# FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing

## (Competition Contribution)

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
kaled.alshmrany@postgrad.manchester.ac.uk

**Abstract.** *FuSeBMC* is a test generator for finding security vulnerabilities in C programs. In Test-Comp 2021, 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 an improved version of *FuSeBMC* that utilizes both engines to produce smart seeds. First, the engines 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 extended time limits using the smart seeds created in the previous round. *FuSeBMC* manages this process in two main ways. 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. In this year's competition, we participate in the *Cover-Error*, *Cover-Branches*, and *Overall* categories. The Test-Comp 2022 results show that we significantly increased our code coverage score from last year, outperforming all tools in all categories.

**Keywords:** Automated Test-Case Generation · Symbolic Execution · Bounded Model Checking · Fuzzing · Security · Seed.

## 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 BMC. *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

* Jury Member

this power of model checkers as a method for smart seed generation. Here, we rate seeds on two metrics. First, the depth of the deepest goal covered by the seed. Second, the number of goals covered uniquely by the seed. Seeds that rate highly on these metrics are called *smart*. 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) that reaches the branch. This model is a smart seed since it covers a previously unexplored branch. It is then added to a seed store. Periodically seeds are selected from the store for further grey-box fuzzing based on the criteria as mentioned above. However, BMC can be slow and resource-intensive. As an alternative, we also carry out a lightweight static program analysis to recognize certain restricted forms of 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-class 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: one BMC and two fuzzing engines. ESBMC [9,10] is a state-of-the-art SMT-based bounded model checker. For the two fuzzers, one is based on the American Fuzzy Lop (AFL) [7,2], and the other is a custom fuzzer, which we refer here to as *selective fuzzer* (see [4] for details). In the sections below, we detail how these components work together.

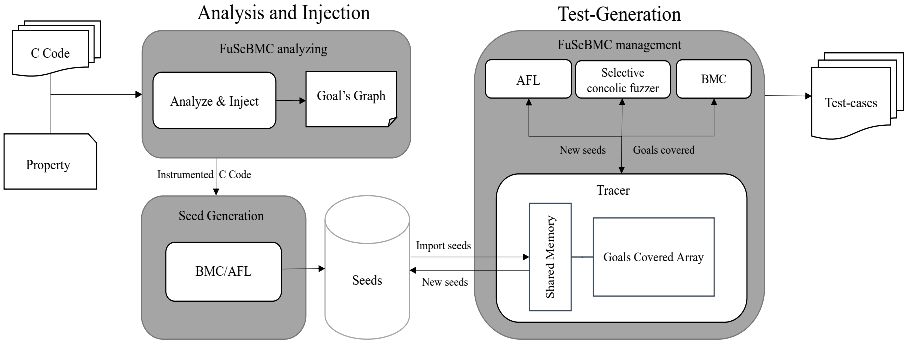

**Fig. 1.** *FuSeBMC* v4 Framework. This figure illustrates the major components of the *FuSeBMC* test generator and how they interact. Note in particular the seed store, which interacts with the BMC/AFL and the shared memory to produce test cases.

**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 assigns each goal label to the code block it is located in. Then, *FuSeBMC* ranks goals depending on the strategy chosen. For example, one strategy, which we used in Test-Comp 2022, 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 concise time limits (60 s for Test-Comp 2022). 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 selected seeds are added to the *seed store*. The impact of a test case is measured using two metrics.

1. The number of labels covered uniquely by that test case.
2. The maximum 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. Note that we do not rely on any specific features of the models returned by the SMT solvers. Instead, the strength of the method lies in the solvers' ability to return *some* model that can satisfy a guard and cover goals lying beyond. A fuzzer on its own, randomly mutating a seed, struggles to explore program sections occurring behind complex guards [12].

**Reachability Analysis Engines** In its primary phase, *FuSeBMC* carries out reachability analysis. Essentially, this involves running the engines in parallel with longer timeouts on the original, non-instrumented code with the fuzzer making use of the smart seeds. ESBMC is run using an incremental BMC strategy with some fixed time limit for each goal it attempts. *FuSeBMC*'s *Tracer* component coordinates the various engines through the use of *shared memory*. In this shared memory, we have two components. The first component is a "goals covered array" that stores the goals covered so far during the execution. Its purpose is to ensure there is no wasting effort through duplication of work. Secondly, the *Tracer* maintains a set of the currently most effective seeds for the fuzzer to use.

