On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers

DOMINIK WINTERER*, ETH Zurich, Switzerland
CHENGYU ZHANG*, East China Normal University, China
ZHENDONG SU, ETH Zurich, Switzerland

We propose type-aware operator mutation, a simple, but unusually effective approach for testing SMT solvers. The key idea is to mutate operators of conforming types within the seed formulas to generate well-typed mutant formulas. These mutant formulas are then used as the test cases for SMT solvers. We realized type-aware operator mutation within the OpFuzz tool and used it to stress-test Z3 and CVC4, two state-of-the-art SMT solvers. Type-aware operator mutations are unusually effective: During one year of extensive testing with OpFuzz, we reported 1,092 bugs on Z3's and CVC4's respective GitHub issue trackers, out of which 819 unique bugs were confirmed and 685 of the confirmed bugs were fixed by the developers. The detected bugs are highly diverse — we found bugs of many different types (soundness bugs, invalid model bugs, crashes, *etc.*), logics and solver configurations. We have further conducted an in-depth study of the bugs found by OpFuzz. The study results show that the bugs found by OpFuzz are of high quality. Many of them affect core components of the SMT solvers' codebases, and some required major changes for the developers to fix. Among the 819 confirmed bugs found by OpFuzz, 184 were soundness bugs, the most critical bugs in SMT solvers, and 489 were in the default modes of the solvers. Notably, OpFuzz found 27 critical soundness bugs in CVC4, which has proved to be a very stable SMT solver.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: SMT solvers, Fuzz testing, Type-aware operator mutation

ACM Reference Format:
Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (November 2020), 25 pages. https://doi.org/10.1145/3428261

1 INTRODUCTION

Satisfiability Modulo Theory (SMT) solvers are important tools for many programming languages advances and applications, *e.g.*, symbolic execution [Cadar et al. 2008; Godefroid et al. 2005], program synthesis [Solar-Lezama 2008], solver-aided programming [Torlak and Bodik 2014], and program verification [DeLine and Leino 2005; Detlefs et al. 2005]. Incorrect results from SMT solvers can invalidate the results of these tools, which can be disastrous in safety-critical domains. Hence, the SMT community has undertaken great efforts to make SMT solvers reliable. Examples include the standardized input/output file formats for SMT solvers, semi-formal logic/theory specifications, extensive benchmark repositories, and yearly-held SMT solver competitions. To date, there are

*Both authors contributed equally to this work.

Authors' addresses: Dominik Winterer, ETH Zurich, Department of Computer Science, Switzerland, dominik.winterer@inf.ethz.ch; Chengyu Zhang, East China Normal University, Software Engineering Institute, China, dale.chengyu.zhang@gmail.com; Zhendong Su, ETH Zurich, Department of Computer Science, Switzerland, zhendong.su@inf.ethz.ch.

2475-1421/2020/11-ART193
https://doi.org/10.1145/3428261

```
; \phi
(assert (forall ((a Int))
        (exists ((b Int))
        ( distinct  (* 2 b) a))))
(check-sat)
```

```
; \phi_{test}
(assert (forall ((a Int))
        (exists ((b Int))
        ( =  (* 2 b) a))))
(check-sat)
```

Fig. 1. Type-aware operator mutation illustrated. We mutate the distinct operator in φ to the equals operator (see φ_{test}). Formula φ_{test} triggers a soundness bug in Z3 which reports sat on this unsatisfiable formula. https://github.com/Z3Prover/z3/issues/3973

