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

Kaled Alshmrany, Mohannad Aldughaim, Ahmed Bhayat and Lucas Cordeiro

March 9, 2021

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

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. We describe and evaluate a novel approach *FuSeBMC* that exploits fuzzing and BMC engines to detect security vulnerability in C programs. It explores and analyzes the target C program by injecting labels that guide those engines to produce test-cases. *FuSeBMC* also exploits 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. As a result, *FuSeBMC* guides the fuzzing and BMC engines to explore more profound in the target C programs and then produce test-cases that achieve higher coverage with lower energy consumption to detect bugs efficiently. We evaluated *FuSeBMC* by participating in Test-Comp 2021 to test the ability of the tool in two categories of the competition, which are *code coverage* and *bug detection*. The competition results show that *FuSeBMC* performs well if 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*.

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 individuals' well-being, software verification became 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. This year's competition, Test-Comp 2021 [8], has two categories *Error Coverage* (or *Cover-Error*) and *Branch Coverage* (or *Cover-Branches*). *Error Coverage* category tests the tool's ability to discover bugs where every C program in the benchmarks contains a bug. *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-suites[3]. 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 named *FuSeBMC* that combines Fuzzing with Symbolic Execution via Bounded Model Checking for detecting security vulnerabilities in C programs. In particular, we use two approaches for verifying C programs. The first one exploits coverage-guided fuzzing to produce random inputs to locate security vulnerabilities in C programs. The second one is based on BMC techniques [16,17]. BMC unfolds a software system 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 resulting formula using SMT solvers. Thus, *FuSeBMC* relies on efficient fuzzing and BMC techniques; it can handle two main features in software testing: *bug detection* and *code coverage*, as defined by Beyer et al. [18]. As a result, our proposed method *FuSeBMC* combines fuzzing and symbolic execution

[3] https://gitlab.com/sosy-lab/software/test-format/

via BMC techniques. We also manage each engine's execution time to improve *FuSeBMC*'s efficiency. 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 describe the details of *FuSeBMC* that guides fuzzing and BMC to produce test-cases that can detect security vulnerabilities, achieve high code coverage, and massively reduce the consumption of both CPU and memory. Furthermore, we employ selective fuzzer as a third engine, where it learns from the test-cases of fuzzing/BMC to produce new test-cases for the uncovered goals to raise the chance of detecting bugs and code coverage.
- FuSeBMC successfully participated in Test-Comp 2021 and achieved first place in the *Cover-Error* category and second place in the *Overall* category. Furthermore, in the subcategories *ReachSafety-BitVectors, ReachSafety-Floats, ReachSafety-Recursive, ReachSafety-Sequentialized and ReachSafety-XCSP*, *FuSeBMC* obtained first place in all these subcategories. Also, *FuSeBMC* shows the ability to achieve high code coverage competitively if compared with other state-of-the-art software testing tools.

2 Preliminaries

2.1 Fuzzing

Fuzzing is a software testing technique to exploit vulnerabilities in software systems [22]. Fuzzing prepares random or semi-random inputs to the target C program. Critical security flaws most often occur because program inputs are not adequately checked [23]. Since these inputs are random, their unexpected and improper appearance in a target C program is highly probable. If the target C program does not reject these improper inputs, it will hang or crash during fuzz testing. Fuzzing is a quick and cost-effective method for locating security vulnerabilities in C programs. 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 tools. 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 is 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. In 2013, DARPA announced a two-year competition titled Cyber Grand Challenge [26]. In this competition, participants are to create tools that automatically detect vulnerabilities and exploitation. This competition motivated researchers to advance state-of-the-art of software testing by utilizing symbolic execution.

2.3 Types of Vulnerabilities

The software, in general, is often prone to vulnerabilities caused by developer mistakes, which include: *buffer overflow*, where a running program attempts to write data outside the memory buffer, which is not intended to store this data [27]; *memory leak*, which occurs when programmers create a memory in a heap and forget to delete it [28]; *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 [29]. Additionally, *string manipulation*, when the string may contain malicious code and is accepted as an input; this is reasonably common in the C programming language [30]. *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 [31]. 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 [32].

