Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing

Muhammad Numair Mansur
MPI-SWS, Germany
numair@mpi-sws.org

Maria Christakis
MPI-SWS, Germany
maria@mpi-sws.org

Valentin Wüstholz
ConsenSys, Germany
valentin.wustholz@consensys.net

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

ABSTRACT

Formal methods use SMT solvers extensively for deciding formula satisfiability, for instance, in software verification, systematic test generation, and program synthesis. However, due to their complex implementations, solvers may contain critical bugs that lead to unsound results. Given the wide applicability of solvers in software reliability, relying on such unsound results may have detrimental consequences. In this paper, we present STORM, a novel blackbox mutational fuzzing technique for detecting critical bugs in SMT solvers. We run our fuzzer on seven mature solvers and find 29 previously unknown critical bugs. STORM is already being used in testing new features of popular solvers before deployment.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

automated testing, blackbox fuzzing, SMT solvers

ACM Reference Format:
Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409763

1 INTRODUCTION

The *Satisfiability Modulo Theories* (SMT) problem [11] is the decision problem of determining whether logical formulas are satisfiable with respect to a variety of background theories. More specifically, an SMT *formula* generalizes a Boolean SAT formula by supplementing Boolean variables with predicates from a set of theories. As an example, a predicate could express a linear inequality over real variables, in which case its satisfiability is determined with the theory of linear real arithmetic. Other theories include bitvectors, arrays, and integers [28], to name a few.

SMT solvers, such as CVC4 [10] and Z3 [24], are complex tools for evaluating the satisfiability of SMT instances. A typical SMT *instance* contains assertions of SMT formulas and a satisfiability check (see Figs. 2 and 3 for examples). SMT solvers are extensively used in formal methods, most notably in software verification (e.g., Boogie [8] and Dafny [34]), systematic test case generation (e.g., KLEE [18] and Sage [30]), and program synthesis (e.g., Alive [36]). Due to their high degree of complexity, it is all the more likely that SMT solvers contain correctness issues, and due to their wide applicability in software reliability, these issues may be detrimental.

Tab. 1 shows classes of bugs that may occur in SMT solvers. We restrict the classification to bugs that manifest themselves as an incorrect solver result. For bugs in class A, the solver is *unsound* and returns `unsat` (i.e., unsatisfiable) for instances that are satisfiable. These bugs are known as *refutational soundness bugs* in the SMT community. Class B refers to bugs where the solver returns `sat` (i.e., satisfiable) for unsatisfiable instances. A solver is *incomplete* when it returns `unknown` for an instance that lies in a decidable theory fragment. We categorize such bugs in class C. Finally, bugs in class D indicate crashes where the solver does not return any result.

We call bugs in class A *critical* for two main reasons. First, such bugs may cause unsoundness in program analyzers that rely on SMT solvers. As an example, consider a software verifier (e.g., Dafny [34]) or a test case generator (e.g., KLEE [18]) that checks reachability of an error location by querying an SMT solver. If the solver unsoundly proves that the error is unreachable (e.g., returns `unsat` for the path condition to the error), then the verifier will verify incorrect code and the testing tool will not generate inputs that exercise the error.

Second, it is much harder to safeguard against bugs in class A than bugs in other classes. Specifically, consider that, when an instance is found to be `sat`, the solver typically provides a *model*, that is, an assignment to all free variables in the instance such that it is satisfiable. Therefore, bugs in class B could be detected by simply evaluating the instance under the model generated by the solver (assuming that the model is correct). If this evaluation returns false, then there is a B bug. Bugs in class C are detected whenever the solver returns `unknown` for an instance that lies in a decidable theory fragment, and bugs in class D are detected when the solver crashes.

Related work. Early work on testing SMT solvers presented FuzzSMT [14], a blackbox grammar-based fuzzer for generating

This work is licensed under a Creative Commons Attribution International 4.0 License.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409763

Table 1: Classes of bugs in SMT solvers. GT stands for ground truth and SR for solver result.

GT \ SR	sat	unsat	unknown	Crash
sat		A	C	D
unsat	B		C	D

syntactically valid SMT instances (from scratch) for bitvectors and arrays. Since the satisfiability of the generated formulas is unknown, the main goal of this fuzzer is to detect crashes in solver implementations (class D). Critical bugs (class A) may only be detected with differential testing, when multiple solvers disagree on the satisfiability of a generated SMT instance.

The above idea was recently extended to enable fuzzing string solvers by a tool called StringFuzz [13]. Similarly to FuzzSMT, StringFuzz generates formulas from scratch. In addition, it can transform existing string instances, but without necessarily preserving their satisfiability. Consequently, critical bugs in a single string solver cannot be detected given that the satisfiability of the formulas is not known a priori—differential testing would again be needed.

Even more recently, there emerged another technique for testing string solvers [16], which synthesizes SMT instances such that their satisfiability is known by construction. This ground truth is used to derive test oracles. Violating these oracles indicates bugs of classes A and B.

Note that finding bugs in classes C and D does not require knowing the ground truth. As a result, any of the above techniques can in principle detect such bugs as a by-product.

Our approach. In this paper, we present a general blackbox fuzzing technique for detecting critical bugs in any SMT solver. In contrast to existing work, our technique does not require a grammar to synthesize instances from scratch. Instead, it takes inspiration from state-of-the-art mutational fuzzers (e.g., AFL [5]) and generates new SMT instances by mutating existing ones, called *seeds*. The key novelty is that our approach generates satisfiable instances from any given seed. As a result, our fuzzer detects a critical bug whenever an SMT solver returns unsat for one of our generated instances. We implement our technique in an open-source tool called STORM, which has the additional ability to effectively minimize the size of bug-revealing instances to facilitate debugging.

Contributions. Our paper makes the following contributions:

(1) We present a novel blackbox mutational fuzzing technique for detecting critical bugs in SMT solvers.
(2) We implement our technique in an open-source fuzzer[1] that is already being used for testing new features of solvers before deployment.
(3) We evaluate the effectiveness of our fuzzer on seven mature solvers and 43 logics. We found 29 previously unknown critical bugs in three solvers (or nine solver variants) and 15 different logics.

Outline. The next section gives an overview of our approach. Sect. 3 explains the technical details, and Sect. 4 describes our implementation. We present our experimental evaluation in Sect. 5, discuss related work in Sect. 6, and conclude in Sect. 7.

2 OVERVIEW

To give an overview of our fuzzing technique for SMT solvers, we first describe a few interesting examples of STORM in action and then explain what happens under the hood on a high level.

In action. One of the critical bugs[2] found by STORM was in Z3's QF_LIA logic, which stands for quantifier-free linear integer arithmetic. We opened a GitHub issue to report this bug, which resulted in an eight-comment discussion between two Z3 developers on how to resolve it. Note that eight comments (or in fact any discussion) on how to fix a bug is typically uncommon. From the GitHub issues we have seen, developers simply acknowledge an issue or additionally ask for a minimized SMT instance. The issue was closed but re-opened a day later with more comments on what still needs to be fixed. The issue was closed for the last time three days after that. Based on our understanding and explanations by the developers, this bug was triggered by applying Gomory's cut on an input that did not satisfy a fundamental assumption of the cut. STORM was able to generate an instance that violated this assumption and led to misapplying Gomory's cut. The fix in Z3 included changing the implementation of the cut.