several mature SMT solvers, among which Z3 [de Moura and Bjørner 2008] (5.8K stars on GitHub) and CVC4 [Barrett et al. 2011] (433 stars on GitHub) are the most prominent ones. Both Z3 and CVC4 are very stable and reliable. In Z3, there have been fewer than 150 reported soundness bugs in more than three years, while fewer than 50 in CVC4 in more than 8 years.[1] Despite this, SMT solvers are complex pieces of software and inevitably still have latent bugs. Various automated testing approaches [Blotsky et al. 2018; Brummayer and Biere 2009b; Bugariu et al. 2018] have been devised for finding bugs in SMT solvers. However, nearly all SMT solver soundness bugs have still been exposed directly by their client applications, not by these techniques. This has only begun to change with the recently proposed Semantic Fusion [Winterer et al. 2020] and STORM [Numair et al. 2020]. Both exposed a number of soundness bugs in Z3, while Semantic Fusion additionally exposed some soundness bugs in CVC4. Yet, it is unclear whether SMT solvers have reached a strong level of maturity and how many latent bugs remain in them.

Type-aware operator mutation. To answer this question, we introduce type-aware operator mutation, a simple, yet unusually effective approach for stress-testing SMT solvers. Its key idea is to mutate functions within SMT formulas with functions of conforming types. Fig. 1 illustrates type-aware operator mutation on an example formula. We replace the "`distinct`" in φ by an operator of conforming type, *e.g.*, the equals operator "=" to obtain formula φ_{test}. We then differentially test SMT solvers with φ_{test} as input and observe their results. If the results differ, *e.g.*, one SMT solver returns `sat` while the other returns `unsat`, we have found a soundness bug in either of the tested solvers. Formula φ_{test} is clearly unsatisfiable, as b cannot exist whenever a is odd. In fact, while CVC4 correctly returns `unsat` on φ_{test}, Z3 incorrectly reports `sat` on φ_{test}. Thus, φ_{test} has triggered a soundness bug in Z3 which was promptly fixed by Z3's main developer.

Bug hunting with OpFuzz. We have engineered OpFuzz, a practical realization of type-aware operator mutation. OpFuzz is unusually effective. During our bug hunting campaign from September 2019 to September 2020, we found and reported 1,092 bugs in Z3 and CVC4 issue trackers, among which 819 were confirmed by the developers and 685 were already fixed. We have found bugs across various logics such as (non-)linear integer and real arithmetic, uninterpreted functions, bit-vectors, strings, sets, sequences, array, floating-point, and combinations of these logics. Among these, most of the bugs (489) were found in the default modes of the solvers, *i.e.*, without additionally supplied options. This underpins the importance of our findings. We have found many high-quality soundness bugs in Z3 and notably also in CVC4, which has been proven to be a very robust SMT solver by previous work. The root causes of the bugs that we found are often complex and, sometimes require the developers to perform major code changes to fix the underlying issues. The developers of Z3 and CVC4 greatly appreciated our bug finding effort with comments like "Great find!", "Thanks a lot for the bug report!" or labelling our bug reports as "major".

[1] Data recorded prior to any SMT fuzzing campaigns: July 2010 to October 2019 for CVC4; April 2015 to October 2019 for Z3.

Approach	Bugs in Z3		Bugs in CVC4	
	soundness	all	soundness	all
StringFuzz [Blotsky et al. 2018]	0 (0)	1 (0)	-	-
BanditFuzz [Scott et al. 2020]	≥ 1 (0)	≥ 1 (0)	-	-
Bugariu *et al.* [Bugariu and Müller 2020]	3 (1)	5 (3)	0 (0)	0 (0)
YinYang [Winterer et al. 2020]	25 (24)	39 (36)	5 (5)	9 (8)
STORM [Numair et al. 2020]	21 (17)	27 (21)	0 (0)	0 (0)
OpFuzz	**157** (114)	**578** (316)	**27** (11)	**241** (185)

Fig. 2. Comparison of confirmed bugs found by OpFuzz against the bug findings of recent SMT solver testing approaches on the trunks of Z3 and CVC4. In parentheses: confirmed bugs in the default modes of the solvers. Performance issues are excluded from this comparison.

