Combining BMC and Fuzzing Techniques for Finding Software Vulnerabilities in Concurrent Programs

FATIMAH K. ALJAAFARI1,2, RAFAEL MENEZES1,3, EDOARDO MANINO1, FEDOR SHMAROV1, MUSTAFA A. MUSTAFA1,4, (Member, IEEE), AND LUCAS C. CORDEIRO1,3

1Department of Computer Science, The University of Manchester, M13 9PL Manchester, U.K. 2Department of Computer Science, King Faisal University, Al Hofuf 31982, Saudi Arabia 3Postgraduate Program in Electrical Engineering (PPGEE), Federal University of Amazonas, Manaus 69077-000, Brazil 4Department of Electrical Engineering, imec-COSIC, KU Leuven, B-3001 Leuven, Belgium

Corresponding author: Lucas C. Cordeiro (lucas.cordeiro@manchester.ac.uk)

This work was supported in part by the Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/T026995/1, Grant EP/V000497/1, and Grant EU H2020 ELEGANT 957286; and in part by the Soteria Project Awarded by the U.K. Research and Innovation for the Digital Security by Design (DSbD) Programme. The work of Mustafa A. Mustafa was supported by the Dame Kathleen Ollerenshaw Fellowship of The University of Manchester.

ABSTRACT
Finding software vulnerabilities in concurrent programs is a challenging task due to the size of the state-space exploration, as the number of interleavings grows exponentially with the number of program threads and statements. 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 find software vulnerabilities in concurrent programs. Since there are no publicly-available GBF tools for concurrent code, we first propose OpenGBF – a new open-source concurrency-aware gray-box fuzzer that explores different thread schedules by instrumenting the code under test with random delays. Then, we build an ensemble of a BMC tool and OpenGBF in the following way. On the one hand, when the BMC tool in the ensemble returns a counterexample, we use it as a seed for OpenGBF, thus increasing the likelihood of executing paths guarded by complex mathematical expressions. On the other hand, we aggregate the outcomes of the BMC and GBF tools in the ensemble using a decision matrix, thus improving the accuracy of EBF. We evaluate EBF against state-of-the-art pure BMC tools and show that it can generate up to 14.9% more correct verification witnesses than the corresponding BMC tools alone. Furthermore, we demonstrate the efficacy of OpenGBF, by showing that it can find 24.2% of the vulnerabilities in our evaluation suite, while non-concurrency-aware GBF tools can only find 0.55%. Finally, thanks to our concurrency-aware OpenGBF, EBF detects a data race in the open-source wolfMqtt library and reproduces known bugs in several other real-world programs, which demonstrates its effectiveness in finding vulnerabilities in real-world software.

INDEX TERMS
Concurrency-aware gray-box fuzzer, bounded model checking, concurrent programs, instrumentation, LLVM pass.

I. INTRODUCTION
Concurrency is becoming prevalent in present-day software systems thanks to the performance benefits provided by multi-core hardware [1]. Examples of such software systems include online banking, auto-pilots, computer games and railway ticket reservation systems [2]. Ensuring the correctness and safety of such software is crucial since software failures may lead to significant financial losses and affect people’s well-being [3]. As an example, the OpenSSL library had a
Heartbleed vulnerability that allows a remote attacker to get access to sensitive information.

Despite the significant resources invested into software testing, much of existing software still features security vulnerabilities [4]. This is because the different possible threads’ interleavings cause the program execution to be non-deterministic, thus making the process of testing and verifying concurrent programs an inherently difficult task [5] (e.g., some bugs may occur only for a specific threads order, making them harder to detect). Furthermore, there exists a wide variety of unwanted concurrent behaviors. On the one hand, the non-determinism of the thread interleavings introduces concurrency bugs such as data races, deadlocks, thread leaks, and resource starvation [4], which may cause the program to produce abnormal results or unforeseen hangs. On the other hand, specific program inputs and thread interleavings may lead to memory corruption and security violations (e.g., access out of bounds) [6].

Due to this complexity, manual testing of concurrent software is not always adequate, and so automated verification and testing are often employed. In this respect, there is a myriad of different automated techniques such as control engineering [7], abstract interpretation [8], and data-flow analysis [9] for detecting bugs and vulnerabilities in concurrent programs [10], [11]. Among those, two methods have seen significant development in recent years: Bounded Model Checking (BMC) and fuzzing [12].

BMC [13] searches for violations in bounded executions (up to some given depth k) of the given program. If no property violation is detected, then k is increased until a bug is found, the verification problem becomes intractable, or a preset upper bound is reached. Although many industrial-grade bounded model checkers [14], [15], [16], [17], [18] have been successfully used for software verification, BMC has several fundamental drawbacks in general. Namely, BMC often experiences difficulties with achieving high path coverage (especially for multi-threaded programs) and reaching deep statements within the code because of state-space explosion and its dependency on Boolean Satisfiability (SAT) [19] or Satisfiability Modulo Theories (SMT) solvers [20].

Fuzzing [11] is an automated software testing technique that involves the repeated generation of inputs (based on some initial guess – seed value) to a Program Under Test (PUT). The PUT is then executed for each given sequence of input values; its behavior is checked for abnormalities, such as crashes or failures [21]. The main advantages of fuzzing include relative ease of integration with the existing testing frameworks, high scalability, and most importantly, exploring the deep execution paths is not as costly as in BMC. However, fuzzing often suffers from low branch coverage since the input generation is based on random mutations [22]. Typically this occurs when a program features conditional statements with complex conditions (e.g., input validation functions). As a result, providing a good initial seed for the fuzzing process is crucial. Moreover, fuzzing techniques face challenges detecting vulnerabilities in multi-threaded programs [23] since existing fuzzing techniques do not focus on thread interleavings that affect execution states.

Efforts toward developing a combined verification technique harnessing the strengths of both BMC and fuzzing have been made in the past. For example, Ognawala et al. [24] combine symbolic execution and fuzzing and apply it to general-purpose software. Alshmrany et al. [25] use BMC to guide a fuzzer in the analysis of sequential C programs. Chowdhury et al. [26] improve the seeding of gray-box fuzzing (GBF) by using BMC as a constraint solver to find execution paths through complex blocks of code. Nevertheless, given the current knowledge in software verification, there are no techniques that harness both BMC and fuzzing for verification of concurrent programs, and the question of whether combining BMC and fuzzing improves bug finding in concurrent programs remains open.

The challenge in answering this fundamental question is twofold. First, while there are many available BMC tools in the literature, all existing concurrency fuzzers are (at least partially) closed source. As a result, employing any of these concurrency fuzzers requires a major reproducibility effort. Second, combining BMC and fuzzing for concurrency is not straightforward. Given the lack of existing baselines, we take inspiration from portfolios [27], [28], the practice of running an ensemble of similar tools in parallel and picking the best result. At the same time, BMC and fuzzing are very dissimilar approaches, thus their cooperation inside the ensemble has to be carefully coordinated.

This paper addresses these challenges and makes the following original contributions:

1) We develop **OpenGBF** – a new open-source state-of-the-art concurrency-aware gray-box fuzzer [29]. Our main technique is instrumenting the PUT with random delays obtained from a random number generator whose seed value is controlled by the fuzzer. In this way, we can discover different thread interleavings and explore deep execution paths. Furthermore, our fuzzer is capable of generating crash reports containing the full program execution path.

2) We introduce **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. EBF incorporates a result decision matrix for coping with the potentially conflicting verdicts produced by the tools in the ensemble. Furthermore, EBF efficiently distributes the available computational resources between the tools to enhance its bug-finding capabilities.

3) We demonstrate that the combination of BMC and fuzzing improves verification outcomes compared to either technique applied separately. More specifically, **EBF** improves the bug-finding abilities of all

---

1. [https://heartbleed.com/](https://heartbleed.com/)
state-of-the-art concurrent BMC tools considered in this work by up to 14.9%. Similarly, EBF can find 24.2% of the vulnerabilities in our evaluation suite, whereas the state-of-the-art gray-box fuzzer AFL++ can only find 0.55%.

4) We apply EBF to the wolfMQTT open-source library that implements the MQTT messaging protocol, and we discover the presence of a data race bug. We reported the bug to the developers of the wolfMQTT library, who fixed it in June 2021. Also, EBF successfully reproduced known bugs in several real-world concurrent programs (i.e., pfscan [30], bzip2sm [31], and swarm 1.1 [32]). This demonstrates the real-world capabilities of EBF.

5) We report that the bug-finding capabilities of EBF are stable across a wide range of parameter values. In detail, we run a comparison experiment along three different axis: time allocation between the BMC tool and OpenGBF, maximum delay inserted by OpenGBF and maximum number of threads allowed by OpenGBF. Our results show a large sweet spot of parameter values that allows EBF to find nearly 50-fold more bugs than the worst setting.

The remainder of the paper is structured as follows: Section II contains the preliminaries on concurrent programs, bounded model checking and fuzzing, while Section III states the main research question of this work. Section IV discusses the main design choices and implementation details of OpenGBF, our state-of-the-art fuzzer for concurrent programs. Section V presents EBF, our ensemble verification technique. Section VI presents the experimental results, while Section VII lists the related work, and Section VIII draws the final conclusions.

II. PRELIMINARIES

A. COMMON SOFTWARE VULNERABILITIES
Concurrent programs feature multiple processes or threads simultaneously operating on shared computing resources [33]. As a result, such programs can feature vulnerabilities specific to sequential problems (e.g., invalid memory accesses, memory leaks [34]) as well as types of bugs that only occur in concurrent programs (e.g., data races, deadlocks, thread leaks [35]). Some software vulnerabilities are considered more dangerous than others. For example, writing out of bounds (a type of invalid memory access) is ranked number 1 in the top 25 MITRE ranking in 2022, while data races are in the 22nd place.

