







FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs



Kaled M. Alshmrany^{1,2}(✉) , Mohannad Aldughaim¹ ,
Ahmed Bhayat¹ , and Lucas C. Cordeiro¹ 



¹ University of Manchester, Manchester, UK

kaled.alshmrany@postgrad.manchester.ac.uk

² Institute of Public Administration, Jeddah, Saudi Arabia

Abstract. We describe and evaluate a novel approach to automated test generation that exploits fuzzing and Bounded Model Checking (BMC) engines to detect security vulnerabilities in C programs. We implement this approach in a new tool *FuSeBMC* that explores and analyzes the target C program by injecting labels that guide the engines to produce test cases. *FuSeBMC* also exploits a selective fuzzer to produce test cases for the labels that fuzzing and BMC engines could not produce test cases. Lastly, we manage each engine's execution time to improve *FuSeBMC*'s energy consumption. We evaluate *FuSeBMC* by analysing the results of its participation in Test-Comp 2021 whose two main categories evaluate a tool's ability to provide *code coverage* and *bug detection*. The competition results show that *FuSeBMC* performs well compared to the state-of-the-art software testing tools. *FuSeBMC* achieved 3 awards in the Test-Comp 2021: first place in the *Cover-Error* category, second place in the *Overall* category, and third place in the *Low Energy Consumption* category.

Keywords: Automated test generation · Bounded model checking · Fuzzing · Security

1 Introduction

Developing software that is secure and bug-free is an extraordinarily challenging task. Due to the devastating effects vulnerabilities may have, financially or on an individual's well-being, software verification is a necessity [1]. For example, Airbus found a software vulnerability in the A400M aircraft that caused a crash in 2015. This vulnerability created a fault in the control units for the engines, which caused them to power off shortly after taking-off [2]. A software vulnerability is best described as a defect or weakness in software design [3]. That design can be verified by Model Checking [4] or Fuzzing [5]. Model-checking and fuzzing are two techniques that are well suited to find bugs. In particular,

model-checking has proven to be one of the most successful techniques based on its use in research and industry [6]. This paper will focus on fuzzing and bounded model checking (BMC) techniques for code coverage and vulnerability detection. Code coverage has proven to be a challenge due to the state space problem, where the search space to be explored becomes extremely large [6]. For example, vulnerabilities are hard to detect in network protocols because the state-space of sophisticated protocol software is too large to be explored [7]. Vulnerability detection is another challenge that we have to take besides the code coverage. Some vulnerabilities cannot be detected without going deep into the software implementation. Many reasons motivate us to verify software for coverage and to detect security vulnerabilities formally. Therefore, these problems have attracted many researchers' attention to developing automated tools.

Researchers have been advancing the state-of-the-art to detect software vulnerabilities, as observed in the recent edition of the International Competition on Software Testing (Test-Comp 2021) [8]. Test-Comp is a competition that aims to reflect the state-of-the-art in software testing to the community and establish a set of benchmarks for software testing. Test-Comp 2021 [8], had two categories *Error Coverage* (or *Cover-Error*) and *Branch Coverage* (or *Cover-Branches*). The *Error Coverage* category tests the tool's ability to discover bugs where every C program in the benchmarks contains a bug. The aim of the *Branch Coverage* category is to cover as many program branches as possible. Test-Comp 2021 works as follows: each tool task is a pair of an input program (a program under test) and a test specification. The tool then should generate a test suite according to the test specification. A test suite is a sequence of test cases, given as a directory of files according to the format for exchangeable test-suites¹. The specification for testing a program is given to the test generator as an input file (either `coverage-error-call.prp` or `coverage branches.prp` for Test-Comp 2021) [8].

Techniques such as fuzzing [9], symbolic execution [10], static code analysis [11], and taint tracking [12] are the most common techniques, which were employed in Test-Comp 2021 to cover branches and detect security vulnerabilities [8]. Fuzzing is generally unable to create various inputs that exercise all paths in the software execution. Symbolic execution might also not achieve high path coverage because of the dependence on Satisfiability Modulo Theories (SMT) solvers and the path-explosion problem. Consequently, fuzzing and symbolic execution by themselves often cannot reach deep software states. In particular, the deep states' vulnerabilities cannot be identified and detected by these techniques in isolation [13]. Therefore, a hybrid technique involving fuzzing and symbolic execution might achieve better code coverage than fuzzing or symbolic execution alone. VeriFuzz [14] and LibKluzzer [15] are the most prominent tools that combine these techniques. VeriFuzz combines the power of feedback-driven evolutionary fuzz testing with static analysis, where LibKluzzer combines the strengths of coverage-guided fuzzing and dynamic symbolic execution.

This paper proposes a novel method for detecting security vulnerabilities in C programs that combines fuzzing with symbolic execution via bounded model

¹ <https://gitlab.com/sosy-lab/software/test-format/>.

checking. We make use of coverage-guided fuzzing to produce random inputs to locate security vulnerabilities in C programs. Separately, we make use of BMC techniques [16, 17]. BMC unfolds a program up to depth k by evaluating (conditional) branch sides and merging states after that branch. It builds one logical formula expressed in a fragment of first-order theories and checks the satisfiability of the resulting formula using SMT solvers. These two methods are combined in our tool *FuSeBMC* which can consequently handle the two main features in software testing: *bug detection* and *code coverage*, as defined by Beyer et al. [18]. We also manage each engine’s execution time to improve *FuSeBMC*’s efficiency in terms of verification time. Therefore, we raise the chance of bug detection due to its ability to cover different blocks of the C program, which other tools could not reach, e.g., KLEE [19], CPAchecker [20], VeriFuzz [14], and LibKluzzer [15].

Contributions. This paper extends our prior work [21] by making the following original contributions.

- We detail how *FuSeBMC* guides fuzzing and BMC engines to produce test cases that can detect security vulnerabilities and achieve high code coverage while massively reducing the consumption of both CPU and memory. Furthermore, we discuss using a custom fuzzer we refer to as a *selective fuzzer* as a third engine that learns from the test cases produced by fuzzing/BMC to produce new test cases for the uncovered goals.
- We provide a detailed analysis of the results from *FuSeBMC*’s successful participation in Test-Comp 2021. *FuSeBMC* achieved first place in *Cover-Error* category and second place in *Overall* category. *FuSeBMC* achieved first place in the subcategories *ReachSafety-BitVectors*, *ReachSafety-Floats*, *ReachSafety-Recursive*, *ReachSafety-Sequentialized* and *ReachSafety-XCSP*. We analyse the results in depth and explain how our research has enabled *FuSeBMC*’s success across these categories as well its low energy consumption.