Comparison of OpFuzz with recent SMT fuzzers. In this paper, we use the term bug trigger to refer to a formula that triggers a bug in an SMT solver and the term bug to refer to a single unique bug in an SMT solver. Note, multiple bug triggers can be caused by the same underlying bug. All bug counts mentioned in this paper refer to unique bugs. Fig. 2 shows a comparison of OpFuzz with recent tools on SMT solver testing from the last two years in terms of bug counts. We have not considered older approaches and defer to the related work section (Section 6) for a detailed discussion and literature review. Recent approaches can be roughly separated into two categories: generators (StringFuzz [Blotsky et al. 2018], Bugariu and Müller's approach [Bugariu and Müller 2020], BanditFuzz [Scott et al. 2020]) and mutators (StringFuzz, YinYang [Winterer et al. 2020], STORM [Numair et al. 2020]). StringFuzz is a string formula generator that also comes with a mutator. It mainly targets performance issues in z3str3 [Berzish et al. 2017], an alternative string solver in Z3. StringFuzz can find correctness bugs as a by-product; the paper mentioned one. Bugariu and Müller's approach is a formula synthesizer for string logics generating formulas that are by construction (un)satisfiable. They found 5 bugs in Z3 in total with 3 soundness bugs, but none in CVC4. Recently, BanditFuzz, a reinforcement learning-based fuzzer has been proposed. Similar to StringFuzz, BanditFuzz's main focus is on performance issues in SMT solvers and less on correctness bugs. The authors have identified inconsistent results for 1600 syntactically different bug triggers on the four SMT solvers Z3, CVC4, MathSAT [Cimatti et al. 2013], Colibri, and 100 bug triggers in z3str3. However, the number of unique bugs in Z3 remains unclear as the authors did not reduce and report the bug triggers to filter out the duplicates. Among the mutation-based fuzzers, YinYang is an approach to stress-test SMT solvers by fabricating fused formula pairs that are by construction either (un)satisfiable. YinYang found 39 bugs in Z3 and 9 in CVC4. Another recent approach is STORM which is based on a three-phase process of seed fragmentation, formula generation and instance generation. STORM has found 27 bugs in Z3 with 21 being soundness bugs, but none in CVC4.

As Fig. 2 illustrates, our realization OpFuzz of type-aware operator mutation compares favorably against all existing approaches by a significant margin — OpFuzz found substantially more bugs in both Z3 and CVC4 in terms of all bugs, the soundness bugs in Z3 and CVC4, and bugs for the default modes of the solvers. Existing approaches also extensively tested Z3 and CVC4, and have missed the bugs found by OpFuzz. In summary, we make the following contributions in this paper:

- We introduce type-aware operator mutation, a simple, but unusually effective approach for stress-testing SMT solvers;
- We have realized type-aware operator mutation within our tool OpFuzz in no more than 212 lines of code. OpFuzz helps SMT solver developers and practitioners to stress-test SMT solver decision procedures regardless of the used logic and solver;

```
(set-logic NRA)
(declare-fun a () Real)
(declare-fun b () Real)
(assert (= (* a b) 1))
(check-sat) (get-model)
```

(a) Invalid model bug in CVC4.
https://github.com/CVC4/CVC4/issues/3407

```
(set-logic NRA)
(declare-fun a () Real)
(declare-fun b () Real)
(assert (> (* a b) 1))
(check-sat) (get-model)
```

(b) Mutating the equals operator in Fig. 3a to a greater operator makes the bug disappear.

```
(declare-fun f (Int) Bool)
(declare-fun g (Int) Bool)
(assert (distinct f g))
(check-sat)
(get-model)
```

(c) Invalid model bug in CVC4.
https://github.com/CVC4/CVC4/issues/3527

```
(declare-fun f (Int) Bool)
(declare-fun g (Int) Bool)
(assert (= f g))
(check-sat)
(get-model)
```

(d) Mutating the equals operator in Fig. 3c to a distinct operator makes the bug disappear.

```
(declare-fun x () Real)
(assert (distinct x (sin 4.0)))
(check-sat)
```

(e) CVC4 crashes on this formula.
https://github.com/CVC4/CVC4/issues/3614

```
(declare-fun x () Real)
(assert (>= x (sin 4.0)))
(check-sat)
```

(f) Mutating the distinct operator in Fig. 3e to a greater than operator makes the bug disappear.