Invalid memory accesses comprise a vast family of memory safety violations. They include accessing memory outside the bounds of the intended buffer for either reading (potentially revealing some sensitive data to the attacker) or writing (causing memory corruptions or injections of executable code), accessing previously freed memory (aka “use-after-free”), and dereferencing of invalid pointers or NULL pointers (causing the program to crash or exit unexpectedly).

Data race is a condition when the program execution results in an undesired behavior due to a particular sequence and/or timing of the instructions executed by each thread. For instance, when a thread modifies the shared memory without acquiring a lock first, causing memory corruption when another thread tries to update the same memory location (see Figure 1a).

Deadlock occurs when the program is not in the terminal state and it cannot progress to any other state. For instance, when a thread does not release a lock after accessing the shared memory, therefore, denying memory access for any other thread (see Figure 1b).

Thread leak is a vulnerability specific to multi-threaded programs that happens when a terminated thread never joins the calling thread, thus never releasing the occupied resources (see Figure 1c).

Similarly, memory leaks are caused by repeated memory allocations which are never released during the program’s execution. This may lead to memory exhaustion resulting in the system hanging or crashing.

B. BOUNDED MODEL CHECKING
Bounded model checking is a verification technique that has been successfully applied to software and hardware verification over the past decades [13]. BMC works with the underlying program’s mathematical model (represented as a finite state transition system). It explores the model’s evolution up to some finite positive bound and determines whether the given safety property (e.g., absence of deadlocks, data races, buffer overflows, assertion violations, etc.) holds. In short,
BMC symbolically executes the given program up to the given bound \( k \) and encodes all the obtained traces \( C \) together with the given property \( P \) as a SAT/SMT [36] formula \( C \land \neg P \). A decision procedure (often referred to as automated theorem prover or solver) then checks the obtained formula and returns the satisfiability verdict. If the formula is satisfiable, then it means that the safety property is violated, and a witness (counter-example) is produced. Otherwise, a proof can be obtained that the program is safe up to the given bound \( k \).

Several drawbacks of BMC include state-space explosion as the verification depth grows, which becomes even more challenging for multi-threaded programs since it is required to explore the combined search space of thread interleavings and program states. Moreover, the verification of logical formulae consumes more CPU time and computer memory as the size of the formulae grows with the increasing verification depth. Finally, since BMC works with a symbolic abstraction (over-approximation) of the underlying program, it may report incorrect results when the devised model does not precisely represent the given program. For example, this can be caused by external libraries whose implementation in the language supported by the given BMC tool does not exist. Consequently, their behavior must be modelled (approximated) inside the BMC tool.

Thus, existing BMC tools like ESBMC [37], CBMC [38], and Cseq [39] differ mainly in their choices of program encoding and symbolic abstractions. We provide more details on their strategies to deal with concurrent programs in Section VII.

C. GRAY-BOX FUZZING

Fuzzing is an automated testing technique that discovers vulnerabilities by repeatedly executing a program with randomly-generated inputs [40]. Since most inputs generated this way are invalid, state-of-the-art fuzzers let users specify a small set of valid program inputs (the seeds) and employ a mutation-based strategy to generate new ones. Gray-box fuzzing improves on this idea by guiding the mutation process with program-specific metrics. To do so, the program under test must be instrumented with some additional code that tracks the required metric (e.g., code coverage) during execution.

Algorithm 1 [23], [41] shows the standard workflow of a gray-box fuzzer. It takes a target PUT and initial seeds \( M \) as inputs. Then, it instruments the PUT (line 1) by inserting some additional code that allows the fuzzer to collect code coverage statistics in the PUT. At every iteration of the main fuzzing loop (line 4), it selects a seed \( t \) (line 5) and chooses a random number \( N \) of mutations (line 6). Then, the fuzzer repeatedly executes the instrumented program \( P_f \) (line 9) with different mutated seed \( t' \) (line 8) as input and obtains 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). Otherwise, 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 may help the fuzzer discover more vulnerabilities in the subsequent iterations. Finally, the execution of the main fuzzing loop continues until the predefined timeout is reached.

Multiple attempts have been made to detect security vulnerabilities in concurrent programs with fuzzing [23], [34], [42], [43], [44]. Here, we organize these past efforts according to five categories in the taxonomy of Table 1. The first three categories concern the usability of each fuzzer: whether they apply to user programs or operating system code (Scope), which type of bugs they are able to detect (Vulnerabilities), and whether their code is easily accessible (Open Source). In this regard, none of the existing state-of-the-art fuzzers satisfy our research requirements. That is, there is no fully-open-source fuzzer that can detect multiple concurrency vulnerabilities in user programs. We address this gap by introducing our own concurrency-aware fuzzer in Section IV.

The last two categories concern the fuzzing techniques themselves. Specifically, the general fuzzing strategy in Algorithm 1 requires some adaptations to produce good results on concurrent programs. First and foremost, a mechanism to force the execution of a large number of different interleaving is required (Interleaving Control). Existing fuzzers like MUZZ [23] and ConAFL [34] manipulate the thread priorities at assembly level, others like Krace [44] inject sleep instruction to force a context switch, while AutoInter-fuzzing [42] and Conzer [45] instrument the code with explicit synchronization barriers or thread locks. Alternatively, the interleaving exploration can be left to the natural non-determinism of the operating system like in ConFuzz [43]. Lastly, some authors propose to change the feedback to the input mutation engine in an attempt to guide the fuzzer towards more interesting interleavings.
We mark such attempts as Thread-Aware as opposed to the default Branch Coverage metrics used in sequential fuzzing. We provide more information on these state-of-the-art fuzzers in Section VII-B.

III. PROBLEM STATEMENT

In general, BMC and GBF tackle the problem of finding vulnerabilities in fundamentally different ways. Consequently, it is natural to ask whether combining the two techniques can lead to better coverage of the search space. More precisely, in this study, we ask the following research question:

**Research Question.** Does an ensemble of bounded model checkers and gray-box fuzzers discover more concurrency vulnerabilities and 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 in the remainder of our paper:

- **Concurrency-aware gray-box fuzzer.** As detailed in Section II-C, there are some recent existing efforts to fuzz concurrent programs, but no mature open-source tool exists. Consequently, designing such a tool is an important step towards answering our research question. In doing so, we aim to draw from the lessons learned in the literature and implement OpenGBF, a tool that is representative of state-of-the-art concurrency-aware GBF. We do so in Section IV.

- **Aggregating BMC and GBF results.** By creating an ensemble of different tools, we run into the risk of them returning conflicting results. The main reason is that BMC relies on abstractions of the program execution states and symbolic execution (see Section II-B), whereas GBF tests concrete inputs and execution schedules. When the two approaches disagree, we have an opportunity to make an informed choice about the verification outcome. We propose to do so via a decision matrix, as detailed in Section V.

- **Resource allocation trade-off.** The main drawback of using an ensemble of different tools is that they all compete for the same computational resources. We must choose how many resources to allocate to each tool for applications with limited time, memory, or computational power. In general, our decisions depend not only on the problem at hand but also on the partial results we obtain from the tools in the ensemble. We discuss strategies to optimize our ensembles in Section V.

IV. DESIGNING A STATE-OF-THE-ART CONCURRENCY GRAY-BOX FUZZER

This section describes the main design challenges we address in implementing our concurrency-aware gray-box fuzzer OpenGBF. Namely, we discuss how we control the thread interleavings (Section IV-B) and how we generate witness information when a violation is found (Section IV-C). Both of these goals require the instrumentation of the PUT as detailed in Sections IV-A and IV-D.

Note that our GBF is based on established techniques: fuzzer-controlled delay injection to force interleaving exploration and branch coverage to guide the fuzzer mutation engine (see Table 1). At the same time, we believe that our design is worth reporting for two reasons. On the one hand, our GBF is the only user-space concurrency fuzzer that is currently available as fully open-source software; thus the present section is a useful reference for future users. On the other hand, our GBF is a transparent effort to reproduce the claims of the existing literature, which are currently impossible to confirm given the lack of open-source codebases.

A. CUSTOM LLVM PASS INSTRUMENTATION

We build our concurrency-aware fuzzer on top of the widely used gray-box fuzzer AFL++ [46], which is designed to find vulnerabilities in sequential programs. AFL++ minimizes the fuzzing overhead by instrumenting the PUT via an LLVM pass [47]. The LLVM pass is an essential framework of the LLVM compiler. It works with the program translated into the LLVM intermediate representation (IR) language and adds additional code to monitor the program behavior [48].

We combine the standard LLVM pass of AFL++ with our custom independent LLVM pass to make our fuzzer aware of concurrent execution (see Algorithm 2). More specifically, we inject five different function calls: a delay
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_alloc();

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 run-time}
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 run-time}
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 run-time}
9: else if $I$ is DECLARATION then
10: $M \leftarrow$ instrument ($\lambda_e$, $I$, $M$) {insert a call to _EBF_alloc() 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 ($\lambda_l$, $I$, $M$) {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 $M$

function (see line 4), two thread-monitoring functions (see lines 6 and 8) and two information-collecting functions (see lines 10 and 12). The first function controls the interleaving schedule, and we explain its implementation details in Section IV-B. The second and the third functions monitor the number of active threads (see Section IV-C) in the PUT during run-time by tracking when the functions _pthread_create and _pthread_join are called. The last two functions record the information required to generate a witness file containing the execution trace. We present a full example of instrumented code in Section IV-D.

We bundle these five instrumentation functions in a runtime library. We compile and link both the runtime library and the instrumented PUT using the AFL++ clang wrapper. The resulting executable can be fuzzed to detect reachability and memory corruption bugs in the default setting. Optionally, the _ThreadSanitizer_ flag can be enabled for finding concurrency bugs.

B. CONTROLLING THE THREAD INTERLEAVING

As previously mentioned in Section I, our main algorithmic idea is to introduce random delays in the PUT to force context switches between threads. However, there are several major corner cases that _OpenGFB_ needs to take care of.