3 FuSeBMC: A White-Box Fuzzer 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 [33] to instrument the C program, uses Map2check [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 the C programs and the specifications as input. Then, *FuSeBMC* invokes the fuzzing and BMC engines sequentially for the *Cover-Error* category to find a path that violates a given property. It uses an iterative BMC approach that incrementally unwinds the program until it finds a property violation or exhausts time or memory limits. As a result, *FuSeBMC* uses incremental BMC to explore the program state space, searching for a property violation since all programs in Test-Comp 2021 are known to have errors. In *Cover-Branches* category, *FuSeBMC* explores and analyzes the target C program using the clang compiler to inject labels incrementally. *FuSeBMC* will compute all C code branches and inject the labels for each branch by adding the label $GOAL_N$, where N is the goal number. Then,

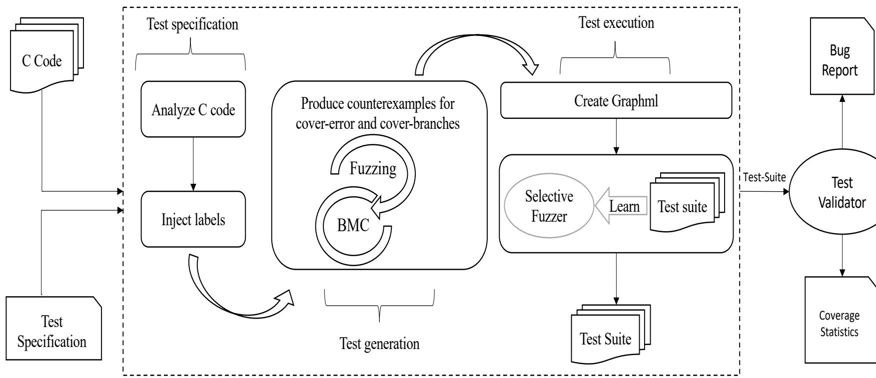

Fig. 1: *FuSeBMC*: White-Box Fuzzing Framework for C Programs.

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 named selective fuzzer (cf. Section 3.6), which produces test-cases for the rest of the labels. The selective fuzzer produces test-cases by learning from the two engines' output: it analyzes the range of the inputs that should be passed to examine the target C program and then produces different test-cases.

FuSeBMC also manages the run-time of the fuzzing, BMC, and selective fuzzer engines to 150s, 700s, and 50s, respectively. *FuSeBMC* further manages the time allocated for each engine. 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. Lastly, 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*.

Fig 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 $\langle entryfunction\rangle$, the coverage criterion for the test suite $\langle specification\rangle$, the programming language of the program under test $\langle sourcecodelang\rangle$, the system architecture the program tests were created for $\langle architecture\rangle$, the creation time $\langle creationtime\rangle$, the SHA-256 hash of the program under test $\langle programhash\rangle$, the producer of counterexample $\langle producer\rangle$ and the name of the target program $\langle programfile\rangle$. A test-case file contains a sequence of tags $\langle input\rangle$ that describes the input values sequence. Fig 3 illustrates an example of the test-case file.

Algorithm 1 describes our algorithm 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 for that goal (line 7).

If our engines find that the property is violated, meaning that there is a valid execution path that reaches the goal, then the goals are marked as found, and the test-case is saved for later (lines 8-10). After that, if our verification engines could not reach some goals, then we employ a selective fuzzer, i.e., we generate random inputs and check whether these inputs can reach those goals (lines 13-18). As a result, the generation of values still depends on the program's internal structure. In the end, we return all test-cases for all the goals we have found in the specified XML format (line 21).

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: **for all** $G \in goals$ **do**
6:     $\phi \leftarrow REACH(G)$
7:     $result, test\_case \leftarrow Fuzzing/BMC(instrumentedP, \phi)$
8:     **if** $result = false$ **then**
9:         $reached\_goals \leftarrow reached\_goals \cup G$
10:         $tests \leftarrow tests \cup test\_case$
11:     **end if**
12: **end for**
13: **for all** $G \in (goals - reached\_goals)$ **do**
14:     $\phi \leftarrow REACH(G)$
15:     $result \leftarrow selectivefuzzer(instrumentedP, \phi)$
16:     **if** $result = false$ **then**
17:         $reached\_goals \leftarrow reached\_goals \cup G$
18:         $tests \leftarrow tests \cup test\_case$
19:     **end if**
20: **end for**
21: **return** $tests$