STORM detected another critical bug[3] in Z3's Z3str3 string solver [12]. According to a developer of Z3str3, the bug existed for a long time before STORM found it. During this time, it remained undetected even though Z3str3 was being tested with fuzzers exclusively targeting string solvers [13, 16]. A simplified version of the SMT instance that revealed the bug is shown on the right of Fig. 2. (We will discuss it in detail later in this section.)

A third critical bug[4] was found in Z3's tactic for applying dominator simplification rules. The instance that was generated by STORM and revealed the bug spanned 194 lines. The minimization component of STORM reduced this instance to 15 lines. A simplified version of the instance is shown on the left of Fig. 3. (We discuss it later in this section.) A developer of the buggy tactic asked us which application generated this instance, thinking that it was a tool he developed during his PhD thesis. When we mentioned that it was STORM, he replied "*What? Your random generator could have done my PhD thesis?? &@#%, you should have told me sooner :)*". This demonstrates STORM's ability to generate realistic SMT instances that can be difficult to distinguish from instances produced by client applications of SMT solvers.

In Sect. 5, we describe in more detail our experience of using STORM to test both mature solver implementations as well as new features before their deployment.

Under the hood. We now give a high-level overview of our fuzzing technique, which operates in three phases. Fig. 1 depicts each of these phases.

[1] https://github.com/Practical-Formal-Methods/storm

[2] https://github.com/Z3Prover/z3/issues/2871

[3] https://github.com/Z3Prover/z3/issues/2994

[4] https://github.com/Z3Prover/z3/issues/3052

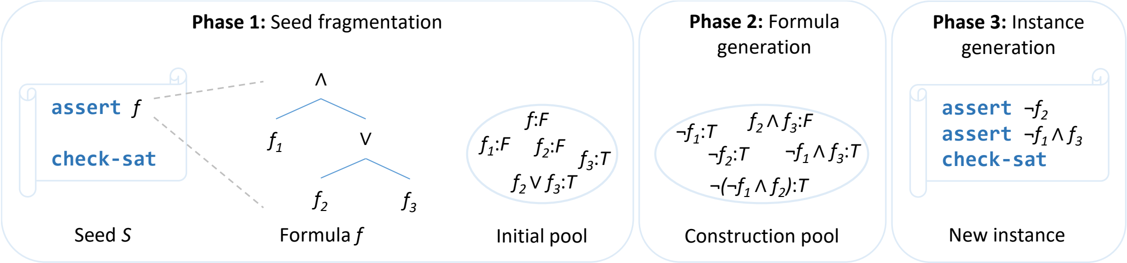

Figure 1: Overview of the three STORM phases.

```
(declare-const S String)
(assert (str.in.re S (re.++ re.allchar (re.++
  (str.to.re "7;") (re.++ re.allchar
    (str.to.re "aa"))))))
(assert (not (str.in.re S (re.union re.allchar
  (str.to.re "X'jafa")))))
(check-sat)
```

```
(declare-const S String)
(assert
(let ((a (str.in.re S (re.++ re.allchar (re.++
  (str.to.re "7;") (re.++ re.allchar
    (str.to.re "aa"))))))
(let ((b (not (str.in.re S (re.union re.allchar
  (str.to.re "X'jafa"))))))
(let ((c (and (not b) (not a))))
(not c))))))
(check-sat)
```

Figure 2: Original seed instance from SMT-COMP 2019 on the left, and simplified instance revealing critical bug in Z3's Z3str3 string solver on the right.

```
(declare-fun A () Bool)
(declare-fun B () Bool)

(assert (not B))
(assert (not (and (not A) B)))
(assert A)

(check-sat-using dom-simplify)
```

```
(declare-fun A () Bool)
(declare-fun B () Bool)

(assert (and (not B) A))

(check-sat-using dom-simplify)
```

Figure 3: Simplified instance revealing critical bug in Z3's `dom-simplify` tactic on the left, and logically equivalent instance not revealing the bug on the right.

The first phase, *seed fragmentation*, takes as input a seed SMT instance S. For instance, imagine an instance with multiple assertions. Each assertion contains a logical formula, such as f in the figure, potentially composed of Boolean sub-formulas (i.e., predicates), such as $f_2 \vee f_3$, f_1, f_2, and f_3 in the figure. Initially, STORM generates a random assignment of all free variables in the formulas in S. Then, STORM recursively fragments the formulas in S into all their possible sub-formulas. For example, f is broken down into f_1 and $f_2 \vee f_3$, each of these is in turn broken down into its Boolean sub-formulas, and so on. The valuation (i.e., truth value) of each (sub-)formula, T or F, is computed based on the random assignment. All formulas together with their valuations are inserted in an initial pool as shown in the figure.

The second phase, *formula generation*, uses the formulas in the initial pool to build new formulas. The valuation of each new formula is computed based on the valuations of its constituent initial formulas. All new formulas with their valuations are inserted in a construction pool as shown in the figure. For instance, initial formulas f_2 and f_3 are used to construct a new formula $f_2 \wedge f_3$.

The third phase, *instance generation*, uses formulas from both pools to generate new SMT instances. The reason for having the two pools is to be able to control the frequency with which initial and constructed formulas appear in the new instances. Instances generated during this phase have a different Boolean structure than the seeds. However, their basic building blocks, that is, the initial formulas that could not be fragmented further, remain unchanged. This is what allows STORM to generate realistic instances. In addition, all new instances are satisfiable by construction.

Therefore, a critical bug is detected whenever an SMT solver returns `unsat` for a STORM-generated instance. In such a case, STORM uses *instance minimization* to minimize the size of the instance revealing the bug.

Examples. The left of Fig. 2 shows a seed instance from the international SMT competition SMT-COMP 2019 [3]. Starting from this seed, STORM generated the (simplified) instance on the right, which revealed the critical bug in Z3str3 described above. Z3str3 derives length constraints from regular-expression membership predicates. The bug that STORM exposed here is that such a length constraint, which is implied by membership in a regular expression, was not asserted by the string solver.

It is easy to see that the first asserted formula on the left corresponds to variable `a` on the right, while the second asserted formula on the left corresponds to variable `b` on the right. Therefore, the seed essentially checks for satisfiability of `a` ∧ `b`. On the right, `c` is equivalent to ¬a ∧ ¬b, and the instance checks for satisfiability of ¬c, thus, of a ∨ b. This shows that even small mutations to the Boolean structure of a formula can be effective in revealing critical issues in solvers. In fact, such mutations can result in triggering different parts of a solver's implementation, e.g., different simplifications, heuristics, or optimizations.

