FuSeBMC: An Automated Energy-Efficient Test Generator for C Programs

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

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

Abstract. *FuSeBMC* is an automated energy-efficient test generator for finding security vulnerabilities in C programs. It incrementally injects labels to guide Bounded Model Checking (BMC) and evolutionary fuzzing engines to produce test cases for code coverage and bug finding. *FuSeBMC* also exploits a custom fuzzer to produce test cases by learning from test cases produced by the BMC and evolutionary fuzzing engines. Lastly, *FuSeBMC* optimises each engine's execution time to reduce energy consumption. As a result, *FuSeBMC* combines BMC and fuzzing engines to produce test cases that achieve high coverage for low energy consumption and thus detects bugs efficiently compared with the state-of-the-art software testing tools. This paper describes the *FuSeBMC* tool and its performance compared to state-of-the-art test generators.

Keywords: Automated Test Generation · Bounded Model Checking · Fuzzing · Security.

1 Introduction

FuSeBMC is an energy-efficient test generator that combines fuzzing with symbolic execution via bounded model checking. *FuSeBMC* can be used to detect security vulnerabilities in C programs. The tool works by first performing coverage-guided fuzzing and BMC independently to produce test cases that can trigger a security vulnerability. After this, *FuSeBMC* runs a selective fuzzer that uses the test cases produced by the other engines to guide a search for other test cases. BMC works by unrolling a program to some given depth k, converting the resulting program into a logical formula expressed in a fragment of first-order theories, and then using SMT solvers to check the satisfiability of the formula [1]. Fuzzing generates inputs in an attempt to achieve broad coverage or trigger a particular piece of code.

This paper details the *FuSeBMC* architecture. In particular, we focus on *(i)* the use of clang to traverse an inject label into the input program; *(ii)* how *FuSeBMC* controls the fuzzing and BMC engines; and *(iii)* the use of the custom fuzzer we which refer to as a *selective fuzzer* to build on the output of the other engines. A short evaluation section is included, detailing the tool's impressive performance at the Test-Comp 2021 competition [2], but for a more in-depth evaluation please view our earlier papers [3,4]. The tool is available under and open source license at: https://github.com/kaled-alshmrany/FuSeBMC.

2 Components and Features

FuSeBMC builds on top of the Clang compiler [5] to instrument C programs. It uses Map2check [6] as an evolutionary fuzzing engine, and ESBMC (Efficient SMT-based Bounded Model Checker) [1,7] as a BMC and symbolic execution engine, thus combining dynamic and static verification techniques to improve code coverage and bug discovery. *FuSeBMC* takes a C program and test specification as input. It checks for array bounds violations, divisions by zero, pointer safety, and all user-defined properties by default. It also has options to check for overflows and memory leaks. Figure 1 illustrates its architecture.

Analysis and Injection. *FuSeBMC* uses clang, a state-of-the-art compiler suite for C/C++/ObjectiveC/ObjectiveC++ widely used in industry [8], as its front-end. By using Clang as a front-end, *FuSeBMC* can provide compilation error messages. When parsing the C program, *FuSeBMC* leverages clang's powerful static analyzer to provide meaningful warnings. Furthermore, clang can simplify the input program (e.g., calculate the size of expressions, evaluate static asserts), which simplifies the code analysis. Finally, *FuSeBMC* traverses the clang AST and injects labels (e.g., the $GOAL_i$ in Figure. 2) incrementally in every branch of the C code to produce *instrumented* code. The labels are used by *FuSeBMC* to measure code coverage. Figure. 2 provides an example of C code and the equivalent instrumented version.

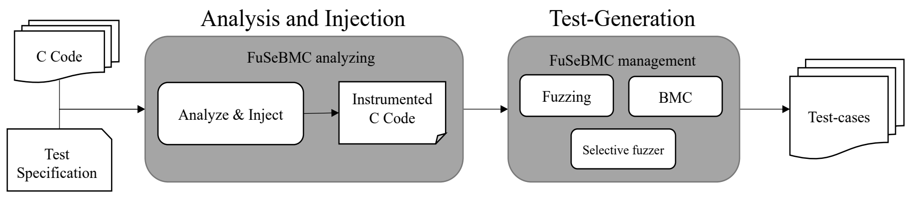

Fig. 1: *FuSeBMC*: An Automated Energy-Efficient Test Generator Framework.