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

```
<?xml version='1.0'>
<!DOCTYPE test-metadata PUBLIC [...]>
<test-metadata>
  <entryfunction>main</entryfunction>
  <specification>COVER(init(main()), FQL(COVER EDGES(@DECISIONEDGE)))
    </specification>
  <sourcecodelang>C</sourcecodelang>
  <architecture>32bit</architecture>
  <creationtime>2021-02-28 20:44:56.117416</creationtime>
  <programhash>e8f2cf545726d8f791bfc137e9eca7e9de4cb696</programhash>
  <producer>FuSeBMC</producer>
  <programfile>sv-benchmarks/c/array-tiling/skippedu.c</programfile>
</test-metadata>
```

Fig. 2: An example of a metadata.

```
<?xml version="1.0"?>
<!DOCTYPE testcase PUBLIC [...]>
<testcase>
  <input>2</input>
  <input>1</input>
  <input>128</input>
  <input>0</input>
  <input>0</input>
  <input>1</input>
  <input>64</input>
  <input>0</input>
  <input>0</input>
</testcase>
```

Fig. 3: An example of test-case file.

3.2 Inject Labels

FuSeBMC injects the labels *GOAL* 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.

3.3 Produce Counterexamples

FuSeBMC generates counterexamples for all goals (e.g., $GOAL_1$, $GOAL_2$, ..., $GOAL_n$) produced in the previous phase by our verification engines. *FuSeBMC* then checks whether it covers all the goals within the C program. If so, *FuSeBMC* continues to the next phase; otherwise, *FuSeBMC* passes the goals that are not covered to the selective fuzzer to produce the test-cases for it using randomly generated inputs learned from the test-cases produced from both engines. Fig. 5 illustrates how the method works.

```c
#include <stdio.h>
int example () {
    int x;
    if ( x < 0 ){
        //...
    }
}
```

(a) Original C code.

```c
#include <stdio.h>
int example () {
    int x;
    if ( x < 0 ){
        GOAL_1:;
        //...
    }
    else{
        GOAL_2:;
    }
    return 0;
}
```

(b) Code instrumented.

Fig. 4: Original C code vs code instrumented.

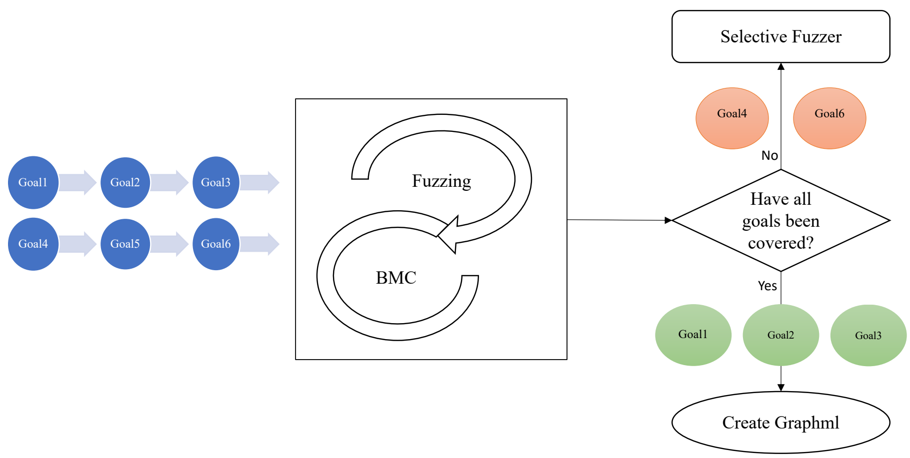

Fig. 5: Produce Counterexamples.

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 variable, which will be used to obtain the values of the target variable. Fig. 6 illustrates a *graphml* file example extracted from *FuSeBMC*.

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 edges that refer to the variable with a type non-deterministic. These variables will store their value in a file called, for example, (*testcase1.xml*). Fig. 7 illustrates the edges and values used to create the test-cases.

```
<?xml version="1.0" encoding="utf-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <key id="frontier" attr.name="isFrontierNode"
         attr.type="boolean" for="node">
    <default>false</default>
  </key>
  .
  .
  .
  <graph edgedefault="directed">
    <data key="producer">FuSeBMC</data>
    <data key="sourcecodelang">C</data>
    <data key="architecture">32bit</data>
    <data key="programfile">
    /my_instrument_outpt/instrumented.c</data>
    <data key="programhash">
    480a6e9ea7d5cd504ef15934ff3cb633121b790f</data>
    <data key="specification">CHECK( init(main()),
     LTL(G ! call(__reach_error())) )</data>
    <data key="creationtime">2021-02-28T14:07:11</data>
    <data key="witness-type">violation_witness</data>
    <node id="N1">
      <data key="entry">true</data>
    </node>
    <node id="N2">
    <edge id="E2" source="N2" target="N3">
      <data key="enterFunction">main</data>
      <data key="createThread">0</data>
    </edge>
    </node>
    .
    .
    .
    <node id="N5"/>
    <edge id="E4" source="N4" target="N5">
      <data key="startline">4</data>
      <data key="assumption">b = 0;</data>
      <data key="threadId">0</data>
    </edge>
    </node>
  </graph>
</graphml>
```

