Finding and Understanding Bugs in Software Model Checkers

Chengyu Zhang
dale.chengyu.zhang@gmail.com
East China Normal University, China

Ting Su*
tingsu@inf.ethz.ch
ETH Zurich, Switzerland

Yichen Yan
sei_yichen@outlook.com
East China Normal University, China

Fuyuan Zhang
fuyuan@mpi-sws.org
MPI-SWS, Germany

Geguang Pu
ggpu@sei.ecnu.edu.cn
East China Normal University, China

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich, Switzerland

ABSTRACT

Software Model Checking (SMC) is a well-known automatic program verification technique and frequently adopted for checking safety-critical software. Thus, the reliability of SMC tools themselves (*i.e.*, software model checkers) is critical. However, little work exists on validating software model checkers, an important problem that this paper tackles by introducing a *practical, automated fuzzing technique*. For its simplicity and generality, we focus on control-flow reachability (*e.g.*, whether or how many times a branch is reached) and address two specific challenges for effective fuzzing: *oracle* and *scalability*. Given a deterministic program, we (1) leverage its concrete executions to synthesize valid branch reachability properties (thus solving the oracle problem) and (2) fuse such individual properties into a single safety property (thus improving the scalability of fuzzing and reducing manual inspection). We have realized our approach as the MCFuzz tool and applied it to extensively test three state-of-the-art C software model checkers, CPAchecker, CBMC, and SeaHorn. MCFuzz has found 62 unique bugs in all three model checkers – 58 have been confirmed, and 20 have been fixed. We have further analyzed and categorized these bugs (which are diverse), and summarized several lessons for building reliable and robust model checkers. Our testing effort has been well-appreciated by the model checker developers, and also led to improved tool usability and documentation.

CCS CONCEPTS

• **Software and its engineering → Model checking**; **Software reliability**; **Software testing and debugging**.

KEYWORDS

Software Testing, Software Model Checking, Fuzz Testing

ACM Reference Format:
Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers.
In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages.
https://doi.org/10.1145/3338906.3338932

*Co-first and corresponding author.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338932

1 INTRODUCTION

Software model checking (SMC) is a well-known verification technique that statically proves program correctness *w.r.t.* properties (or specifications) of interest [23, 29]. Specifically, given a program P and a safety property ϕ, the primary goal of SMC is to prove that the property ϕ holds on all executions of P. It returns *safe* if every execution of P satisfies ϕ, or *unsafe* otherwise. In practice, this goal is formulated as *reachability checking* of a particular *error* location $\mathcal{E}$ in P because any safety property ϕ (*e.g.*, expressed in a temporal logic) reduces to checking the reachability of $\mathcal{E}$ [29] — P is safe *w.r.t.* ϕ if the error location $\mathcal{E}$ is unreachable; otherwise, P is unsafe, and a counterexample, *i.e.*, an execution that reaches $\mathcal{E}$, is given.

Significant recent advances have made SMC a practical, automated approach for verifying real-world software. For example, it was successfully applied in verifying OS device drivers [3, 7, 8] and generating test cases for bug detection [6, 9, 25]. These successes have further driven active research (*e.g.*, the annual SV-COMP competition [44]) and industrial adoption [3, 24, 31, 41].

However, like all software, SMC implementations (*i.e.*, software model checkers) also have bugs, and may give wrong verification results even for simple programs. The bugs in software model checkers can lead to missed program errors, which may be disastrous for safety-critical software (*e.g.*, aerospace, airborne, and railway signal systems). However, no prior work exists that systematically validates the correctness of software model checkers themselves. The goal of this paper is to fill this important gap.

To this end, we develop an effective fuzzing technique that can *automatically* generate a large number of verification tasks (*i.e.*, a program P and a *valid property* ϕ for P) to test software model checkers. We stress "automatically" as creating manual tests is labor-intensive and error-prone. For example, the de facto SV-COMP benchmarks were manually created by injecting artificial bugs or collecting real-world bugs (cf. [5], Section 4). The prominent challenge is the oracle problem [4]. To tackle it in our problem setting, we focus on control-flow reachability and specifically branch reachability. In particular, given a valid, deterministic and terminating program P, our key idea is to leverage information on P's concrete executions to synthesize a valid branch reachability property ϕ for P, and couple it with P to generate a verification task. More specifically, for a particular branch b in P, given an input i, we monitor the control-flow information of b (*e.g.*, whether b is reached or how

many times it is reached). This information is used to synthesize a *guaranteedly valid* safety property ϕ for P. If a model checker concludes that the oracle ϕ does not hold for P, the model checker has an actual bug.

The above approach allows us to test model checkers as black-boxes despite many state-of-the-art software model checkers (*e.g.*, CPAchecker [8], SeaHorn [27], CBMC [18]) combine different techniques [5] (*e.g.*, CEGAR [3, 7, 8], IC3 [11, 12, 16, 27], BMC [18]). In particular, we realize our approach as two distinct property-synthesizing strategies for each individual branch, *i.e.*, enumerative reachability (ER) and enumerative counting reachability (ECR). However, these approach instances face scalability issues because (1) a seed program P can have a large number of branches, and (2) the model checkers can spend much time checking each test program. To tackle this scalability issue, we develop an optimized strategy, fused counting reachability (FCR), by validating all branch properties at once to improve fuzzing performance. Section 3 details our approach and the three testing strategies.

We implemented our approach for validating C software model checkers in a tool called MCFuzz (Model Checker Fuzzer), and applied it to test three state-of-the-art model checkers, CPAchecker [8], SeaHorn [27] and CBMC [18]. For evaluation, we selected 4,609 seed programs from the GCC regression test suite [48], from which we systematically generated 48,505 test programs. Finally, we found 62 unique bugs – 58 bugs have been confirmed, and 20 have already been fixed. Our bug reports and testing effort have been very well-received by the developers. We have also carefully analyzed these bugs and the developer comments, and found that the bugs are diverse and reside in different modules of the model checkers, including front-end, memory model, pointer alias analysis, third-party components, *etc.* Our work has also led to improvements in the usability (*e.g.*, better warning messages) and documentation of these model checkers.

This paper makes the following main contributions:

- It proposes a fully-automated branch reachability fuzzing approach to validating software model checkers. To our knowledge, this is the first systematic and extensive effort to automatically validate the correctness of software model checkers.
- It realizes the approach as a tool MCFuzz with three property-synthesizing strategies, which can effectively and efficiently validate C software model checkers. MCFuzz is general and flexible to test any software model checker, and will be made publicly available to benefit the community once our paper is deanonymized. We have already made our detailed bug reports available from an anonymized GitHub repository [1].
- It reports our experience in applying MCFuzz to test three state-of-the-art model checkers (CPAchecker, SeaHorn and CBMC) and found 62 unique bugs — 58 were confirmed and 20 already fixed. We analyzed the characteristics of these bugs and summarized several lessons for building reliable and robust model checkers.

The rest of the paper is organized as follows. Section 2 motivates and illustrates our approach via two examples, and Section 3 formulates the problem, and presents our approach and implementation details. Next, Section 4 reports the evaluation of MCFuzz and discusses some representative bugs that we found. Finally, we survey related work (Section 5) and conclude (Section 6).

2 ILLUSTRATIVE EXAMPLES

This section presents two examples to motivate the problem and illustrate our approach. Both examples are real bugs that we found in CPAchecker, a state-of-the-art CEGAR-based model checker.