First, if the program features many active threads, we need to limit their number during the PUT execution. Limiting the number of threads is an unfortunate but necessary approximation of the PUT run-time behavior. Increasing the number of active threads slows down the PUT execution and consumes more compute resources during fuzzing. Furthermore, the PUT may attempt to create an “infinite” number of threads, which can either be an undefined behavior or just undecidable to solve. We limit the number of threads by assuming that interleavings that create more threads than a pre-defined threshold are safe and start a new run with different interleavings. We discuss the effect of different threshold values on the bug-finding capabilities of our fuzzer in Section VI-B5.

Secondly, deadlocks in the PUT may cause the current interleaving to be stuck during execution. To avoid this problem, we force the fuzzer to terminate non-deterministically by introducing a probability $p$ of exiting at every instruction.

Thirdly, in _EBF_ we provide a mechanism for defining atomic blocks (via _EBF_atomic_begin and _EBF_atomic_end functions) within the PUT. They can be used to ensure that all instructions inside these blocks are executed atomically.

To this end, our delay function will force all other threads to wait until the atomic block has finished. We achieve this by initializing a global mutex (i.e., _EBF_mutex_) which the active thread can lock. If the global mutex is locked and the current interleaving does not own the global mutex, then we wait for the mutex owner to finish its execution. Additionally, _EBF_ will not insert delays inside the atomic blocks (thus, improving the performance of the instrumented program) since no thread interleavings can take place within these blocks.

Finally, we force different interleavings by changing the amount of delay (in milliseconds) inserted after each instruction. The delay values are drawn uniformly at random from a pre-set range. More specifically, we let _AFL++_ produce a seed value for the random number generator providing the delay values. We explore the impact of different delay ranges on the bug-finding ability of our fuzzer in Section VI-B5.

The above design choices have been incorporated into the implementation of function _delay_function_, whose definition is illustrated in Algorithm 3. In lines 2-4 we implement our strategy for limiting the number of active threads. If the number of active threads $T_N$ is greater than the given threshold $T_T$ or we extract 1 from a Bernoulli distribution with success probability $p$, then the fuzzer exits this analysis normally (see line 3) and starts a new run with different inter-
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: φ ← wait_for_timeout {wait until EBF_mutex is released}
10: if φ is timeout then
11: exit {exit this analysis normally}
12: end if
13: sleep(*) {run a delay for * nanoseconds}
14: EndFunction

Algorithm 4 Function pthread_add()

Global: Mutex_lock, T_N - active threads counter.

1: Function pthread_add()
2: lock thread ← Mutex_lock
3: T_N + +
4: unlock thread ← Mutex_lock
5: EndFunction

leavings (i.e., different delay values). Otherwise, we check whether the current thread owns the global mutex (line 5), and if so, we let it finish its execution and release the mutex (lines 6 and 7). If the global mutex is not released before the timeout (line 9), we also allow the fuzzer to exit this analysis normally (line 11). This is done to prevent deadlocks if the global mutex is never released. Finally, the delay is executed by running a sleep function for the duration value produced by the fuzzing engine (line 13).

Additionally, the number of active threads in the PUT is monitored by the functions pthread_add and pthread_release, whose definitions are shown in Algorithms 4 and 5, respectively. The former increments (decrements) the active threads counter T_N (see line 3) atomically by locking the current thread (see line 2) before changing the value of T_N and unlocking it afterwards (see line 4).

C. WITNESS GENERATION

If OpenGBF finds a violation, we need to support the users and tools in reproducing the identified bug. To do so, we generate a crash report file with all the necessary information to reproduce the property violation. We use functions EBF_alloca and EBF_add_store_pointer to record all the information needed for automated witness generation:

Algorithm 5 Function pthread_release()

Global: Mutex_lock, T_N - active thread counter.

1: Function pthread_release()
2: lock thread ← Mutex_lock
3: T_N - -
4: unlock thread ← Mutex_lock
5: EndFunction

Algorithm 6 Function EBF_alloca()

Inputs: a – variable name, f – function name, &a – variable address.

Global: Mutex_lock, witnessInfoAFL pid – witness file for the process with ID = pid.

1: Function EBF_alloca(a, f, &a)
2: lock thread ← Mutex_lock
3: witnessInfoAFL pid ← write(a, f, &a)
4: unlock thread ← Mutex_lock
5: EndFunction

Algorithm 7 Function EBF_add_store_pointer()

Inputs: &a - variable address, l - line number in the code, f - function name, 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, v)
2: lock thread ← Mutex_lock
3: witnessInfoAFL pid ← write(&a, l, f, v)
4: unlock thread ← Mutex_lock
5: EndFunction

assumption values, thread ID, variable names, and function names as shown in Algorithms 6 and 7.

As the fuzzing process begins, we run an initialization function before the main method is called in the PUT. This function creates a witness file uniquely identified by the current process ID (i.e., witnessInfoAFL pid) and sets the environment (i.e., initializes the global mutex EBF_mutex, getting process id (pid)). Then, our custom LLVM pass inserts a function call to EBF_alloca (see line 10 in Algorithm 2) after each declaration instruction in the PUT, and a function call to EBF_pointer_add_store_pointer (see line 12 in Algorithm 2) after each loading store instruction.

Algorithms 6 and 7 demonstrate the definitions of functions EBF_alloca and EBF_add_store_pointer, respectively. The former records the declared variable’s name, its address, and the name of the function where it has been declared in the PUT (see line 3). The latter records the assigned variable’s address, the assigned value, the name of the function and the line of code where the assignment takes place in the PUT (see line 3). Both functions record information atomically – the thread is locked (see line 2) before the writing occurs and unlocked afterwards (see line 4). If the fuzzing run finishes
normally (i.e., a timeout is reached or the process finishes with the exit code 0), we delete the created witness files in

a destructor function [49]. If the fuzzer causes a crash in

one of the PUT executions, we save the ID of the process

that has crashed and generate a crash report by extracting

the data from the witness file associated with this process

ID. The resulting crash report contains the exact sequence of

operations (i.e., memory accesses) that led to the PUT’s crash

(see Appendix B for more details).

D. FULL ILLUSTRATIVE EXAMPLE

To tie all these design choices together, we present an illustrative example. Assume that we have a concurrent PUT that has

one reachability bug, as illustrated in Listing 1. The program

contains two threads calling the same function foo (see line

3), which contains a loop of 5 iterations. At the end of the

execution, the value of $a$ should be 10: Line 18 consists of a

conditional statement that checks whether this is not the case,

and reports an error (property violation). This error can only

be reached when the reads and writes over $a$ are not correctly

synchronized between the two threads.

Figure 2a illustrates an interleaving that causes a violation:

thread 1 (T1) reads the variable $a = 0$ which is initialized to

0 in line 2, then thread two (T2) reads $a = 0$ before T1 writes

$a = 1$ (see line 7). This pattern repeats until the end, when

the value of $a$ will be 5 rather than the expected 10.

Ideally, each thread will read the content of variable $a$

and increment it without interference from the other thread.

Figure 2b, illustrates an interleaving scenario where the two

threads are synchronized: thread 1 ($T1$) reads $a = 0$ and

writes the new value $a = 1$, then thread two ($T2$) reads the

updated value $a = 1$ and increments it to $a = 2$. This pattern

yields a final value of $a = 10$, which makes the property hold.

When we verify the code in Listing 1 with EBF, we find

two different bugs. Namely, OpenGBF reports the data race,

whereas all BMC tools we tested report a reachability bug at

Line 18.

Let us now present an example of our instrumentation.

Recall that we instrument the PUT at the LLVM-IR level.

For our example of Listing 1, we report the LLVM-IR encod-

ing for function foo in Listing 2.Listing 3 illustrates the

IR after the instrumentation 1 define.

1 define dso_local i8* @foo(i8* %0) #0 {
2 %2 = alloc %a, %align 8
3 %3 = alloc %a, %align 4
4 %4 = alloc %a, %align 4
5 store i8* %4, i8* %2, %align 8
6 store i32 1, i32 4, %align 4
7 br label %5
8 %6 = load %a, i32 4, %align 4
9 %7 = icmp eq %a, %0, 0
10 br label %6, label %8, label %14
11 %9 = load %a, i32 4, %align 4
12 store %a, %4, %3, %align 4
13 %10 = load %a, i32 4, %align 4
14 %11 = add %a, %10 %10, 1
15 store %a, %4, %4, %align 4
16 br label %5
17 ret i8* null
18 }

LISTING 2. Fragment of the corresponding IR before instrumentation.

1 define dso_local i8* @foo(i8* %0) #0 {
2 %2 = alloc %a, %align 8
3 %3 = alloc %a, %align 4
4 %4 = alloc %a, %align 4
5 %5 = bitcast i8* %0 to i1
6 %6 = bitcast i8* %2 to %a
7 call void @EBF_add_store_pointer(i8* %6, i16 → 0, %a getelementptr inbounds ([4 x i16], [4 x i16], %0, %a), i16 %b)
8 store i8* %b, i8* %2, %align 8
9 %7 = bitcast i8* %b to %a
10 call void @EBF_alloc(i8* @getelementptr inbounds ([4 x i16], [4 x i16], %a, %a), i16 %b)
11 call void @_delay_function()
12 %8 = bitcast i32 %b to %a
13 call void @EBF_alloc(i8* @getelementptr inbounds ([4 x i16], [4 x i16], %a, %a), i16 %b)
14 call void @_delay_function()
15 %9 = bitcast i32 %a to %a
16 call void @EBF_alloc(i8* @getelementptr inbounds ([4 x i16], [4 x i16], %a, %a), i16 %b)
17 call void @_delay_function()
18 %10 = sec %a %a to %a
19 call void @EBF_add_store_pointer(i8* %b, i16 %b → 0, %a getelementptr inbounds ([4 x i16], [4 x i16], %0, %a))
20 store %a, i8* %4, %align 4,
21 call void @_delay_function()
In lines 10, 13 and 16, we call a function called EBF_alloc, which stores the metadata of any variable declared in the PUT. This information is also used to generate the witness file. In lines 11, 14, 17 and 22, we inserting a function call to the _delay_function(), as we describe in Algorithm 3.