Fig. 3. Left column: bug-triggering formulas in SMT-LIB format. Right column: formulas that were transformed from the corresponding bug-triggering formulas by a single operator change.

- Between September 2019 and September 2020, we stress-tested Z3 and CVC4 using OpFuzz, and reported 1,092 unique bugs on the respective GitHub issue trackers of Z3 and CVC4. Out of these, 819 bugs were confirmed, and 685 bugs were fixed. Most confirmed bugs were triggered in the default modes (489) of the solvers, and many were soundness bugs (184)[2];
- We have conducted an in-depth analysis of the bugs to understand in which parts of the SMT solvers these bugs occur. Furthermore, we examined the effort necessary for the developers to fix OpFuzz's bugs. Our results show that many bugs occur in the core parts of the SMT solvers, and some require the developers to perform major code changes.

Organization of the paper. Section 2 illustrates the idea behind type-aware operator mutation. Section 3 presents type-aware operator mutation formally, and shows how we apply it to SMT solver testing through our realization OpFuzz. We then present our empirical evaluation (Section 4) which includes detailed statistics about our bug findings. Section 5 introduces our quantitative analysis of the detected bugs and in-depth investigation on a set of sampled bugs to provide further insight. Finally, we survey related work (Section 6) and conclude (Section 7).

2 MOTIVATING EXAMPLES

This section gives a short introduction on the SMT-LIB [Barrett et al. 2010] language, the standard for describing SMT formulas, and then motivates our technique *type-aware operator mutation*.

SMT-LIB language. We consider the following subset of statements: `declare-fun`, `assert`, `check-sat` and `get-model`. Variables are declared as zero-valued functions. For example, the declaration

[2]All links to the bug reports are provided under the URL testsmt.github.io/opfuzz_bugs.html

```
(declare-fun a () Real)
(assert (> (/ (* 2 a) a) (* a a) 1))
(check-sat)
```

(a) Original formula

```
(declare-fun a () Real)
(assert (* (/ (* 2 a) a) (* a a) 1))
(check-sat)
```

(b) Syntactically incorrect mutant.

```
(declare-fun a () Real)
(assert (= (/ (* 2 a) a) (* a a) 1))
(check-sat)
```

(c) Syntactically correct mutant.

```
(declare-fun a () Real)
(assert (= (/ (* 2 a) a) (/ a a) 1))
(check-sat)
```

(d) Bug triggering mutant.

Fig. 4. Motivating examples for type-aware operator mutation.

`"(declare-fun a () Real)"` declares a variable of type real. An `assert` statement specifies constraints. The predicates within the constraints have different types, *e.g.*, the constraint `"(assert (<= (/ x 4) (* 5 x)))"` includes predicates of real and boolean types. Multiple constraints can be viewed as the conjunction of the constraints in each individual constraint statement. The `(check-sat)` statement queries the solver to decide on the satisfiability of a formula. If all constraints are satisfied, the formula is satisfiable; otherwise, the formula is unsatisfiable. We can obtain a model, *i.e.*, a satisfiable assignment, for a satisfiable formula by the `(get-model)` statement.

Type-aware operator mutation. We first examine three exemplary bugs that were found by our technique. Consider the formula in Fig. 3a on which CVC4 returns the following model: $a = -\frac{3}{2}$ and $b = -\frac{1}{2}$. This model is invalid as $a \cdot b \neq 1$. Mutating the equals operator = to the greater operator > hides this bug (see Fig. 3b). As another example, consider the formula in Fig. 3c. CVC4 gives an invalid model on this formula by setting $f = g = \mathit{false}$. Furthermore CVC4 crashes on the formula in Fig. 3e. Again in both cases, the bug disappears with a single operator change (see Fig. 3b and Fig. 3d). All illustrated cases show that operators play an important role in triggering SMT solver bugs. This inspired our technique, type-aware operator mutation, that is to stress-test SMT solvers via mutating operators and use the so mutated formulas for stress-testing SMT solvers.

