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

Kaled Alshmrany, Mohannad Aldughaim, Ahmed Bhayat and Lucas Cordeiro
**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 \textit{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].

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

\begin{itemize}
  \item We describe the details of \textit{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.
  \item \textit{FuSeBMC} successfully participated in Test-Comp 2021 and achieved first place in the \textit{Cover-Error} category and second place in the \textit{Overall} category. Furthermore, in the subcategories \textit{ReachSafety-BitVectors}, \textit{ReachSafety-Floats}, \textit{ReachSafety-Recursive}, \textit{ReachSafety-Sequentialized}, and \textit{ReachSafety-XCSP}, \textit{FuSeBMC} obtained first place in all these subcategories. Also, \textit{FuSeBMC} shows the ability to achieve high code coverage competitively if compared with other state-of-the-art software testing tools.
\end{itemize}

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,
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 ⟨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. 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.

```plaintext
Require: program P
1: goals ← clang_extract_goals(P)
2: instrumentedP ← clang_instrument_goals(P, goals)
3: reached_goals ← ∅
4: tests ← ∅
5: for all G ∈ goals do
6:   φ ← REACH(G)
7:   result, test_case ← Fuzzing/BMC(instrumentedP, φ)
8:   if result = false then
9:     reached_goals ← reached_goals ∪ G
10:    tests ← tests ∪ test_case
11: end if
12: end for
13: for all G ∈ (goals − reached_goals) do
14:   φ ← REACH(G)
15:   result ← selective_fuzzer(instrumentedP, φ)
16:   if result = false then
17:     reached_goals ← reached_goals ∪ G
18:     tests ← tests ∪ 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.
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.


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">32 bit</data>
    <data key="programfile">/my_instrument_outpt/instrumented.c</data>
    <data key="programhash">480a6e9ea7d5e04ef15934ff3cb633121b790f</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

<? xml version="1.0" encoding="utf-8"?>
<edge id="E2" source="N2" target="N3">
  <data key="startline">3</data>
  <data key="assumption">a = -2147483647;</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, 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/khaled-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.

<table>
<thead>
<tr>
<th>Cover-Error</th>
<th>Task Num</th>
<th>FuSeBMC</th>
<th>CMA-ES Fuzz</th>
<th>CoVeriTest</th>
<th>HybridTiger</th>
<th>KLEE</th>
<th>Legion</th>
<th>LibKluzzer</th>
<th>PRTest</th>
<th>Symbiotic</th>
<th>Tracer-X</th>
<th>VeriFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReachSafety-Arrays</td>
<td>100</td>
<td>93</td>
<td>0</td>
<td>59</td>
<td>69</td>
<td>88</td>
<td>67</td>
<td>90</td>
<td>11</td>
<td>73</td>
<td>75</td>
<td>95</td>
</tr>
<tr>
<td>ReachSafety-BitVectors</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>8</td>
<td>6</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>ReachSafety-ControlFlow</td>
<td>32</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>7</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>ReachSafety-ECA</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>14</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>15</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>ReachSafety-Floats</td>
<td>33</td>
<td>32</td>
<td>0</td>
<td>16</td>
<td>22</td>
<td>6</td>
<td>0</td>
<td>30</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>ReachSafety-Heap</td>
<td>57</td>
<td>45</td>
<td>0</td>
<td>37</td>
<td>38</td>
<td>46</td>
<td>0</td>
<td>47</td>
<td>9</td>
<td>47</td>
<td>44</td>
<td>47</td>
</tr>
<tr>
<td>ReachSafety-Loops</td>
<td>158</td>
<td>131</td>
<td>0</td>
<td>35</td>
<td>53</td>
<td>96</td>
<td>4</td>
<td>138</td>
<td>102</td>
<td>82</td>
<td>78</td>
<td>136</td>
</tr>
<tr>
<td>ReachSafety-Recursive</td>
<td>20</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>16</td>
<td>0</td>
<td>17</td>
<td>1</td>
<td>17</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>ReachSafety-Sequentialized</td>
<td>107</td>
<td>101</td>
<td>0</td>
<td>61</td>
<td>53</td>
<td>86</td>
<td>0</td>
<td>83</td>
<td>0</td>
<td>79</td>
<td>57</td>
<td>99</td>
</tr>
<tr>
<td>ReachSafety-XCSP</td>
<td>59</td>
<td>53</td>
<td>0</td>
<td>46</td>
<td>52</td>
<td>37</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>41</td>
<td>31</td>
<td>25</td>
</tr>
<tr>
<td>SoftwareSystems-BusyBox-MemSafety</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DeviceDrivers-Linmx64-ReachSafety</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>699</td>
<td>405</td>
<td>0</td>
<td>225</td>
<td>266</td>
<td>339</td>
<td>35</td>
<td>359</td>
<td>79</td>
<td>314</td>