2.1 CPAchecker Bug #529

Figure 1b shows the test program that manifests an incorrect checking result of CPAchecker. It is generated from the seed program in Figure 1a by inserting the error label __VERIFIER_error() before line 5. Here, the error label __VERIFIER_error() can be interpreted as an error location under checking, which enforces CPAchecker to do reachability querying. Obviously, by concretely executing the program in Figure 1b, the error label should be reached since `i` is assigned to 1 on line 9 (thus > 0) after the first loop iteration.

However, CPAchecker concludes the program as *safe*, which means the error label is unreachable. This incorrect result is caused by an intricate bug in the predicate analysis component, which is the core component of such CEGAR-based model checkers as CPAchecker. The reason is the following: Firstly, CPAchecker detects a spurious counterexample in the first loop iteration. Then the refinement procedure produces the predicate `i>0`, which rules out the counterexample. The analysis continues and encounters the last statement (line 9) of the loop body. It notices that the address of `i` is taken. The predicate analysis thus switches from tracking `i` directly to tracking `i` indirectly via its address (as if it were on the heap). When the analysis encounters the loop head again after the first loop iteration, the counterexample trace formula is something like `!(i@2>0) && ((i@2!=0) || (p))`[1], with `p` being the formula that indirectly increments the value of i. Unfortunately, when applying the predicate `i>0`, the variable `i` gets matched with the old value i@2 from the beginning of the first loop iteration, which is outdated and its value is always equal to `0`. This leads to the wrong proof (since the trace formula is always unsatisfiable). The CPAchecker developers confess this bug is very difficult to fix and they are still working on it at the time of submission.

Approach I: Enumerative Reachability. As the above example shown, the test program in Figure 1b is created by inserting an error label at a particular program branch (line 4). Thus, our basic approach is designed to enumeratively checking the reachability of each program branch[2], which we called *enumerative reachability*. For example, Figure 1 illustrates this approach. Figure 1a is the seed program. Figure 1b, 1c and 1d are the three test programs created from Figure 1 by inserting the error labels at the three branches, respectively. By running the executables of these test programs, we can get the test oracles (*i.e.*, whether the error labels are reachable). Then we run model checkers to do reachability checking on each test program. If the checking result is different from the oracle, a bug may exist. This approach is effective but has two limitations.

- Querying the reachability of an error label only checks whether the target branch is reachable or not, but cannot check how many times the branch can be reached. Thus, it may miss those intricate bugs related to the computations of loops.

[1] The clause `!(i@2>0)` denotes the previous state before the loop against the predicate `i > 0` in the first loop iteration. `i@2` denotes the value of `i` during the second iteration of CEGAR analysis.

[2] Note that checking the reachability of each branch is semantically equal to checking each program location.

```c
void main() {
  int i = 0;
  while (1) {
    if (i > 0) {
      break;
    }
    if (i == 0)
      *(&i) = *(&i) + 1;
  }
}
```

(a) Seed program

```c
void main() {
  int i = 0;
  while (1) {
    if (i > 0) {
    __VERIFIER_error();
      break;
    }
    if (i == 0)
      *(&i) = *(&i) + 1;
  }
}
```

(b) Test program 1 which leads to the bug #529 in CPAchecker.

```c
void main() {
  int i = 0;
  while (1) {
    __VERIFIER_error();
    if (i > 0) {
      break;
    }
    if (i == 0)
      *(&i) = *(&i) + 1;
  }
}
```

(c) Test program 2.

```c
void main() {
  int i = 0;
  while (1) {
    if (i > 0) {
      break;
    }
    if (i == 0){
    __VERIFIER_error();
      *(&i) = *(&i) + 1;
    }
  }
}
```

(d) Test program 3.

Figure 1: Illustrative example for the bug #529 of CPAchecker.

```c
int main(void){
  int a[3] = {1};
  int i = 0;
  while(i < 3){
    if(a[i] == 1) {
      i++;continue;
    }
    i++;
  }
  return 0;
}
```

(a) Seed program

```c
int main(void){
  int a[3] = {1};
  int i = 0;
  int br = 0;
  while(i < 3){
    if(a[i] == 1) {
      br++;
      i++;continue;
    }
    i++;
  }
  if(br != 1)
    __VERIFIER_error();
  return 0;
}
```

(b) Test program 1 which leads to the bug #534 in CPAchecker.

```c
int main(void){
  int a[3] = {1};
  int i = 0;
  int br = 0;
  while(i < 3){
    br++;
    if(a[i] == 1) {
      i++;continue;
    }
    i++;
  }
  if(br != 3)
    __VERIFIER_error();
  return 0;
}
```

(c) Test program 2.

```c
int main(void){
  int a[3] = {1};
  int i = 0;
  int br1=0;int br2=0;
  while(i < 3){
    br1++;
    if(a[i] == 1) {
      br2++;
      i++;continue;
    }
    i++;
  }
  if(br1!=3||br2!=1)
    __VERIFIER_error();
  return 0;
}
```

(d) Test program generated by Approach III.

Figure 2: Illustrative example for the bug #534 of CPAchecker.

- The number of test programs could be large if the given set of seed programs have many branches. It may bring high overhead and reduce the testing performance.

Section 2.2 will use the example in Figure 2 to illustrate how we tackle these limitations.

2.2 CPAchecker Bug #534

Approach II: Enumerative Counting Reachability. To solve the first limitation, we evolve the basic approach by synthesizing a safety property of the number of times a branch should be reached. Figure 2a, 2b and 2c illustrate this *enumerative counting reachability* approach. Given the seed program in Figure 2a, it introduces a counter variable `br` and initializes it to `0`. It then inserts the counting instruction `br++;` at each branch to create the two test programs in Figure 2b and 2c. By running the executables of these test programs, we can get the exact value of `br` for each test program. We then insert a reachability query in the form of value equivalent checking (*e.g.*, the code of lines 12-13 in Figure 2b and 2c) before the program exit. Specifically, it checks whether `br` is equal to the exact value obtained in the concrete executions. If the check fails, the error label `__VERIFIER_error()` is reachable. Obviously, the two test programs should always be *safe*. If a model checker answers *unsafe* (*i.e.*, the error label is reachable), a bug may exist. We can see *Approach II* is stronger than *Approach I* since *Approach II* not only checks the reachability of each branch but also checks the number of times that each branch should be reached.

Figure 2b shows an intricate bug in CPAchecker (bug #534) that can only be manifested by *Approach II*. The bug is introduced because CPAchecker fails to correctly handle the array initializer with a single element. According to the C standard [49], if there are fewer initializers in a brace-enclosed list than the elements of an aggregate, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration. For example, the object with an arithmetic type will be initialized to (positive or unsigned) zero. In Figure 2b, the array `a` should be initialized to `{1,0,0}`, and has only one element 1 (the value of `br` should be 1). The error label `__VERIFIER_error()` should be unreachable. However, CPAchecker incorrectly initializes the array `a` to `{1,1,1}`. As a result, CPAchecker answers *unsafe* which means the error label is reachable. In fact, we find this bug is a regression error caused by the fix of another bug #342. *Approach I* will miss this bug since CPAchecker returns correct checking results when doing reachability querying of each branch, while *Approach II* can

find such deeper bugs by countering the number of times that each branch is reached.

