FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing

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

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

Abstract. *FuSeBMC* is a test generator for finding security vulnerabilities in C programs. In earlier work [4], we described a previous version that incrementally injected labels to guide Bounded Model Checking (BMC) and Evolutionary Fuzzing engines to produce test cases for code coverage and bug finding. This paper introduces a new version of *FuSeBMC* that utilizes both engines to produce smart seeds. First, the engines are run with a short time limit on a lightly instrumented version of the program to produce the seeds. The BMC engine is particularly useful in producing seeds that can pass through complex mathematical guards. Then, *FuSeBMC* runs its engines with more extended time limits using the smart seeds created in the previous round. *FuSeBMC* manages this process in two main ways using its *Tracer* subsystem. Firstly, it uses *shared memory* to record the labels covered by each test case. Secondly, it evaluates test cases, and those of high impact are turned into seeds for subsequent test fuzzing. As a result, we significantly increased our code coverage score from last year, outperforming all tools that participated in this year's competition in every single category.

1 Overview

Software testing is one of the most crucial phases in software development [11]. Tests often expose critical bugs in software applications. In earlier work [4], we presented *FuSeBMC*, an automated test generation tool that exploits the combination of Fuzzing and Bounded Model Checking. *FuSeBMC* achieved second place in Test-Comp 2021 [5,3] and first place in the *Cover-Error* category. It ranked fourth in the *Cover-Branches* category. This year, we introduce a new version of *FuSeBMC* (v4) that adds smart seed generation and shared memory amongst other improvements and features. The new version significantly improves on the previous version, particularly relating to code coverage. One of the primary contributions of this paper is the linking of a grey-box fuzzer with a bounded model checker. A bounded model checker works by treating a program as a state transition system and then checking whether there exists a transition in this system of length less than a bound k that violates the property to be verified [6,8]. We leverage this power of model checkers as a method for smart seed generation. During grey-box fuzzing, if a particular branch has not been explored, BMC can be used to provide a model (set of assignments to input variables), which reaches the branch. This model is then shared in memory and seeded for further grey-box fuzzing. However, BMC can be slow and resource-intensive. We also carry out a lightweight static program analysis to recognize input verification. We analyze the code for conditions on the input variables and ensure that seeds are only selected if they pass these conditions. Together, these contributions turn *FuSeBMC* into a world-leading fuzzer.

2 Test Generation Approach

Figure 1 provides an overview of the components within *FuSeBMC* and how these interact. *FuSeBMC* makes use of the Clang tooling infrastructure [1] to instrument programs. In addition, *FuSeBMC* employs three engines in its reachability analysis. *FuSeBMC* starts by injecting goal labels into the given input (C program) and ranks them depending on the chosen strategy. After that, *FuSeBMC* employs two fuzzers, one based on the American Fuzzy Loop (AFL) [7,2], and a second custom fuzzer, which we refer to as *selective fuzzer* (see [4] for details). ESBMC [10,9] is a state-of-the-art SMT-based bounded model checker which *FuSeBMC* utilises to produce seeds and test cases. Besides for these engines, *FuSeBMC* also incorporates a subsystem we call the *Tracer*. This component maintains a program graph and records which labels in the graph have been covered by which test cases. It coordinates the other tools and handles the passing of information between them. The fuzzers generate test cases by randomly mutating the program's input and running it to analyze code coverage. The BMC engine executes the given program symbolically to determine the reachability of particular goal labels. In case of success, a witness – a set of inputs leading to the goal – is produced. If any engine manages to reach a label, we say that label is *covered*. The *Tracer* records which goals have been covered by which test cases. This information is used to prevent the computationally expensive BMC engine from trying to reach an already covered goal. More importantly, it is used to coordinate the fuzzing and BMC engines. Let L_n be the deepest label on some branch that ESBMC covers and let L_{n+1} be the next deepest label on the branch. *Tracer* records the test case produced by ESBMC that covers L_n and then passes this as a seed to the fuzzer. Mutating such a seed has a far larger chance of leading to a test case that covers L_{n+1} than starting the fuzzer from scratch. *FuSeBMC* runs until all goals are covered or a timeout is reached.

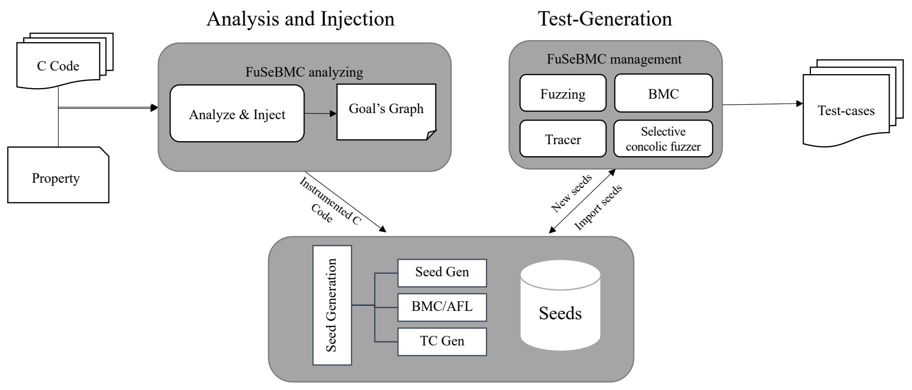

Fig. 1. *FuSeBMC* v4 Framework. This figure displays the major components of the *FuSeBMC* test generator and how they interact. Note in particular the seed store.