V. EBF: ENSEMBLES OF BMC AND FUZZING

Thanks to our OpenGBF, we now have access to both state-of-the-art BMC and GBF tools. This section explains how we combine them in EBF and maximize their effectiveness in finding vulnerabilities in concurrent software.

We remark that types of software vulnerabilities that can be detected by EBF solely depend on the capabilities of each individual tool used in the ensemble. For example, most BMC tools can detect most types of illegal memory accesses (e.g., buffer overflows, use-after-free, invalid pointer dereference) and memory leaks, as well as some BMC tools, can detect concurrency bugs (i.e., thread leaks, data races, and deadlocks). Regarding OpenGBF, its main function is exploring different executions of the program by sampling different thread schedules and different program inputs. In order to evaluate whether each such execution leads to a bug, OpenGBF relies on the bug-detecting capabilities of sanitizers. They perform some additional instrumentation to the PUT, making it crash when a vulnerability has been detected. Namely, AddressSanitizer [50] is capable of identifying memory-related vulnerabilities, while ThreadSanitizer [51] can identify concurrency bugs.

We present a high-level overview of the structure of our ensembles in Figure 3. Overall, the ensemble executes both BMC and GBF tools on the PUT. The execution of these two tools is not fully independent like it would be in a portfolio [27], [28]. In fact, the result of the BMC tool run can be used to seed the GBF tool under specific conditions. We elaborate on this in Section V-A.

Furthermore, our ensemble structure in Figure 3 requires addressing the two main challenges outlined in Section III.

First, the results of the BMC and GBF runs must be aggregated in a coherent assessment of the safety of the PUT. Second, the BMC and GBF tools in the ensemble compete for the same computational resources, which might reduce the ability of each tool to find violations. We present our solution to these challenges in Sections V-B and V-C.

A. SEEDING

Before discussing our solution to the two problems of aggregation and resource sharing, let us propose a further optimization of the ensemble. Specifically, if sequential execution of the ensemble is possible, we can improve the GBF seeds by initializing them with the counterexample produced by BMC. This is only possible when the BMC tool reports a failed verification outcome. These seeds are concrete values that cause an assertion to fail. It is important to mention that despite the BMC tools providing thread scheduling information in their bug reports, it is not always straightforward which delay values must be applied for replicating these bug-inducing schedules. In general, it is impossible to guarantee a particular thread order by only injecting time delays and without an explicit scheduling algorithm or another locking technique. This is because the effect of the introduced delays on the thread order depends on the implementation of multi-threading in the corresponding operating system and its current workload (e.g., sometimes, the same delay values

| T1  | R(a) a=0 | W(a) a=1 | R(a) a=1 | W(a) a=2 | R(a) a=2 | W(a) a=3 | R(a) a=3 | W(a) a=4 | R(a) a=4 | W(a) a=5 |
|-----|---------|----------|---------|----------|---------|----------|---------|----------|---------|----------|
| T2  | R(a) a=0 | W(a) a=1 | R(a) a=1 | W(a) a=2 | R(a) a=2 | W(a) a=3 | R(a) a=3 | W(a) a=4 | R(a) a=4 | W(a) a=5 |

(a) Mis-synchronized memory accesses.

| T1  | R(a) a=0 | W(a) a=1 | R(a) a=2 | W(a) a=3 | R(a) a=4 | W(a) a=5 | R(a) a=6 | W(a) a=7 | R(a) a=8 | W(a) a=9 |
|-----|---------|----------|---------|----------|---------|----------|---------|----------|---------|----------|
| T2  | R(n) a=1 | W(n) a=2 | R(n) a=3 | W(n) a=4 | R(n) a=5 | W(n) a=6 | R(n) a=7 | W(n) a=8 | R(n) a=9 | W(n) a=10 |

(b) Synchronized memory accesses.

FIGURE 3. Visualization of the memory accesses on variable a in Listing 1 for two different interleavings. In Figure 2a, the accesses are not synchronized: both T1 and T2 read a before simultaneously incrementing it to a = 1. This pattern continues until the end, where the final value will be a = 5. Conversely, Figure 2b depicts synchronized accesses: T1 reads a and increments it to a = 1, then T2 reads a and increments it to a = 2. In the end, the final value will be a = 10.
may lead to the execution of different thread schedules). As a result, OpenGBF uses only the bug-inducing inputs and not the thread schedule information as the seed. On the one hand, this means that OpenGBF cannot reproduce every bug produced by the BMC tool since it might not be able to sample the sequence of delays replicating the bug-inducing thread schedule. On the other hand, this allows OpenGBF to explore other randomly generated schedules that may lead to other bugs.

However, if the BMC engine timeouts, proves (partial) correctness or produces Unknown, then we generate the fuzzer seed with pseudo-random integer numbers ranging between 0 to 5000. We determined experimentally that this range provides a good trade-off between functionality and efficiency since larger numbers (e.g., more than 5000) lead the fuzzer to generate a lot of inputs that do not result in triggering a bug. Note that these values are not directly used inside the delay function. The delays are produced by a different random number generator that is seeded from one of the inputs given by the fuzzer.

### B. AGGREGATION

After running all ensemble members, we need to aggregate their outcomes. This is especially challenging since BMC and GBF may disagree on the safety of the PUT (cf. Section III). We summarize our aggregation rule in the decision matrix in Table 2. Some decisions are straightforward: we must trust the other when either method cannot conclude. Accordingly, when GBF reports Unknown, our decision matrix aligns with the outcome of BMC. Vice versa, when BMC cannot prove or disprove the PUT’s safety, we trust the bugs found by GBF. A more interesting scenario happens when there is a conflict in the ensemble: BMC can declare a PUT as Safe, but GBF may still be able to find a Bug. In general, we report such instances as Conflict. This can be caused by the over-approximations in the computational models used by the BMC tool or by the bugs from the code instrumentations introduced by the GBF tool. Each such Conflict may be resolved by analyzing the witness file produced by the GBF.

We remark that the decision matrix proposed in Table 2 was motivated by the SV-COMP [52] competition rules where interactive verification is not available, while incorrect answers are punished by deducting competition points (see Section VI-B3 for more details). However, verification of more complex software systems can benefit from a more descriptive decision matrix. For example, it may be useful to distinguish between different Unsafe outcomes in Table 2.

### C. CPU TIME ALLOCATION

CPU time allocation is another important design choice in optimizing the performance of EBF. More specifically, we need to split the available CPU time between the two components of the ensemble in order to increase the search space coverage as much as possible for each of them.

| EBF | GBF |
|-----|-----|
| Safe | Conflict |
| Bug | Safe |
| Unknown | Unsafe |

In Section VI-B5, we discuss how different CPU time distribution strategies affect the overall EBF performance.

### VI. EXPERIMENTAL EVALUATION

In this section, we demonstrate the effectiveness of BMC and GBF ensembles in a diverse set of scenarios. We will reiterate our experimental objectives before detailing the deployed benchmarks and our results.

#### A. OBJECTIVES

The present experimental evaluation has the following goals:

**EG1** - Detection of violations in concurrent programs

Demonstrate that EBF can detect more violations in concurrent programs than state-of-the-art BMC tools on their own.

**EG2** - Real-world performance of OpenGBF

Demonstrate that the concurrency-aware GBF we implement in EBF can find violations in real-world programs.

**EG3** - Parameter trade-offs in our concurrency-aware fuzzer

Demonstrate that EBF produces consistent results across a wide range of parameter settings.

Note that the latter two objectives EG2 and EG3 are oriented towards demonstrating that OpenGBF (see Section IV) is representative of state-of-the-art gray-box fuzzing techniques.

#### B. RESULTS

We gathered our experimental results over a substantial period, beginning in February 2021. During this period, the design of EBF has evolved and improved. To avoid confusion, we report our results separately for each version of EBF. Namely, we start with the participation of EBF 2.3 in the Concurrency Safety category of SV-COMP 2022 (see Section VI-B1). This EBF version was based on CBMC v5.43 and a more rudimentary implementation of our concurrency-aware fuzzer. For comparison, we also report the performance of our latest version EBF 4.0 on the same set of benchmarks (see Section VI-B2). EBF 4.0 includes the full implementation of OpenGBF described in Section IV, and a large number of different BMC tools. Then, we demonstrate the ability of our fuzzer to find a data race in the wolfMQTT cryptographic library (see Section VI-B3). Historically, we first found this bug in February 2021 with an earlier version of our fuzzer. Here, we repeat our previous
experiment with the latest version of *OpenGBF* included in *EBF* 4.0. Finally, we run an extensive comparison of the performance of *EBF* 4.0 across a wide range of parameter settings (see Section VI-B5).

1) *EBF* 2.3 PARTICIPATION IN SV-COMP 2022

*EBF* 2.3 took part in SV-COMP 2022 in the *Concurrency Safety* category [53]. This category features a set of 763 concurrent C programs, 398 of which are *safe*. The bugs in the remaining 365 programs are formulated in terms of reachability conditions: the program is deemed *unsafe* if a predefined error function is reachable within the given program, and *safe* otherwise. These programs contain a number of intrinsic functions [52]. We explain how we model them in Appendix A.

In the SV-COMP 2022 *Concurrency Safety* category, each participating tool is asked to produce one of the following six verification outcomes for a given concurrent benchmark (see the first column in Table 3):

- **Correct True.** The tool correctly confirms that the program is safe.
- **Correct False.** The tool correctly confirms the presence of a bug.
- **Correct False Unconfirmed.** The tool correctly confirms the presence of a bug, but the associated counterexample cannot be reproduced by the *witness validator* tool developed by the competition organizers.
- **Incorrect True.** The tool confirms that a program is safe when it contains a bug.
- **Incorrect False.** The tool confirms that the program contains a bug when it is, in fact, safe.
- **Unknown.** The tool cannot conclude within the given CPU time and memory limit.

Every verification outcome is assigned a score value (see the fourth column in Table 3), which strongly discourages incorrect results. The resulting score for each tool is comprised of the sum of the scores obtained for all benchmarks.