Approach III: Fused Counting Reachability. *Approach II* is more effective than *Approach I*, but may still generate a large number of test programs. To solve the second limitation, we evolved *Approach II* to the *fused counting reachability* approach. Figure 2d illustrates this approach, which is as effective as *Approach II* but only generates one test program. Specifically, this approach introduces the counting variables $br_1, br_2, \ldots, br_n$ for each branch and inserts the corresponding loop counting instructions br_k++; $(k \in 1...n)$ at each branch. It then checks the equivalence between these counting variables and their actual values altogether before the program exit (*e.g.*, the codes of lines 13-14). Obviously, the test program generated by *Approach III* should be always *safe*, if a model checker answers *unsafe*, a bug may exist. *Approach III* effectively reduces the testing time from $O(n)$ to $O(1)$ by creating the disjunction checking for all of the counting variables. Section 3 will give the formal definition of these approaches and discuss their effectiveness.

3 APPROACH AND IMPLEMENTATION

This section gives the details of our approach and implementation. In particular, Section 3.1 gives the problem definition and our high-level approach, Section 3.2 shows the three instances of our approach to effectively validating software model checkers, and Section 3.3 details the implementation.

3.1 Definition and High-level Approach

Definition 3.1 (Test Program)**.** A valid test program for a software model checker M is a source program P with a safety property ϕ.

Given a test program, a software model checker M verifies whether the safety property ϕ is satisfied on P. Specifically, this safety verification problem is formulated to a check for the reachability of a particular error location $\mathcal{E}$ in P. Here, the reachability of $\mathcal{E}$ represents the safety property ϕ. If $\mathcal{E}$ is unreachable, M concludes that P is *safe* (*i.e.*, ϕ holds on all computations of P). Otherwise, P is *unsafe* (*i.e.*, ϕ is violated by one computation ending at $\mathcal{E}$).

Problem Definition. Our problem is to generate a set of valid test programs to validate the correctness of a model checker M. Given a test program, if the answer of M is contradictory to the ground-truth, a potential bug is reported.

The key challenge of this problem is how to generate valid test programs with known oracles (*i.e.*, the ground-truth of the safety for P *w.r.t.* ϕ). To resolve this problem, we propose the approach of *branch reachability fuzzing* that can automatically generate safe or unsafe test programs.

Assume that we are given a seed program P_{seed}. We insert to P_{seed} *counter variables* to monitor branches of P_{seed}. For an arbitrary branch in P_{seed}, we are interested in (1) whether the branch is reachable and (2) how many times the branch can be reached. The values of counter variables are updated accordingly whenever the corresponding branches are executed. We construct safety properties based on the final values of counter variables (derived through dynamic execution). In the rest of the paper, we assume that seed programs are valid, terminating and deterministic. We formalize our way of generating test programs for software model checkers in the following definition.

Definition 3.2 (Branch Reachability Fuzzing)**.** Let P_{seed} be a seed program and P be derived by inserting counter variables $c_1, ..., c_n$ to n different branches of P_{seed}. Assume that we execute P with a valid input and that $c_i = v_i$ $(1 \leq i \leq n)$ on the final state of P, i.e when P terminates. The test program we construct is P together with a safety property $\phi := c_1 = v_1 \wedge ... \wedge c_n = v_n$, which is meant to be satisfied on the final state of P.

In practice, for a test program P and ϕ, we can construct a corresponding program P' with an error location $\mathcal{E}$ such that whether P satisfies ϕ is reduced to whether $\mathcal{E}$ is reachable in P'. In the following, we also refer to the programs with an error location as test programs for software model checkers.

Our high-level approach. To validate software model checkers, we apply Definition 3.2 to generate a large number of test programs with known oracles. If the conclusion of a model checker is inconsistent with the known oracle, a bug will be reported. In this way, our approach can stress-test model checkers as black-box. Note that since the reachability problem is in general undecidable, a model checker may give *unknown* (*i.e.*, the model checker cannot terminate within a given time bound). In this case, we cannot conclude any oracle violation.

3.2 Approach Instances

We introduce in this section three specific approach instances to construct test programs with known oracles. We first introduce Approach I, which is a simplified application of Definition 3.2. The safety properties involved there specify the reachabilities of branches in the seed programs.

Approach I: Enumerative Reachability (ER). Let P_{seed} be a seed program. For each branch b in P_{seed}, we construct a test program P_b by adding an error label $\mathcal{E}$ to branch b of P_{seed}.

The oracle of the test program in Approach I can be derived as follows. Imagine that we construct an intermediate program P by initializing a boolean variable c to 0 at the beginning of P_{seed} and then inserting the instruction $c := 1$ to branch b of P_{seed}. Assume that v is the value of c when P terminates. It is easy to see that branch b is reachable (resp. unreachable) in P_{seed} iff $v = 1$ (resp. $v = 0$). Therefore, we have that P_b is safe iff $v = 0$.

Example. For the seed program in Figure 1a, Approach I generates three test programs in Figure 1b, Figure 1c and Figure 1d. All of these programs are unsafe since the error labels are reachable in the concrete executions.

Approach II further considers safety properties that constrain the number of times a branch can be executed. For each branch in a seed program, we insert a counter variable and increment its value by 1 in each execution of the branch.

Approach II: Enumerative Counting Reachability (ECR) Let P_{seed} be a seed program. For each branch b in P_{seed}, we first construct a program P by initializing a counter variable c to 0 at the beginning of P_{seed} and then inserting instruction $c := c + 1$ to branch b of P_{seed}. Let $\phi := c = v$ be a safety property, where v is the value of c when P terminates. We construct a test program $P_b := P;$ `if` $(\neg\phi)$ $\{\mathcal{E}\}$ and we have that P_b is safe.

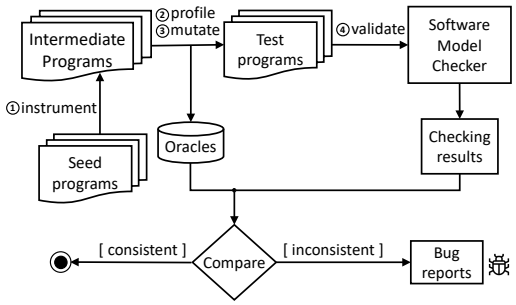

Figure 3: Workflow of our testing framework MCFuzz.

Example. For the seed program in Figure 2a, Approach II generates two test programs in Figure 2b and Figure 2c, where br is the counter variable. All of these two programs are safe.

For a seed program with n branches, both Approach I and II construct n test programs, each of which monitors a different branch in the seed program. To optimize the test cost, we propose Approach III that reduces testing time from $O(n)$ to $O(1)$.

Approach III: Fused Counting Reachability (FCR). Let P_{seed} be a seed program. Assume that $b_1, ..., b_n$ are all the branches in P_{seed}. We construct a program P by initializing counter variables $c_1, ..., c_n$ to 0 at the beginning of P_{seed} and then inserting instruction $c_i := c_i + 1$ $(1 \le i \le n)$ into branch b_i of P_{seed}. Let $\phi := c_1 = v_1 \wedge ... \wedge c_n = v_n$ be a safety property, where v_i $(1 \le i \le n)$ is the value of c_i when P terminates. We construct a test program $P_{all} := P$; `if` $(\neg\phi)$ $\{\mathcal{E}\}$ and we have that P_{all} is safe.