This is also evidenced by the example in Fig. 3. The instance on the left reveals the critical bug in Z3's `dom-simplify` tactic described earlier. It essentially checks the satisfiability of ¬B ∧ ¬(¬A ∧ B) ∧ A, which is logically equivalent to ¬B ∧ A. Observe, however, that the logically equivalent formula, shown on the right of Fig. 3, does not trigger the bug.

Consequently, the benefit of fuzzing the Boolean structure of seed instances is two-fold. First, it is effective in detecting critical issues in solvers. Such issues are by definition far more serious and complex than other types of bugs, such as crashes, since they can, for instance, result in verifying incorrect safety-critical code. Second, fuzzing only the Boolean structure of seeds helps generate realistic SMT instances. This is confirmed by the above comments on the tactic bug from the Z3 developer who thought that the STORM instance was generated by his own PhD tool. This was also confirmed by other solver developers with whom we interacted.

3 OUR APPROACH

In this section, we describe our fuzzing technique and how it solves two key challenges in detecting critical bugs in SMT solvers: (1) how to generate non-trivial SMT instances, and (2) how to determine if a critical bug is exposed. The latter demonstrates how STORM addresses the oracle problem [9] in the context of soundness testing for solvers. Finally, we describe how we minimize bug-revealing instances to reduce their size. This step is crucial for solver developers as it significantly facilitates debugging.

3.1 Fuzzing Technique

Given an SMT instance as seed input, our fuzzing approach proceeds in three main phases: (1) seed fragmentation, (2) formula generation, and (3) instance generation. Seed fragmentation extracts sub-formulas from the seed. These will be used as building blocks for generating new formulas in the second phase. Lastly, instance generation creates new, satisfiable SMT instances based on the generated formulas, invokes the SMT solver under test on each of these instances, and uses the solver result as part of the test oracle to detect critical bugs.

Alg. 1 describes these three phases in detail. Function Fuzz takes the initial seed S and several additional parameters that bound the fuzzing process (explained below). As a first step, the function populates an initial pool P_{init} of formulas (line 13) with formula fragments of the seed S.

Algorithm 1: Core fuzzing procedure in STORM.

```
1  procedure PopulateInitialPool($S, D_{max}$)
2      $A \leftarrow$ GetAsserts($S$)
3      $M \leftarrow$ RandAssignment($A$)
4      $P \leftarrow$ EmptyPool()
5      foreach $pred \in S$
6          if $\neg$ExceedsDepth($pred, D_{max}$) then
7              $v \leftarrow$ IsTrue($M, pred$)
8              $P \leftarrow$ Add($P, pred, v$)
9      return $P$
10
11 procedure Fuzz($S, NC, NM, D_{max}, A_{max}$)
12     // Phase 1: Seed fragmentation
13     $P_{init} \leftarrow$ PopulateInitialPool($S, D_{max}$)
14
15     // Phase 2: Formula generation
16     $P_{constr} \leftarrow$ EmptyPool()
17     while Size($P_{constr}$) $<$ $NC$ do
18         $f_1, v_1 \leftarrow$ RandFormula($P_{init}, P_{constr}$)
19         $op \leftarrow$ RandOp()
20         if $op = AND$ then
21             $f_2, v_2 \leftarrow$ RandFormula($P_{init}, P_{constr}$)
22             $f \leftarrow AND(f_1, f_2)$
23             $v \leftarrow v_1 \wedge v_2$
24         else
25             $f \leftarrow NOT(f_1)$
26             $v \leftarrow \neg v_1$
27         if $\neg$ExceedsDepth($f, D_{max}$) then
28             $P_{constr} \leftarrow$ Add($P_{constr}, f, v$)
29
30     // Phase 3: Instance generation
31     $B \leftarrow$ EmptyList()
32     $m \leftarrow 0$
33     while $m < NM$ do
34         // Number of generated assertions
35         $ac \leftarrow$ (RandInt() % $A_{max}$) + 1
36         $A \leftarrow$ EmptyList()
37         while $0 < ac$ do
38             $f, v \leftarrow$ RandFormula($P_{init}, P_{constr}$)
39             if $\neg v$ then
40                 // Negation of $f$ to guarantee assertion satisfiability
41                 $f \leftarrow NOT(f)$
42             $A \leftarrow$ Append($A, f$)
43             $ac \leftarrow ac - 1$
44         // Invocation of SMT solver under test
45         $r \leftarrow$ CheckSAT($A$)
46         // Test oracle
47         if $r = UNSAT$ then
48             $B \leftarrow$ Append($B, A$)
49         $m \leftarrow m + 1$
50     return $B$
```

To this purpose, function PopulateInitialPool extracts all assertions in the seed and generates a random assignment M, i.e., an assignment of values to free variables. In our implementation, we use a separate SMT solver (i.e., different from the one under test) to generate a model for the assertions (or their negation if the assertions are unsatisfiable). Next, we iterate over all predicates (i.e., tree-shaped Boolean sub-formulas as in Fig. 1) in the seed. We use assignment M to evaluate those predicates for which the tree depth does not exceed a bound D_{max}. This valuation v is crucial for subsequent phases of the fuzzing process, and we add both the formula *pred* and v to the initial pool, which is essentially a map from formulas to valuations. Note that, by fragmenting the seed, the initial pool already contains a large number of non-trivial formulas that would be difficult to generate from scratch (e.g., with a grammar-based fuzzer).

Algorithm 2: Depth-minimization procedure in STORM.

```
1 procedure MinimizeDepth($S, NC, NM, D_{min}, D_{max}, A_{max}$)
2   if $D_{max} \leq D_{min}$ then
3     return $S$
4   $D \leftarrow (D_{min} + D_{max})/2$
5   $B \leftarrow$ Fuzz($S, NC, NM, D, A_{max}$)
6   if $0 <$ Size($B$) then
7     $S_{min} \leftarrow$ SelectSeedWithSmallestDepth($B$)
8     return MinimizeDepth($S_{min}, NC, NM, D_{min}, D, A_{max}$)
9   return MinimizeDepth($S, NC, NM, D + 1, D_{max}, A_{max}$)
```

In the second phase, we populate the construction pool P_{constr} by adding *NC* new formulas of maximum depth D_{max}. These formulas are generated randomly by selecting one of two Boolean operators, logical *AND* (lines 21–23) and *NOT* (lines 25–26). Note that this set of operators is functionally complete, thus allowing us to generate any Boolean formula. We construct a new formula f by conjoining two existing formulas (f_1 and f_2 with valuations v_1 and v_2) in the case of *AND* and negating an existing formula (f_1 with valuation v_1) in the case of *NOT*. Existing formulas are randomly selected from the pools. Before adding the resulting formula f to the construction pool, we derive its valuation v from the valuations of its sub-formulas (lines 23 and 26).

In essence, the second phase enriches the set of existing formulas by generating new ones without requiring a complete grammar for all syntactic constructs. Instead, we use a minimal, but functionally complete, grammar for *Boolean formulas*. This significantly simplifies formula generation without sacrificing expressiveness. Note that a separate pool for newly constructed formulas allows having control over how many of them are used in the instances generated in the third phase. On the right of Fig. 2, this step is responsible for generating the formulas on lines 8 and 9 that ultimately amount to checking the satisfiability of `a ∨ b`.