The competition took place on the SV-COMP servers featuring 8 CPUs (Intel Xeon E3-1230 v5 @ 3.40 GHz) and 33 GB of RAM. Each benchmark verification task was limited to 15 minutes of CPU time and 15 GB of RAM.

The version of our tool that we submitted to the competition, *EBF* 2.3, is based on *CBMC* v5.43 as a BMC engine and an earlier implementation of *OpenGBF*. Namely, we selected *CBMC* as it is a state-of-the-art BMC tool that has consistently been achieving high rankings in the concurrency category of SV-COMP over the past decade. Also, the implementation of *OpenGBF* we used in *EBF* 2.3 was more rudimentary. Namely, it had no limit on the number of threads, no probability of terminating early, and no mechanism to avoid injecting delays inside atomic blocks.

*EBF* 2.3 reached 7th place out of 20 participants in SV-COMP 2022, by scoring a total of 496 points. Crucially, *EBF* 2.3 outperformed *CBMC* 5.43, which finished 10th with 460 points. We report the official SV-COMP 2022 results of these two tools in Table 3. Note that *CBMC* achieved a higher score than *EBF* in predicting programs safety (148 vs 139, respectively). This is an expected outcome, since *EBF* dedicates only 6 minutes out of 15 minutes to BMC, and the rest are used by *OpenGBF*, which cannot prove whether a program is safe. At the same time, *EBF* was better than *CBMC* at detecting bugs that could be confirmed by the *witness validator* (234 vs 212), thus scoring extra points.

Moreover, *EBF* reported only one *Incorrect False* outcome, while *CBMC* produced 3 incorrect verdicts resulting in 48 penalty points. Interestingly, *EBF* avoided reproducing the latter three incorrect outcomes (returning *Unknown* instead) since *CBMC* did not have enough time to wrongly detect these bugs running as a part of the ensemble (reporting *Unknown* as the result), while *OpenGBF* also could not find any bugs in these benchmarks within the remaining time (hence, another set of *Unknown’s*). In contrast, the only incorrect outcome (different from the three false positives obtained by *CBMC*) produced by *EBF* was caused by a bug inside *OpenGBF*, which made it generate a spurious counterexample. This issue has been resolved in our most recent *EBF* 4.0 version.

Overall, *EBF* 2.3 improved the competition result of *CBMC* 5.43 by ~ 7.8%. In addition, *EBF* could detect and confirm a property violation in one benchmark, which could not be detected by any other dynamic tool in the competition. These results are a first positive answer to goal *EG1*; we present further experimental evidence in Section VI-B2.

2) *EBF* 4.0 WITH DIFFERENT STATE-OF-THE-ART BMC TOOLS

After the participation of *EBF* in SV-COMP 2022, we improved *OpenGBF* following the algorithmic ideas we describe in Section IV. Here, we present the results of further experiments that test whether any BMC tool can be improved by adding our latest version of *OpenGBF* on top of it (goal *EG1*). To avoid confusion, we refer to the latest implementation of our ensemble technique as *EBF* 4.0.

In our evaluation, we run *EBF* 4.0 over the same benchmarks from the SV-COMP 2022 *Concurrency Safety* category (see Section VI-B1). However, we omit the SV-COMP aggregate scoring system (see Table 3), since its different weights can obfuscate the advantages of each verification technique. Instead, we focus on analyzing the trade-off between proving

| Verification outcome | Tool | Score per benchmark |
|----------------------|------|---------------------|
| Correct True         | *EBF* 2.3 | 119 * CBMC | 148 | × 2 |
| Correct False        | *EBF* 2.3 | 234 * CBMC | 212 | × 1 |
| Correct False Unconfirmed | *EBF* 2.3 | 55 * CBMC | 90 | × 0 |
| Incorrect True       | *EBF* 2.3 | 0 * CBMC | 0 | × 32 |
| Incorrect False      | *EBF* 2.3 | 1 * CBMC | 3 | × 16 |
| Unknown              | *EBF* 2.3 | 334 * CBMC | 310 | × 0 |

*Overall SV-COMP 2022 score: 496*
safety\(^4\) (BMC only) and bug-finding abilities (both BMC and GBF) from the raw results.

Furthermore, we consider three additional BMC tools in our experiments (see Table 4), rather than just CBMC [38]. Namely, ESBMC [37] is a powerful BMC tool that has been successfully participating in SV-COMP over the past decade. Similarly, Deagle [54] and Cseq [39] achieved 1st and 2nd place, respectively, in the Concurrency Safety category at SV-COMP 2022.

We conduct all our experiments on a virtual machine running Ubuntu 20.04 LTS with 160 GB RAM and 25 CPU cores of Intel Core Processor (Broadwell, IBRS) @ 2.1 GHz. Moreover, we run EBF 4.0 with the following parameters: maximum thread threshold 5, delay range from 0 [\(\mu s\)] to \(10^5 [\mu s]\). Additionally, we distribute the available runtime in the following way: we allocate 6 minutes to the BMC engine, 5 minutes to OpenGBF, and 4 minutes for the seeding, aggregation, and witness file generation. These parameter settings are optimal for the SV-COMP 2022 benchmark we are using, as we discuss in Section VI-B5. Note that the user can specify the time distribution between the tools in the ensemble in EBF via command-line arguments.

Table 4 reports a pair-wise comparison between EBF 4.0 and the four different BMC tools on their own. The results demonstrate that EBF finds more bugs than all four BMC engines on their own while reducing the number of Unknown instances. More in detail, EBF achieves the best improvement concerning ESBMC, by finding \(\sim 14.9\)% more bugs and correcting one wrong outcome while reducing the number of safety proofs by only \(\sim 7.6\)%.

Similarly, the ability to double-check any counterexample produced by BMC allows EBF to correct all three erroneous outcomes produced by CBMC while showing a marginal difference between the improvement in bug-finding (\(\sim 5.6\)% and the degradation in safety proof (\(\sim 5\)%). In contrast, when compared to Deagle, EBF shows no decrease in the Correct True outcomes, but can increase the number of discovered bugs by \(\sim 5.3\)%.

For Cseq, the number of safety proofs produced by EBF declines by only \(\sim 2.9\)%, while the number of Correct False results rises by \(\sim 6.3\)%.

Overall, EBF provides a better trade-off between bug-finding and safety proving than each BMC engine. On average, EBF finds over 8% more concurrency bugs while reducing the number of programs declared safe by only 3.8%. Hence, this evaluation achieves our first experimental goal (EG1).

3) DETECTING A DATA RACE IN wolfMQTT

We evaluate EBF 4.0 on the wolfMQTT library [55]. MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol developed for constrained environments like the Internet of Things (IoT). It employs the publish-subscribe messaging pattern of publishing messages and subscribing to topics. The wolfMQTT library is a client implementation of the MQTT protocol written in C for embedded devices. We use its API to verify the concurrent part of the protocol implementation.

OpenGBF detects a data race\(^5\) in wolfMQTT after running for 15 minutes and consuming 24 MB of RAM. In detail, MQTT contains 4 packet structures (i.e., Connect, Publish, Subscribe and Unsubscribe). The Subscribe function accepts an acknowledgment from the server (i.e., broker). This acknowledgment was received through an unprotected pointer due to the data race detected in function MqttClient_WaitType, which can lead to an information leak or data corruption. This issue has been successfully replicated and consequently fixed\(^6\) by the wolfMQTT developers.

Our setup for the experiment is the following. We run EBF 4.0 on an Intel Core i7 2.7Ghz machine with 8 GB of RAM running Ubuntu 18.04.5 LTS. We use a Mosquito server for the communication with the wolfMQTT client [56]. We enable ThreadSanitizer on top of OpenGBF for detecting the concurrency bugs that are not formulated explicitly in terms of reaching a predefined error function (i.e., like it is done in the SV-COMP 2022 concurrency benchmarks) or violating a safety assertion. Finally, we run our fuzzer with a thread threshold of \(T_h = 5\) and a delay range from 0 [\(\mu s\)] to \(10^5 [\mu s]\).

Other tools fail to discover the same vulnerability. More specifically, both bounded model checkers CBMC v5.43 and ESBMC v6.8 are unable to detect the data race within the given time limit. Moreover, the BMC tool Deagle v1.3 cannot parse the program correctly, since it is using an outdated version of C parser. Similarly, Cseq v3.0 does not support programs featuring multiple source files. Finally, neither the fuzzer AFL nor AFL++ can detect this bug in wolfMQTT.

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

4) DETECTING MEMORY VIOLATIONS IN REAL WORLD CONCURRENT PROGRAMS

To show scalability and robustness of EBF, we evaluate it on several real-world concurrent programs using the same machine as in Section VI-B3. We consider three multi-threaded real-world programs: pfscan [30], a multi-threaded file scanner; bzip2smp [31], a parallel implementation of bzip2 compressor; swarms.1 [32], a library that provides a framework for parallel programming on multi-core systems. Table 5 presents the number of lines of code (LOC) for each PUT, the number (\(N_N\)) and types of vulnerabilities detected by EBF, and the median time EBF takes to find these bugs. We give more detailed information on the latter in Section VI-B6.

Both tools in EBF detect a NULL pointer deference in pfscan that is caused by a malloc instruction whose result

\(^4\)The term “prove safety” means that the BMC procedure could verify all reachable states and could not find an execution path that violates the safety property.

\(^5\)https://github.com/wolfSSL/wolfMQTT/issues/198

\(^6\)https://github.com/wolfSSL/wolfMQTT/pull/209
is not checked for successful memory allocation leading to a crash due to writing to a NULL pointer. As for bzip2smp, EBF finds two bugs. The BMC engine detects a vulnerability in the BZ2_bzclose() function, which receives a pointer that can be NULL. Meanwhile, OpenGBF finds a memory leak in the writerThread() function of bzip2smp. Regarding swarm 1.1, EBF (in particular, the fuzzer) finds an invalid pointer dereference caused by an incorrect thread initialization (i.e., calling the pthread_create function with a NULL pointer as an argument).