Test-Generation. *FuSeBMC* invokes the BMC and fuzzing engines sequentially to find an execution 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. In particular, it unrolls loops k times, generates the static single assignments (SSA) form of the unrolled program, and from this generates an SMT formula. Then, SMT solvers are used to check the satisfiability of the formula. *FuSeBMC* simplifies the program to generate small SSA sets, using constant folding and various arithmetic simplifications for integer and floating-point data types. For measuring coverage, *FuSeBMC* explores and analyzes the target C program using the clang compiler to inject labels incrementally. *FuSeBMC* traverses every branch of the clang AST and injects a label in each of the form $GOAL_i$ for $i \in \mathbb{N}$. Then, both engines will check whether these injected labels are reachable to produce test cases for branch coverage. After that, *FuSeBMC* checks whether the BMC and fuzzing engines could produce test cases or not. If that is not the case, *FuSeBMC* employs the selective fuzzer

```c
extern void abort(void);
#include <assert.h>
void reach_error() { assert(0); }
extern int __VERIFIER_nondet_int(void);
int main() {
  int a = __VERIFIER_nondet_int();
  int b = __VERIFIER_nondet_int();
  int c = a + b;
  if (a > 0)
    return 1;
  else
    reach_error();
return 0;
}
```

```c
void FuSeBMC_custom_func(void){}
extern void abort(void);
#include <assert.h>
void reach_error() { assert(0);GOAL_1:;}
extern int __VERIFIER_nondet_int(void);
int main() {
  int a = __VERIFIER_nondet_int();
  int b = __VERIFIER_nondet_int();
  int c = a + b;

  if (a > 0) {
    GOAL_3:;
        return 1;
  } else {
    GOAL_4:;
        reach_error();
  }
  GOAL_2:;
  return 0;
}
```

Fig. 2: An example of C program and *FuSeBMC* instrumented file.

(cf. Section 2.1) to produce test cases for the remaining labels. The selective fuzzer produces test cases by learning from the two engines' output. It analyzes the input range to examine the target C program and then produces different test cases to improve code coverage or find bugs.

FuSeBMC supports the solvers Boolector (default) [9], Z3 [10], and Yices [11]. We use these solvers to check the satisfiability of $C \land \neg P$, where C is the set of constraints derived from the SSA form of the program and P is the set of properties. If the formula is satisfiable, the program contains a bug: *FuSeBMC* will generate test cases that lead to the property violation.

FuSeBMC also manages the run-time of the evolutionary fuzzer, BMC, and selective fuzzer engines to 150s, 700s, and 50s, respectively. *FuSeBMC* further manages the time allocated for each engine. If the evolutionary fuzzing engine finishes before the time allocated to it, the remaining time will be carried over and added to the allocated time of the BMC engine. Similarly, we add the remaining time from the BMC engine to the selective fuzzer allocated time. Lastly, *FuSeBMC* prepares valid test cases with metadata to test a target C program using TestCov [12] as a test validator.

2.1 Selective Fuzzer

The selective fuzzer learns from the test cases produced by either the BMC or fuzzing engines to produce inputs for the goals that have not been covered by them. The selective fuzzer utilises the test cases produced by the other engines by extracting from them the number of inputs required to trigger a property violation. For example, in Figure. 3, assume that fuzzing/BMC produced a test case that contains values 18 (1000 times) generated from a random seed. The selective fuzzer will produce random numbers (1000 times) since this is the number of inputs required to trigger a property violation. In several cases, the BMC engine exhausts the time limit before providing such information, e.g., when large arrays need to be initialized at the beginning of the program.