2 Preliminaries

2.1 Fuzzing

Fuzzing is a cost-effective software testing technique to exploit vulnerabilities in software systems [22]. The basic idea is to generate random inputs and check whether an application crashes; it is not testing functional correctness (compliance). Critical security flaws most often occur because program inputs are not adequately checked [23]. Therefore, fuzzing prepares random or semi-random inputs, which might consider, e.g., (1) very long or completely blank strings; (2) min/max values of integers, or only zero and negative values; and (3) include unique values or characters likely to trigger bugs. Software systems that cannot endure fuzzing could potentially lead to security holes. For example, a bug was found in Apple wireless driver by utilizing file system fuzzing. The driver could not handle some beacon frames, which led to out-of-bounds memory access.

2.2 Symbolic Execution

Introduced in the 1970s, symbolic execution [24] is a software analysis technique that allowed developers to test specific properties in their software. The main idea is to execute a program symbolically using a symbolic execution engine that keeps track of every path the program may take for every input [24]. Moreover, each input represents symbolic input values instead of concrete input values. This method treats the paths as symbolic constraints and solves the constraints to output a concrete input as a test case. Symbolic execution is widely used to find security vulnerabilities by analyzing program behavior and generating test cases [25]. BMC is an instance of symbolic execution, where it merges all execution paths into one single logical formula instead of exploring them individually.

2.3 Types of Vulnerabilities

Software, in general, is prone to vulnerabilities caused by developer errors, which include: *buffer overflow*, where a running program attempts to write data outside the memory buffer, which is intended to store this data [26]; *memory leak*, which occurs when programmers create a memory in a heap and forget to delete it [27]; *integer overflows*, when the value of an integer is greater than the integer’s maximum size in memory or less than the minimum value of an integer. It usually occurs when converting a signed integer to an unsigned integer and vice-versa [28]. Another example is *string manipulation*, where the string may contain malicious code and is accepted as an input; this is reasonably common in the C programming language [29]. *Denial-of-service attack* (DoS) is a security event that occurs when an attacker prevents legitimate users from accessing specific computer systems, devices, services, or other IT resources [30]. For example, a vulnerability in the Cisco Discovery Protocol (CDP) module of Cisco IOS XE Software Releases 16.6.1 and 16.6.2 could have allowed an unauthenticated, adjacent attacker to cause a memory leak, which could have lead to a DoS condition [31]. Part of our motivation is to mitigate the harm done by these vulnerabilities by the proposed method *FuSeBMC*.

3 *FuSeBMC*: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs

We propose a novel verification method named *FuSeBMC* (cf. Fig. 1) for detecting security vulnerabilities in C programs using fuzzing and BMC techniques. *FuSeBMC* builds on top of the Clang compiler [32] to instrument the C program, uses Map2check [33,34] as a fuzzing engine, and ESBMC (Efficient SMT-based Bounded Model Checker) [35,36] as BMC and symbolic execution engines, thus combining dynamic and static verification techniques.

The method proceeds as follows. First, *FuSeBMC* takes a C program and a test specification as input. Then, *FuSeBMC* invokes the fuzzing and BMC engines sequentially to find an execution path that violates a given property.

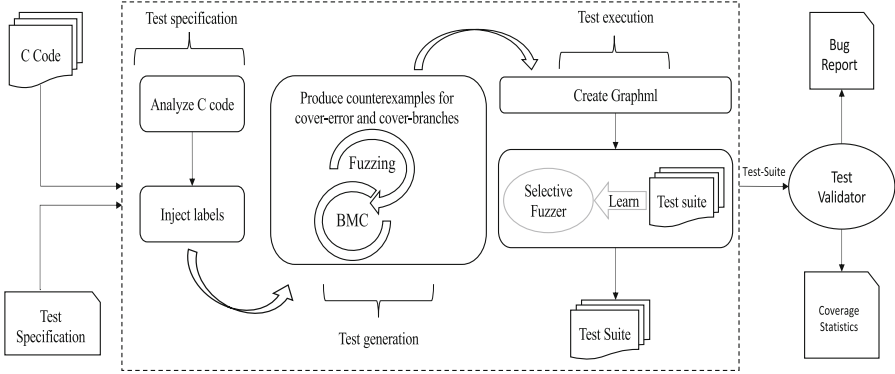


Fig. 1. *FuSeBMC*: an energy-efficient test generator framework.

It uses an iterative BMC approach that incrementally unwinds the program until it finds a property violation or exhausts time or memory limits. In code coverage mode, *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* 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, the so-called selective fuzzer (cf. Sect. 3.6), which attempts to produce test cases for the rest of the labels. The selective fuzzer produces test cases by learning from the two previous engines' output.

FuSeBMC introduces a novel algorithm for managing the time allocated to its component engines. In particular, *FuSeBMC* manages the time allocated to each engine to avoid wasting time for a specific engine to find test cases for challenging goals. For example, let us assume we have 100 goals injected by *FuSeBMC* and 1000s to produce test cases. In this case, *FuSeBMC* distributes the time per engine per goal so that each goal will have 10s and recalculate the time for the goals remaining after each goal passed. If an engine succeeds on a particular goal within the time limit, the extra time is redistributed to the other goals; otherwise, *FuSeBMC* kills the process that passes the time set for it.

Furthermore, *FuSeBMC* has a minimum time, which a goal must be allocated. If there are too many goals for all to receive this minimum time, *FuSeBMC* will select a subset to attempt using a quasi-random strategy (e.g., all even-numbered goals). *FuSeBMC* also manages the global time of the fuzzing, BMC, and selective fuzzing engines. It gives 13% of the time for fuzzing, 77% for BMC, and 10% for selective fuzzing. *FuSeBMC* further carries out time management at this global level to maximize engine usage. If, for example, the fuzzing engine is finished before the time allocated to it, its 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.