5) OPTIMIZING EBF’s SETTINGS
In the following experiments, we explore different settings for EBF and OpenGBF. For the first two experiments, we run EBF with the BMC engine switched off, allowing the fuzzer to run for 11 minutes. While for the third evaluation, we run EBF with both engines enabled but with a different amount of time allotted (out of total 11 minutes) to each of them.

a: MAXIMUM NUMBER OF THREADS IN OpenGBF
Figure 4 shows the result of choosing different values for the thread threshold on the number of bugs (i.e., the number of Correct False outcomes) discovered by OpenGBF. We ran this experiment with the delay range from 0 [µs] to $10^5$ [µs] and probability of exiting $p = 0.01\%$. It can be seen that the most optimal value lies in the region around $Th = 5$, and raising the threshold value leads to fewer bugs being detected due to the increase in the number of computer resources required to maintain a more significant number of active threads. Perhaps, we can suggest that many bugs can be discovered without considering a large number of threads, which was also demonstrated by the wolfMQTT data race that was discovered 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.

b: MAXIMUM AMOUNT OF DELAY IN OpenGBF
Figure 5 illustrates the effect of the amount of delay we insert to force scheduling in OpenGBF. We use a logarithmic scale to compare different delay ranges in OpenGBF. Similar to the evaluations of different thread thresholds, we use the number of Correct False to assess the efficacy of a given delay bound. We set the thread threshold to 5 active threads in this experiment. We change the upper bound of the delay’s range from $0$ [µs] (i.e., no delay) to $10^7$ [µs] (i.e., 10 seconds). The results show that increasing the delay upper bound from 0 to $10^5$ [µs] gradually improves the bug-finding capabilities of OpenGBF from 68 to 88 benchmarks. When we set a large 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 threads interleavings that our GBF explores. At the same time, choosing a larger upper bound (e.g., $10^6$ or $10^7$) leads to a decrease in the number of bugs found due to a higher number of timeouts. This is expected, as 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 delay range is benchmark-dependent.

c: COMPARISON WITH THE “NON-INSTRUMENTING” GBF
As a sanity check, we compare our GBF implementation against the “non-instrumenting” version of the fuzzer, which does not feature the PUT instrumentation stage described in Algorithm 2. We run this experiment with the optimal set of parameters reported above: i.e., $Th = 5$, $p = 0.01\%$ and the random delay upper bound of $10^5$ [µs]. The results show a nearly 50-fold increase in the number of detected bugs. More specifically, OpenGBF detects 88 out of 365 vulnerabilities (i.e., 24.2% of the total), while the “non-instrumenting” GBF

| Verification outcome | Tool | EBF | Deagle | EBF | Czeq | 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    | 1   | 0     | 3   |      |
| Unknown              |      | 187 | 204    | 258 | 273  | 390 | 424   | 304 | 311  |

FIGURE 4. Number of bugs (i.e., Correct False outcomes) discovered by OpenGBF in EBF 4.0 for different values of threshold on the maximum number of active threads.

VOLUME 10, 2022
121377
TABLE 5. Evaluation of EBF on real-world concurrent programs. For each program we present its size in terms of the number of lines of code (LOC), the number of vulnerabilities detected by EBF (N_N), types of corresponding vulnerabilities (N_T), the median time (in seconds) of 20 EBF re-runs, and which EBF engine (i.e., ESBMC or OpenGBF) detects the corresponding vulnerability.

| 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, Memory leak | 10.6s       | ✔     | ✔       |
| swarm 1.1           | 2.8k| 1   | Invalid pointer dereference | 339.6s      | ✔     | ✔       |

FIGURE 5. Number of bugs (i.e., Correct False outcomes) discovered by OpenGBF in EBF 4.0 for different upper bounds of the distributions for the random delay.

detects only 2 out of 365 (0.55%). This highlights the necessity of using concurrency-aware fuzzers in our EBF.

d: CPU TIME ALLOCATION INSIDE EBF
In this experiment we compare different ways of distributing the total verification time (11 minutes overall) between the fuzzer and the BMC engines in EBF. The results demonstrate a relatively wide range of values (between 3 and 8 minutes per engine) within which EBF 4.0 produces identical results detecting 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 drops by ~5% to 303 out of 365. Conversely, the overall bug-finding performance of EBF 4.0 decreases drastically by over 72.5% when all 11 minutes are devoted to OpenGBF. On the whole, this result confirms that BMC tools are better than our GBF tool when used in isolation. However, combining them both in an ensemble is going to yield better results across a very different time allocation choices.

e: EARLY THREAD TERMINATION IN OpenGBF
Recall that our GBF fuzzer terminates the execution of each thread with probability p (see Section IV-B). This implementation detail is crucial for avoiding potential deadlocks in the PUT. Figure 7 shows the impact of different values of p on the bug-finding performance of our GBF on the SV-COMP 2022 concurrency benchmark suite. For comparison, we implemented an alternative mechanism, which deterministically terminates the execution of each thread after n instructions. Note that both termination mechanisms are local to each thread; thus, they do not introduce any synchronization overhead. Furthermore, we align the plots according to each thread’s average number of instructions, which is the mean 1/p of an exponential distribution.

The results in Figure 7 show that the performance of our fuzzer is stable across a wide range of values of p. Interestingly, removing the termination mechanism altogether causes only minimal degradation in the fuzzer performance. Moreover, there is no significant difference between the probabilistic and deterministic termination mechanisms as the average number of instructions per thread increases. However, the performance of the probabilistic mechanism degrades more slowly as the average number of instructions per thread increases. We speculate that the probabilistic termination mechanism allows our GBF to explore a large number of shallow paths and a few deeper ones, thus slightly increasing the chance of finding a bug for a low average number of instructions. Finally, we select the best parameter setting p = 0.01% for the rest of our experiments.

6) ANALYZING THE NON-DETERMINISM OF OUR FUZZER
Fuzzers are fundamentally non-deterministic programs. As such, the performance of EBF may vary across different
FIGURE 7. Number of bugs (i.e., Correct False outcomes) discovered by OpenGBF in EBF 4.0 for different early thread termination strategies.

We show the effects of non-determinism by re-running our GBF 20 times on the benchmarks of the present experimental section.

a: NON-DETERMINISM ON SV-COMP 2022 SUITE
We run our GBF 20 times with the same optimal settings described in VI-B5. In the worst case, our fuzzer finds only 82 bugs, whereas in the best case, it finds 89. Given that there are 365 bugs in the SV-COMP 2022 suite, we expect the distribution to be approximately Gaussian, with an empirical mean 85.2 and standard deviation 2.0. Given that the variance in the total number of bugs is small, we can trust the results of Figures 4, 5, 6 and 7 to give us robust values for the optimal EBF setting.

Furthermore, we inspect the impact of fuzzer non-determinism on each individual file in the SV-COMP 2022 benchmark suite. Specifically, there are 74 files for which our fuzzer always finds a bug across the 20 independent runs. Among those, we select the ones with the smallest, median, and largest variance. We plot the performance of our GBF in these three representative cases in Figure 8. Note that the violin shows the extremes of the distributions, together with their median and kernel density estimation. Since these distributions are highly non-Gaussian, we omit the mean.

b: NON-DETERMINISM ON wolfMQTT AND REAL-WORLD PROGRAMS
We re-run our GBF 20 times on the wolfMQTT library and the three real-world programs listed in Table 5. The results are shown in the violin plot of Figure 8. In the case of pfscan and bzip2smp, our GBF is able to find bugs almost instantly (see also Table 5). In contrast, we can observe more variance on wolfMQTT and swarm 1.1. In the former case, the distribution is fairly compact in its support [10.6s, 66.8s]. In the latter case, the distribution has a long tail. More specifically, the median time is 9.5s, 75% of the runs find a bug in less than 40s, but there are also occasional outliers where the first bug is reported after 300s.

C. LIMITATIONS
We identify several possible limitations to our current work.

1) INCOMPLETENESS OF FUZZING FOR PROVING SAFETY
Fuzzing works by executing the program along many concrete paths, hoping to find the one that leads to vulnerability detection. Consequently, it cannot formally guarantee that we can exhaustively explore the entire state-space of the program. As a result, by design, EBF prioritizes bug-finding over proving a program’s safety.

2) SOURCES OF INCORRECT VERDICTS IN EBF
Although EBF does not produce conflicting verdicts using the aggregation matrix from Table 2, the correctness of EBF’s verification verdicts largely depends on the implementation of the tools used in the assembly. For example, if the BMC engine produces a wrong Safe outcome while the GBF cannot find any violations within the given time limit (thus returning Unknown) the final verification verdict becomes Safe. Similarly, our GBF may become a source of an incorrect Bug verdict when BMC reports Unknown and the GBF crashes because of an internal bug within the GBF’s implementation rather than an actual vulnerability inside the PUT. Fortunately, this is not critical since EBF generates a witness file that can be further evaluated using witness validators (see Appendix B).

3) CHOICE OF PARAMETER SETTINGS IN EBF
Although EBF does not produce conflicting verdicts using the aggregation matrix from Table 2, the correctness of EBF’s verification verdicts largely depends on the implementation of the tools used in the assembly. For example, if the BMC engine produces a wrong Safe outcome while the GBF cannot find any violations within the given time limit (thus returning Unknown) the final verification verdict becomes Safe. Similarly, our GBF may become a source of an incorrect Bug verdict when BMC reports Unknown and the GBF crashes because of an internal bug within the GBF’s implementation rather than an actual vulnerability inside the PUT. Fortunately, this is not critical since EBF generates a witness file that can be further evaluated using witness validators (see Appendix B).

VOLUME 10, 2022
benchmarks will follow similar patterns to the ones shown in Figures 4 and 5.

VII. RELATED WORK
Throughout our paper, we describe various existing studies that cover relevant tools and techniques. In this section, we collate and expand on these references. Our goal here is to clarify the context in which our research occurs.