Example. For the seed program in Figure 2a, Approach III generates the test program in Figure 2d, where br_1 and br_2 are the two counter variables. Obviously, this test program subsume the two test programs in Figure 2b and Figure 2c.

Since the test program generated in Approach III subsumes all test programs that Approach II generates, Approach III can be viewed as an optimization of Approach II. Approach III is the most efficient and scalable one among the three approach instances without sacrificing effectiveness. The evaluation of these three approach instances will be given in Section 4. In the next subsection, we will introduce the workflow of our tool MCFuzz and its implementation with Approach III.

3.3 Implementation

Figure 3 gives the overview of our testing framework MCFuzz. Given a set of seed programs, MCFuzz instruments the seed programs and transforms them to intermediate programs. By profiling the execution of these intermediate programs, MCFuzz generates the test programs with the corresponding oracle. Given a test program, if the checking result and the oracle is inconsistent, a bug is reported for further inspection. The three approach instances presented in Section 3.2 can be easily adapted to this framework.

Algorithm 1 details our implementation by using Approach III. It takes as input a model checker M and a closed program P, and then follows the following four steps.

Step 1: Instrument. In the function `Instrument()`, MCFuzz instruments probes (*i.e.*, counter variables) into the program P. Specifically, the declarations of counter variables are inserted at the beginning of P, and the counting instructions are inserted at each branch.

Algorithm 1: Fused counting reachability

Procedure `Test`(*Software model checker M, Program P*)**:**
- $P', C \leftarrow$ `Instrument`(P);
- $(C, V) \leftarrow$ `Profile`(P', C);
- $\widehat{P} \leftarrow$ `Mutate` $(P', (C, V))$;
- **if** *M.check*$(\widehat{P})$ == *unsafe* **then**
  - reportBug();

Function `Instrument`(*Program P*)**:**
- $P' \leftarrow P$, $i \leftarrow 0$;
- $B \leftarrow P'$.Branches();
- $C \leftarrow \emptyset$;
- **foreach** $b \in B$ **do**
  - $P' \leftarrow P'$.firstLoc().insert($c_i \leftarrow 0$);
  - $P' \leftarrow P'$.getBr(b).firstLoc().insert($c_i \leftarrow c_i + 1$);
  - $C \leftarrow C \cup c_i$;
  - $i \leftarrow i + 1$;
- **return** P', C;

Function `Profile`(*Program P, Counters C*)**:**
- $E \leftarrow P$.execute();
- pair$(C, V) \leftarrow E$.getFinalValue(C);
- **return** (C, V);

Function `Mutate`(*Program P, Pair* (C, V))**:**
- $O \leftarrow true$;
- **foreach** $(c, v) \in (C, V)$ **do**
  - $O \leftarrow O \wedge (c = v)$;
- $\widehat{P} \leftarrow P$.lastLoc().insert($\neg O \Rightarrow \mathcal{E}$);
- **return** $\widehat{P}$;

At last, an intermediate program P' is created and C contains all inserted counter variables.

Step 2: Profile. In the function `Profile()`, MCFuzz compiles P' by using an off-the-shelf compiler, *e.g.*, GCC [48], and executes P'. The final values of counter variables in C (when P' terminates) are recorded in V.

Step 3: Mutate. In the function `Mutate()`, MCFuzz exploits the execution results from the profiling step to generate a safety property. This property is represented as a reachability query $\neg O \Rightarrow \mathcal{E}$, and inserted before the program exit. At last, it generates the test program $\widehat{P}$ from P' with the oracle *safe*.

Step 4: Validate. MCFuzz runs $\widehat{P}$ to get the verification result. If the result is inconsistent with the provided oracle (*i.e.*, *unsafe* for Approach III), a bug is reported.

In our actual implementation, Algorithm 1 is realized using python scripts and C++. In particular, we use LLVM's LibTooling library [45] to instrument and mutate the seed program, and generate test programs. GCC [48] is used to compile the intermediate programs for concrete execution. MCFuzz has approximately 2000 lines of python scripts and 800 lines of C++ code.

4 EMPIRICAL EVALUATION

In this section, we applied MCFuzz on three state-of-the-art software model checkers, *i.e.*, CPAchecker, CBMC, SeaHorn, to demonstrate its effectiveness. We finally found 62 unique bugs with diverse types. Our testing effort has been well-appreciated by the model checker developers, also leads to improved tool usability and documentation. All the detailed bug reports submitted by us to the developers can be accessed at [1].

4.1 Evaluation Setup

Testing environment. MCFuzz and the three model checkers all run on a workstation with the following configurations: Ubuntu 14.04, Intel i7 7800X 3.5GHz 6-core CPU, and 32GB memory.

Software model checkers. We selected the three model checkers, *i.e.*, CPAchecker, CBMC, and SeaHorn, as the testing subjects. We chose them based on the following considerations. First, they implemented different SMC techniques, including CEGAR, BMC and IC3. Second, they all support C programs, which makes the tool comparison and result analysis more fair and convenient. Third, they are mature and widely-used in both industry and academia. Validating their correctness is of importance.

CPAchecker is a state-of-art CEGAR [17] based model checker proposed by Beyer *et al.* [8]. It is the champion in SV-COMP 2018, and performs the best in the *ReachSafety* track [50]. Besides CEGAR, CPAchecker also supports other SMC techniques, *e.g.*, k-induction [22]. In the evaluation, we use the default predicate abstraction [2, 26] configuration `predicateAnalysis.properties` with `-skipRecursion`. The CPAchecker version is `svn-28771`.

CBMC is a state-of-art BMC [10] based software model checker proposed by Clarke *et al.* [18]. It won the silver medal in SV-COMP 2018 falsification overall track [50]. CBMC is an under-approximation model checker with bit-precision. If the given checking bound is not large enough, it is allowed to answer *safe* for *unsafe* programs. Thus, we only generate *safe* programs in the evaluation for Approach II and III. We used the default CBMC configuration without specifying any checking bounds. The CBMC version is `5.10`.

SeaHorn is a state-of-art LLVM [33] based model checker [27]. It employs several IC3 [11, 12] based model checking engines. It also supports abstract interpretation [19] and BMC. In the evaluation, we use the default configuration `pf`, which uses the IC3 engine Spacer [32]. The SeaHorn version is `21a1c0`.

Although the outputs of these tools can be different, we consistently record *safe* if the tool concludes the error label is unreachable, or *unsafe* if reachable. For each model checker, we set the time bound of each checking as 5 minutes. If the tool timeouts, we record the conclusion as *unknown*.

Seed programs. In theory, the seed program can be any program that can be handled by software model checkers. In the evaluation, we selected seed programs from the GCC regression test suite, which is a set of programs for testing GCC. Most of them are closed program, which can be directly handled by model checkers. Specifically, we only selected the programs that can be independently compiled with a single file since most model checkers cannot handle programs with multiple files. We selected 4,609 out of 25,292 GCC regression test cases as seed programs, which contains 219,636 lines of code.

Table 1: Bugs overview.

	CPAchecker	CBMC	SeaHorn	Total
Fixed	14	4	2	20
Confirmed	22	2	14	38
Unconfirmed	0	4	0	4
Total	36	10	16	62

4.2 Results

This section presents our results and anlysis via the following three research questions *RQ1~RQ3*.