Fig. 6: An example of Graphml file

```
    <edge id="E2" source="N2" target="N3">
      <data key="startline">3</data>
      <data key="assumption"> a = -2147483647;</data>
      <data key="threadId">0</data>
    </edge>

    <edge id="E4" source="N4" target="N5">
      <data key="startline">4</data>
      <data key="assumption">b = 0;</data>
      <data key="threadId">0</data>
    </edge>
```

Fig. 7: An example of target edges

3.6 Selective Fuzzer

In this phase, our third engine Selective Fuzzer will learn from the test-cases produced by either Fuzzing or BMC engines to produce test-cases for the goals that have not been covered by the two engines Fuzzing/BMC. The test-cases information will help our selective fuzzer by providing information about the number of inputs required to trigger a property violation, i.e., the number of assignments required to reach an error. For example, in Fig. 8, 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 information about the number of inputs required to trigger a property violation, i.e., the number of assignments required to reach an error. In several cases, the BMC engine can exhaust the time limit before providing such information, e.g., when there are large arrays that need to be initialized at the beginning of the program.

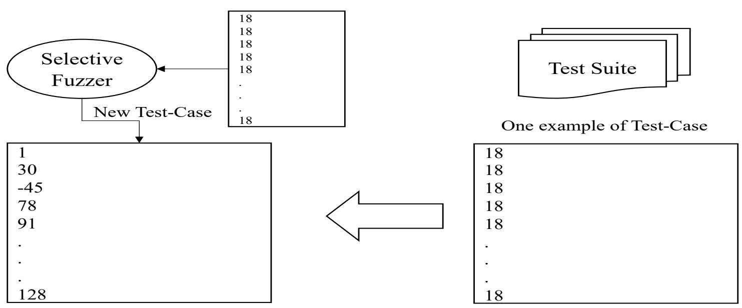

Fig. 8: The Selective Fuzzer

3.7 Test Validator

The test validator takes as input the test-cases produced by *FuSeBMC* and then validates it 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, as well as 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

FuSeBMC defines three main criteria as follows. First, the ability to detect bugs that can be evaluated by validating software against their specifications. Second, the ability to obtain high-coverage of the program compared to state-of-the-art software testing tools. Third, reducing the consumption of CPU and memory.

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,[4] 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 and suitability of *FuSeBMC* to detect security vulnerabilities in open-source C programs. Our experimental evaluation aims to answer three experimental goals:

EG1 **(Security Vulnerability Detection)** Could *FuSeBMC* detect security vulnerabilities in a target C software?
EG2 **(Coverage Capacity)** Could *FuSeBMC* achieve a higher coverage when compared with other state-of-the-art software testing tools?
EG3 **(Low Energy Consumption)** Could *FuSeBMC* reduce the consumption of CPU and memory compared with the state-of-the-art tools?

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*

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

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* could not perform that well in the *ReachSafety-ECA* subcategory if compared with top tools in the Test-Comp 2021 since these benchmarks contain too many nested branches. 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, which makes the logical formula hard to solve or difficult to create random inputs to reach the error.

> Overall, the results of *FuSeBMC* showed its efficiency in detecting bugs in different types of C programs, which successfully answers **EG1**.

Table 1: *Cover-Error* Results[5]. 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	699	**405**	0	225	266	339	35	359	79	314	246	385

Also, we applied *FuSeBMC* to the *Branch Coverage* category. Table 2 shows the experiments' results compared with other tools in the Test-Comp 2021 [39], where *FuSeBMC* achieved 4th place in this category by successfully achieving 1161 out of 2566 scores, which was behind the 3rd place by 8 scores only.