Once the two pools are populated, we use them to generate *NM* SMT instances that we feed to the solver under test. To assemble a new instance *A*, we create up to A_{max} assertions (*ac* on line 35) by randomly picking formulas from the pools. If the valuation of a selected formula is true, we directly assert it, otherwise we assert its negation. This ensures that all assertions are satisfiable. Of course, the same holds for instance *A* consisting of these assertions in addition to a satisfiability check. We now leverage this fact when feeding the SMT instance to the solver under test (line 45). The oracle reveals a critical bug if the solver returns *UNSAT*.

3.2 Instance Minimization

In practice, our fuzzing technique often generates bug-revealing instances that are very large, containing deeply nested formulas and several assertions. This can considerably complicate debugging for solver developers.

Adapting established minimization techniques based on delta debugging [46] might seem like a natural fit for this use case. However, the special nature of critical bugs complicates this task in comparison to other classes of bugs, such as crashes. For minimizing crashing instances, it is sufficient to minimize the original instance (e.g., by dropping assertions) *while preserving the crash*. In contrast, for instances that exhibit a critical bug, the behavior that should be preserved is more involved, that is, *the instance should be minimized such that the buggy solver still returns* `unsat` *while the ground truth remains* `sat`. This requires either satisfiability-preserving minimizations or a trusted second solver that can act as a ground-truth oracle by rejecting minimizations that do not preserve satisfiability. Unfortunately, the only state-of-the-art delta debugger for SMT instances, ddSMT [40], does not preserve satisfiability. (Note that ddSMT is the successor of deltaSMT [2], which was used to minimize instances generated by FuzzSMT [14].) Moreover, a second trusted solver is not always available (e.g., for new theories or solver-specific features and extensions).

To overcome these limitations, we developed a specialized minimization technique that directly leverages the bounds of our fuzzing procedure to obtain smaller instances (see Alg. 2 for depth minimization). By repeatedly running the fuzzing procedure on a *buggy* seed instance, this algorithm attempts to find the minimum values for D_{max} and A_{max} that still reveal a critical bug. It uses binary search to first minimize the number of assertions (analogous to MinimizeDepth in Alg. 2) and subsequently the depth of asserted formulas. Note that the fuzzing procedure may report multiple bug-revealing instances, and we recursively minimize the smallest with respect to the bound being minimized (line 8). Our evaluation shows that this technique works more reliably than leveraging ddSMT and a second solver (see Sect. 5.5).

4 IMPLEMENTATION

Seeds. STORM uses the Python API in Z3 to manipulate SMT formulas for generating new instances. It can, therefore, only fuzz instances within the logics supported by Z3. In practice, this is not an important restriction since Z3 supports a very large number of logics. Moreover, STORM requires seeds to be expressed in an extension of the SMT-LIB v2 input format [4] supported by Z3. Note that SMT-LIB is the standard input format used across solvers.

Random assignments. STORM uses Z3 to generate a random model for a given seed (line 3 of Alg. 1). Note, however, that bugs in Z3 resulting in a wrong model do not affect our fuzzer. In fact, given any assignment, our technique just requires correct valuations for predicates in the initial pool. In theory, computing these valuations is relatively straightforward since the assignment provides concrete values for all free variables; simply substituting variables with values should be sufficient for quantifier-free predicates. In practice, we use Z3 to compute predicate valuations and have not encountered any bugs in this solver component.

Random choices. Our implementation provides concrete instantiations of functions RandOp and RandFormula from Alg. 1 as follows. RandOp returns *AND* with probability 50% and *NOT* otherwise. Function RandFormula selects a formula from one of the pools uniformly at random, but with probability 30% from the initial pool and from the construction pool otherwise.

Incremental mode. Many solvers support a feature called *incremental mode*. It allows client tools to push and pop constraints when performing a large number of similar satisfiability queries (e.g., checking feasibility of paths with a common prefix during symbolic execution). To efficiently support this mode, solvers typically use dedicated algorithms that reuse results from previous queries;

in fact, SMT-COMP [3] features a separate track to evaluate these algorithms. To test incremental mode, STORM is able to generate SMT instances that contain push and pop instructions in addition to regular assertions.

5 EXPERIMENTAL EVALUATION

In this section, we address the following research questions:

RQ1: How effective is STORM in detecting new critical bugs in SMT solvers?

RQ2: How effective is STORM in detecting known critical bugs in SMT solvers?

RQ3: How do the assertion and depth bounds of STORM impact its effectiveness?

RQ4: How effective is our instance minimization at reducing the size of bug-revealing instances?

RQ5: To what extent do STORM-generated instances increase code coverage of SMT solvers?

We make our implementation open source[5]. To support open science, we include all data, source code, and documentation necessary for reproducing our experimental results.

5.1 Solver Selection

We used STORM to test seven popular SMT solvers, which support the SMT-LIB input format [4] and regularly participate in the international SMT competition SMT-COMP [3]. Specifically, we selected Boolector [42], CVC4 [10], MathSAT5 [20], SMTInterpol [19], STP [29], Yices2 [26], and Z3 [24].

In addition to the above mature implementations, STORM was also used to test new features of solvers. In particular, the developers of Yices2 asked us to test the new bitvector theory in the MCSAT solver [31] of Yices2, which is based on the model-constructing satisfiability calculus [25]. MCSAT is an optional component of Yices2, which is dedicated to quantifier-free non-linear real arithmetic. STORM did not find bugs in this new theory of MCSAT, and the theory was integrated with the main version of Yices2 shortly after. In our experimental evaluation, it is therefore tested as part of Yices2.

Moreover, the developers of Z3 asked us to test a new arithmetic solver (let us refer to it as Z3-AS), which they have been preparing for the last two years. It comes with better non-linear theories and has just replaced the legacy arithmetic solvers in Z3. According to the Z3 developers, STORM could help expedite the integration of this new feature by finding bugs early, which it did. Since Z3-AS has just now been integrated with the current version of Z3 and we have only been testing it independently, we include it separately in our evaluation.

Due to the success of STORM in detecting intricate critical bugs in Z3-AS, the Z3 developers described our fuzzer as being "*extremely useful*" and have now asked us to test Z3's current debug branch (let us refer to it as Z3-DBG). Z3-DBG implements a variety of new solver features in which STORM has already detected a critical bug (see Sect. 5.5).

Finally, the developers of the Z3str3 string solver [12] asked us to provide them with STORM-generated string instances. They became aware of STORM since it detected several critical issues in Z3str3, which we reported. Note that Z3str3 is developed by the same group of people as StringFuzz [13]. We, therefore, suspect that STORM found bugs in Z3str3 that StringFuzz could not find, especially since StringFuzz does not target critical bugs. The STORM-generated instances that we provided (in addition to the bug-revealing ones that we reported) were used as a regression test suite during the development of performance enhancements in Z3str3. According to a developer of Z3str3, our instances helped reveal critical bugs introduced by these enhancements. Most of these bugs were due to missing or incorrect axioms in Z3str3.