FuSeBMC prepares valid test cases with metadata to test a target C program using TestCov [37] as a test validator. The metadata file is an XML file that describes the test suite and is consistently named *metadata.xml*. Figure 2 illustrates an example metadata file with all available fields [37]. Some essential fields include the program function that is tested by the test suite (*entryfunction*), the coverage criterion for the test suite (*specification*), the programming language of the program under test (*sourcecodelang*), the system architecture the program tests were created for (*architecture*), the creation time (*creationtime*), the SHA-256 hash of the program under test (*programhash*), the producer of counterexample (*producer*) and the name of the target program (*programfile*). A test case file contains a sequence of tags (*input*) that describes the input values sequence. Figure 3 illustrates an example of the test case file.

Algorithm 1 describes the main steps we implemented in *FuSeBMC*. It consists of extracting all *goals* of a C program (line 1). For each goal, the instrumented C program, containing the goals (line 2), is executed on our verification engines (fuzzing and BMC) to check the reachability property produced by REACH(G) for that goal (lines 8 & 20). REACH is a function; it takes a goal (G) as input and produces a corresponding property for fuzzing/BMC (line 7 & 19). If our engines find that the property is violated, meaning that there is a valid execution path that reaches the goal (counterexample), then the goals are marked as covered, and the test case is saved for later (lines 9–11). Then, we continue if we still have time allotted for each engine. Otherwise, if our verification engines could not reach some goals, then we employ the selective fuzzer in attempt to reach these as yet uncovered goals. In the end, we return all test cases for all the goals we have found in the specified XML format (line 41).

```

1 <?xml version='1.0'?>
2 <!DOCTYPE test-metadata PUBLIC [...]>
3 <test-metadata>
4   <entryfunction>main</entryfunction>
5   <specification>COVER(init(main()), FQL(COVER EDGES(@DECISIONEDGE)))
6   </specification>
7   <sourcecodelang>C</sourcecodelang>
8   <architecture>32bit</architecture>
9   <creationtime>2021-02-28 20:44:56.117416</creationtime>
10  <programhash>e8f2cf545726d8f791bfc137e9eca7e9de4cb696</programhash>
11  <producer>FuSeBMC</producer>
12  <programfile>sv-benchmarks/c/array-tiling/skippedu.c</programfile>
13 </test-metadata>

```

Fig. 2. An example of a metadata.

3.1 Analyze C Code

FuSeBMC explores and analyzes the target C programs as the first step using Clang [38]. In this phase, *FuSeBMC* analyzes every single line in the C code and considers the conditional statements such as the *if*-conditions, *for*, *while*,

Algorithm 1. Proposed *FuSeBMC* algorithm.

```

Require: program  $P$ 
1:  $goals \leftarrow clang\_extract\_goals(P)$ 
2:  $instrumentedP \leftarrow clang\_instrument\_goals(P, goals)$ 
3:  $reached\_goals \leftarrow \emptyset$ 
4:  $tests \leftarrow \emptyset$ 
5:  $FuzzingTime = 150$ 
6: for all  $G \in goals$  do
7:    $\phi \leftarrow REACH(G)$ 
8:    $result, test\_case \leftarrow Fuzzing(instrumentedP, \phi, FuzzingTime)$ 
9:   if  $result = false$  then
10:      $reached\_goals \leftarrow reached\_goals \cup G$ 
11:      $tests \leftarrow tests \cup test\_case$ 
12:   end if
13:   if  $FuzzingTime = 0$  then
14:      $break$ 
15:   end if
16: end for
17:  $BMCTime = FuzzingTime + 700$ 
18: for all  $G \in (goals - reached\_goals)$  do
19:    $\phi \leftarrow REACH(G)$ 
20:    $result, test\_case \leftarrow BMC(instrumentedP, \phi, BMCTime)$ 
21:   if  $result = false$  then
22:      $reached\_goals \leftarrow reached\_goals \cup G$ 
23:      $tests \leftarrow tests \cup test\_case$ 
24:   end if
25:   if  $BMCTime = 0$  then
26:      $break$ 
27:   end if
28: end for
29:  $SelectiveFuzzerTime = BMCTime + 50$ 
30: for all  $G \in (goals - reached\_goals)$  do
31:    $\phi \leftarrow REACH(G)$ 
32:    $result \leftarrow selective\_fuzzer(instrumentedP, \phi, SelectiveFuzzerTime)$ 
33:   if  $result = false$  then
34:      $reached\_goals \leftarrow reached\_goals \cup G$ 
35:      $tests \leftarrow tests \cup test\_case$ 
36:   end if
37:   if  $SelectiveFuzzerTime = 0$  then
38:      $break$ 
39:   end if
40: end for
41: return  $tests$ 

```

and *do while* loops in the code. *FuSeBMC* takes all these branches as path conditions, containing different values due to the conditions set used to produce the counterexamples, thus helping increase the code coverage. It supports blocks, branches, and conditions. All the values of the variables within each path are

```

1 <?xml version="1.0"?>
2 <!DOCTYPE testcase PUBLIC [...]>
3 <testcase>
4   <input>2</input>
5   <input>1</input>
6   <input>128</input>
7   <input>0</input>
8   <input>0</input>
9   <input>1</input>
10  <input>64</input>
11  <input>0</input>
12  <input>0</input>
13 </testcase>

```

Fig. 3. An example of test case file.

taken into account. Parentheses and the *else*-branch are added to compile the target code without errors.

3.2 Inject Labels

FuSeBMC injects labels of the form $GOAL_i$ in every branch in the C code as the second step. In particular, *FuSeBMC* adds *else* to the C code that has an *if*-condition with no *else* at the end of the condition. Additionally, *FuSeBMC* will consider this as another branch that should produce a counterexample for it to increase the chance of detecting bugs and covering more statements in the program. For example, the code in Fig. 4 consists of two branches: the *if*-branch is entered if condition $x < 0$ holds; otherwise, the *else*-branch is entered implicitly, which can exercise the remaining execution paths. Also, Fig. 4 shows how *FuSeBMC* injects the labels and considers it as a new branch.

```

1 #include <stdio.h>
2 int example () {
3   int x;
4   if ( x < 0 ){
5     //...
6   }
7 }

```

(a) Original C code.

```

1 #include <stdio.h>
2 int example () {
3   int x;
4   if ( x < 0 ){
5     GOAL_1;;
6     //...
7   }
8   else{
9     GOAL_2;;
10  }
11  return 0;
12 }

```

(b) Code instrumented.

