EBF 4.2: Black-Box Cooperative Verification for Concurrent Programs
(Competition Contribution)

Fatimah Aljaafari[1,2], Fedor Shmarov[1], Edoardo Manino[1], Rafael Menezes[1], and Lucas C. Cordeiro[1]

[1] The University of Manchester, UK
[2] King Faisal University, SA

Abstract. Combining different verification and testing techniques together could, at least in theory, achieve better results than each individual one on its own. The challenge in doing so is how to take advantage of the strengths of each technique while compensating for their weaknesses. *EBF* 4.2 addresses this challenge for concurrency vulnerabilities by creating Ensembles of Bounded model checkers and gray-box Fuzzers. In contrast with portfolios, which simply run all possible techniques in parallel, *EBF* strives to obtain closer cooperation between them. This goal is achieved in a black-box fashion. On the one hand, the model checkers are forced to provide seeds to the fuzzers by injecting additional vulnerabilities in the program under test. On the other hand, off-the-shelf fuzzers are forced to explore different interleavings by adding lightweight instrumentation and systematically re-seeding them.

1 Overview

Finding vulnerabilities in concurrent programs presents the combined challenge of exploring the search space of program inputs and execution schedules, or *interleavings*. Recently, there have been attempts at solving complex verification problems by combining different techniques into hybrid verification tools [1,2,3]. More generally, these attempts belong to a larger trend in automated software analysis called *cooperative verification* [4,5]. In this paradigm, the main idea is implementing some form of communication interface between different tools (i.e., a common information exchange format), which allows the exchange of partial results (artifacts). In this way, we can harness the strengths of multiple verification techniques and solve more complex problems [6,7,8].

In *EBF* [9], we are the first to implement a cooperative approach that combines Bounded Model Checking (BMC) and concurrency-aware Gray-Box Fuzzing (GBF) for finding vulnerabilities in concurrent C programs. In order to simplify the communication interface between the cooperating tools, we adopt a *black-box* design philosophy where verification artifacts are implicitly shared via appropriate transformation and instrumentation of the program under test (PUT). The advantage of this design philosophy is its universality: in fact, *EBF* can incorporate any BMC or GBF tool that takes a C program as input.

More specifically, *EBF 4.2* expands the cooperative verification capabilities of previous versions of *EBF*. First, we introduce a new seed generation module for the GBF.

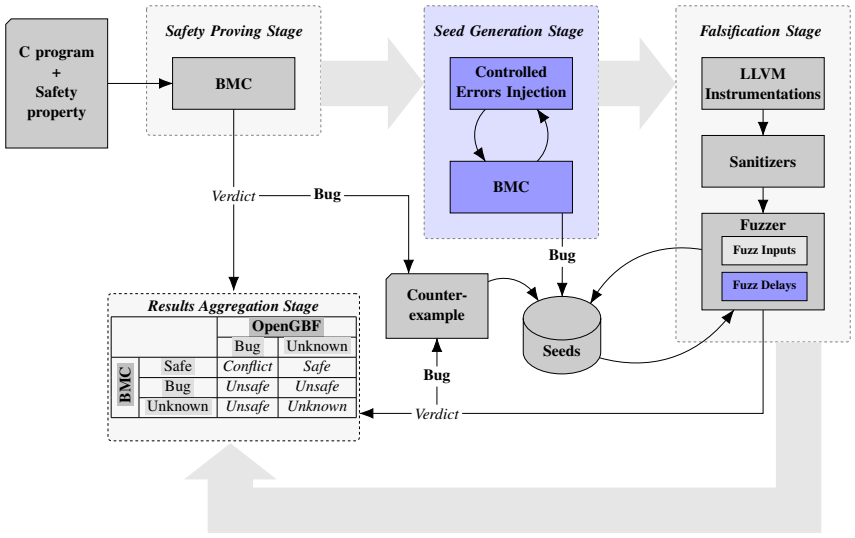

Fig. 1: The workflow of *EBF 4.2* comprises four stages (dashed rectangles). The safety proving and seed generation stages use a BMC tool. The falsification stage uses our *OpenGBF* tool. The result aggregation stage generates a verification verdict and counter-example (if any). Areas of improvement over *EBF 4.0* [9] are shown in blue.