5.2 Logic Selection

In our experimental evaluation, for each solver, we identified well supported logics based on its participation in SMT-COMP 2019 [3]. In certain cases, we also added logics identified as error-prone by the solver developers, such as `QF_FP`. In general however, STORM can handle the intersection of all logics supported by the SMT-LIB v2 input format and all logics supported by Z3. The latter constraint emerges because our implementation relies on Z3's APIs for generating the mutated SMT instances (see Sect. 4).

Tab. 2 shows the tested logics for each solver. (The second column and second to last row of the table should be ignored for now.) The logic abbreviations are explained in the SMT-LIB standard [4], but generally speaking, the following rules hold. `QF` stands for quantifier-free formulas, `A` for arrays, `AX` for arrays with extensionality, `BV` for bitvectors, `FP` for floating-point arithmetic, `IA` for integer arithmetic, `RA` for real arithmetic, `IRA` for integer real arithmetic, `IDL` for integer difference logic, `RDL` for rational difference logic, `L` before `IA`, `RA`, or `IRA` for the linear fragment of these arithmetics, `N` before `IA`, `RA`, or `IRA` for the non-linear fragment, `UF` for the extension that allows free sort and function symbols, `S` for strings, and `DT` for datatypes.

5.3 Benchmark Selection

For our experiments, we used as seeds all non-incremental SMT-LIB instances in SMT-COMP 2019 [3]. We also used all SMT-LIB instances in the regression test suites of CVC4, Yices2, and Z3. The second column of Tab. 2 shows how many seeds correspond to each tested logic. The second to last row of the table ("Unsp.") refers to instances in which the logic is unspecified—the solver may use any.

In general, we only tested each solver with logics, and thus instances, it supports. For seeds without a specified logic, we only generated mutations of those that each solver could handle.

5.4 Experimental Setup

For our experiments, we used the following setting for STORM unless stated otherwise: $D_{max} = 64$, $A_{max} = 64$, NC between 200 and 1500, and NM between 300 and 1000 (see Alg. 1). Both NC and NM were adjusted dynamically within the above ranges based on the size of the initial pool. The goal was to use larger values for larger initial pools, and thus, larger seeds.

We performed all experiments on a 32-core Intel ® Xeon ® E5-2667 v2 CPU @ 3.30GHz machine with 256GB of memory, running Debian GNU/Linux 10 (buster).

[5] https://github.com/Practical-Formal-Methods/storm

Table 2: The tested logics per solver and the number of seed instances per logic.

		SMT Solvers						
Logic	**Seeds**	Boolector	CVC4	MathSAT5	SMTInterpol	STP	Yices2	Z3
`ALIA`	42		✓		✓			✓
`AUFNIA`	3		✓					✓
`LRA`	2444		✓		✓			✓
`QF_ALIA`	42		✓		✓		✓	✓
`QF_AUFNIA`	3		✓	✓				✓
`QF_DT`	1602		✓					✓
`QF_LRA`	1049		✓		✓		✓	✓
`QF_RDL`	261		✓				✓	✓
`QF_UFIDL`	444		✓				✓	✓
`QF_UFNRA`	38		✓	✓			✓	✓
`UFDTLIA`	327		✓					✓
`AUFDTLIA`	728		✓					✓
`AUFNIRA`	1490		✓					✓
`NIA`	14		✓					✓
`QF_ANIA`	8		✓	✓				✓
`QF_AX`	555		✓	✓	✓		✓	✓
`QF_FP`	40418		✓	✓				✓
`QF_NIA`	23901		✓	✓			✓	✓
`QF_S`	24323		✓					✓
`QF_UFLIA`	580		✓		✓		✓	✓
`UFLIA`	9524		✓		✓			✓
`AUFLIA`	3273		✓		✓			✓
`BV`	5750	✓	✓					✓
`NRA`	3813		✓					✓
`QF_AUFBV`	49	✓	✓				✓	✓
`QF_BV`	3872	✓	✓			✓	✓	✓
`QF_IDL`	843		✓		✓		✓	✓
`QF_NIRA`	3		✓	✓			✓	✓
`QF_UF`	7481		✓		✓		✓	✓
`QF_UFLRA`	936		✓		✓		✓	✓
`UF`	7596		✓		✓			✓
`UFLRA`	17		✓		✓			✓
`AUFLIRA`	2268		✓		✓			✓
`LIA`	388		✓		✓			✓
`QF_ABV`	8310	✓	✓				✓	✓
`QF_AUFLIA`	1310		✓		✓			✓
`QF_BVFP`	17196		✓	✓				✓
`QF_LIA`	2104		✓		✓		✓	✓
`QF_NRA`	4067		✓	✓			✓	✓
`QF_UFBV`	1238	✓	✓				✓	✓
`QF_UFNIA`	478		✓	✓			✓	✓
`UFDT`	4527		✓					✓
`UFNIA`	4446		✓					✓
Unsp.	5825	–	–	–	–	–	–	–
Total	193586	5	43	10	17	1	19	43

Comparison with state of the art. Except for a single tool [16], all existing testing tools for SMT solvers do not use oracles to detect critical bugs. They, therefore, require differential testing of multiple solvers to identify such bugs. In RQ2, we evaluate the effectiveness of STORM at detecting existing critical bugs, including the publicly reported bugs found by the most closely related tool [16]. Recall that this tool supports only the theory of strings.

Table 3: Previously unknown, unique, and confirmed critical bugs found by STORM in the tested SMT solvers.

SMT Solver	Incremental Mode	Logics	Critical Bugs
MathSAT5		QF_FP QF_BVFP	2
Yices2		QF_UFIDL QF_UF	2
Yices2	✓	QF_UFIDL QF_UFLRA	2
Z3		QF_UFLIA QF_BV UF LIA QF_BVFP QF_LIA	8
Z3	✓	QF_FP QF_S	3
Z3str3		QF_S	6
Z3-AS		AUFNIRA QF_NIA AUFLIRA QF_NRA	4
Z3-AS	✓	AUFNIRA	1
Z3-DBG		QF_NIA	1

5.5 Experimental Results

We now discuss our experimental results for each of the above research questions.

RQ1: New critical bugs. Tab. 3 shows critical bugs found by STORM in the SMT solvers we tested. The first column of the table shows the solvers. We list Z3str3 separately as it is not the default string solver in Z3. The second column denotes whether bugs were found in the incremental mode of a solver, which essentially corresponds to a different solver variant. The third column lists the logics in which bugs were found, and the last column shows the number of bugs. **Overall, STORM found 29 critical bugs in three mature solvers (or nine solver variants) and 15 different logics.**

All of these bugs are previously unknown, unique, and confirmed by the solver developers. Out of the 29 critical bugs, 19 have already been fixed in the latest solver versions. Note that the bugs were only detected by STORM-generated instances, i.e., none were detected by the seeds. In addition to the bugs in the table, STORM was also able to detect known bugs as well as other issues (i.e., of classes C and D) as a by-product, which we do not report here.