Fig. 4. Original C code vs code instrumented.

3.3 Produce Counterexamples

FuSeBMC uses its verification engines to generate test cases that can reach goals amongst $GOAL_1$, $GOAL_2$, ..., $GOAL_n$ inserted in the previous phase.

FuSeBMC then checks whether all goals within the C program are covered. If so, *FuSeBMC* continues to the next phase; otherwise, *FuSeBMC* passes the goals that are not covered to the selective fuzzer to produce test cases for it using randomly generated inputs learned from the test cases produced from both engines. Figure 5 illustrates how the method works.

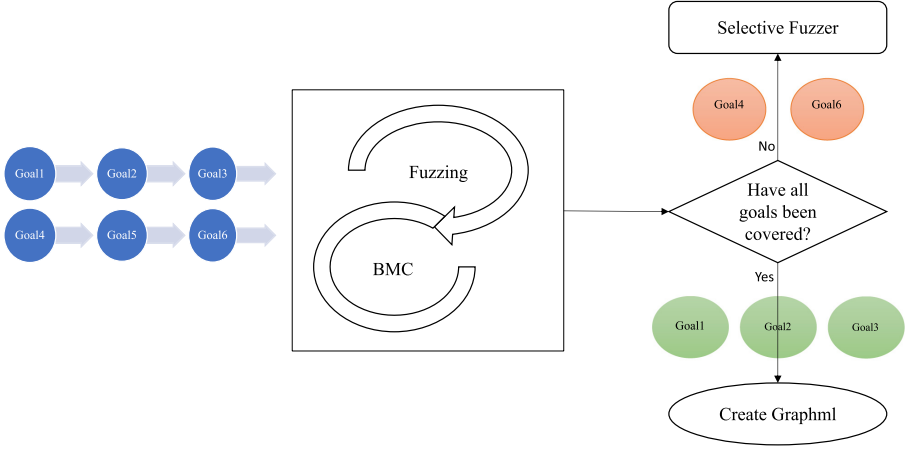


Fig. 5. Produce counterexamples.

3.4 Create Graphml

FuSeBMC will generate a *graphml* for each goal injected and then name it. The name of the *graphml* takes the number of the goal extended by the *graphml* extension, e.g., (*GOAL1.graphml*). The *graphml* file contains data about the counterexample, such as data types, values, and line numbers for the variables, which will be used to obtain the values of the target variable.

3.5 Produce Test Cases

In this phase, *FuSeBMC* will analyze all the *graphml* files produced in the previous phase. Practically, *FuSeBMC* will focus on the `<edge>` tags in the *graphml* that refer to the variable with a type non-deterministic. These variables will store their value in a file called, for example, (*testcase1.xml*). Figure 6 illustrates the edges and values used to create the test cases.

3.6 Selective Fuzzer

In this phase, we apply the selective fuzzer to learn from the test cases produced by either fuzzing or BMC engines to produce test cases for the goals that have

```

1 <edge id="E2" source="N2" target="N3">
2 <data key="startline">3</data>
3 <data key="assumption"> a = -2147483647;</data>
4 <data key="threadId">0</data>
5 </edge>
6
7 <edge id="E4" source="N4" target="N5">
8 <data key="startline">4</data>
9 <data key="assumption">b = 0;</data>
10 <data key="threadId">0</data>
11 </edge>

```

Fig. 6. An example of target edges

not been covered by the two. The selective fuzzer uses the previously produced test cases by extracting from each the number of assignments required to reach an error. For example, in Fig. 7, we assumed that the 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) based on the test case produced by the fuzzer. In several cases, the BMC engine can exhaust the time limit before providing the information needed by the selective fuzzer, such as the number of inputs, when large arrays need to be initialized at the beginning of the program.

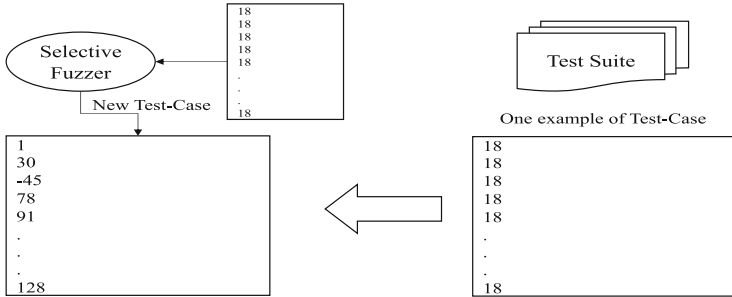


Fig. 7. The selective fuzzer

3.7 Test Validator

The test validator takes as input the test cases produced by *FuSeBMC* and then validates these test cases by executing the program on all test cases. The test validator checks whether the bug is exposed if the test was bug-detection, and it reports the code coverage if the test was a measure of the coverage. In our experiments, we use the tool TESTCOV [37] as a test validator. The tool provides coverage statistics per test. It supports block, branch, and condition coverage and covering calls to an error function. TESTCOV uses the XML-based exchange format for test cases specifications defined by Test-Comp [16].

TESTCOV was successfully used in recent editions of Test-Comp 2019, 2020, and 2021 to execute almost 9 million tests on 1720 different programs [37].

4 Evaluation

4.1 Description of Benchmarks and Setup

We conducted experiments with *FuSeBMC* on the benchmarks of Test-Comp 2021 [39] to check the tool’s ability in the previously mentioned criteria. Our evaluation 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 [40]. 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.

The experiments were conducted on the server of Test-Comp 2021 [39]. Each run was limited to 8 processing units, 15 GB of memory, and 15 min of CPU time. The test suite validation was limited to 2 processing units, 7 GB of memory, and 5 min of CPU time. Also, the machine had the following specification of the test node was: one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86-64-Linux, Ubuntu 20.04 with Linux kernel 5.4).

FuSeBMC source code is written in C++; it is available for downloading at GitHub,² 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*.

4.2 Objectives

This evaluation’s main goal is to check the performance of *FuSeBMC* and the system’s suitability for detecting security vulnerabilities in open-source C programs. Our experimental evaluation aims to answer three experimental goals:

- EG1 (**Security Vulnerability Detection**) Can *FuSeBMC* generate test cases that lead to more security vulnerabilities than state-of-the-art software testing tools?
- EG2 (**Coverage Capacity**) Can *FuSeBMC* achieve a higher coverage when compared with other state-of-the-art software testing tools?
- EG3 (**Low Energy Consumption**) Can *FuSeBMC* reduce the consumption of CPU and memory compared with the state-of-the-art tools?