[5] https://test-comp.sosy-lab.org/2021/results/results-verified/

Practically, 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, such as KLEE [19], CPAchecker [20], Symbiotic [41], LibKluzzer [15], and VeriFuzz [14]. Further, *FuSeBMC* obtained the first place in the subcategory *ReachSafety-XCSP* by achieving 97 out of 119 scores. However, *FuSeBMC* could not perform well in the subcategory *ReachSafety-ECA* compared with top tools in the Test-Comp 2021 because of the same problem that we explained in the previous subsection.

> These results answer our **EG2**: *FuSeBMC* showed its efficiency in the *Branch Coverage* category, especially in these subcategories *ReachSafety-Floats and ReachSafety-XCSP*, where it ranked in the first place.

Table 2: *Cover-Branches* Results[6]. 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**

FuSeBMC overall results achieved 2nd place in Test-Comp 2021, achieving 1776 out of 3173 scores. Table 3 and Fig. 9 shows the overall results comparing with other tools in the competition. Overall, *FuSeBMC* performed well compared with top tools KLEE [19], CPAchecker [20], Symbiotic [41], LibKluzzer [15], and VeriFuzz [14] in the subcategories *ReachSafety-BitVectors, ReachSafety-Floats, ReachSafety-Recursive, ReachSafety-Sequentialized and ReachSafety-XCSP.*

[6] https://test-comp.sosy-lab.org/2021/results/results-verified/

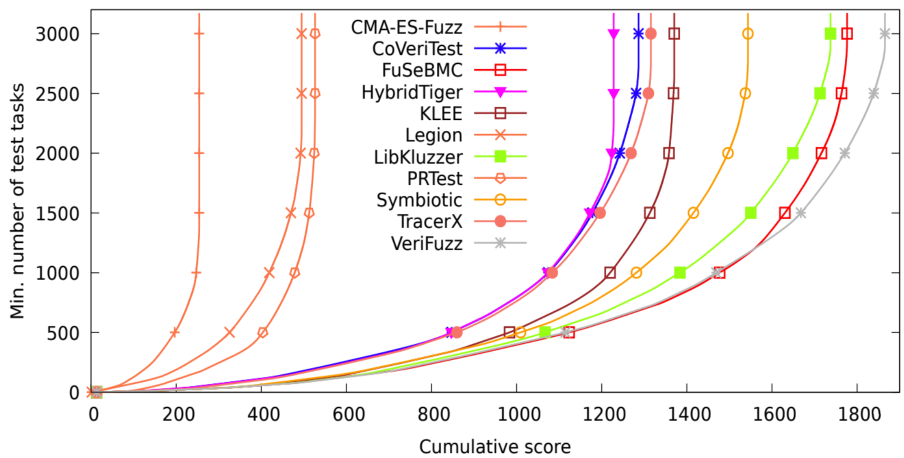

Fig. 9: Quantile functions for category *Overall.* [8]

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 Fig. 10). *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 [42] for measuring the energy.

Rank	**Test Generator**	**Quality (sp)**	**CPU Time (h)**	**CPU Energy (kWh)**	**Rank Measure**
Green Testing					**(kJ/sp)**
1	TracerX	1 315	210	2.5	6.8
2	Klee	1 370	210	2.6	6.8
3	FuSeBMC	1 776	410	4.8	9.7
worst					51

Fig. 10: The Consumption of CPU and Memory [8].

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

[7] https://test-comp.sosy-lab.org/2021/results/results-verified/

Table 3: Test-Comp 2021 *Overall* Results[7].

Cover-Error and Branches	Task-Num	*FuSeBMC*	CMA-ES Fuzz	CoVeriTest	HybridTiger	KLEE	Legion	LibKluzzer	PRTest	Symbiotic	Tracer-X	VeriFuzz
OVERALL	3230	1776	254	1286	1228	1370	495	1738	526	1543	1315	**1865**

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 [34] 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 was starting with the tool that earned first place in Test-Comp 2020 [52], VeriFuzz [14]. VeriFuzz is a state-of-the-art tool we have compared to *FuSeBMC*. It is a program-aware fuzz testing 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 [53]. LibKluzzer is one of the top state-of-the-art tools in the Test-Comp 2020 that we compared to our *FuSeBMC* approach. Driller [54] is a hybrid vulnerability excavation tool, which leverages fuzzing and selective concolic execution in a complementary manner to find bugs deeply. 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.

