637109`. [Online]. Available: `https://doi.org/10.1109/SIOT48044.2019.9637109` (cited on p. 39).

[118] `https://github.com/google/AFL`, 2021 (cited on p. 39).

[119] C. Lie, *Personal communications*, email, 2022 (cited on p. 39).

[120] D. Beyer, "Software verification and verifiable witnesses: (report on sv-comp 2015)," in *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, Springer, 2015, pp. 401–416 (cited on p. 40).

[121] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann, "Correctness witnesses: Exchanging verification results between verifiers," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 326–337 (cited on p. 40).

[122] M. Y. R. Gadelha, F. R. Monteiro, L. C. Cordeiro, and D. A. Nicole, "Towards counterexample-guided k-induction for fast bug detection," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds., ACM, 2018, pp. 765–769. DOI: `10.1145/3236024.3264840`. [Online]. Available: `https://doi.org/10.1145/3236024.3264840` (cited on p. 41).

[123] O. M. Alhawi, H. Rocha, M. R. Gadelha, L. C. Cordeiro, and E. B. de Lima Filho, "Verification and refutation of C programs based on k-induction and invariant inference," *Int. J. Softw. Tools Technol. Transf.*, vol. 23, no. 2, pp. 115–135, 2021. DOI: `10.1007/s10009-020-00564-1`. [Online]. Available: `https://doi.org/10.1007/s10009-020-00564-1` (cited on p. 41).

[124] *Graphml-based exchange format for violation witnesses and correctness witnesses*, `https://github.com/sosy-lab/sv-witnesses/blob/main/README-GraphML.md`, 2023 (cited on p. 41).

[125] D. Beyer, "Second competition on software testing: Test-comp 2020.," in *FASE*, 2020, pp. 505–519 (cited on p. 44).

[126] D. Beyer, "Advances in automatic software testing: Test-comp 2022.," in *FASE*, 2022, pp. 321–335 (cited on p. 44).

[127] D. Beyer, "Software testing: 5th comparative evaluation: Test-comp 2023," *Fundamental Approaches to Software Engineering LNCS 13991*, p. 309, 2023 (cited on p. 44).

[128] H. Rocha, R. Menezes, L. C. Cordeiro, and R. Barreto, “Map2check: Using symbolic execution and fuzzing: (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II 26*, Springer, 2020, pp. 403–407 (cited on p. 44).

[129] R. Menezes, H. Rocha, L. Cordeiro, and R. Barreto, “Map2check using llvm and klee: (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II 24*, Springer, 2018, pp. 437–441 (cited on p. 44).

[130] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs.,” in *OSDI*, vol. 8, 2008, pp. 209–224 (cited on p. 44).

[131] `https://llvm.org/docs/LibFuzzer.html`, 2021 (cited on p. 44).

[132] P. Godefroid, M. Y. Levin, and D. Molnar, “Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft.,” *Queue*, vol. 10, no. 1, pp. 20–27, 2012 (cited on p. 44).

[133] H. M. Le, “Llvm-based hybrid fuzzing with libkluzzer (competition contribution).,” in *FASE*, 2020, pp. 535–539 (cited on pp. 44, 68, 69).

[134] C. Wen, M. He, B. Wu, Z. Xu, and S. Qin, “Controlled concurrency testing via periodical scheduling,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 474–486 (cited on p. 45).

[135] Y. Cai, C. Ye, Q. Shi, and C. Zhang, “Peahen: Fast and precise static deadlock detection via context reduction,” 2022 (cited on p. 45).

[136] S. Mukherjee, P. Deligiannis, A. Biswas, and A. Lal, “Learning-based controlled concurrency testing,” *Proceedings of the ACM on Programming Languages*, vol. 4, 2020. DOI: `10.1145/3428298`. [Online]. Available: `https://doi.org/10.1145/3428298` (cited on p. 45).

[137] Y. Cai, H. Yun, J. Wang, L. Qiao, and J. Palsberg, "Sound and efficient concurrency bug prediction," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 255–267, ISBN: 9781450385626. DOI: 10.1145/3468264.3468549. [Online]. Available: https://doi.org/10.1145/3468264.3468549 (cited on p. 45).

[138] U. Mathur, A. Pavlogiannis, and M. Viswanathan, "Optimal prediction of synchronization-preserving races," *Proceedings of the ACM on Programming Languages*, vol. 5, Jan. 2021. DOI: 10.1145/3434317. [Online]. Available: https://doi.org/10.1145/3434317 (cited on p. 45).