² <https://github.com/kaled-alshmrany/FuSeBMC>.

4.3 Results

First, we evaluated *FuSeBMC* on the *Error Coverage* category. Table 1 shows the experimental results compared with other tools in Test-Comp 2021 [39], where *FuSeBMC* achieved the 1st place in this category by solving 500 out of 607 tasks, an 82% success rate.

In detail, *FuSeBMC* achieved 1st place in the subcategories *ReachSafety-BitVectors*, *ReachSafety-Floats*, *ReachSafety-Recursive*, *ReachSafety-XCSP* and *ReachSafety-Sequentialized*. *FuSeBMC* solved 10 out of 10 tasks in *ReachSafety-BitVectors*, 32 out of 33 tasks in *ReachSafety-Floats*, 19 out of 20 tasks in *ReachSafety-Recursive*, 53 out of 59 tasks in *ReachSafety-XCSP* and 101 out of 107 tasks in *ReachSafety-Sequentialized*.

FuSeBMC outperformed the top tools in Test-Comp 2021, such as KLEE [19], CPAchecker [20], Symbiotic [41], LibKluzzer [15], and VeriFuzz [14] in these subcategories. However, *FuSeBMC* did not perform as well in the *ReachSafety-ECA* subcategory if compared with leading tools in the competition. We suspect that this is due to the prevalence of nested branches in these benchmarks. The *FuSeBMC*'s verification engines and the selective fuzzer could not produce test cases to reach the error due to the existence of too many path conditions, making the logical formula hard to solve and making it difficult to create random inputs to reach the error.

Overall, the results show that *FuSeBMC* produces test cases that detect more security vulnerabilities in C programs than state-of-the-art tools, which successfully answers **EG1**.

FuSeBMC also participated in the *Branch Coverage* category at Test-Comp 2021. Table 2 shows the experimental results from this category. *FuSeBMC* achieved 4th place in the category by successfully achieving a score of 1161 out of 2566, behind the 3rd place system by 8 scores only. In the subcategory *ReachSafety-Floats*, *FuSeBMC* obtained the first place by achieving 103 out of 226 scores. Thus, *FuSeBMC* outperformed the top tools in Test-Comp 2021. Further, *FuSeBMC* obtained the first place in the subcategory *ReachSafety-XCSP* by achieving 97 out of 119 scores. However, *FuSeBMC* did not perform well in the subcategory *ReachSafety-ECA* compared with the leading tools in the Test-Comp 2021. Again we suspect the cause to be the prevalence of nested branches in these benchmarks.

These results validate **EG2**. *FuSeBMC* proved its capability in *Branch Coverage* category, especially in the subcategories *ReachSafety-Floats* and *ReachSafety-XCSP*, where it ranked first.

FuSeBMC achieved 2nd place overall at Test-Comp 2021, with a score of 1776 out of 3173. Table 4 and Fig. 8 shows the overall results compared with other tools in the competition. Overall, *FuSeBMC* performed well compared

Table 1. *Cover-Error* results^a. We identify the best for each tool in bold.

Cover-Error	Task-Num	FuSeBMC	CMA-ES Fuzz	CoVeriTest	HybridTiger	KLEE	Legion	LibKluzzer	PRTest	Symbiotic	Tracer-X	VeriFuzz
ReachSafety-Arrays	100	93	0	59	69	88	67	96	11	73	75	95
ReachSafety-BitVectors	10	10	0	8	6	9	0	9	5	8	7	9
ReachSafety-ControlFlow	32	8	0	8	8	10	0	11	0	7	9	9
ReachSafety-ECA	18	8	0	2	1	14	0	11	0	15	2	16
ReachSafety-Floats	33	32	0	16	22	6	0	30	3	0	0	30
ReachSafety-Heap	57	45	0	37	38	46	0	47	9	47	44	47
ReachSafety-Loops	158	131	0	35	53	96	4	138	102	82	78	136
ReachSafety-Recursive	20	19	0	0	5	16	0	17	1	17	14	13
ReachSafety-Sequentialized	107	101	0	61	93	86	0	83	0	79	57	99
ReachSafety-XCSP	59	53	0	46	52	37	0	3	0	41	31	25
SoftwareSystems-BusyBox-MemSafety	11	0	0	0	0	0	0	0	0	0	0	0
DeviceDriversLinux64-ReachSafety	2	0	0	0	0	0	0	0	0	0	0	0
Overall	607	405	0	225	266	339	35	359	79	314	246	385

^a<https://test-comp.sosy-lab.org/2021/results/results-verified/>.

Table 2. *Cover-Branches* results^a. We identify the best for each tool in bold.

Cover-Branches	Task-Num	FuSeBMC	CMA-ES Fuzz	CoVeriTest	HybridTiger	KLEE	Legion	LibKluzzer	PRTest	Symbiotic	Tracer-X	VeriFuzz
ReachSafety-Arrays	400	284	139	229	225	96	195	296	119	226	223	295
ReachSafety-BitVectors	62	37	23	39	13	28	29	40	27	37	37	38
ReachSafety-ControlFlow	67	15	4	16	3	8	8	16	5	18	15	18
ReachSafety-ECA	29	5	0	6	2	7	3	10	2	10	7	12
ReachSafety-Floats	226	103	51	98	84	16	64	90	41	50	48	99
ReachSafety-Heap	143	88	19	79	74	81	69	90	40	84	86	86
ReachSafety-Loops	581	412	152	402	338	274	271	419	252	383	385	424
ReachSafety-Recursive	53	36	19	31	31	18	20	36	9	38	34	35
ReachSafety-Sequentialized	82	62	0	61	39	26	1	55	8	36	41	71
ReachSafety-XCSP	119	97	0	80	80	81	2	80	79	93	69	88
ReachSafety-Combinations	210	15	0	31	8	82	18	139	2	135	99	180
SoftwareSystems-BusyBox-MemSafety	72	1	0	5	4	6	0	6	4	7	4	8
DeviceDriversLinux64-ReachSafety	290	35	13	60	6	25	56	58	16	44	56	57
SoftwareSystemsSQLite-MemSafety	1	0	0	0	0	0	0	0	0	0	0	0
Termination-MainHeap	231	202	138	193	189	119	166	199	51	178	185	204
Overall	2566	1161	411	1128	860	784	651	1292	519	1169	1087	1389

^a<https://test-comp.sosy-lab.org/2021/results/results-verified/>.