A. BOUNDED MODEL CHECKING (BMC)
Bounded model checking has been successfully applied to the verification of concurrent C programs over the past years [37]. There exist several state-of-the-art bounded model checkers, such as ESBMC [14] and CBMC [15] that can handle both sequential and multi-threaded C programs and detect concurrency bugs (e.g., data races, deadlocks, etc.) and other vulnerabilities (e.g., buffer overflows, dangling pointers, etc.). In particular, ESBMC handles concurrency by performing a depth-first search through all possible thread interleavings, up to the given maximum number of context switches [58]. In contrast, CBMC encodes each concurrent execution unit separately and joins them with partial order formulae [59].

Many other BMC tools demonstrate their efficacy in the verification of concurrent programs at the annual SV-COMP software verification competition [60]. For example, Lazy-CSeq [18], [61] - one of the leaders in the concurrency category at SV-COMP over the past decade - works by translating a multi-threaded C program into a non-deterministic sequential program that considers all round-robin schedules up to a given number of rounds. Then the obtained sequential program is verified using a bounded model checker for sequential programs (e.g., CBMC, ESBMC). Similarly, Daegle - the winner in the concurrency category in SV-COMP 2022 [62] - introduces a novel ordering consistency theory for multi-threaded programs [17], and implements a more efficient solver for this theory on top of CBMC (front-end) and MathSAT [63] (back-end).

B. FUZZING
Traditional techniques for fuzzing sequential programs do not translate well for concurrent programs since they let the fuzzer control only the input of the program and not the scheduling of its threads [23]. Existing proposals towards concurrency-aware fuzzing attempt to rectify this issue (see Table 1 for an overview).

ConAFL [34] is a gray-box fuzzer that specializes on user-space multi-threaded programs. ConAFL employs static analysis to locate sensitive concurrent operations to determine the execution order, focusing on three types of invalid memory access vulnerabilities: buffer-overflow, double-free, or use-after-free. The thread interleavings are controlled indirectly by changing the execution priority of each thread at assembly level. As an alternative, the authors mention the possibility of injecting sleep commands at the code level, but they do not test it. Finally, the authors rely on the default mutation feedback of the sequential fuzzer AFL [64], which is based on branch coverage. Due to its heavy thread-aware static and dynamic analysis, ConAFL cannot scale to large programs. Furthermore, the authors’ static analysis tool is not publicly available [65].

Similarly, the gray-box fuzzer MUZZ [23] employs static analysis to identify blocks of code that have a higher chance of triggering a concurrency vulnerability. When the code is instrumented, such blocks receive heavier instrumentation which helps the fuzzer dynamically track the execution of different schedules. To encourage the exploration of a large number of interleavings, MUZZ manipulates the execution order by assigning random priorities to the threads at assembly level. Despite the promising experimental results, MUZZ is not yet publicly available.

A simpler approach is implemented in the tool ConFuzz [43], which lets the natural non-determinism of the operating system guide the exploitation of different interleavings by random chance. To compensate for that, ConFuzz modifies the standard branch coverage feedback of the mutation engine by measuring how far each block of code is from a thread-related instruction. Seeds that execute blocks of code closer to such instructions have a higher chance of survival at each mutation. Unfortunately, the ConFuzz tool [43] is not publicly available.

Recently, another concurrency-aware gray-box fuzzer has been proposed in [42]. This tool, called AutoInter-fuzzing, uses static analysis to identify instruction pairs that access the same memory location but are executed by different threads. Then, the program code is instrumented with synchronization barriers that control the order of execution of the instruction in each pair. Every time one such pair is encountered during regular fuzzing, the program is re-run, forcing the opposite execution order of the pair. Unfortunately, this strategy for exploring interleavings makes AutoInter-fuzzing suffer from low path coverage compared to other fuzzers. In line with most of the fuzzers listed in the present section, AutoInter-fuzzing is not publicly available.

Conzzer [45] improves upon the ideas of AutoInter-fuzzing. More specifically, the instruction pairs are obtained at runtime and contain information about the execution trace. The authors argue that the fuzzer can be used to explore different interleavings for a critical region by being context-aware. They also implemented their own mutation algorithm, resulting in the fuzzer being able to explore more interleaving than AutoInter-fuzzing.

On a different note, Krace [44] is a fuzzer for kernel file systems that specializes in finding data races. We mention it here because it also employs the interleaving control strategy of injecting delays in the program code. Furthermore, it augments the standard branch coverage metrics by explicitly tracking the order of execution of any pair of instructions that access the same memory location. This feedback induces the mutation engine to explore a larger number of thread interleavings. The source code of [44] is available but cannot
be used in our research as it targets data races in the kernel space.

*OpenGBF* (see Section IV) implements many of these ideas, including instrumenting the code with `sleep` instructions, forcing the exploration of random interleavings and letting the fuzzer control the randomness through its mutation engine. In the future, if the aforementioned concurrency-aware fuzzers become open source [29], it will be possible to test their efficacy when paired with BMC tools, as we do here with our GBF tool.

### C. Hybrid Techniques

Recently, several efforts have combined fuzzing with various forms of symbolic execution and static analysis [66]. The rationale behind these efforts is that fuzzing alone struggles to find “deep” bugs and vulnerabilities because the random mutations introduced in the input have a low probability of hitting complex paths in the program. In contrast, if the fuzzer is given a set of input seeds that are already close to the correct target, the evolutionary algorithm has a higher chance of exposing the bugs and vulnerabilities.

To this end, Ognawala et al. [24] propose to increase the coverage of fuzzing by augmenting the set of input seeds with a round of concolic execution. With it, the code coverage rises significantly. There are other examples of tools employing concolic execution, such as *Driller* [67] and *QSYM* [68]. Similarly, Chowdhury et al. [26] are concerned with the inability of off-the-shelf fuzzers to discover inputs that pass complex blocks of program logic. Their solution is using a bounded model checker to solve the corresponding reachability problem and produce concrete input seeds that satisfy the complex conditions of the program under analysis. The fuzzer is then free to explore the search space beyond that. On a different note, Alshmrany et al. [25] employs a selective fuzzer if the model checker of their *FuSeBMC* tool fails to find all vulnerabilities. Such fuzzers use the statistics collected by the model checker to create a particular set of input seeds.

*EBF* is similar to these hybrid tools in the sense that it exploits the combined advantages of fuzzing and model checking. However, 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 ensembles are more flexible and allow virtually any existing tool to be combined together. Finally, none of the existing hybrid approaches can verify concurrent programs.

### D. Other Techniques

Other techniques for finding vulnerabilities in concurrent programs have been proposed. Wen et al. [69] propose a controlled concurrency testing technique called *Period*, which uses a periodical execution to model the execution of concurrent programs. They feed the periodical executor with a key point slice of the target program and apply an analyzer to collect feedback on runtime information. In contrast, *Peahen* [70] is an approach to combine context-sensitive and context-insensitive static techniques, namely context reduction. This context reduction consists of filtering vulnerabilities found by a context-insensitive technique with a path feasibility check. Afterward, a context-sensitive approach is used to validate the vulnerability. Finally, *QL* [71] is a tool that employs reinforcement learning to guide the exploration of interleavings. This tool uses an explicit scheduler.

On a different note, there are a few methods that improve on classic verification techniques. For example, in dynamic analysis, some works focus on improving soundness and completeness [72], [73], while other works focus on creating a new value flow analysis for interprocedural data flow that detects concurrency issues [74]. At the same time, there are techniques that employ a different flavor of Model Checking, specifically stateless model checking (SMC) [75]. The method was born from the intuition that caching states in Model Checking was not as effective as a stateless approach. For example, *RCMC* [76] and *GenMC* [77] rely on having a code interpreter that is able to compute a reachability graph over the program, and use system calls during the analysis to provide more accurate results.

### VIII. Conclusion

Discovering vulnerabilities in concurrent programs remains a challenging problem due to the extreme explosion of the search space in the number of possible interleavings. In this paper we focus on two existing approaches to this problem: Bounded Model Checking (BMC) and Gray-Box Fuzzing (GBF). When used on their own, each approach can only find a subset of the vulnerabilities present in state-of-the-art concurrent benchmarks. Our contribution is building ensembles comprising both BMC and GBF tools, thus exploiting the complementary advantages of these two approaches. We call such ensembles *EBF*.

A major hindrance to the use of *EBF* ensembles is the current lack of mature open-source GBF tools that support concurrent testing. For this reason, we first propose our own implementation of state-of-the-art concurrency-aware fuzzing techniques, and make *OpenGBF* publicly available. Then, we combine it with a large variety of state-of-the-art BMC tools, and show that the *EBF* ensembles so created can find up to 14.9% more concurrency vulnerabilities than the BMC tools on their own. Furthermore, thanks to *OpenGBF*, we are able to discover a data race vulnerability in the open-source *wolfMqtt* library.

Overall, we demonstrate that *EBF* is an effective technique for finding vulnerabilities in concurrent programs. Still, the capability of each ensemble is directly related to the complementary qualities of its BMC and GBF building blocks. As a consequence, we believe that improving and specializing each of the two ensemble components is the most promising direction for future works. More in detail, we need faster BMC tools that rely on rougher approximations of the program under test, in order to produce a larger number of meaningful counterexamples that the GBF tool can exploit as seeds.
APPENDIX A
HARNESSING FUNCTION

Evaluating the SV-COMP 2022 benchmarks [52] requires specific functions that must be supported by every tool participating in the competition. As a result, we model some functions for non-determinism and synchronization. The non-determinism is used to get the value of the input from the fuzzer. The synchronization is implemented using a set of functions that guarantee atomicity (i.e., to ensure no thread interleavings during a block of instructions). In order to make AFL++ understand the SV-COMP specific semantics, we implement these functions as a run-time specific semantics, we implement these functions as a run-time

function coverage with Munch: A hybrid fuzzing and directed symbolic execution approach,'' in Proc. 17th Int. Symp. Princ. Pract. Declarative Program. Symp., vol. 29, no. 2, pp. 63–69, Jan. 2018.