RQ1: Can MCFuzz find model checker bugs? The answer is yes. MCFuzz discovered 62 unique bugs in the three software model checkers, of which 58 were confirmed. In particular, we found 36 bugs in CPAchecker, 10 bugs in CBMC and 15 bugs in SeaHorn, respectively. Since the semantics of undefined behaviors are not defined in C standard, the undefined behaviors in general allow a model checker to output any result. Thus, we did not report any undefined behaviors related issues as bugs. Table 1 gives an overview of the bugs we found by using MCFuzz. The numbers in Table 1, represent the bugs that have been fixed, confirmed (but not fixed) or unconfirmed in each software model checker, respectively. *Fixed* means developers have confirmed the bugs and fixed them by committing to the up-to-date version. *Confirmed* means developers have confirmed and explained the reason but have not fixed yet. *Unconfirmed* means we have confirmed the bugs and reported to the developers, but they have not replied yet.

We used both manual inspection and automated filtering via scripts to facilitate bug reporting from the generated incorrect test programs. For automated script filtering, we used CompCert's C interpreter [46] to automatically remove invalid bugs caused by undefined behaviors (UBs). Through reporting and discussing bugs with the SMC developers, we gained more domain knowledge, which helped further automatically remove duplicate cases (*e.g.*, we used the keywords "float" and "double" to remove duplicate cases for Seahorn as it does not support floating-point types). We then manually inspected the remaining cases. In total, it took us about 3-4 person-weeks for bug reporting. We tried our best to report high-quality bugs and not to burden the developers unnecessarily.

In summary, MCFuzz can indeed find many bugs in all three state-of-the-art software model checkers. Specifically, MCFuzz discovered 62 unique bugs, of which 58 were confirmed.

RQ2: Which fuzzing approach instance is more effective and efficient? As we proposed the three fuzzing approach instances in Section 3.2, we intend to evaluate which approach instance is more effective (*i.e.*, find more bugs) and efficient (*i.e.*, cost lower testing time). We evaluated these three approaches, respectively, by running MCFuzz on the three model checkers. Table 2 shows the statistics of the evaluation. Column *#Mutants* means the number of test programs generated by the corresponding approach instance from the selected GCC test suite. Column *#Incorrect cases* means the number of test programs which lead to inconsistent checking results. In particular, for Approach II and III, the oracles of test programs generated by MCFuzz are *safe*, so all *unsafe* results given by model checkers are incorrect cases; for Approach I, the oracle can be *safe* or *unsafe*, so any inconsistent results are incorrect cases. Note that each incorrect case in Table 2 may not indicate a valid bug,

Table 2: Statistics of the three approach instances in terms of the generated test programs (mutants), incorrect cases, average checking time of one test program (in seconds), and the total fuzzing time (in hours). The time is measured in CPU time.

Approach instance	#Mutants	#Incorrect cases			Average time (s)			Total time (h)		
		CPAchecker	CBMC	SeaHorn	CPAchecker	CBMC	SeaHorn	CPAchecker	CBMC	SeaHorn
Approach I: ER	21,948	2,091	1,567	3,665	28.55	25.63	6.30	168.41	151.23	37.17
Approach II: ECR	21,948	5,972	4,690	6,752	42.32	21.67	15.65	258.03	132.08	95.35
Approach III: FCR	4,609	699	356	727	31.81	15.83	9.86	32.95	16.89	10.50

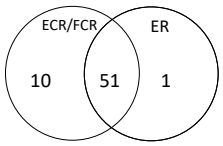

Figure 4: Relation of the bugs found by the three fuzzing approach instances

since the case could be duplicated, caused by undefined behaviors or invalid for software model checkers (*e.g.*, macro, third-party library). Column *Average time* means the average checking time of a model checker for one test program. Column *Total time* means the total fuzzing time of an approach instance.

From Table 2, we can see Approach III generated the least number of test programs, *i.e.*, 4,609 tests, while Approach I and II generated the same number of test programs, *i.e.*, 21,948 tests, respectively. Approach III reduced over 79% of test programs. From the total testing time, Approach III saved hundreds of hours, compared with Approach I and II. In particular, Approach III reduced more than 80% of fuzzing time in total. This huge performance improvement is enabled by fusing all the reachability checkings into one test program. Additionally, although Approach I and II generated the same number of test programs, Approach II can find more incorrect cases. It indicates Approach II is stronger than Approach I, *i.e.*, Approach II may find more bugs.

The Venn diagram in Figure 4 shows the relation of the bugs found by three approach instances. We can see Approach I found 52 bugs in total, while Approach II and III found the same number of bugs, *i.e.*, 61 bugs. Approach II and III can find all the bugs found by Approach I except one bug, which is shown in Figure 1b. The reason is that CPAChecker reports *safe* when this bug was triggered. This cannot be detected by Approach II and III as they use *safe* as the oracle. On the other hand, there are 10 bugs that can only be found by Approach II and III but cannot be found by Approach I. Figure 2 is one sample bug of these ten bugs.

In summary, Approach III is the most effective and efficient fuzzing approach instance. It fuses oracles to achieve testing scalability without sacrificing effectiveness.

RQ3: What types of bugs can be found by MCFuzz? In total, we found 62 unique bugs in the three software model checkers. To understand these bugs found by MCFuzz, we categorized them into seven categories according to the module in which the bug resides: *front-end* related, *memory model* related, *pointer alias* related, *third-party component* related, *C standard library* related, *language feature* related and *configuration* related bugs. Front-end related bugs are usually caused when incorrectly compiling or optimizating the program source code. Incomplete memory modeling usually

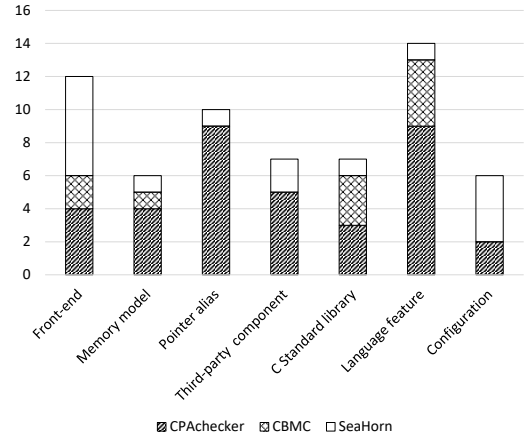

Figure 5: Numbers of the bugs found by MCFuzz across different categories.

causes memory model related bugs. Incorrect pointer alias analysis may lead to pointer alias related bugs. The bugs in third-party components of software model checkers cause third-party component related bugs. The incomplete supporting for C standard library functions and language features causes C standard library and language feature related bugs, respectively. Configuration related bugs are the bugs that can be solved by switching the configurations of software model checker via the command line options.

Figure 5 shows the number of bugs in each category. Most bugs of CPAchecker found by us are related to pointer alias analysis and language features, while most bugs of SeaHorn are related to the unexpected behaviors of the front-end and tool configurations. As for CBMC, we found a few front-end and memory model related bugs, but most bugs are related to standard library functions and language features. In the next subsection, we will explain and discuss more details about each bug category, and give assorted bug samples for each bug category.

In summary, the bugs found by MCFuzz in software model checkers are diverse and categorized into seven groups, of which front-end, pointer alias and language features related bugs are common.