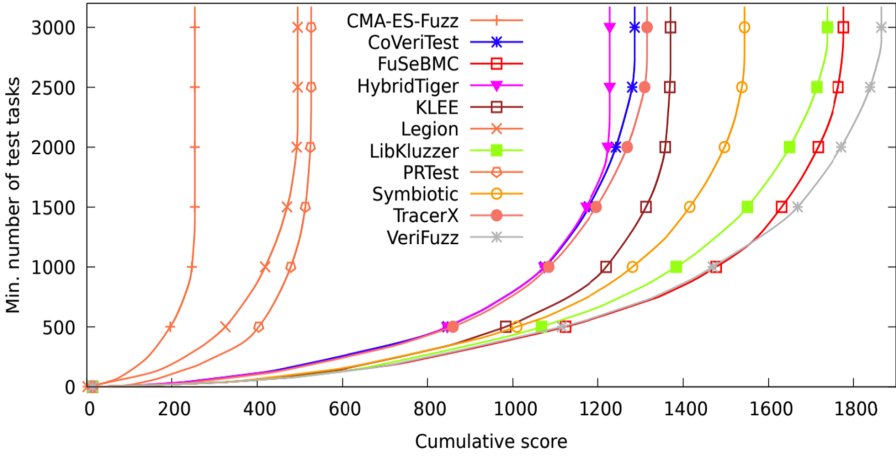


Fig. 8. Quantile functions for category *Overall*. [8]

Table 3. The consumption of CPU and memory [8].

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

with top tools in the subcategories *ReachSafety-Bit Vectors*, *ReachSafety-Floats*, *ReachSafety-Recursive*, *ReachSafety-Sequentialized* and *ReachSafety-XCSP*.

Test-Comp 2021 also considers 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 3). *FuSeBMC* consumed less energy than many 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 [42] for measuring the energy.

These experimental results showed that *FuSeBMC* could reduce the consumption of CPU and memory efficiently and effectively in C programs, which answers **EG3**.

Table 4. Test-Comp 2021 *Overall* results^a.

Cover-Error and Branches	Task-Num	<i>FuSeBMC</i>	CMA-ES Fuzz	CoVeriTest	HybridTiger	KLEE	Legion	LibKluzzer	PRTTest	Symbiotic	Tracer-X	VeriFuzz
OVERALL	3173	1776	254	1286	1228	1370	495	1738	526	1543	1315	1865

^a<https://test-comp.sosy-lab.org/2021/results/results-verified/>.

5 Related Work

For more than 20 years, software vulnerabilities have been mainly identified by fuzzing [43]. American fuzzy lop (AFL) [44, 45] is a tool that aims to find software vulnerabilities. AFL increases the coverage of test cases by utilizing genetic algorithms (GA) with guided fuzzing. Another fuzzing tool is LibFuzzer [46]. LibFuzzer generates test cases by using code coverage information provided by LLVM’s Sanitizer Coverage instrumentation. It is best used for programs with small inputs that have a run-time of less than a fraction of a second for each input as it is guaranteed not to crash on invalid inputs. AutoFuzz [47] is a tool that verifies network protocols using fuzzing. First, it determines the specification for the protocol, then utilizes fuzzing to find vulnerabilities. Additionally, Peach [48] is an advanced and robust fuzzing framework that provides an XML file to create a data model and state model definition.

Symbolic execution has also been used to identify security vulnerabilities. One of the most popular symbolic execution engines is KLEE [19]. It is built on top of the LLVM compiler infrastructure and employs dynamic symbolic execution to explore the search space path-by-path. KLEE has proven to be a reliable symbolic execution engine for its utilization in many specialized tools such as TracerX [49] and Map2Check [33] for software verification, also SymbexNet [50] and SymNet [51] for verification of network protocols implementation.

The combination of symbolic execution and fuzzing has been proposed before. It started with the tool that earned first place in Test-Comp 2020 [18], VeriFuzz [14]. VeriFuzz is a state-of-the-art tool we have compared to *FuSeBMC*. It is a program-aware fuzz tester that combines the power of feedback-driven evolutionary fuzz testing with static analysis. It is built based on grey-box fuzzing to exploit lightweight instrumentation for observing the behaviors that occur during test runs. There is also LibKluzzer [15], which is a novel implementation that combines the strengths of coverage-guided fuzzing and white-box fuzzing. LibKluzzer is a combination of LibFuzzer and an extension of KLEE called KLUZZER [52]. Driller [53] is a hybrid vulnerability excavation tool, which leverages fuzzing and selective concolic execution in a complementary manner to find deeply embedded bugs. The authors avoid the path explosion inherent in concolic analysis and the incompleteness of fuzzing by combining the two techniques’ strengths and mitigating the weaknesses. Driller splits the application into *compartments* based on checks of particular values of a specific input. The proficiency of fuzzing allows it

to explore possible values of general input in a compartment. However, when it comes to values that satisfy checks on an input that guides the execution between *compartments*, fuzzing struggles to identify such values. In contrast, selective concolic execution excels at identifying such values required by checks and drive the execution between *compartments*.

Another example is hybrid fuzzer [54], which provides an efficient way to generate provably random test cases that guarantee the execution of unique paths. It uses symbolic execution to determine frontier nodes that lead to a unique execution path. Given some resource constraints, the tool collects as many frontier nodes as possible. With these nodes, fuzzing is employed with provably random input, preconditioned to lead to each frontier node. Badger [55] is a hybrid testing approach for complexity analysis. It uses Symbolic PathFinder [56] to generate new inputs and provides the Kelinci fuzzer with worst-case analysis. Munch [57] is a hybrid tool introduced to increase function coverage. It employs fuzzing with seed inputs generated by symbolic execution and targets symbolic execution when fuzzing saturates. SAGE (Scalable Automated Guided Execution) [58] is a hybrid fuzzer developed at Microsoft Research. It extends dynamic symbolic execution with a generational search; it negates and solves the path predicates to increase the code coverage. SAGE is used extensively at Microsoft, where it has been successful at finding many security-related bugs. SAFL [59] is an efficient fuzzer for C/C++ programs. It generates initial seeds that can get an appropriate fuzzing direction by employing symbolic execution in a lightweight approach. He et al. [60] describe a new approach for learning a fuzzer from symbolic execution; they instantiated it to the domain of smart contracts. First, it learns a fuzzing policy using neural networks. Then it generates inputs for fuzzing unseen smart contracts by this learning fuzzing policy. In summary, many tools have combined fuzzers with BMC and symbolic execution to perform software verification. However, our approach’s novelty lies with the addition of the selective fuzzer and time management algorithm between engines and goals. These features were what distinguished *FuSeBMC* from other tools at Test-Comp 2021.

