FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs
(Competition Contribution)

Kaled M. Alshmrany[1] *, Rafael S. Menezes[2], Mikhail R. Gadelha[3], and Lucas C. Cordeiro[4]

[1] University of Manchester, Manchester, UK
Institute of Public Administration, Jeddah, Saudi Arabia
kaled.alshmrany@manchester.ac.uk
[2] Federal University of Amazonas, Manaus, Brazil
[3] SIDIA Instituto de Ciência e Tecnologia, Manaus, Brazil
[4] University of Manchester, Manchester, UK

Abstract. We describe and evaluate a novel white-box fuzzer for C programs named FuSeBMC, which combines fuzzing and symbolic execution, and applies Bounded Model Checking (BMC) to find security vulnerabilities in C programs. FuSeBMC explores and analyzes C programs (1) to find execution paths that lead to property violations and (2) to incrementally inject labels to guide the fuzzer and the BMC engine to produce test-cases for code coverage. FuSeBMC successfully participates in Test-Comp'21 and achieves first place in the `Cover-Error` category and second place in the `Overall` category.

1 Test Generation Approach

Automated test-case generation is a method to check whether the software matches expected requirements [2]. It involves the automated execution of software components to evaluate intricate properties and achieve code coverage metrics (e.g., decision, branch, instruction). Here, we describe and evaluate a novel white-box fuzzer, FuSeBMC, capable of automatically producing test-cases for C programs. FuSeBMC provides an innovative software testing framework that detects security vulnerabilities in C programs by using fuzzing and symbolic execution in combination with Bounded Model Checking (BMC) (cf. Fig. 1). FuSeBMC builds on top of clang [1] to instrument the C program, uses Map2check [6] as a fuzzing engine, and ESBMC (Efficient SMT-based Bounded Model Checker) [4] as a BMC engine, thus combining dynamic and static verification techniques.

FuSeBMC takes a C program and a test specification [3] as input. In the `Cover-Error` category, FuSeBMC invokes the fuzzing and BMC engines to find a path that violates a given property. It uses an iterative BMC approach that incrementally unwinds the program until it finds a property violation or exhausts time or memory limits. FuSeBMC uses incremental BMC to explore the program state space searching for a property violation since all programs in Test-Comp'21

* Jury Member

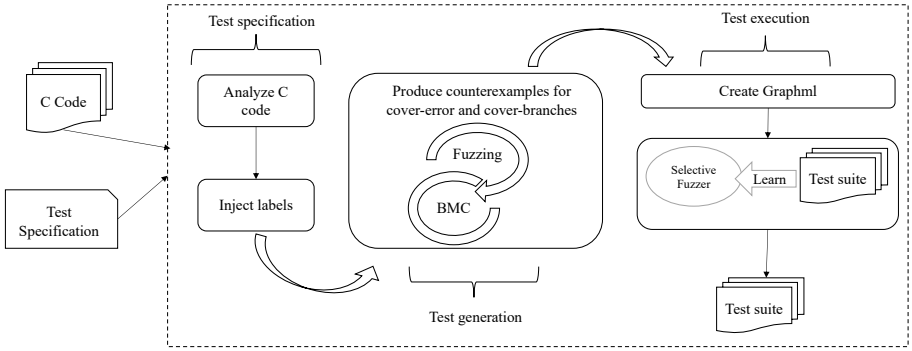

Fig. 1: FuSeBMC: a white-box fuzzer framework for C Programs.

are known to have issues. In the `Cover-Branches` category, FuSeBMC explores and analyzes the target C program using the clang compiler to inject labels incrementally. FuSeBMC will compute all branches and inject the labels for each branch by adding the label `GOAL-`N, where N is the goal number. Both engines will check whether these injected labels are reachable to produce test-cases for branch coverage.

FuSeBMC analyzes the counterexamples and saves them as a *graphml* file. It checks whether the fuzzing and BMC engines could produce counterexamples for both categories `Cover-Error` and `Cover-Branches`. If that is not the case, FuSeBMC employs a second fuzzing engine based on selective fuzzer to produce test-cases for the rest of the labels. The selective fuzzer produces test-cases by learning from the two engines' output: it analyzes the range of the inputs that should be passed to examine the target C program and then produce different test-cases. Lastly, FuSeBMC prepares valid test-cases with metadata to test a target C program using TestCov [3] as a test validator.

FuSeBMC sets a 150 seconds limit for the fuzzing engine and a 700 seconds limit for the BMC engine and sets a 50 seconds limit for the selective fuzzer. These numbers were obtained empirically by analyzing the Test-Comp'21 results.