4.3 Assorted Bug Samples

Figure 6 gives eight bug samples to illustrate each bug category we found. All of the samples are reduced from more complicated test programs and all these error labels __VERIFIER_error() should be *unreachable*, *i.e.*, all these test programs are *safe*.

Front-end related bugs. In software model checkers, the front-end component is used to transform and optimize the program source code to an intermediate representation for model checking.

```c
void f(int a, int b){
  if (a != b)
    __VERIFIER_error();
}
int main(){
  int c = 4, d = 7;
  int e = 2;
  f (d=c&&e, 1);
  return 0;
}
```

(a) The program triggers front-end related bug in CPAchecker.

```c
int main(){
  int array[6] =
  { 1,2,3,4,5,6 };
  for (int i = 0;
       i < 5; i++)
      array[i] = 0;
  if (array[5] != 6)
    __VERIFIER_error();
  return 0;
}
```

(b) The program triggers front-end related bug in SeaHorn.

```c
union{int a;int b; } u;
int main() {
  u.a = 1;
  u.b = 2;
  if (u.a != 2) {
    __VERIFIER_error();
  }
  return 0;
}
```

(c) The program triggers memory model related bug in CPAchecker.

```c
typedef int T[];
struct f{int w;T x;};
static struct f f =
  { 4, { 0, 1, 2, 3 } };
int main(){
  for (int i = 0;
       i < f.w;++i) {
    if (f.x[i] != i)
     __VERIFIER_error();}
  return 0;}
```

(d) The program triggers memory model related bug in CBMC.

```c
int x,*p=&x;
int main()
{
  int i=0;
  x=1;
  p[i]=2;
  if (x != 2)
    __VERIFIER_error();
  return 0;
}
```

(e) The program triggers pointer alias related bug in CPAchecker.

```c
typedef _Bool bool;
const bool false = 0;

int main ()
{
  bool a = 1;
  if (a != false)
    __VERIFIER_error();
  return 0;
}
```

(f) The program triggers third-party component related bug in CPAchecker.

```c
struct {
int a:4; int :4;
int b:4; int c:4;
} x = { 2,3,4 };
int main (){
  if (x.b != 3)
    __VERIFIER_error();
  return 0;
}
```

(g) The program triggers language feature related bug in CBMC.

```c
void test(int x,int y,
          int q){
  if ((x / y) != q )
    __VERIFIER_error();
}
int main (){
  test (7, 6, 1);
  test (-7, -6, 1);
  return 0;
}
```

(h) The program triggers configuration related bug in SeaHorn.

Figure 6: Assorted samples that trigger software model checker bugs.

For example, in SeaHorn, the front-end Clang transforms and optimizes the C language code into LLVM IR bitcode before model checking [27]. In CPAchecker, the front-end CDT [47] constructs the control-flow automata (CFA), which consist of control-flow locations and edges. The CEGAR algorithm later runs on CFA. Incorrect transformations or optimizations could lead to incorrect checking results. Figure 6a is a program which triggers a front-end bug in CPAchecker. CPAchecker answers *unsafe* because the front-end creates a temporary variable `__CPAchecker_TMP_0` for the `c&&e` expression, but `d=__CPAchecker_TMP_0` is added to the CFA before the value of the temporary variable is computed. This bug is due to the incorrect transformation. Figure 6b is a program that triggers a font-end bug in SeaHorn. The loop (lines 4-6) is silently optimized into a `memset` (a standard library function), which however is not supported in SeaHorn and leads to the *unsafe* conclusion. This bug is due to the incorrect optimization. These front-end bugs affects the first step of SMC. The model checker developers should be careful when choosing the front-end implementations, and pay more attention to the reliability and the unexpected behavior.

Memory model related bugs. The memory model is a key component that simulates the memory operations when model checking. It would be error-prone when some memory operations involve complex data structures, *e.g.* `unions` and `structs`. Figure 6c is the program which triggers the `union` related bug in CPAchecker. In Figure 6c, `u.a` should be assigned to 2 at line 4 since the variables `a` and `b` share the same memory location in the `union u`. CPAchecker should always treat the `union` as heap memory. Unfortunately, it fails on this case and answers *unsafe*, unless the `union` address is explicitly taken in the program. Similarly, CBMC does not initialize the nested `struct` correctly in the case of Figure 6d, and thus answers *unsafe*. Such memory model bugs usually involve complex data structures or corner cases that were easily ignored.

Pointer alias related bugs. The pointer alias analysis is a challenging problem in program analysis area [28], and also difficult for SMC. To precisely check programs, software model checkers analyze pointer aliases in the front-end or during the checking process. For example, SeaHorn analyzes pointer aliases by transforming the source code into the static single assignment (SSA) form of LLVM IRs. CPAchecker analyzes pointer aliases while doing predicate analysis. Figure 6e is the program that triggers a pointer alias related bug in CPAchecker. CPAchecker has an optimization in the predicate analysis component, which reduces a program by ignoring the irrelevant variables *w.r.t.* the property under check. The problem here is that the predicate analysis does not detect that the address of `x` is taken, and thus regards `x` as a primitive variable instead of a potentially aliased variable. As a result, it ignores the assignment of `x` via the pointer alias `p` at line 6. Thus, to avoid the pointer alias related bugs, it is important to implement the pointer alias analysis algorithm correctly.

Third-party component related bugs. In practice, software model checkers usually integrate a few third-party components. For example, CPAchecker integrates CDT as the front-end and SMTInterpol [51] as the SMT solver, while SeaHorn integrates the LLVM framework into its implementation. However, the third-party components are not always reliable. Figure 6f is the program that crashes CPAchecker due to an issue of using SMTInterpol. In the

program, a variable named `false` is defined, which is a reserved token in SMTInterpol. In fact, the variables `false`, `true` or `select` are all reserved tokens in SMTInterpol. Using any of them will crash SMTInterpol and then fail CPAchecker. It is usually hard for developers to note such issues when using third-party components. Additionally, the bugs of third-party components themselves can also affect the correctness of model checkers. Thus, ensuring the reliability of these components (*e.g.*, compilers, solvers) is important for building a more reliable software model checker.

C standard library related bugs. In C, there are many standard library functions. For example, `malloc` allocates a block of memory, `memset(s,c,n)` replaces each of the first `n` characters of `s` by `c` and returns `s`, and `isdigit(c)` returns nonzero if `c` is a numeric digit. However, we find such standard library functions are not well supported in most software model checkers. SeaHorn and CPAchecker do not support most of the standard library functions, and usually give incorrect checking results silently without any warning. It lets user erroneously believe that the checking results are correct. As for CBMC, it supports quite a number of standard library functions by implementing the corresponding built-in functions. But the built-in functions are not always reliable. For example, CBMC sometimes falls into infinite unwindings while there are some string related C standard library functions (*e.g.* `strlen` in the program). It may be infeasible for model checkers to support all functions in the standard library. But model checkers should give clear warning messages if some unsupported functions exist in the program or conservatively answer *unknown*.

Language feature related bugs. There are many language features defined in C standard. In principle, model checkers for C program should support all of the language features in C standard to correctly comprehend the semantics of program. In practice, it is difficult for software model checker developers to completely follow the C standard. For example, Figure 6g is the program that leads to an incorrect checking result of CBMC. In this program, there is an unnamed member in the `struct x`. The C standard says "unnamed members of objects of structure and union type do not participate in initialization", and "unnamed members of structure objects have indeterminate value even after initialization". Thus, the variables `a`, `b`, and `c` should be assigned as 2, 3, 4, respectively. But CBMC does not follow these rules, and the member `b` is not correctly assigned as 3. The bug in Figure 2b is also a language feature related bug. Although it may not be possible to follow every rule in C standard, software model checkers should clearly explain which rules are supported, and define the boundary of their abilities.