6 Conclusions and Future Work

We proposed a novel test case generation approach that combined Fuzzing and BMC and implemented it in the *FuSeBMC* tool. *FuSeBMC* explores and analyzes the target C programs by incrementally injecting labels to guide the fuzzing and BMC engines to produce test cases. We inject labels in every program branch to check for their reachability, producing test cases if these labels are reachable. We also exploit the selective fuzzer to produce test cases for the labels that fuzzing and BMC could not produce test cases. *FuSeBMC* achieved two significant awards from Test-Comp 2021. First place in the *Cover-Error* category and second place in the *Overall* category. *FuSeBMC* outperformed the leading state-of-the-art tools because of two main factors. Firstly, the usage of the selective fuzzer as a third engine that learns from the test cases of fuzzing/BMC to produce new test cases for the as-yet uncovered goals. Overall, it substantially

increased the percentage of successful tasks. Secondly, we apply a novel algorithm of managing the time allocated for each engine and goal. This algorithm prevents *FuSeBMC* from wasting time finding test cases for difficult goals so that if the fuzzing engine is finished 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. As a result, *FuSeBMC* raised the bar for the competition, thus advancing state-of-the-art software testing. Future work will investigate the extension of *FuSeBMC* to test multi-threaded programs [61, 62] and reinforcement learning techniques to guide our selective fuzzer to find test cases that path-based fuzzing and BMC could not find.

A Appendix

A.1 Artifact

We have set up a zenodo entry that contains the necessary materials to reproduce the results given in this paper: <https://doi.org/10.5281/zenodo.4710599>. Also, it contains instructions to run the tool.

A.2 Tool Availability

FuSeBMC contents are publicly available in our repository in GitHub under the terms of the MIT License. *FuSeBMC* provides, besides other files, a script called *fusebmc.py*. In order to run our *fusebmc.py* script, 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. *FuSeBMC* participated in the 3rd international competition, Test-Comp 21, and met all the requirements each tool needs to meet to qualify and participate. The results in our paper are also available on the Test-Comp 21 website. Finally, instructions for building *FuSeBMC* from the source code are given in the file README.md in our GitHub repository, including the description of all dependencies.

A.3 Tool Setup

FuSeBMC is available to download from the link.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.

³ <https://doi.org/10.5281/zenodo.4710599>.

References

1. Rodriguez, M., Piattini, M., Ebert, C.: Software verification and validation technologies and tools. *IEEE Softw.* **36**(2), 13–24 (2019)
2. Airbus issues software bug alert after fatal plane crash. *The Guardian*, May 2015. <https://tinyurl.com/xw67wtd9>. Accessed Mar 2021
3. Liu, B., Shi, L., Cai, Z., Li, M.: Software vulnerability discovery techniques: a survey. In: 2012 Fourth International Conference on Multimedia Information Networking and Security, pp. 152–156. IEEE (2012)
4. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking, pp. 196–215 (2008)
5. Godefroid, P.: Fuzzing: hack, art, and science. *Commun. ACM* **63**(2), 70–76 (2020)
6. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Meyer, B., Nordio, M. (eds.) LASER 2011. LNCS, vol. 7682, pp. 1–30. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35746-6_1
7. Shameng, W., Feng Chao, E.A.: Testing network protocol binary software with selective symbolic execution. In: CIS, pp. 318–322. IEEE (2016)
8. Beyer, D.: 3rd competition on software testing (test-comp 2021) (2021)
9. Miller, B.P., et al.: Fuzz revisited: a re-examination of the reliability of UNIX utilities and services. Technical report, UW-Madison (1995)
10. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (1976)
11. Faria, J.: Inspections, revisions and other techniques of software static analysis. *Software Testing and Quality, Lecture*, vol. 9 (2008)
12. Qin, S., Kim, H.S.: LIFT: a low-overhead practical information flow tracking system for detecting security attacks. In: MICRO 2006, pp. 135–148. IEEE (2006)
13. Ognawala, S., Kilger, F., Pretschner, A.: Compositional fuzzing aided by targeted symbolic execution. arXiv preprint [arXiv:1903.02981](https://arxiv.org/abs/1903.02981) (2019)
14. Basak Chowdhury, A., Medicherla, R.K., Venkatesh, R.: VeriFuzz: program aware fuzzing. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 244–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_22
15. Le, H.M.: LLVM-based hybrid fuzzing with LibKluzzer (competition contribution). In: FASE, pp. 535–539 (2020)
16. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) *Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 457–481. IOS Press (2009)
17. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.* **38**(4), 957–974 (2012)
18. Beyer, D.: Second competition on software testing: Test-Comp 2020. In: Wehrheim, H., Cabot, J. (eds.) *Fundamental Approaches to Software Engineering. FASE 2020*. LNCS, vol. 12076, pp. 505–519. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45234-6_25
19. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8, pp. 209–224 (2008)