Another example is hybrid fuzzer [55], which provides an efficient way to generate provably random test-cases that will guarantee the unique paths' execution. Also, Badger [56], a hybrid testing approach for complexity analysis. It uses Symbolic PathFinder [57] to generate new inputs and provides the Kelinci fuzzer with worst-case analysis. Munch [58] 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) [59] 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 [60] 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. [61] describe a new approach for learning a fuzzer from symbolic execution and 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 combined fuzzers with BMC and symbolic execution to perform software verification. However, our approach's novelty lies within the combination of the selective fuzzer and time management between engines. They distinguished *FuSeBMC* from other tools and made it outperform them in Test-Comp 2021.

6 Conclusions and Future work

We proposed a novel software testing approach named *FuSeBMC* that combines Fuzzing and BMC. *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, thus producing test-cases if these labels are reachable. We also exploit selective fuzzer to produce test-cases for the labels that fuzzing and BMC could not produce test-cases. Consequently, *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 top state-of-the-art tools because of two major reasons. First, employing a selective fuzzer as a third engine learns from the test-cases of fuzzing/BMC to produce new test-cases for the uncovered goals by previous test-cases. Overall, it substantially increased the percentage of successful tasks. Second, we manage the time allocated for each engine. 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 reinforcement learning techniques to guide our selective fuzzer to find test-cases that path-based fuzzing and BMC could not find.

References

1. M. Rodriguez, M. Piattini, and C. Ebert, "Software verification and validation technologies and tools," *IEEE Software*, vol. 36, no. 2, pp. 13–24, 2019.
2. "Airbus issues software bug alert after fatal plane crash." the Guardian https://tinyurl.com/xw67wtd9, May 2015. [Online; accessed March-2021].
3. B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 fourth international conference on multimedia information networking and security*, pp. 152–156, IEEE, 2012.
4. E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," in *25 Years of Model Checking* (O. Grumberg and H. Veith, eds.), pp. 196–215, 2008.
5. P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.
6. E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," *LASER Summer School*, vol. 7682 LNCS, no. 2005, pp. 1–30, 2012.
7. W. Shameng and e. a. Feng Chao, "Testing network protocol binary software with selective symbolic execution," in *CIS*, pp. 318–322, IEEE, 2016.
8. D. Beyer, "3rd Competition on Software Testing (Test-Comp 2021)," 2021.
9. Miller and e. a. Barton, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," tech. rep., UW-Madison, 1995.
10. J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
11. J. Faria, "Inspections, revisions and other techniques of software static analysis," *Software Testing and Quality, Lecture*, vol. 9, 2008.
12. Qin, S, and K, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *MICRO'06*, pp. 135–148, IEEE, 2006.
13. S. Ognawala, F. Kilger, and A. Pretschner, "Compositional fuzzing aided by targeted symbolic execution," *arXiv preprint arXiv:1903.02981*, 2019.
14. A. B. Chowdhury, R. K. Medicherla, and R. Venkatesh, "Verifuzz: Program aware fuzzing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 244–249, Springer, 2019.
15. H. M. Le, "Llvm-based hybrid fuzzing with libkluzzer (competition contribution).," in *FASE*, pp. 535–539, 2020.
16. A. Biere, "Bounded model checking," in *Handbook of Satisfiability* (A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds.), vol. 185 of *Frontiers in Artificial Intelligence and Applications*, pp. 457–481, IOS Press, 2009.
17. L. C. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ANSI-C software," *IEEE Trans. Software Eng.*, vol. 38, no. 4, pp. 957–974, 2012.
18. D. Beyer, "Second competition on software testing: Test-comp 2020," in *FASE* (H. Wehrheim and J. Cabot, eds.), vol. 12076 of *LNCS*, pp. 505–519, Springer, 2020.
19. Cadar, Cristian, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.
20. D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*, pp. 184–190, Springer, 2011.