Configuration related bugs. Software model checkers usually have many configuration options. For example, CBMC has the options to set the loop unwinding bounds, and CPAchecker has the options to setup the specification file, SMT solver, and model checking mode (*e.g.*, k-inductive, BMC). Figure 6h shows the program which triggers a bug in SeaHorn but can be solved by changing the configuration. SeaHorn incorrectly gives the *unsafe* result on this program. This bug appears when using both the IC3 and BMC engines of SeaHorn. The developers comment that SeaHorn can give the expected *safe* result with the `inline` configuration. Besides this bug, SeaHorn gives different answers in many other cases when switching between the `pf` and `bpf` configurations. While `pf` is the default configuration in SeaHorn which unrolls the loop dynamically, `bpf` means unrolling the loop statically. In many cases, SeaHorn gives incorrect answers when using `pf`, because `pf` unrolls the loop incorrectly. Using `bpf` can avoid these issues. However, without clear documentations, it is hard for users to figure out which configurations should be used to get the correct results.

4.4 Discussions

SV-COMP benchmarks & Undefined behaviors. We also selected the seed programs from SV-COMP benchmarks [44], which are the *de facto* benchmarks for software verification competition. These benchmarks consist of six categories of programs with different features. We selected the programs from two categories, *i.e.*, *ReachSafety* and *SoftwareSystems*, as the seed programs, since these categories focus on reachability checking. To adapt the benchmark programs for our purposes, we (1) removed the original error labels, and (2) replaced the symbolic inputs (*e.g.*, specified by `__VERIFIER_nondet_int`) in each program with the randomly generated concrete values for the variables of different types. In particular, we generated 5 groups of random inputs for each program. At last, we selected 1,123 seed programs and generated 5,615 test programs, and used Approach III to test the three model checkers.

We finally found 5 unique bugs (3 SeaHorn bugs and 2 CPAchecker bugs). Among these bugs, 1 bug in SeaHorn and 1 bug in CPAchecker are duplicate with the bugs we found by the GCC regression test suite, and the rest are new bugs. Figure 7a shows a new bug found by a SV-COMP program in SeaHorn. The variables `st1` and `st2` are less than `0`, so that `st1+st2` should be less than `0`. Then `br1` should be 1 and `br2` should be `0`, the error label should be unreachable. However, SeaHorn answers *unsafe* to this case which means the error label should be reachable. The SeaHorn developers have confirmed this bug and fixed in the newest version.

Compared with the GCC test suite, the SV-COMP benchmarks only found a few bugs. Then, we carefully inspected the test programs, and found this can be explained by three main reasons. First, the software model checkers have been thoroughly tested by these benchmark programs when participating in the competition. If some bugs were found, they have already been fixed. Second, most of the inconsistent cases reported by using SV-COMP benchmarks are related to undefined behaviors (*e.g.*, uninitialized arrays and local variables, visiting the memory out of bound), and some unsupported features (*e.g.*, floating point variables and recursions). Figure 7b shows a case reduced from a SV-COMP program with undefined behavior. The array `arg` is uninitialized, and thus the elements in this array are non-deterministic. The variable `br1` equals to 5 at line 10 when MCFuzz profiles the program. However, `br1` may not be equal to 5 since the elements in `arg` could be assigned by any value due to the undefined behavior. In such a case, model checkers are allowed to output any checking result. Therefore, all the inconsistent cases due to undefined behaviors cannot be concluded as bugs of model checkers.

Usability of software model checkers. The usability of software model checkers is very important in practice. However, we find it is not that satisfactory due to unclear user manuals, lack of necessary warning messages, or even crash failures. First, we find the user manuals are not very informative for us (as experienced users of program analyzers) to choose appropriate configurations. Especially

```c
int st1,st2,br1,br2;
void check(){
  if (st1 + st2 <= 1)
    br1++;
  else br2++;
}
int main(){
  st1 = -3; st2 = -1;
  br1 = 0; br2 = 0;
  check();
  if (br2 != 0)
    __VERIFIER_error();
  return 0;
}
```

(a) Sample bug found in SeaHorn

```c
int main(){
  int arg[5];
  int i=0, br1=0, br2=0;
  int num = 5;
  while (i < num){
    if(arg[i]%2==0)
      br1++;
    i++;
  }
  if(br1 != 5)
    __VERIFIER_error();
  return 0;
}
```

(b) Undefined behavior example

Figure 7: Evaluation on SV-COMP benchmarks

for SeaHorn, sometimes when we chose different configurations, the opposite verification results may be given. We believe it should not be the obligation for users to infer the effect of different configurations. An informative user manual with clear examples can greatly improve usability. Second, we find the model checkers do not fully support the features of C language. However, they sometimes silently give incorrect verification results without any warnings if some unsupported library functions or language features exists in the program under check. It is necessary for model checkers to give some warning messages to alert the users, or conservatively answer *unknown*. Third, it is also better to clearly define which language features are supported in user manual. Benefited from our bug reports, the developers of software model checkers have improved their user manuals and warning messages.

Developers feedback. We have received very positive feedback from the developers of software model checkers. The CPAchecker developers said, "Thank you very much for finding and reporting these cases! This is very helpful for us". They enhanced the original test suite by the test programs we reported, and further found more bugs in other verification modes by using these programs. The SeaHorn developers said, "I am working on improving soundness on such small examples. Thank you for the examples. They are very useful.". The CBMC developers also thanked for the bugs we reported and added the test programs to their test suite.

Generality and limitations. In this paper, we propose the branch reachability fuzzing approach to validate software model checkers. In general, our approach could be adapted to any other static analysers or verifiers. The modular design of MCFuzz is flexible to test any software model checker. Our approach can also be used to generate benchmark programs for the software model checker competition. But our approach requires the seed program should be valid, deterministic and terminating for generating test oracles.

5 RELATED WORK

The reliability of program analysis tools is very crucial since these tools are now widely used to improve the quality or ensure the correctness of software systems [14]. There exists work on validating various program analyzers, *e.g.*, static analyzers [20], symbolic execution engines [30], pointer analyses [52], refactoring engines [21], abstract interpreters [13, 39], and compliers [34, 36, 43, 53, 54].

Kapus *et al.* [30] use random program generation with differential testing to find bugs in several symbolic execution tools (*e.g.*, KLEE, CREST and FuzzBALL). Wu *et al.* [52] cross-check the pointer aliases during concrete program execution with the aliases found by static pointer alias analyzers to find bugs. Daniel *et al.* [21] detect potential bugs in refactoring engines by inspecting whether the two tools yield different refactored programs. Bugariu *et al.* [13] automatically test the soundness and precision of the implementations of abstract domains, which is the core of abstract interpretation-based tools. Qiu *et al.* [42] compare the results of three static analyzers (*e.g.*, FlowDroid, AmanDroid and DroidSafe) for identifying information flows in mobile applications, and reveal many inaccuracies based on a same set of benchmark. Pauck *et al.* [40] similarly evaluate several taint-analysis tools for Android applications to see whether they keep their promises. Different from these, ours is the first work to systematically validate software model checkers, a type of formal program verifier, and uncover many, diverse bugs.