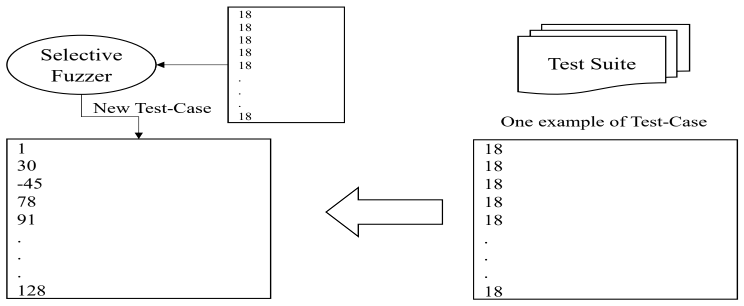

Fig. 3: The Selective Fuzzer Implemented in *FuSeBMC*.

3 Tool Setup

FuSeBMC is invoked via a Python script[3]. To generate test cases for a C program a command of the following form is run:

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

where `-a` sets the architecture (either 32- or 64-bit), `-p` sets the property file path, `-s` sets the strategy (one of `kinduction`, `falsi`, `incr`, or `fixed`) and <file>.c is the C program to be checked. *FuSeBMC* produces the test cases in the XML format.

4 Evaluation

We conducted experiments with *FuSeBMC* on the benchmarks of Test-Comp 2021 [2]. Our benchmarks are taken from the largest and most diverse open-source repository of software verification tasks. The same benchmark collection is used by SV-COMP [13]. These benchmarks yield 3173 test tasks, namely 607 test tasks for the category *Error Coverage* and 2566 test tasks for the category *Code Coverage*. Both categories contain C programs with loops, arrays, bit-vectors, floating-point numbers, dynamic memory allocation, and recursive functions.

We evaluated *FuSeBMC* on the *Error Coverage* category. The plot 4 shows the experimental results compared with other tools in Test-Comp 2021 [2], where *FuSeBMC* achieved the 1st place in this category by solving 500 out of 607 tasks, an 82% success rate.

Also, we applied *FuSeBMC* to the *Branch Coverage* category. The plot 5 shows the experiments' results compared with other tools in the Test-Comp 2021 [2], where *FuSeBMC* achieved fourth place in this category by successfully achieving 1161 out of 2566 scores and a 45% success rate, which was behind the 3rd place by 8 scores only.

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

FuSeBMC overall results achieved 2nd place in Test-Comp 2021, achieving 1776 out of 3173 scores. Figure 6 shows the overall results comparing with other tools in the competition.

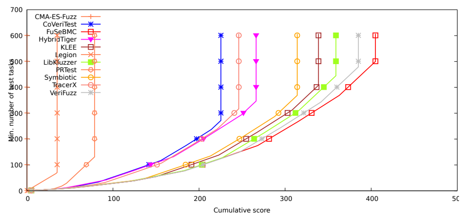

Fig. 4: Results of Cover-Error

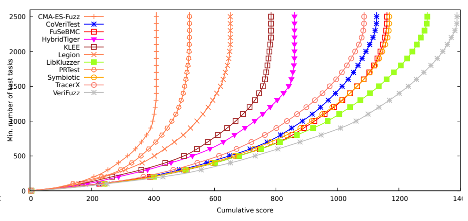

Fig. 5: Results of Cover-Branches

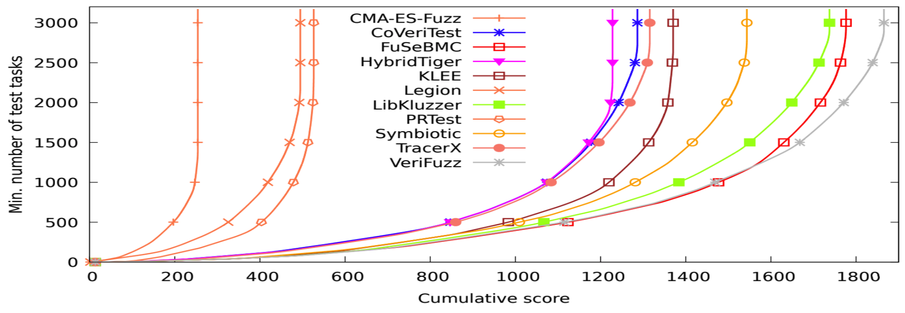

Fig. 6: Quantile functions for category *Overall.* [2]

Test-Comp 2021 also considers the energy efficiency in rankings since a large part of the cost of test generation is caused by energy consumption. *FuSeBMC* is classified as a Green-testing tool - Low Energy Consumption tool (see Table 1). *FuSeBMC* consumed less energy than other tools in the competition. This ranking category uses the energy consumption per score point as a rank measure: CPU Energy Quality, with the unit kilo-joule per score point (kJ/sp). It uses CPU Energy Meter [14] for measuring the energy.