2 Strengths and Weaknesses

Incremental BMC allows FuSeBMC to keep unwinding the program until a property violation is found or time or memory limits are exhausted. This approach is advantageous in the `Cover-Error` category as finding one error is the primary goal. Another strength of FuSeBMC is that it can accurately model C programs that use the IEEE floating-point arithmetic [5]. The floating-point encoding layer in our BMC engine extends the support for the SMT `FP` theory to solvers that do not support it natively. FuSeBMC can test programs with floating-point arithmetic using all currently supported solvers in BMC engine (ESBMC), including Boolector [7], which does not support the SMT `FP` theory natively.

In both `Cover-Error` and `Cover-Branches` categories, various test-cases produced by FuSeBMC are validated successfully. The majority of our test-cases were produced by the BMC engine and the selective fuzzer; our fuzzing engine did not produce many test-cases because it does not model the C library, so it mostly guesses the inputs. However, our fuzzing engine is not limited to only produce test-cases: it helps our selective fuzzer by providing information about the number of inputs required to trigger a property violation, i.e., the number of assignments required to reach an error. In several cases, the BMC engine can exhaust the time limit before providing such information, e.g., when there are large arrays that need to be initialized at the beginning of the program.

Apart from that, our employed verification engines also demonstrate a certain level of weakness to produce test-cases due to the many optimizations we perform when converting the program to SMT. In particular, two techniques affected the test-case generation significantly: *constant folding* and *slicing*. *Constant folding* evaluates constants (which includes nondeterministic symbols) and propagates then throughout the formula during encoding, and *slicing* removes expression not in the path to trigger a property violation. These two techniques can significantly reduce SMT solving time. However, they can remove the expressions required to trigger a violation when the program is compiled, i.e., variable initialization might be optimized away, forcing FuSeBMC to generate a test-case with undefined behavior.

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

3 Tool Setup and Configuration

In order to run our `fusebmc.py` script[5], one must set the architecture (*i.e.*, 32 or 64-bit), the competition strategy (i.e., k-induction, falsification, or incremental BMC), the property file path, and the benchmark path, as:

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

where `-a` sets the architecture, `-p` sets the property file path, and `-s` sets the strategy (e.g., `kinduction`, `falsi`, `incr`, or `fixed`). For Test-Comp'21, FuSeBMC uses `incr` for incremental BMC.

By choosing fuzzing engine, we set the following options when executing Map2Check: timeout of 150 seconds for Map2Check in `Cover-Error`, and a timeout of 70 seconds in `Cover-Branches`; `--fuzzer-mb` 1000 limits memory to 1000

[5] https://gitlab.com/sosy-lab/test-comp/archives-2021/-/blob/master/2021/FuSeBMC.zip

MB; `--target-function-name reach_error` defines the function name to be searched; `--target-function` checks whether the target-function is reachable; `--nondet-generator fuzzer` uses only libfuzzer [8]; `--generate-witness` sets the witness output path.

By choosing incremental BMC, the following options are set when executing ESBMC: `--no-div-by-zero-check` disables the division by zero check (required by Test-Comp); `--force-malloc-success` sets that all dynamic allocations succeed (a Test-Comp requirement); `--floatbv` enables floating-point SMT encoding; `--incremental-bmc` enables incremental BMC; `--unlimited-k-steps` removes the upper limit of iteration steps for incremental BMC; `--witness-output` sets the witness output path; `--no-bounds-check` and `--no-pointer-check` disable bounds-check and pointer-safety checks, resp., since we are only interested in finding reachability bugs; `--k-step 5` sets the incremental BMC to 5; `--no-allign-check` disables pointer alignment checks; and `--no-slice` disables slicing of unnecessary instructions.

The Benchexec tool info module is named `fusebmc.py` and the benchmark definition file is `FuSeBMC.xml`.

4 Software Project

The FuSeBMC source code is written in C++ and it is available for downloading at GitHub[6], which includes the latest release of FuSeBMC v3.6.6. FuSeBMC is publicly available under the terms of the MIT License. Instructions for building FuSeBMC from the source code are given in the file `README.md` (including the description of all dependencies).

References

1. Clang documentation. http://clang.llvm.org/docs/index.html.
2. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test-case generation. J. Syst. Softw. **86**(8), 1978–2001, 2013.
3. Beyer, D.: Second competition on software testing: Test-Comp 2020. In FASE, LNCS 12076, pp. 505–519, 2020.
4. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In ASE, pp. 888–891, 2018.
5. Gadelha, M.R., Menezes, R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC: scalable and precise test generation based on the floating-point theory - (competition contribution). In FASE, LNCS 12076, pp. 525–529, 2020.
6. Menezes, R., Rocha, H., Cordeiro, L., Barreto, R.: Map2check using LLVM and KLEE. In TACAS, LNCS 10806, pp. 437–441, 2018.
7. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. Journal on Satisfiability, Boolean Modeling and Computation **9**, 53–58 (2014)
8. Libfuzzer – a library for coverage-guided fuzz testing https://llvm.org/docs/LibFuzzer.html

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