[139] Y. Cai, P. Yao, and C. Zhang, "Canary: Practical static detection of inter-thread value-flow bugs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1126–1140, ISBN: 9781450383912. DOI: 10.1145/3453483.3454099. [Online]. Available: https://doi.org/10.1145/3453483.3454099 (cited on p. 45).

[140] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency {use-after-free} bugs in linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 255–268 (cited on p. 45).

[141] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 174–186 (cited on p. 45).

[142] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for c/c++ concurrency," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–32, 2017 (cited on p. 45).

[143] M. Kokologiannakis and V. Vafeiadis, "Genmc: A model checker for weak memory models," in *International Conference on Computer Aided Verification*, Springer, 2021, pp. 427–440 (cited on p. 46).

[144] M. Kokologiannakis, A. Raad, and V. Vafeiadis, "Effective lock handling in stateless model checking," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, 2019 (cited on p. 46).

[145] K. Yu, C. Wang, Y. Cai, X. Luo, and Z. Yang, "Detecting concurrency vulnerabilities based on partial orders of memory and thread events," in *Proceedings of the 29th ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 280–291 (cited on p. 46).

[146] L. C. Cordeiro and E. B. de Lima Filho, "Smt-based context-bounded model checking for embedded systems: Challenges and future trends," *ACM SIGSOFT Softw. Eng. Notes*, vol. 41, no. 3, pp. 1–6, 2016. DOI: `10.1145/2934240.2934247`. [Online]. Available: `https://doi.org/10.1145/2934240.2934247` (cited on p. 49).

[147] `https://aflplus.plus/docs/notes_for_asan/`, 2023 (cited on pp. 50, 51, 66).

[148] H. O. Rocha, R. S. Barreto, and L. C. Cordeiro, "Hunting memory bugs in C programs with map2check - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, M. Chechik and J. Raskin, Eds., ser. Lecture Notes in Computer Science, vol. 9636, Springer, 2016, pp. 934–937. DOI: `10.1007/978-3-662-49674-9\_64`. [Online]. Available: `https://doi.org/10.1007/978-3-662-49674-9%5C_64` (cited on p. 51).

[149] Y. Jeon, W. Han, N. Burow, and M. Payer, "{Fuzzan}: Efficient sanitizer metadata design for fuzzing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 249–263 (cited on p. 51).

[150] `https://manpages.ubuntu.com/manpages/focal/en/man1/afl-clang-fast.1.html`, 2023 (cited on p. 51).

[151] H. Rocha, R. S. Barreto, L. C. Cordeiro, and A. D. Neto, "Understanding programming bugs in ANSI-C software using bounded model checking counter-examples," in *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, J. Derrick, S. Gnesi, D. Latella, and H. Treharne, Eds., ser. Lecture Notes in Computer Science, vol. 7321, Springer, 2012, pp. 128–142. DOI: `10.1007/978-3-642-30729-4\_10`. [Online]. Available: `https://doi.org/10.1007/978-3-642-30729-4%5C_10` (cited on p. 51).

[152] `https://llvm.org/docs/LangRef.html#phi-instruction`, 2020 (cited on p. 53).

[153] *Declaring attributes of functions*, `https://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc/Function-Attributes.html`, 2022 (cited on p. 57).

[154] D. Beyer, M. Huisman, V. Klebanov, and R. Monaham, “Evaluating software verification systems: Benchmarks and competitions,” *Dagstuhl Reports*, vol. 4, no. 4, pp. 1–19, 2014 (cited on p. 58).

[155] O. Llorente-Vazquez, I. Santos-Grueiro, and P. G. Bringas, “When memory corruption met concurrency: Vulnerabilities in concurrent programs,” *IEEE Access*, 2023 (cited on p. 64).

[156] D. Beyer, S. Kanav, and C. Richter, “Construction of verifier combinations based on off-the-shelf verifiers,” in *Fundamental Approaches to Software Engineering*, E. B. Johnsen and M. Wimmer, Eds., Cham: Springer International Publishing, 2022, pp. 49–70, ISBN: 978-3-030-99429-7 (cited on pp. 64, 65).

[157] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Satzilla-07: The design and analysis of an algorithm portfolio for sat,” in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 712–727, ISBN: 978-3-540-74970-7 (cited on p. 65).

[158] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer, “Witness validation and stepwise testification across software verifiers,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 721–733 (cited on p. 66).

[159] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, “Bounded verification of multi-threaded programs via lazy sequentialization,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 44, no. 1, pp. 1–50, 2021 (cited on p. 66).

[160] Z. Sun, H. Fan, and F. He, “Consistency-preserving propagation for smt solving of concurrent program verification,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 929–956, 2022 (cited on p. 66).