The feedback from solver developers is very positive, and we have been discussing it throughout the paper. As another example, a Yices2 developer told us that STORM found real bugs and that it is especially useful to have the ability to test the incremental mode of solvers. He also mentioned that they used to run FuzzSMT [14] on all theories, and that now this fuzzer runs continuously on new theories generating "*infinite*" instances. FuzzSMT, however, does not target critical bugs, and for this reason, they run VoteSMT [6] to differentially test solvers and detect incorrect Yices2 results. Despite this, STORM detected four new critical bugs in Yices2.

Another Yices2 developer commented on the severity of two of the bugs that STORM found. He mentioned that one was in the pre-processing component and "*easy to fix (and an obvious mistake in retrospect) but it was in a part of Yices that had probably not been exercised much*". "*The other one was much more tricky to trace and fix, it was related to a combination of features and optimization in the E-graph, not localized to a single module*".

RQ2: Known critical bugs. In this research question, we evaluate the effectiveness of STORM in reproducing known critical bugs. We, therefore, collected all critical bugs that were reported for the solvers under test during the three-month period between Nov 15 and Feb 15, 2020. We focused only on bugs with a subsequent fix (i.e., closed issues on GitHub). Out of the seven solvers, we exclude MathSAT5 because it is closed source, and bugs may only be reported via email. We also exclude Boolector, SMTInterpol, and STP because no critical bugs were reported for these solvers during the above time period. For the remaining three solvers, CVC4, Yices2, and Z3, there were 6, 1, and 14 critical bugs with a fix, respectively, after excluding all the bugs that we reported.

We ran STORM on the solver version in which each bug was found. Since developers typically add fixed bugs to their regression tests, we removed all seeds that revealed any of these bugs (without being mutated). We collected all generated instances for which each solver incorrectly returned `unsat`. To ensure that STORM actually found the reported bug (and not a different one), we ran all bug-revealing instances against the first solver version with the corresponding fix. If the solver now returned `sat` for at least one of the instances, we counted the bug as reproduced.

For each of the three solvers, STORM was able to reproduce 1 (CVC4), 1 (Yices2), and 4 (Z3) critical bugs, so 6 out of a total of 21. Therefore, **if STORM had run on these solver versions, it would have prevented approximately 1/3 of the critical-bug reports in a three-month period.** Given that during this period we reported 10 additional bugs detected by STORM in these solvers, it is possible that our fuzzer would have been able to reproduce more bugs if it had run longer or if it was being run continuously.

We also ran STORM on the publicly reported critical bugs found by Bugariu and Müller [16] (regardless of when they were reported). STORM was able to reproduce them.

RQ3: Fuzzing bounds. To evaluate the effect of the fuzzing bounds of STORM, we only considered closed bugs. We used all 19 closed bugs reported by us from RQ1 except for those in Z3-AS (the original commits could not be retrieved due to a rebase in the branch) for a remaining of 14 bugs. In addition, we used all reproduced bugs from RQ2 for a total of 21 bugs.

For each of these bugs, we randomly selected a seed file that had allowed STORM to detect the bug in RQ1 or RQ2. We performed eight independent runs of STORM (with random seeds different from the ones used in RQ1 and RQ2 to avoid bias) to evaluate the effect of the different fuzzing bounds. STORM was unable to reproduce one Yices2 bug from RQ1 with any of the eight random seeds; we therefore do not include it in the results shown in Fig. 4.

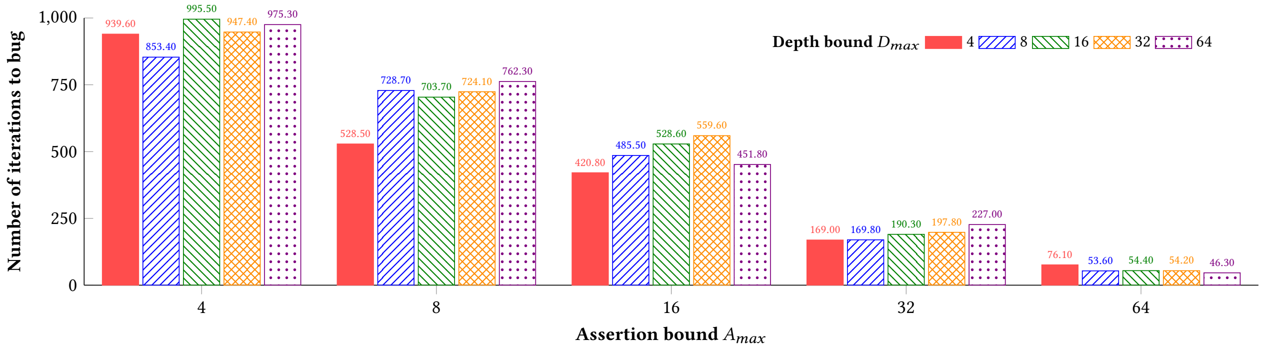

Figure 4: Median number of iterations to find bugs with different configurations of STORM. Each bar corresponds to a configuration with a certain depth and assertion bound.

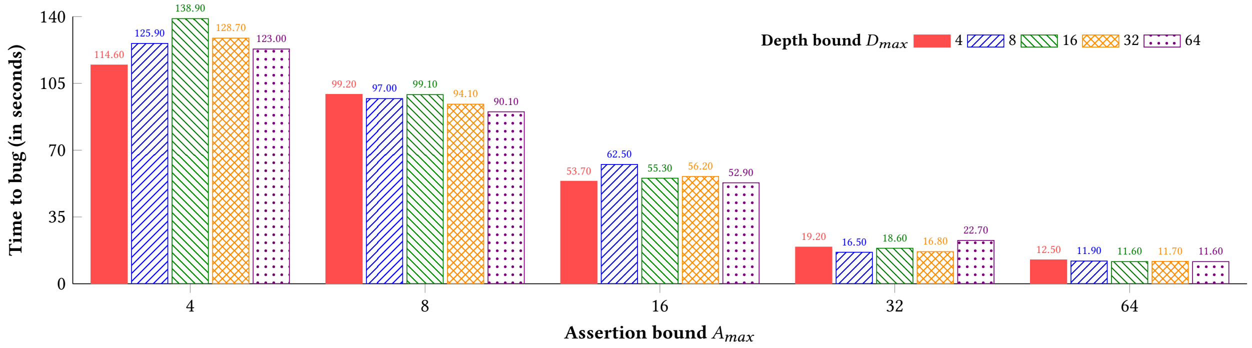

Figure 5: Median time (in seconds) to find bugs with different configurations of STORM. Each bar corresponds to a configuration with a certain depth and assertion bound.

For the assertion and depth bounds A_{max} and D_{max}, we used five different settings: 4, 8, 16, 32, and 64. Fig. 4 shows the median number of iterations (i.e., generated instances) until the bug was found for different combinations of these settings. We can observe that **a large assertion bound reduces the number of iterations significantly (e.g., up to 12x for D_{max} = 4).** In contrast, the trend for the depth bound is less clear, which suggests that it has a less significant effect and is mostly useful for minimizing instances. We can observe very similar trends when comparing the median time to find the bug (see Fig. 5).