Table 1: The Consumption of CPU and Memory [15].

Rank	Test Generator	Quality(sp)	CPU Time(h)	CPU Energy(kWh)	Rank Measure
Green Testing					(kj/sp)
1	TRACERX	1315	210	2.5	6.8
2	KLEE	1370	210	2.6	6.8
3	FuSeBMC	1776	410	4.8	9.7
worst					51

5 Conclusions and Future work

We presented *FuSeBMC*, an automated energy-efficient test generator tool. Here, we focused on two novel features of *FuSeBMC*. First, we use a selective fuzzer as a third engine that learns from the test cases produced by other engines to produce new test cases for the uncovered goals. Overall, it substantially increases the percentage of successful tasks. Second, *FuSeBMC*'s time allocation strategy. If the fuzzing engine finishes before its allocated time, the remaining time is carried over and added to the allocated time of the BMC engine. Similarly, we add the remaining time from the BMC engine to the selective fuzzer's allocated time. Results over the Test-Comp 2021 benchmark suite show that *FuSeBMC* is the strongest tool currently available in detecting bugs in terms of resources and time management. Future work will investigate using reinforcement learning techniques to guide our selective fuzzer to find test cases that path-based fuzzing and BMC could not find.

References

1. M. Y. R. Gadelha, R. Menezes, F. R. Monteiro, L. C. Cordeiro, and D. A. Nicole, "ESBMC v6.0: Verifying C programs using k-induction and invariant inference," in *TACAS*, vol. 11429 of *LNCS*, pp. 209–213, 2019.
2. D. Beyer, "Status report on software testing: Test-comp 2021," *FASE*, vol. 12649, pp. 341–357, 2021.
3. K. M. Alshmrany, R. S. Menezes, M. R. Gadelha, and L. C. Cordeiro, "FuSeBMC: A white-box fuzzer for finding security vulnerabilities in c programs," *FASE*, vol. 12649, pp. 363–367, 2020.
4. K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in c programs," *TAP*, 2021. Awaiting Publication.
5. "Clang documentation." http://clang.llvm.org/docs/index.html, 2015.
6. H. Rocha, R. Barreto, and L. C. Cordeiro, "Hunting memory bugs in C programs with map2check," in *TACAS*, vol. 9636 of *LNCS*, pp. 934–937, 2016.
7. M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "ESBMC 5.0: an industrial-strength c model checker," in *ASE*, pp. 888–891, 2018.
8. C. Metz, "Why Apple's Swift language will instantly remake computer programming." https://www.wired.com/2014/07/apple-swift/, 2004.
9. M. Preiner, A. Niemetz, and A. Biere, "Better lemmas with lambda extraction," in *FMCAD*, pp. 128–135, 2015.
10. L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, vol. 4963 of *LNCS*, pp. 337–340, 2008.
11. B. Dutertre, "Yices 2.2," in *Computer-Aided Verification*, vol. 8559 of *LNCS*, pp. 737–744, 2014.
12. D. Beyer and T. Lemberger, "Testcov: Robust test-suite execution and coverage measurement," in *ASE*, pp. 1074–1077, IEEE, 2019.
13. D. Beyer, "Software verification: 10th comparative evaluation (sv-comp 2021)," *Proc. TACAS (2). LNCS*, vol. 12652.
14. D. Beyer and P. Wendler, "CPU energy meter: A tool for energy-aware algorithms engineering," in *TACAS*, vol. 12079 of *LNCS*, pp. 126–133, 2020.
15. D. Beyer, "3rd Competition on Software Testing (Test-Comp 2021)," 2021.