<td>246</td>
<td>385</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Cover-Branches</th>
<th>Task-Num</th>
<th>FuSeBMC</th>
<th>CMA-ES Fuzz</th>
<th>CoVeriTest</th>
<th>HybridTiger</th>
<th>KLEE</th>
<th>Legion</th>
<th>LibKluzzer</th>
<th>PRTest</th>
<th>Symbiotic</th>
<th>Tracer-X</th>
<th>VeriFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReachSafety-Arrays</td>
<td>400</td>
<td>284</td>
<td>139</td>
<td>229</td>
<td>225</td>
<td>96</td>
<td>195</td>
<td>206</td>
<td>119</td>
<td>226</td>
<td>223</td>
<td>295</td>
</tr>
<tr>
<td>ReachSafety-BitVectors</td>
<td>62</td>
<td>37</td>
<td>23</td>
<td>39</td>
<td>13</td>
<td>28</td>
<td>29</td>
<td>40</td>
<td>27</td>
<td>37</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>ReachSafety-ControlFlow</td>
<td>67</td>
<td>15</td>
<td>4</td>
<td>16</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>5</td>
<td>15</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>ReachSafety-ECA</td>
<td>29</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>ReachSafety-Floats</td>
<td>226</td>
<td>108</td>
<td>51</td>
<td>98</td>
<td>84</td>
<td>16</td>
<td>64</td>
<td>90</td>
<td>41</td>
<td>50</td>
<td>48</td>
<td>99</td>
</tr>
<tr>
<td>ReachSafety-Heap</td>
<td>143</td>
<td>88</td>
<td>19</td>
<td>79</td>
<td>74</td>
<td>81</td>
<td>69</td>
<td>90</td>
<td>40</td>
<td>84</td>
<td>86</td>
<td>86</td>
</tr>
<tr>
<td>ReachSafety-Loops</td>
<td>581</td>
<td>412</td>
<td>152</td>
<td>402</td>
<td>338</td>
<td>274</td>
<td>271</td>
<td>419</td>
<td>252</td>
<td>383</td>
<td>385</td>
<td>424</td>
</tr>
<tr>
<td>ReachSafety-Recursive</td>
<td>53</td>
<td>36</td>
<td>19</td>
<td>31</td>
<td>31</td>
<td>18</td>
<td>20</td>
<td>36</td>
<td>9</td>
<td>38</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>ReachSafety-Sequentialized</td>
<td>82</td>
<td>62</td>
<td>0</td>
<td>61</td>
<td>39</td>
<td>26</td>
<td>1</td>
<td>55</td>
<td>8</td>
<td>36</td>
<td>41</td>
<td>71</td>
</tr>
<tr>
<td>ReachSafety-XCSP</td>
<td>119</td>
<td>97</td>
<td>0</td>
<td>80</td>
<td>80</td>
<td>81</td>
<td>2</td>
<td>80</td>
<td>79</td>
<td>93</td>
<td>69</td>
<td>88</td>
</tr>
<tr>
<td>ReachSafety-Combinations</td>
<td>210</td>
<td>15</td>
<td>0</td>
<td>31</td>
<td>8</td>
<td>82</td>
<td>18</td>
<td>139</td>
<td>2</td>
<td>135</td>
<td>99</td>
<td>180</td>
</tr>
<tr>
<td>SoftwareSystems-BusyBox-MemSafety</td>
<td>72</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>DeviceDrivers-Linux64-ReachSafety</td>
<td>290</td>
<td>35</td>
<td>13</td>
<td>60</td>
<td>6</td>
<td>25</td>
<td>56</td>
<td>58</td>
<td>16</td>
<td>44</td>
<td>56</td>
<td>57</td>
</tr>
<tr>
<td>SoftwareSystems-SQLite-MemSafety</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Termination-MainHeap</td>
<td>231</td>
<td>202</td>
<td>138</td>
<td>193</td>
<td>189</td>
<td>119</td>
<td>166</td>
<td>199</td>
<td>51</td>
<td>178</td>
<td>185</td>
<td>204</td>
</tr>
<tr>
<td>Overall</td>
<td>2566</td>
<td>1161</td>
<td>411</td>
<td>1128</td>
<td>860</td>
<td>784</td>
<td>651</td>
<td>1292</td>
<td>519</td>
<td>1169</td>
<td>1087</td>
<td>1389</td>
</tr>
</tbody>
</table>

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-Floats, ReachSafety-FLOATS, ReachSafety-Recursive, ReachSafety-Sequentialized and ReachSafety-XCSP.

\(^6\) https://test-comp.sosy-lab.org/2021/results/results-verified/
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.

![Fig. 9: Quantile functions for category Overall. [8]](image)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Test Generator</th>
<th>Quality (sp)</th>
<th>CPU Time (h)</th>
<th>CPU Energy (kWh)</th>
<th>Rank Measure (kJ/sp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TracerX</td>
<td>1315</td>
<td>210</td>
<td>2.5</td>
<td>6.8</td>
</tr>
<tr>
<td>2</td>
<td>Klee</td>
<td>1370</td>
<td>210</td>
<td>2.6</td>
<td>6.8</td>
</tr>
<tr>
<td>3</td>
<td>FuSeBMC</td>
<td>1776</td>
<td>410</td>
<td>4.8</td>
<td>9.7</td>
</tr>
<tr>
<td>worst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51</td>
</tr>
</tbody>
</table>

![Fig. 10: The Consumption of CPU and Memory [8].](image)

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/](https://test-comp.sosy-lab.org/2021/results/results-verified/)
Table 3: Test-Comp 2021 Overall Results.

<table>
<thead>
<tr>
<th>Cover-Error and Branches</th>
<th>FuSeBMC</th>
<th>AFL/AMC</th>
<th>CMA-ES</th>
<th>FaStTest</th>
<th>CoverIT</th>
<th>KLEE</th>
<th>Legion</th>
<th>LibKluzzer</th>
<th>PRTest</th>
<th>Symbiotic</th>
<th>Tracer-X</th>
<th>VeriFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERALL</td>
<td>3230</td>
<td>1776</td>
<td>254</td>
<td>1268</td>
<td>1228</td>
<td>1370</td>
<td>1738</td>
<td>1543</td>
<td>526</td>
<td>1315</td>
<td>1865</td>
<td></td>
</tr>
</tbody>
</table>

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.
Verifying Security Vulnerabilities using Fuzzing and BMC

References