RQ4: Instance minimization. We now evaluate the effectiveness of our instance minimization. To this end, we collect all instances revealing the 20 bugs of RQ3 that are generated by STORM with its default configuration (Sect. 5.4).

The results of minimizing these instances using binary search (BS) and delta debugging (ddSMT [40]) are shown in Tab. 4. We perform eight independent minimization runs and report median results. Instance size is measured in terms of the number of bytes, the number of assertions, and the maximum formula depth in an assertion. A dash for ddSMT means either that the instance could not be minimized or that ddSMT does not support a construct in the instance. As outlined in Sect. 3.2, we had to adapt ddSMT for this use case by invoking a second solver to reject minimizations that would not preserve satisfiability; we used the version of the solver that fixed the corresponding bug for this purpose.

Despite these adaptations, we observed that ddSMT could not minimize the instances for bugs 2, 3, 4, 5, 13, 14, and 18. We suspect that its search space of possible minimizations might not contain more complex transformations that would be required to both preserve satisfiability *and* the bug. We observed the same outcome when running ddSMT on instances that were first minimized using binary search.

For bugs 10, 11, and 19, ddSMT does not support `str.to.re` and `str.at`, which are supported by Z3str3. For bugs 7, 8, 9, 12, 15, and 16, ddSMT does not support **`check-sat-using`**, which is supported by Z3. Recall that STORM accepts seed instances expressed in the extension of the SMT-LIB format that is supported by Z3 (Sect. 4), whereas ddSMT only supports the standard.

Overall, this experiment shows that **our minimization procedure works more reliably and is able to significantly reduce buggy instances (median reduction of 82.7%).** However, for the cases where both procedures produced results, the ddSMT-based minimization procedure was able to produce smaller instances. This is not entirely surprising given that BS uses the fuzzer, which treats predicates not containing other predicates (i.e., ground- or leaf-predicates) as atomic building blocks. For instance, for bug 17, the instance that was minimized with BS contains several complex ground-predicates that ddSMT is able to minimize further. We expect that more involved combinations of the two approaches could produce even better results.

Table 4: Size of original and minimized bug-revealing instances. Instance size is shown in terms of the number of bytes / number of assertions / maximum formula depth.

Bug ID	Unminimized Instances	Minimized by BS	Minimized by ddSMT
1	23430/ 64/ 5	20801/ 61/ 5	321/ 4/ 0
2	3756/ 6/ 12	3756/ 6/ 12	–/ –/ –
3	9641/ 20/ 8	5276/ 19/ 9	–/ –/ –
4	66209/ 33/ 56	13086/ 8/ 2	–/ –/ –
5	64071/ 44/ 27	24326/ 24/ 6	–/ –/ –
6	37943/ 51/ 2	4247/ 5/ 0	575/ 4/ 0
7	19408/ 64/ 3	1025/ 5/ 2	–/ –/ –
8	19235/ 27/ 2	4002/ 5/ 0	–/ –/ –
9	23659/ 51/ 4	2004/ 5/ 0	–/ –/ –
10	4275/ 6/ 1	1514/ 5/ 1	–/ –/ –
11	39585/ 64/ 16	5832/ 16/ 2	–/ –/ –
12	22017/ 58/ 5	1013/ 5/ 2	–/ –/ –
13	180082/ 62/ 8	7210/ 5/ 2	–/ –/ –
14	7934/ 10/ 8	4431/ 10/ 5	–/ –/ –
15	72490/ 50/ 0	5455/ 5/ 2	–/ –/ –
16	35725/ 33/ 3	2591/ 5/ 2	–/ –/ –
17	17180/ 21/ 57	1146/ 5/ 0	421/ 1/ 0
18	10176/ 14/ 0	2586/ 14/ 0	–/ –/ –
19	16812/ 51/ 4	13137/ 33/ 6	–/ –/ –
20	16826/ 30/ 1	5163/ 5/ 1	601/ 7/ 0

RQ5: Code coverage. A Yices2 developer mentioned that they use fuzzer-generated instances to enrich their regression tests such that they achieve higher coverage. In this research question, we therefore evaluate whether STORM is able to increase coverage.

We selected one of the solvers (Z3) and four random logics in which we found bugs (`QF_UFLIA, AUFNIRA, UF, LIA`). We then computed the line and function coverage when running Z3 on all the instances from SMT-COMP 2019 [3] for these logics (10054 seeds). The result is shown in the first row of Tab. 5. At the same time, we randomly selected 5 instances from each logic and ran STORM with $NM = 500$ and a single new random seed to generate exactly 500 new instances for each of the 20 seed instances. Tab. 5 shows that, as more instances are generated, coverage increases noticeably (9540 more lines and 4605 more functions after only 500 generated instances). This demonstrates that **running STORM on only a small number of seed instances is able to result in a noticeable coverage increase over a large number of instances from a well known benchmark set.**

5.6 Threats to Validity

We identify the following threats to the validity of our experiments.

Selection of seeds. STORM requires seed instances as input, and our results do not necessarily generalize to other seeds [43]. However, we selected as seeds instances from SMT-COMP 2019 [3] as well as regression test suites of solvers. We believe that our selection is sufficiently broad to mitigate this threat. In addition, we make our tool open source so it may be run with different seeds.

Table 5: Code coverage increase as more instances are generated by STORM.

Generated Instances	Line Coverage	Function Coverage
0	58219	26256
100	66945	30498
200	67063	30524
300	67119	30547
400	67208	30598
500	67759	30861

Selection of solvers. The bugs found by STORM depend on the solvers and logics that we tested. However, we selected a wide range of different, mature solvers and logics to mitigate this threat.

Randomness in fuzzing. A common threat when evaluating fuzzers is related to the internal validity [43] of their results. To mitigate systematic errors that may be introduced due to random choices of our fuzzer, we used random seeds to ensure deterministic results and performed experiments for eight different seeds.

6 RELATED WORK

SMT solvers are core components in many program analyzers, and as a result, their reliability is of crucial importance. Although it is feasible to verify SAT and SMT *algorithms* [27, 35, 37], it is challenging and time consuming to verify even very basic SAT- or SMT-solver *implementations*. Verifying state-of-the-art, high-performance solver implementations, such as CVC4 [10] and Z3 [24], is completely impractical. For these reasons, there is a growing interest in testing such solvers, alongside related efforts that focus on testing entire program analyzers.

Testing SAT and SMT solvers. FuzzSMT [14] focuses on finding crashes of SMT solvers for bitvector and array instances. It uses grammar-based blackbox fuzzing to generate crash-inducing instances and minimizes any such instances with delta debugging [2, 46]. Brummayer et al. [15] extend this line of work to SAT and QBF solvers. In contrast, STORM performs mutational fuzzing, and its minimization procedure leverages the fuzzer and its bounds regarding the number of assertions and the formula depth.