This module works by injecting additional vulnerabilities in critical areas of the PUT, and then using a BMC engine to generate program inputs that trigger them. These inputs represent higher quality seeds for the fuzzer than randomly-generated ones. Second, we propose an improved light-weight instrumentation based on the Clang/LLVM toolchain that turns any compatible off-the-shelf GBF into a concurrency-aware fuzzer. We do so by injecting fuzzer-controlled delays in the PUT, which implicitly force the exploration of different interleavings.

2 Architecture

Figure 1 illustrates the workflow of *EBF*, which comprises four verification stages: safety proving, seed generation, falsification and results aggregation. Each of these stages take a concurrent C program and a given safety property as input.

Safety Proving Stage. During this stage, *EBF* calls the BMC engine with the given inputs. The BMC tool produces one of the three possible *verdicts*: *Safe* if the model checker deems the PUT safe with respect to the given property, *Bug* if a vulnerability is detected, or *Unknown* encompassing a variety of different outcomes including reaching a timeout, running out of memory, or crashing unexpectedly. If the BMC tool finds a bug, it generates a counter-example – a sequence of program inputs and a thread schedule leading to the vulnerability. The input values are stored for later use as a seed.

Seed Generation Stage. This is a new feature of *EBF 4.2*, which harnesses the strength of BMC in resolving complex path conditions. For instance, the branch `if(x*x -2*x +1 == 0)` may be extremely difficult for the fuzzer to explore. *EBF* tackles this issue by repeatedly injecting the error statement `assert(0)` in each conditional branch of the PUT (similar to the approach in [2]). Then, each transformed program (which contains one unique error statement) is independently verified with the BMC tool. If the BMC reaches the error within a timeout, *EBF* converts the resulting counter-example into a fuzzing seed. The seed generation process continues until all injected errors have been detected or the stage timeout has been reached. The seeds we collect during this stage greatly improve the fuzzer performance in the next stage.

Falsification Stage. During this stage, *EBF* checks whether the PUT contains any vulnerabilities by fuzzing its inputs and thread interleavings. Due to the current lack of open-source GBF tools for concurrent programs [9], *EBF* uses our own concurrency-aware gray-box fuzzer *OpenGBF*. Its implementation extends *AFL++*, a state-of-the-art GBF for single-threaded programs, by introducing the following concurrency-aware lightweight instrumentation in the PUT.

First, *OpenGBF* injects delays after each instruction at the *LLVM* intermediate representation level. The value of these delays (typically several micro-seconds) is controlled by the fuzzer and implicitly forces the execution of different thread interleavings. Second, *OpenGBF* inserts functions for recording all the information needed for witness generation: assumption values, thread ID, variable names, and function names. Third, *OpenGBF* supports the use of *UndefinedBehaviorSanitizer* [10], *AddressSanitizer* [11] and *ThreadSanitizer* [12] for the detection of vulnerabilities that cannot be expressed as reachability errors (e.g., buffer overflows, thread leaks).

Results Aggregation Stage. Finally, *EBF* aggregates the outcomes of the *Safety Proving* and the *Falsification* stages as depicted in the table in Fig. 1. The majority of cases are straightforward: if one of the tools produces an inconclusive verdict (i.e., *Unknown*), then *EBF* relies on the decision provided by the other tool. However, if *OpenGBF* finds a bug in the PUT that is deemed to be safe by BMC, *EBF* reports a *Conflict*. In this case extra information can be obtained from the counter-example produced by the fuzzer.

3 Strengths and Weaknesses

EBF 4.2 participated in the *ConcurrencySafety* category of *SV-COMP 2023*, which comprises four subcategories: *ConcurrencySafety-Main*, *NoDataRace-Main*, *ConcurrencySafety-NoOverflows* and *ConcurrencySafety-MemSafety*.