[161] A. Coto, O. Inverso, E. Sales, and E. Tuosto, “A prototype for data race detection in cseq 3: (competition contribution),” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2022, pp. 413–417 (cited on p. 66).

[162] H. Feng, L. Yin, W. Lin, X. Zhao, and W. Dong, “Rchecker: A cbmc-based data race detector for interrupt-driven programs,” in *2020 IEEE 20th International Confer-*

ence on Software Quality, Reliability and Security Companion (QRS-C), IEEE, 2020, pp. 465–471 (cited on p. 66).

[163] A. Horadam, "A generalized fibonacci sequence," *The American Mathematical Monthly*, vol. 68, no. 5, pp. 455–459, 1961 (cited on p. 72).

[164] G. D. Maayan. "Sv-comp rules." (2021), [Online]. Available: `https://sv-comp.sosy-lab.org/2022/rules.php` (visited on 2021) (cited on pp. 73, 90).

[165] D. Beyer and J. Strejček, "Case study on verification-witness validators: Where we are and where we go," in *International Static Analysis Symposium*, Springer, 2022, pp. 160–174 (cited on p. 73).

[166] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: Requirements and solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, pp. 1–29, 2019 (cited on pp. 80, 91).

[167] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, 2004, p. 75 (cited on p. 80).

[168] G. Van Rossum and F. L. Drake Jr, *Python reference manual.* Centrum voor Wiskunde en Informatica Amsterdam, 1995 (cited on p. 80).

[169] `https://github.com/fatimahkj/EBF/releases/tag/EBF4.1`, 2022 (cited on p. 80).

[170] `https://gitlab.com/sosy-lab/sv-comp/archives-2023/-/blob/main/2023/ebf.zip`, 2023 (cited on p. 80).

[171] `https://sv-comp.sosy-lab.org/2022/benchmarks.php`, 2022 (cited on p. 90).

[172] *Deagle*, `https://https://github.com/thufv/Deagle`, 2022 (cited on p. 93).

[173] D. Spanti, "Http and mqtt: A comparison in the context of industry 4.0," Ph.D. dissertation, Politecnico di Torino, 2020 (cited on p. 95).

[174] `https://github.com/wolfSSL/wolfMQTT`, 2021 (cited on pp. 95, 96).

[175] `https://www.allaboutcircuits.com/news/wolfmqtt-client-library-end-to-end-encryption-m2m-IoT-MQTT/`, 2023 (cited on p. 96).

[176] *Mosquitto*, `https://mosquitto.org/`, 2021 (cited on p. 97).

[177] D. Beyer and K. Friedberger, “Violation witnesses and result validation for multi-threaded programs,” in *International Symposium on Leveraging Applications of Formal Methods*, Springer, 2020, pp. 449–470 (cited on p. 113).

[178] `https://www.arm.com/architecture/cpu/morello`, 2023 (cited on p. 136).