As the engines run and produce new test cases, the *Tracer* monitors these and evaluates them, adding those with the highest impact, as measured by the metrics above, to the seed store. Thus, the seed store is dynamically updated as the analysis progresses. Periodically, it selects a number of the most effective seeds from the store and adds them to shared memory for the fuzzers to use in their next fuzzing round. In parallel, ESBMC uses the "goals covered array" to select an as yet uncovered goal and attempts to find a test case that covers it. Test cases produced by ESBMC are passed directly to the store because they are likely to be beneficial for future fuzzing attempts.

For example, assume that the fuzzers are unable to cover some goal $L$ due to a complex condition guarding it. ESBMC can be used to create a seed that covers $L$. This seed is then passed to the store and later selected for fuzzing. The fuzzers, armed with a seed that covers $L$, may well now be able to reach goals deeper than $L$ along $L$'s path. 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.

## 3 Strengths and Weaknesses

The strengths of the latest version of *FuSeBMC* are as follows. It runs a dedicated seed generation phase to start the main fuzzing effort with high-quality, high-impact seeds. Furthermore, these seeds are constantly being updated during the main test-generation phase. 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, *FuSeBMC* achieved first place in the Combination subcategory and took second place in the ECA subcategory of the 2022 Test-Comp competition. Since the benchmarks in the ECA category have remained stable between last year's and this year's competitions, we can measure FuSeBMC's improvement in terms of the combined coverage it achieves across the 29 tasks. This improvement stands at a remarkable 60%. The 2022 Test-Comp results also show that *FuSeBMC* has achieved first place in the *Cover-Branches* category with high coverage and validation statistics. However, 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`}. For Test-Comp'22, *FuSeBMC* uses `incr` for incremental BMC, which relies on the ESBMC's symbolic execution engine to increasingly unwind the program loops using an iterative technique. The `incr` strategy verifies the program for each unwind bound up to a maximum default value of 50 or indefinitely (until it exhausts the time or memory limits). 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[1]. 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. Test-Comp 2022 provides the script, benchmarks, and *FuSeBMC* binary to reproduce the competition's results[2].

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

[2] https://test-comp.sosy-lab.org/2022/

## Acknowledgment

The Institute of Public Administration - IPA - Saudi Arabia [1] supports the FuSeBMC development. The work in this paper is also partially funded by the EPSRC grants EP/T026995/1, EP/V000497/1, EU H2020 ELEGANT 957286, and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme.

## References

1. Clang documentation. http://clang.llvm.org/docs/index.html.
2. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.
3. Kaled Alshmrany et al. FuSeBMC: A white-box fuzzer for finding security vulnerabilities in C programs. In FASE, pages 363–367, 2021.
4. Kaled Alshmrany et al. FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. International Conference on TAP, pages 85-105, 2021.
5. Beyer, D.: Status report on software testing: Test-Comp 2021. In FASE, pages 341–357, 2021.
6. Armin Biere. Bounded model checking. Frontiers in Artificial Intelligence and Applications. In Handbook of satisfiability, pages 457–481, 2009.
7. Böhme et al. Directed greybox fuzzing. In CCS, pages 2329–2344, 2017.
8. Lucas C. Cordeiro et al. SMT-Based bounded model checking for embedded ANSI-C software. IEEE Trans. Software Eng. 38(4): 957–974, 2012.
9. Gadelha, M.R. et al. ESBMC: scalable and precise test generation based on the floating-point theory:(Competition Contribution). In FASE, pages 525–529, 2020.
10. Gadelha, M.R. et al. ESBMC v6.0: verifying C programs using *k*-induction and invariant inference - (Competition Contribution). In TACAS, pages 209–213, 2019.
11. Nicha Kosindrdecha and Jirapun Daengdej: A test case generation process and technique. Journal of Software Engineering, 4(4):265-287, 2010.
12. Stephens, Nick et al. Driller: Augmenting fuzzing through selective symbolic execution. In NDSS, pages 1–16, 2016.

[1] https://www.ipa.edu.sa/en-us/Pages/default.aspx