StringFuzz [13] targets testing of string solvers. In addition to randomly generating syntactically valid instances using a grammar, it is also able to mutate or transform formulas in existing instances. However, since not all of its transformations preserve satisfiability, it is not easily possible to leverage metamorphic testing [9] to detect critical bugs. In contrast to both FuzzSMT and StringFuzz, the satisfiability of all STORM-generated instances is known.

Recently, Bugariu and Müller [16] proposed an automated testing technique that synthesizes SMT instances for the string theory. The true satisfiability of the generated instances is derived by construction and used as a test oracle. In contrast, STORM performs mutational fuzzing and supports a wide range of theories.

Unlike the above approaches that fuzz the input instances of solvers, Artho et al. [7] and Niemetz et al. [41] develop model-based API testing frameworks for SAT and SMT solvers. These focus on testing various API parameters and solver options.

Testing program analyzers. Kapus and Cadar [32] combine random program generation with differential testing [39] to find bugs in symbolic-execution engines. Their technique is inspired by existing compiler-testing techniques (e.g., Csmith [45]) and used to test KLEE [18], CREST [1], and FuzzBALL [38].

Cuoq et al. [23] use randomly generated programs to test the Frama-C static-analysis platform [21]. Bugariu et al. [17] present a fuzzing technique for detecting soundness and precision issues in implementations of abstract domains—the core components of abstract interpreters [22]. They use algebraic properties of abstract domains as test oracles and find bugs in widely used domains. Recently, Taneja et al. [44] proposed a testing technique for identifying soundness and precision issues in static dataflow analyses by comparing results with a sound and maximally precise SMT-based analysis; they rely on the SMT solver to provide correct results.

Zhang et al. [47] develop a practical and automated fuzzing technique to test software model checkers. They focus on testing control-flow reachability properties of programs. More specifically, they synthesize valid branch reachability properties using concrete program executions and then fuse individual properties of different branches into a single safety property.

Klinger et al. [33] propose an automated technique to test the soundness and precision of program analyzers in general. Their approach is based on differential testing. From seed programs, they generate program-analysis benchmarks on which they compare the results of different analyzers.

7 CONCLUSION

In this paper, we have presented a novel fuzzing technique for detecting critical bugs in SMT solvers—key components of many state-of-the-art program analyzers. Conceptually, STORM is a blackbox mutational fuzzer that uses fragments of existing SMT instances to generate new, realistic instances. Its formula-generation phase takes inspiration from grammar-based fuzzers; it leverages a minimal, but functionally complete, grammar for Boolean formulas to generate new formulas from fragments found in seeds. Finally, it solves the oracle problem by generating instances that are satisfiable by construction.

ACKNOWLEDGMENTS

We are grateful to the reviewers for their constructive feedback and to the solver developers for their valuable help.

This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

REFERENCES

[1] [n.d.]. CREST: Concolic Test Generation Tool for C. http://www.burn.im/crest/.
[2] [n.d.]. DeltaSMT. http://fmv.jku.at/deltasmt.
[3] [n.d.]. The International Satisfiability Modulo Theories Competition. https://smt-comp.github.io.
[4] [n.d.]. The Satisfiability Modulo Theories Library. http://smtlib.cs.uiowa.edu.
[5] [n.d.]. Technical "Whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[6] [n.d.]. VoteSMT. http://fmv.jku.at/votesmt.
[7] Cyrille Artho, Armin Biere, and Martina Seidl. 2013. Model-Based Testing for Verification Back-Ends. In *TAP (LNCS, Vol. 7942)*. Springer, 39–55.
[8] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (LNCS, Vol. 4111)*. Springer, 364–387.
[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *TSE* 41 (2015), 507–525. Issue 5.
[10] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV (LNCS, Vol. 6806)*. Springer, 171–177.
[11] Clark W. Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*. Springer, 305–343.
[12] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A String Solver with Theory-Aware Heuristics. In *FMCAD*. IEEE Computer Society, 55–59.
[13] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *CAV (LNCS, Vol. 10982)*. Springer, 45–51.
[14] Robert Brummayer and Armin Biere. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *SMT*. ACM, 1–5.
[15] Robert Brummayer, Florian Lonsing, and Armin Biere. 2010. Automated Testing and Debugging of SAT and QBF Solvers. In *SAT (LNCS, Vol. 6175)*. Springer, 44–57.
[16] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. To appear.
[17] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*. ACM, 768–778.
[18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX, 209–224.
[19] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *SPIN (LNCS, Vol. 7385)*. Springer, 248–254.
[20] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *TACAS (LNCS, Vol. 7795)*. Springer, 93–107.
[21] Loïc Correnson, Pascal Cuoq, Florent Kirchner, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. 2011. *Frama-C User Manual*. http://frama-c.com//support.html.
[22] 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*. ACM, 238–252.
[23] 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 (LNCS, Vol. 7226)*. Springer, 120–125.
[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340.
[25] Leonardo de Moura and Dejan Jovanovic. 2013. A Model-Constructing Satisfiability Calculus. In *VMCAI (LNCS, Vol. 7737)*. Springer, 1–12.
[26] Bruno Dutertre. 2014. Yices 2.2. In *CAV (LNCS, Vol. 8559)*. Springer, 737–744.
[27] Jonathan Ford and Natarajan Shankar. 2002. Formal Verification of a Combination Decision Procedure. In *CADE (LNCS, Vol. 2392)*. Springer, 347–362.
[28] Vijay Ganesh. 2007. *Decision Procedures for Bit-Vectors, Arrays and Integers*. Ph.D. Dissertation. Stanford University, USA.
[29] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *CAV (LNCS, Vol. 4590)*. Springer, 519–531.
[30] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*. The Internet Society, 151–166.
[31] Dejan Jovanovic, Clark Barrett, and Leonardo de Moura. 2013. The Design and Implementation of the Model-Constructing Satisfiability Calculus. In *FMCAD*. IEEE Computer Society, 173–180.
[32] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *ASE*. IEEE Computer Society, 590–600.
[33] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *ISSTA*. ACM, 239–250.
[34] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*. Springer, 348–370.
[35] Stéphane Lescuyer and Sylvain Conchon. 2008. A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLs*.
[36] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *PLDI*. ACM, 22–32.
[37] Filip Maric. 2010. Formal Verification of a Modern SAT Solver by Shallow Embedding into Isabelle/HOL. *TCS* 411 (2010), 4333–4356. Issue 50.
[38] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *ASPLOS*. ACM, 337–348.
[39] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10 (1998), 100–107. Issue 1.
[40] Aina Niemetz and Armin Biere. 2013. ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In *SMT*. 36–45.
[41] Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Model-Based API Testing for SMT Solvers. In *SMT*. 10 pages.

[42] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. 2018. Btor2, BtorMC and Boolector 3.0. In *CAV (LNCS, Vol. 10981)*. Springer, 587–595.

[43] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *ICSE*. IEEE Computer Society, 9–19.

[44] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *CGO*. ACM, 81–93.

[45] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. ACM, 283–294.

[46] Andreas Zeller. 2009. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann.

[47] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *ESEC/FSE*. ACM, 763–773.