[7] T. Kelly, Y. Wang, S. Lafontaine, and S. Mahlke, “Eliminating concurrency bugs with control engineering,” Computer, vol. 42, no. 12, pp. 52–60, Dec. 2009.

[8] Q. Steivenart, J. Nicolay, W. De Meuter, and C. De Roover, “Detecting concurrency bugs in higher-order programs through abstract interpretation,” in Proc. 17th Int. Symp. Princ. Pract. Declarative Program., Jul. 2015, pp. 232–243.

[9] M. B. Dwyer and L. A. Clarke, “Data flow analysis for verifying properties of concurrent programs,” ACM SIGSOFT Softw. Eng. Notes, vol. 19, no. 5, pp. 62–75, Dec. 1994.

[10] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” Commun. ACM, vol. 56, no. 2, pp. 82–90, 2013.

[11] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, “V-Fuzz: Vulnerability-oriented evolutionary fuzzing,” 2019, arXiv:1901.01142.

[12] M. Aizatulin, A. D. Gordon, and J. Jürjens, “Extracting and verifying cryptographic models from C program code by symbolic execution,” 2011, arXiv:1107.1017.

[13] A. Biere, “Bounded model checking,” in Handbook Satisfiability, IOS Press, 2009, pp. 457–481.

[14] M. R. Gadelia, R. S. Menezes, and L. C. Cordeiro, “ESBMCI: 6.1: Automated test case generation using bounded model checking,” Int. J. Softw. Tools Technol. Transf., vol. 23, pp. 1–5, Dec. 2020.

[15] D. Kroening and M. Tautscheg, “CMB—C bounded model checker,” in Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2014, pp. 389–391.

[16] D. Beyer and M. E. Keremoglu, “CPACHEcker: A tool for configurable software verification,” in Computer Aided Verification, G. Gopalakrishnan and S. Qadeer, Eds. Springer, 2011, pp. 184–190.

[17] F. He, Z. Sun, and H. Fan, “Satisfiability modulo ordering consistency theory for multi-threaded program verification,” in Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement., Jun. 2021, pp. 1264–1279.

[18] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlati, “Lazy-CSeq: A lazy sequentialization tool for C,” in Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2018, pp. 394–401.

[19] M. Vanhoef and F. Plessens, “Symbolic execution of security protocol implementations: Handling cryptographic primitives,” in Proc. WOOT, 2018, pp. 1–11.

[20] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, “Chopped symbolic execution,” in Proc. 40th Int. Conf. Softw. Eng., May 2018, pp. 350–360.

[21] P. Tsankov, M. T. Dashki, and D. Basin, “SECFUZZ: Fuzz-testing security protocols,” in Proc. 7th Int. Workshop Autom. Softw. Test (AST), Jun. 2012, pp. 1–7.

[22] B. S. Pak, “Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution,” School Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, 2012.

[23] H. Chen, S. Guo, Y. Xue, Y. Su, C. Zhang, Y. Li, H. Wang, and Y. Liu, “MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs,” in Proc. 29th USENIX Secur. Symp. (USENIX Secur.), 2020, pp. 2325–2342.

[24] S. Ognwala, T. Hutzellmann, E. Psalidila, and A. Pretschner, “Improving function coverage with Munch: A hybrid fuzzing and directed symbolic execution approach,” in Proc. 53rd Annua ACM Symp. Appl. Comput., Apr. 2018, pp. 1475–1482.

[25] K. Ashmarmany, R. S. Menezes, M. R. Gadelha, and L. C. Cordeiro, “fuzeBMC: A white-box fuzzer for finding security vulnerabilities in C programs,” in Fundamental Approaches to Software Engineering. Nature Publishing Group, 2020.

[26] A. B. Chowdhury, R. K. Medicherla, and R. Venkatesh, “VeriFuzz: Program aware fuzzing,” in Tools and Algorithms for the Construction and Analysis of Systems. Cham, Switzerland: Springer, 2019, pp. 244–249.

[27] 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, C. Bessière, Ed. Berlin, Germany: Springer, 2007, pp. 712–727.

[28] 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, Switzerland: Springer, 2022, pp. 49–70.

[29] (2022). EBF Repository. [Online]. Available: https://github.com/fatimahjk/EBF

[30] (2022). pfscan. [Online]. Available: https://github.com/manpages/focal/man1/pfscan.1.html

[31] (2022). bzip2/2mp. [Online]. Available: http://bzip2/mp.sourceforge.net/.

[32] D. A. Bader, V. Kanade, and K. Madduri, “SWARM: A parallel program- ming framework for multicore processors,” in Proc. IEEE Int. Parallel Distrib. Process. Symp., Mar. 2007, pp. 1–8.

[33] P. Pereira, H. Albuquerque, H. Marques, I. Silva, C. Carvalho, L. Cordeiro, V. Santos, and R. Ferreira, “Verifying CUDA programs using SMPS-based context-bounded model checking,” in Proc. 31st Annua ACM Symp. Appl. Comput., Apr. 2016, pp. 1648–1653.

APPENDIX B
COUNTER EXAMPLE EXTRACTION

EBF needs to convert the crash reports discussed in Section IV-C into GraphML-based format to allow automatic witness checkers to validate the produced witness by tracking the execution path leading to the reported bug [78]. This feature of EBF is utilized in two cases: 1) when OpenGDB reports a bug, and/or 2) when the BMC engine produces a counterexample.

ACKNOWLEDGMENT

Fatimah K. Aljaafari acknowledges the scholarship she is receiving from King Faisal University (KFU).

REFERENCES

[1] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, “Parallelism via multithreaded and multi-core CPUs,” Computer, vol. 43, no. 3, pp. 24–32, Mar. 2010.

[2] Vinod. (2022). Multithreading Realtime Examples. [Online]. Available: https://androidmaniaco.wordpress.com/2016/12/16/multithreading-realtime-examples/

[3] L. C. Cordeiro, E. B. De Lima Filho, and I. V. De Bessa, “Survey on automated symbolic verification and its application for synthesising cyber-physical systems,” IET Cyber-Phys. Syst., Theory Appl., vol. 5, no. 1, pp. 1–24, 2020.

[4] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” in Proc. ASPLOS, 2008, pp. 329–339.

[5] P. Pereira, H. Albuquerque, I. Da Silva, H. Marques, F. Monteiro, R. Ferreira, and L. Cordeiro, “SMT-based context-bounded model checking for CUDA programs,” Concurreny Computat., Pract. Exp., vol. 29, no. 22, p. e3934, Nov. 2017.

[6] F. R. Monteiro, E. H. Da S. Alves, I. S. Silva, H. I. Ismail, L. C. Cordeiro, and E. B. De Lima Filho, “ESBMCI-GPU a context-bounded model checking tool to verify CUDA programs,” Sci. Comput. Program., vol. 152, pp. 63–69, Jan. 2018.

[7] A. Deshmeh, K. Macnaughton, and B. Esbaugh, “Detecting concurrency bugs in higher-order programs through abstract interpretation,” in Proc. 17th Int. Symp. Princ. Pract. Declarative Program., Jul. 2015, pp. 232–243.

[8] Q. Steivenart, J. Nicolay, W. De Meuter, and C. De Roover, “Detecting concurrency bugs in higher-order programs through abstract interpretation,” in Proc. 17th Int. Symp. Princ. Pract. Declarative Program., Jul. 2015, pp. 232–243.
EDOARDO MANINO is a Research Associate with the Department of Computer Science at the University of Manchester. He is part of the Systems and Software Security Group and focuses on automated verification of neural networks. His background is in Bayesian machine learning, a topic he got awarded a Ph.D. from the University of Southampton in 2020. His other research interests include network science to algorithmic game theory and reinforcement learning.

FEDOR SHMAROV received the Ph.D. degree in computer science from Newcastle University, in 2018. He is currently a Research Associate with the Systems and Software Security (S3) Group, Department of Computer Science, The University of Manchester. His research interests include formal methods and model checking for verification of software and cyber-physical systems.

MUSTAFA A. MUSTAFA (Member, IEEE) received the B.Sc. and M.Sc. degrees in communications from the Technical University of Varna, Bulgaria, and Newcastle University, U.K., respectively, and the Ph.D. degree in computer science from The University of Manchester, U.K. He is currently a Senior Lecturer (Associate Professor) and a DKO Fellow with the Department of Computer Science, The University of Manchester, and an Affiliated Researcher with imec-COSIC, KU Leuven. He also leads the Trusted Digital Systems Cluster at the Centre for Digital Trust and Society, The University of Manchester. His research interests include information security, user privacy, and applied cryptography.

LUCAS C. CORDEIRO is currently a Reader with the Department of Computer Science, The University of Manchester (UoM), U.K., where he leads the Systems and Software Security (S3) Research Group. He is also the Arm Centre of Excellence Director with UoM. In addition, he is affiliated with the Trusted Digital Systems Cluster at the Centre for Digital Trust and Society, the Formal Methods Group, UoM, and the Post-Graduate Programs in Electrical Engineering (PPGEE) and Informatics (PPGI) at the Federal University of Amazonas, Brazil. Before joining the University of Manchester, he worked as a Postdoctoral Researcher at the University of Oxford and a Research Engineer at Diffblue. In addition, he worked for five years as a Software Engineer at Siemens/BenQ Mobile and CTPIM/NXP Semiconductors. He has coauthored more than 140 peer-reviewed publications in the most prestigious venues. His research interests include software model checking, automated testing, program synthesis, software security, embedded, and cyber-physical systems. He has received various international awards, including the Distinguished Paper Award at ACM ICSE’11 and 32 awards from the international competitions on software verification (SV-COMP) and testing (Test-Comp) 2012–2022. He has a proven track record of securing research funding from EPSRC, Intel, Motorola, Samsung, Nokia Institute of Technology, CNPq, FAPEAM, British Council, and Royal Society.

* * *