Code Instrumentation *FuSeBMC* front-end uses Clang tooling infrastructure [1] to parse a C program and produce an Abstract Syntax Tree (AST). While traversing the AST, *FuSeBMC* injects labels into each branch, including every conditional statement, loop, and function. Using these labels, *FuSeBMC* can measure the code coverage.

Reachability Graph Analysis After instrumenting the C program, *FuSeBMC* analyzes it and produces a reachability graph. The graph will assign each goal label to the code block it is located in. Then, *FuSeBMC* ranks goals depending on the strategy chosen. For example, one strategy is to prefer deeper goals over shallower goals. This strategy improves the performance of *FuSeBMC* since a test case that covers a deep goal will also cover shallower goals on the path to it. *FuSeBMC* also ranks coverage metrics over others, such as conditional coverage over loop coverage.

Seed Generation A unique aspect of the latest version of *FuSeBMC* is a seed generation phase that is run prior to the start of the principal reachability analysis. In this phase, *FuSeBMC* first lightly instruments the code under test by limiting loop bounds and assuming a narrow range of values for input variables. The bounds on input variables are further limited by carrying out a lightweight static analysis to recognize code that applies verification conditions to input variables. After instrumenting the code, *FuSeBMC* runs its fuzzing and BMC engines with very short time limits. The test cases generated by these engines are ranked, and the highest impact test cases are selected as smart seeds for the next round. The impact of a test case is measured using two metrics. First, the number of labels covered uniquely by that test case, and second, the program depth achieved by the test case. ESBMC is particularly effective at seed generation as its underlying SMT solvers can be used to discover test cases that circumvent complex mathematical guards.

Reachability Analysis Engines In its primary phase, *FuSeBMC* carries out reachability analysis. Essentially this involves running the engines with longer timeouts on the original non-instrumented code with the fuzzer making use of the smart seeds. The *Tracer* coordinates the various engines through the use of *shared memory*. For example, assume that ESBMC is unable to cover some goal L and let L' be the deepest goal on the path to L that ESBMC *can* cover. The tracer records the test case that covers L' and passes it to the fuzzer via shared memory as an incomplete seed. Thus, *FuSeBMC* combines the strengths of both types of engines. The BMC engine produces seeds that bypass complex guards and thereby help the fuzzers explore paths deep within the program. During reachability analysis, the *Tracer* constantly monitors the test cases produced by the various engines. High impact test cases, as measured by the metrics discussed above, are passed by the *Tracer* back to the seed store. Thus, the seed store is dynamically updated as the analysis progresses.

3 Strengths and Weaknesses

The strengths of the latest version of *FuSeBMC* are as follows. It runs a dedicated seed generation phase in order to start the main fuzzing effort with high quality, high impact seeds. Furthermore, during the main test-generation phase, these seeds are constantly being updated. Beyond this, it incorporates a dedicated subsystem, the *Tracer*, that uses a shared memory store to manage the various engines. By combining the engines, the *Tracer* ensures that *FuSeBMC* far outperforms the individual engines, or even the running of the engines in parallel, but isolated. The outcome of these improvements can be seen in the ECA and Combination benchmark sets. Previously, these posed a challenge to *FuSeBMC*. With the latest changes, according to preliminary results, *FuSeBMC* achieved first place in the Combination subcategory and took second place in the ECA subcategory of the 2022 Test-Comp competition. In the ECA subcategory *FuSeBMC* displayed a remarkable 60% improvement over its performance from the previous competition. The preliminary results also indicate that *FuSeBMC* has achieved first place in the *Cover-Branches* category with high coverage and validation statistics. One of the weaknesses of *FuSeBMC* that we plan to work on, is that for large programs, particularly for programs that redefine C library functions, seed generation can be slow and consume too much of the tool's time.

4 Tool Setup and Configuration

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

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

where `-a` sets the architecture to 32 or 64, `-p` sets the property file to `PROPERTY_FILE`, where it has a list of all the properties to be tested. `-s` sets the BMC strategy to one of the listed strategies`{kinduction,falsi,incr,fixed}`. The Benchexec tool info module is `fusebmc.py` and the benchmark definition file is `FuSeBMC.xml`.

5 Software Project

FuSeBMC is implemented using C++, and it is publicly available under the terms of the MIT License at GitHub[3]. The repository includes the latest version of *FuSeBMC* (version 4.1.14). *FuSeBMC* dependencies and instructions for building from source code are all listed in the `README.md` file.

References

1. Clang documentation. http://clang.llvm.org/docs/index.html, 2015.
2. American fuzzy lop, https://lcamtuf.coredump.cx/afl/, 2021.
3. Kaled Alshmrany et al. FuSeBMC: A white-box fuzzer for finding security vulnerabilities in c programs. *(FASE)*, 12649:363–367, 2020.
4. Kaled Alshmrany et al. FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in c programs. In *International Conference on TAP*, pages 85–105, 2021.
5. Dirk Beyer. Status report on software testing: Test-comp 2021. *Proc. FASE. LNCS*, 12649.
6. Armin Biere. Bounded model checking. Frontiers in Artificial Intelligence and Applications, pages 457–481. 2009.
7. Marcel Böhme et al. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
8. Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.
9. Mikhail R Gadelha et al. ESBMC: Scalable and precise test generation based on the floating-point theory:(competition contribution). *FASE 2020*, pages 525–529.
10. Mikhail R Gadelha et al. ESBMC v6. 0: Verifying c programs using k-induction and invariant inference. In *International Conference on TACAS*, pages 209–213. Springer, 2019.
11. Nicha Kosindrdecha and Jirapun Daengdej. A test case generation process and technique. *journal of Software Engineering*, 4(4):265–287, 2010.

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