Regarding the *ConcurrencySafety-Main* subcategory, *EBF 4.2* provided 357 correct results out of 692, with only 1 incorrect false and the rest unknown. More in detail, *EBF* correctly identified 67 safe benchmarks and 249 unsafe benchmarks, thus highlighting the *EBF* strengths in bug-finding. In addition, *EBF* labeled an extra 41 benchmarks as unsafe, which were not confirmed by the witness validator. Among these benchmarks, there are 10 verification tasks (beginning with *goblint-regression/28-race_reach_**) where only two tools can find bugs: *EBF* and *Infer* [13]. At the same time,

we hypothesise that the counter-examples provided by *EBF* are more trustworthy than those provided by *Infer* for these 10 tasks. This is because *EBF* is very conservative in its bug-finding claims, with 290 correct false outcomes, 41 unconfirmed, and only 1 incorrect. In contrast, *Infer* produces 330 correct false outcomes and 331 incorrect ones.

Regarding the *NoDataRace-Main* subcategory, *EBF 4.2* only offered partial support for data race detection by enabling *ThreadSanitizer* inside *OpenGBF*. Unfortunately, the BMC engine we used in this year's competition, *ESBMC*, does not yet maintain full support of this safety property. As a consequence, *EBF* provided only 199 correct verification verdicts out of 904, of which 112 were correct true and 87 correct false. At the same time, *EBF* also reported 46 incorrect verdicts (23 incorrect true and 23 incorrect false), which resulted in a negative score for this subcategory.

Regarding the *ConcurrencySafety-NoOverflows* and *ConcurrencySafety-MemSafety* subcategories, *EBF 4.2* did provide support for detecting arithmetic overflows and memory safety violations by enabling *UndefinedBehaviorSanitizer* and *AddressSanitizer*. However, we did not succeed in providing an implementation that was compliant with the competition standards in time. As a result, *EBF* did not feature in these subcategories.

4 Tool Setup and Configuration

In order to use *EBF*[3], the user must set the architecture (32 or 64-bit) with flag `-a`, the property file path with flag `-p`, the benchmark file paths, and run the following command from the *EBF* root directory:

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

Furthermore, there are optional flags that can be enabled (e.g., set the time and memory limit for each engine). In SV-COMP 2023 we divided the allotted 15 minutes of CPU time per verification task across the verification stages inside *EBF 4.2* as follows: $400s$ for the safety proving stage, $120s$ for the seed generation stage, $240s$ for the falsification stage, and the remaining $140s$ were allocated for the results aggregation, counter-example generation and potential execution overheads.

5 Software Project

We released *EBF* 4.2 under the MIT License, and its code is publicly available on GitHub[4]. All dependencies and installation instructions are listed in the repository `README.md` file.

[3] `https://gitlab.com/sosy-lab/sv-comp/archives-2023/-/blob/main/2023/ebf.zip`

[4] `https://github.com/fatimahkj/EBF`

6 Data-Availability Statement

The tool and all necessary files are available[5] on Zenodo [14].

References

1. Ognawala, S., Hutzelmann, T., Psallida, E., Pretschner, A.: Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. In: SAC. (2018) 1475–1482
2. Alshmrany, K.M., Menezes, R.S., Gadelha, M.R., Cordeiro, L.C.: Fusebmc: A white-box fuzzer for finding security vulnerabilities in c programs. FASE (2020)
3. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, Cham (2019) 244–249
4. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In Margaria, T., Steffen, B., eds.: Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles, Cham, Springer International Publishing (2020) 143–167
5. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In Schlingloff, B.H., Chai, M., eds.: Software Engineering and Formal Methods, Cham, Springer International Publishing (2022) 111–128
6. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. Volume 16. (2016) 1–16
7. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In: USENIX). (2018) 745–761
8. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1) (2018) 1–13
9. Aljaafari, F.K., Menezes, R., Manino, E., Shmarov, F., Mustafa, M.A., Cordeiro, L.C.: Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs. IEEE Access **10** (2022) 121365–121384
10. Zannoni, E.: Improving application security with undefinedbehaviorsanitizer (ubsan) and gcc. Accessed: 2022-11-01.
11. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: USENIX, USA (2012) 28
12. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: Data race detection in practice. In: WBIA. (2009) 62–71
13. Kettl, M., Lemberger, T.: The static analyzer Infer in SV-COMP (competition contribution). In: Proc. TACAS (2). LNCS 13244, Springer (2022) 451–456
14. Aljaafar, F.: Ebf a participated version in sv-comp 2023 (December 2022)

[5] `https://doi.org/10.5281/zenodo.7467746`