21. 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," *In 24th International Conference on Fundamental Approaches to Software Engineering (FASE)*, vol. 12649, pp. 363–367, 2020.
22. Munea, T. Legesse, Lim, Hyunwoo, Shon, and Taeshik, "Network protocol fuzz testing for information systems and applications: a survey and taxonomy," *Multimedia Tools and Applications*, vol. 75, no. 22, pp. 14745–14757, 2016.
23. Wang, Jiajie, T. Guo, P. Zhang, and Q. Xiao, "A model-based behavioral fuzzing approach for network service," in *2013 Third International Conference on IMCCC*, pp. 1129–1134, IEEE, 2013.
24. R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, May 2018.
25. Chipounov, Vitaly, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," in *Proceedings of the 5th Workshop on (HotDep)*, 2009.
26. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, 2016.
27. P. E. Black and I. Bojanova, "Defeating buffer overflow: A trivial but dangerous bug," *IT professional*, vol. 18, no. 6, pp. 58–61, 2016.
28. S. Zhang, J. Zhu, A. Liu, W. Wang, C. Guo, and J. Xu, "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.
29. W. Jimenez, A. Mammar, and A. Cavalli, "Software vulnerabilities, prevention and detection methods: A review1," *Security in model-driven architecture*, vol. 215995, p. 215995, 2009.
30. E. H. Boudjema, C. Faure, M. Sassolas, and L. Mokdad, "Detection of security vulnerabilities in c language applications," *Security and Privacy*, vol. 1, no. 1, p. e8, 2018.
31. US-CERT, "Understanding Denial-of-Service Attacks — CISA," 2009.
32. Cisco, "Cisco IOS XE Software Cisco Discovery Protocol Memory Leak Vulnerability," 2018.
33. "Clang documentation." http://clang.llvm.org/docs/index.html, 2015. [Online; accessed August-2019].
34. H. Rocha, R. Barreto, and L. C. Cordeiro, "Hunting memory bugs in C programs with map2check," in *Tools And Algorithms For The Construction And Analysis Of Systems*, vol. 9636 of *LNCS*, pp. 934–937, 2016.
35. M. R. Gadelha, F. Monteiro, L. Cordeiro, and D. Nicole, "Esbmc v6. 0: Verifying c programs using k-induction and invariant inference," in *International Conference on TACAS*, pp. 209–213, Springer, 2019.
36. 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 888–891, 2018.
37. D. Beyer and T. Lemberger, "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. B. C. Lopes and R. Auler, *Getting started with LLVM core libraries.* Packt Publishing Ltd, 2014.
39. D. Beyer, "Status report on software testing: Test-comp 2021," *Proc. FASE. LNCS*, vol. 12649.

40. D. Beyer, "Software verification: 10th comparative evaluation (sv-comp 2021)," *Proc. TACAS (2). LNCS*, vol. 12652.
41. Chalupa, Marek, Novák, J, Strejček, and Jan, "Symbiotic 8: Parallel and targeted test generation (competition contribution)," in *FASE*, vol. 12649 of *LNCS*, 2021.
42. D. Beyer and P. Wendler, "Cpu energy meter: A tool for energy-aware algorithms engineering," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 126–133, Springer, 2020.
43. Barton, J. H., E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using fiat," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 575–582, 1990.
44. M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, 2017.
45. 2021.
46. K. Serebryany, "libfuzzer–a library for coverage-guided fuzz testing," *LLVM project*, 2015.
47. S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *IJCSNS*, vol. 10, no. 8, p. 239, 2010.
48. M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, vol. 34, 2011.
49. J. Jaffar, R. Maghareh, S. Godboley, and X.-L. Ha, "Tracerx: Dynamic symbolic execution with interpolation (competition contribution).," in *FASE*, pp. 530–534, 2020.
50. Song, JaeSeung, C. Cadar, and P. Pietzuch, "Symbexnet: testing network protocol implementations with symbolic execution and rule-based specifications," *IEEE TSE*, vol. 40, no. 7, pp. 695–709, 2014.
51. Sasnauskas, Raimondas, P. Kaiser, R. L. Jukić, and K. Wehrle, "Integration testing of protocol implementations using symbolic distributed execution," in *ICNP*, pp. 1–6, IEEE, 2012.
52. D. Beyer, "Second competition on software testing: Test-comp 2020.," in *FASE*, pp. 505–519, 2020.
53. H. M. Le, "Llvm-based hybrid fuzzing with libkluzzer (competition contribution)," in *Fundamental Approaches to Software Engineering* (H. Wehrheim and J. Cabot, eds.), (Cham), pp. 535–539, Springer International Publishing, 2020.
54. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, pp. 1–16, 2016.
55. B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," *School of Computer Science Carnegie Mellon University*, 2012.
56. Y. Noller, R. Kersten, and C. S. Păsăreanu, "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.
57. C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 179–180, 2010.
58. S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, "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.
59. P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.
60. M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 61–64, 2018.

61. J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "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.