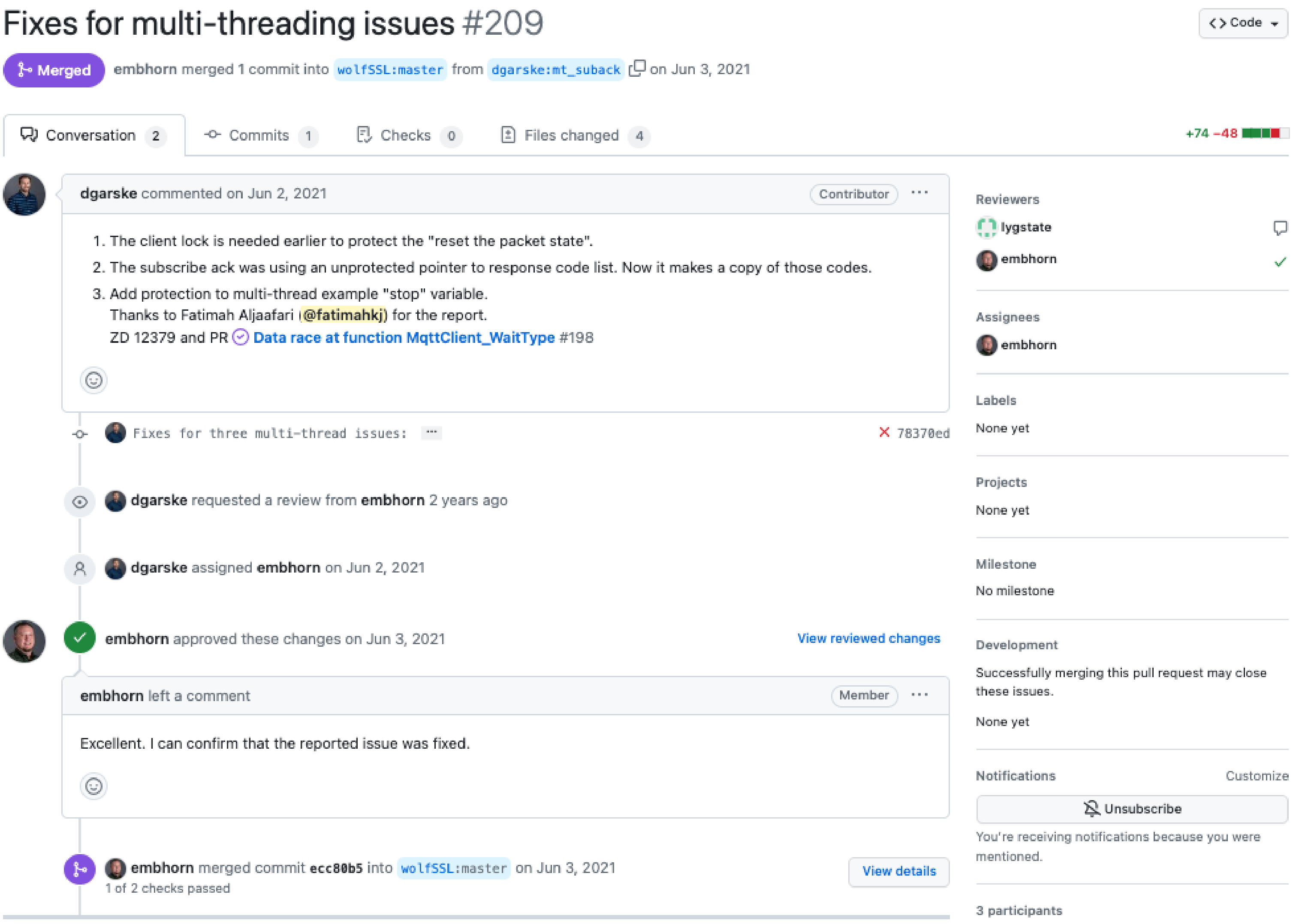
Appendices

Appendix A

Detecting a data race in *wolfMQTT*

After reporting to the *wolfMQTT* team about the bug, they expressed appreciation by sending me a complimentary gift. This gesture recognized my contribution to discovering the data race bug within their implementation and providing insights into the underlying issue. Furthermore, they addressed the bug immediately, and my name was acknowledged in the release notes, as shown in the pictures A.1 and A.2.

Figure A.1. A complimentary gift from *wolfSSL* team for finding the data race bug.



Fixes for multi-threading issues #209

Merged embhorn merged 1 commit into `wolfSSL:master` from `dgarske:mt_suback` on Jun 3, 2021

dgarske commented on Jun 2, 2021 — Contributor

1. The client lock is needed earlier to protect the "reset the packet state".
2. The subscribe ack was using an unprotected pointer to response code list. Now it makes a copy of those codes.
3. Add protection to multi-thread example "stop" variable.
   Thanks to Fatimah Aljaafari (@fatimahkj) for the report.
   ZD 12379 and PR Data race at function MqttClient_WaitType #198

`Fixes for three multi-thread issues:` … ✗ `78370ed`

dgarske requested a review from embhorn 2 years ago

dgarske assigned embhorn on Jun 2, 2021

embhorn approved these changes on Jun 3, 2021 — View reviewed changes

embhorn left a comment — Member

Excellent. I can confirm that the reported issue was fixed.

embhorn merged commit `ecc80b5` into `wolfSSL:master` on Jun 3, 2021 — View details
1 of 2 checks passed

Reviewers
lygstate
embhorn ✓

Assignees
embhorn

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

Notifications — Customize
Unsubscribe
You're receiving notifications because you were mentioned.

3 participants

Figure A.2. Wolfmqtt Github fixed issue.

Appendix B

Volunteering for the community

During my Ph.D., I participated in the implementation of an interactive game designed to illustrate the concept and potential risks of buffer overflow in C programs as a type of vulnerability. Our objective was to demonstrate how this vulnerability can be automatically detected using *ESBMC*. Furthermore, we investigated how such vulnerabilities can be exploited through specialized hardware capable of detecting specific vulnerabilities at the hardware level [178]. We presented this game to diverse groups of school students at multiple open exhibitions, including events such as ScienceX and British Science Week.