Compilers can be viewed as another particular type of program analyzer. Yang *et al.* [53] propose Csmith, which uses differential testing to hunt bugs in C compilers. Csmith found more than 325 previously-unknown bugs in compilers. Later, the equivalence modulo inputs (EMI) technique [34] and its variants [35, 43] were applied to stress-test C compilers, and found over 1,000 unique bugs in GCC and LLVM. Inspired by EMI, Lidbury *et al.* [37] test OpenCL compilers to find miscompilation bugs. These work mainly use the idea of differential testing [38] or metamorphic testing [15] since the test oracles are not available. In contrast, our work leverage dynamic execution to produce test oracles, which can directly validate model checkers. As for differential testing, since the SMC implementations do not have rigorous standardisation (*e.g.*, CPAchecker supports floating point while SeaHorn does not), it may not work well for testing software model checkers.

6 CONCLUSION

In this paper, we have proposed a branch reachability fuzzing approach to automatically validating software model checkers and realized it in the MCFuzz tool. MCFuzz discovered 62 unique bugs in three state-of-the-art software model checkers, almost all of which have been confirmed or fixed, clearly demonstrating its effectiveness. We have also carefully analyzed these bugs and how they have been triaged by the developers. The bugs are diverse, which we classified into seven categories. The developers appreciated our bug reports and promptly responded to them, further highlighting the importance of the problem. Our approach is also general and can be adapted to validate other static analyzers or verifiers.

ACKNOWLEDGMENTS

We thank the anonymous ESEC/FSE reviewers for their valuable feedback. Our special thanks go to the SMC developers, specifically Philipp Wendler, Arie Gurfinkel, Jorge Navas, Daniel Kroening, Michael Tautschnig and Matthias Güdemann, who provided us with much help, insight and advice. Chengyu Zhang was partially supported by the China Scholarship Council, and NSFC Projects No. 61572197 and No. 61632005, Yichen Yan by ECNU Project of Funding Overseas Short-term Studies, and Geguang Pu by China HGJ Project No. 2017ZX01038102-002 and NSFC Project No. 61532019.

REFERENCES

[1] Anonymous. 2019. List of bugs found by MCFuzz. Retrieved Feb. 2019 from https://github.com/MCFuzzer

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic predicate abstraction of C programs. In *PLDI*. 203–213.

[3] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *POPL*. 1–3.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE TSE* 41, 5 (2015), 507–525.

[5] Dirk Beyer. 2017. Software Verification with Validation of Results (Report on SV-COMP 2017). In *TACAS*. 331–349.

[6] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Generating Tests from Counterexamples. In *ICSE*. 326–335.

[7] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast. *STTT* 9, 5-6 (2007), 505–525.

[8] Dirk Beyer and M Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *CAV*. 184–190.

[9] Dirk Beyer and Thomas Lemberger. 2017. Software Verification: Testing vs. Model Checking. In *HVC*. 99–114.

[10] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *TACAS*. 193–207.

[11] Aaron R. Bradley. 2011. SAT-based model checking without unrolling. In *VMCAI*. 70–87.

[12] Aaron R. Bradley. 2012. IC3 and beyond: Incremental, Inductive Verification.. In *CAV*. 4.

[13] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically testing implementations of numerical abstract domains. In *ASE*. 768–778.

[14] Cristian Cadar and Alastair F. Donaldson. 2016. Analysing the program analyser. In *ICSE Companion*. 765–768.

[15] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. HKUST-CS98-01, Hong Kong University of Science and Technology.

[16] Alessandro Cimatti and Alberto Griggio. 2012. Software model checking via IC3. In *CAV*. 277–293.

[17] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *JACM* 50, 5 (2003), 752–794.

[18] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *TACAS*. 168–176.

[19] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. 238–252.

[20] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NFM*. 120–125.

[21] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *ESEC/FSE*. 185–194.

[22] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software verification using k-induction. In *SAS*. 351–368.

[23] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE TCAD* 27, 7 (2008), 1165–1178.

[24] Facebook. 2019. Infer static analyzer. Retrieved Feb. 2019 from http://fbinfer.com/

[25] Gordon Fraser, Franz Wotawa, and Paul Ammann. 2009. Testing with model checkers: a survey. *STVR* 19, 3 (2009), 215–261.

[26] Susanne Graf and Hassen Saïdi. 1997. Construction of abstract state graphs with PVS. In *CAV*. 72–83.

[27] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn verification framework. In *CAV*. 343–361.

[28] Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *PASTE*. 54–61.

[29] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM CSUR* 41, 4 (2009), 21.

[30] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *ASE*. 590–600.

[31] Alexey V. Khoroshilov, Vadim S. Mutilin, Alexandre Petrenko, and Vladimir Zakharov. 2009. Establishing Linux Driver Verification Process. In *PSI*. 165–176.

[32] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M Clarke. 2013. Automatic abstraction in SMT-based unbounded software model checking. In *CAV*. 846–862.

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. 75.

[34] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. 216–226.

[35] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*. 386–399.

[36] Xavier Leroy. 2009. Formal verification of a realistic compiler. *CACM* 52, 7 (2009), 107–115.

[37] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. *PLDI* (2015), 65–76.

[38] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[39] Jan Midtgaard and Anders Møller. 2017. Quickchecking static analysis properties. *STVR* 27, 6 (2017).

[40] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *ESEC/FSE*. 331–341.

[41] Zvonimir Pavlinovic, Akash Lal, and Rahul Sharma. 2016. Inferring annotations for device drivers from verification histories. In *ASE*. 450–460.

[42] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *ISSTA*. 176–186.

[43] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *OOPSLA*. 849–863.

[44] SV-COMP. 2019. Competition on Software Verification (SV-COMP). Retrieved Feb. 2019 from https://sv-comp.sosy-lab.org/2019/

[45] The Clang Team. 2019. Clang 9 documentation: LibTooling. Retrieved Feb. 2019 from https://clang.llvm.org/docs/LibTooling.html

[46] The CompCert Team. 2019. Using the CompCert C interpreter. Retrieved Feb. 2019 from http://compcert.inria.fr/man/manual004.html

[47] The Eclipse Team. 2019. Eclipse CDT (C/C++ Development Tooling). Retrieved Feb. 2019 from https://www.eclipse.org/cdt/

[48] The GCC Team. 2019. GCC, the GNU Compiler Collection. Retrieved Feb. 2019 from https://gcc.gnu.org/

[49] The ISO Team. 2019. ISO/IEC 9899:2018:Information technology – Programming languages – C. Retrieved Feb. 2019 from https://www.iso.org/standard/74528.html

[50] The SVCOMP Team. 2019. 7th Competition on Software Verification (SV-COMP). Retrieved Feb. 2019 from https://sv-comp.sosy-lab.org/2018/results/results-verified/

[51] The SMTInterpol Team. 2019. SMTInterpol: an Interpolating SMT Solver. Retrieved Feb. 2019 from https://ultimate.informatik.uni-freiburg.de/smtinterpol/

[52] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. 2013. Effective dynamic detection of alias analysis errors. In *ESEC/FSE*. 279–289.

[53] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. 283–294.

[54] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *PLDI*. 347–361.