20. Beyer, D., Keremoglu, M.E.: CPACHECKER: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
21. Alshmrany, K.M., Menezes, R.S., Gadelha, M.R., Cordeiro, L.C.: FuSeBMC: a white-box fuzzer for finding security vulnerabilities in c programs. In: 24th International Conference on Fundamental Approaches to Software Engineering (FASE), vol. 12649, pp. 363–367 (2020)
22. Munea, T.L., Lim, H., Shon, T.: Network protocol fuzz testing for information systems and applications: a survey and taxonomy. *Multimedia Tools Appl.* **75**(22), 14745–14757 (2016)
23. Wang, J., Guo, T., Zhang, P., Xiao, Q.: A model-based behavioral fuzzing approach for network service. In: 2013 Third International Conference on IMCCC, pp. 1129–1134. IEEE (2013)
24. Baldoni, R., Coppa, E., D’elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. *ACM Comput. Surv.* **51**, 1–39 (2018)
25. Chipounov, V., Georgescu, V., Zamfir, C., Candea, G.: Selective symbolic execution. In: Proceedings of the 5th Workshop on HotDep (2009)
26. Black, P.E., Bojanova, I.: Defeating buffer overflow: a trivial but dangerous bug. *IT Prof.* **18**(6), 58–61 (2016)
27. Zhang, S., Zhu, J., Liu, A., Wang, W., Guo, C., Xu, J.: A novel memory leak classification for evaluating the applicability of static analysis tools. In: 2018 IEEE International Conference on Progress in Informatics and Computing (PIC), pp. 351–356. IEEE (2018)
28. Jimenez, W., Mammari, A., Cavalli, A.: Software vulnerabilities, prevention and detection methods: a review. In: Security in Model-Driven Architecture, vol. 215995, p. 215995 (2009)
29. Boudjema, E.H., Faure, C., Sassolas, M., Mokdad, L.: Detection of security vulnerabilities in C language applications. *Secur. Priv.* **1**(1), e8 (2018)
30. US-CERT: Understanding denial-of-service attacks | CISA (2009)
31. Cisco: Cisco IOS XE software cisco discovery protocol memory leak vulnerability (2018)
32. Clang documentation (2015). <http://clang.llvm.org/docs/index.html>. Accessed Aug 2019
33. Rocha, H.O., Barreto, R.S., Cordeiro, L.C.: Hunting memory bugs in C programs with Map2Check. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 934–937. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_64
34. Rocha, H., Menezes, R., Cordeiro, L.C., Barreto, R.: Map2Check: using symbolic execution and fuzzing. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2020. LNCS, vol. 12079, pp. 403–407. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_29
35. Gadelha, M.R., Monteiro, F., Cordeiro, L., Nicole, D.: ESBMC v6.0: verifying C programs using k -induction and invariant inference. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 209–213. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_15
36. 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: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 888–891 (2018)

37. Beyer, D., Lemberger, T.: TestCov: robust test-suite execution and coverage measurement. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1074–1077. IEEE (2019)
38. Lopes, B.C., Auler, R.: Getting started with LLVM core libraries. Packt Publishing Ltd. (2014)
39. Beyer, D.: Status report on software testing: Test-Comp 2021. In: Guerra, E., Stoelinga, M. (eds.) *Fundamental Approaches to Software Engineering FASE 2021*. LNCS, vol. 12649, pp. 341–357. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71500-7_17
40. Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Groote, J.F., Larsen, K.G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2021*. LNCS, vol. 12652, pp. 401–422. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_24
41. Chalupa, M., Novák, J., Strejček, J.: SYMBIOTIC 8: parallel and targeted test generation. *FASE 2021*. LNCS, vol. 12649, pp. 368–372. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71500-7_20
42. Beyer, D., Wendler, P.: CPU energy meter: a tool for energy-aware algorithms engineering. *TACAS 2020*. LNCS, vol. 12079, pp. 126–133. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_8
43. Barton, J.H., Czeck, E.W., Segall, Z.Z., Siewiorek, D.P.: Fault injection experiments using fiat. *IEEE Trans. Comput.* **39**(4), 575–582 (1990)
44. Böhme, M., Pham, V.-T., Nguyen, M.-D., Roychoudhury, A.: Directed greybox fuzzing. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344 (2017)
45. American fuzzy lop (2021). <https://lcamtuf.coredump.cx/afl/>
46. Serebryany, K.: libFuzzer-a library for coverage-guided fuzz testing. LLVM project (2015)
47. Gorbunov, S., Rosenbloom, A.: AutoFuzz: automated network protocol fuzzing framework. *IJCSNS* **10**(8), 239 (2010)
48. Eddington, M.: Peach fuzzing platform. *Peach Fuzzer*, vol. 34 (2011)
49. Jaffar, J., Maghareh, R., Godbole, S., Ha, X.-L.: TracerX: dynamic symbolic execution with interpolation (competition contribution). In: *FASE*, pp. 530–534 (2020)
50. Song, J., Cadar, C., Pietzuch, P.: SymbexNet: testing network protocol implementations with symbolic execution and rule-based specifications. In: *IEEE TSE*, vol. 40, no. 7, pp. 695–709 (2014)
51. Sasnauskas, R., Kaiser, P., Jukić, R.L., Wehrle, K.: Integration testing of protocol implementations using symbolic distributed execution. In: *ICNP*, pp. 1–6. IEEE (2012)
52. Le, H.M.: LLVM-based hybrid fuzzing with LibKluzzer (competition contribution). In: Wehrheim, H., Cabot, J. (eds.) *Fundamental Approaches to Software Engineering. FASE 2020*. LNCS, vol. 12076, pp. 535–539. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45234-6_29
53. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: *NDSS*, pp. 1–16 (2016)
54. Pak, B.S.: Hybrid fuzz testing: discovering software bugs via fuzzing and symbolic execution. School of Computer Science Carnegie Mellon University (2012)
55. Noller, Y., Kersten, R., Păsăreanu, C.S.: Badger: complexity analysis with fuzzing and symbolic execution. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 322–332 (2018)

56. Pășăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of Java bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 179–180 (2010)
57. Ognawala, S., Hutzelmann, T., Psallida, E., Pretschner, A.: Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pp. 1475–1482 (2018)
58. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. *Queue* **10**(1), 20–27 (2012)
59. Wang, M., et al.: SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, pp. 61–64 (2018)
60. He, J., Balunović, M., Ambroladze, N., Tsankov, P., Vechev, M.: Learning to fuzz from symbolic execution with application to smart contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 531–548 (2019)
61. Cordeiro, L.C.: SMT-based bounded model checking for multi-threaded software in embedded systems. In: International Conference on Software Engineering, pp. 373–376. ACM (2010)
62. Pereira, P.A., et al.: Verifying CUDA programs using SMT-based context-bounded model checking. In: Ossowski, S. (ed.) Annual ACM Symposium on Applied Computing, pp. 1648–1653. ACM (2016)