However, substituting an operator with another arbitrary operator may not always yield a syntactically correct formula. As an example, consider Fig. 4a that presents a syntactically correct seed formula. By substituting the first operator greater than operator > to $*$, the formula becomes syntactically incorrect (see Fig. 4b). This is because the `assert` statement expects a boolean expression, while $*$ returns a real. The formula of Fig. 4b is of little value to testing an SMT solver's decision procedures since the solvers would reject such formulas already at a preprocessing stage. Hence, we have to consider the operator types for the substitutions, *i.e.*, avoid substituting an operator returning a boolean value, such as =, by an operator returning a real, such as $*$; neither should we substitute an operator with a single argument, like `not`, by an operator of two or more arguments, such as =. Instead we mutate the operators in a *type-aware* fashion. Consider the first > of the formula in Fig. 4a. It takes an arbitrary number of numeral arguments and returns a boolean. Candidates for its substitution are `<=`, `>=`, `<`, `=`, and `distinct`, all of which have a conforming type, *i.e.*, read more than one numerals and return a boolean. Therefore, we can safely substitute > of the formula in Fig. 4a with a random candidate, *e.g.*, =. As a result, we obtain the mutant formula in Fig. 4c. This formula is syntactically correct and can successfully pass the preprocessing phase of the SMT solvers. We call such mutations *type-aware operator mutations*. As we have the guarantee that the mutant is a type-correct formula, we can do iterative type-aware operator mutations. Given the mutant formula in Fig. 4c, we further substitute the second occurrence of $*$ with `/` safely. This yields the formula in Fig. 4d which triggered a soundness bug in Z3. Division by zero terms are specified in the Real and Int theories of the SMT-LIB as the uninterpreted terms, meaning that for a

Function types	Function Symbols
$\Gamma, A <: \top \vdash A \times \cdots \times A \rightarrow Bool$	`=, distinct`
$\Gamma \vdash Quantifier \times Bool \rightarrow Bool$	`exists, forall`
$\Gamma \vdash Bool \times \cdots \times Bool \rightarrow Bool$	`and, or, =>`
$\Gamma, Int <: Real \vdash Real \times \cdots \times Real \rightarrow Bool$	`<=, >=, <, >`
$\Gamma, Int <: Real \vdash Real \times \cdots \times Real \rightarrow Real$	`+, -, *, /`
$\Gamma \vdash Int \times \cdots \times Int \rightarrow Int$	`div`
$\Gamma \vdash Int \times Int \rightarrow Int$	`mod`

Fig. 5. SMT function symbols categorized by their type.

term `(/ t 0)` and arbitrary value `v`, the equation `(= v (/ t 0))` is satisfiable. In fact, we can set $a = 0$ to realize a model for the formula in Fig. 4d, *i.e.*, let the division by zero terms be 1 to satisfy the assert. Hence, the formula in Fig. 4d is satisfiable. However, Z3 incorrectly reports `unsat` on it.

3 APPROACH

In this section, we formally introduce type-aware operator mutation and propose OpFuzz, a fuzzer for stress-testing SMT solvers.

Background. We consider first-order logic formulas of the satisfiability modulo theories (SMT). Such a formula φ is satisfiable if there is at least one assignment on its variables under which φ evaluates to true. Otherwise, φ is unsatisfiable. We consider formulas to be realized by SMT-LIB programs[3] in which operators correspond to functions of the SMT theories. We use the terms functions and operators interchangeably unless stated otherwise.

For a formula φ, we define $F(\varphi)$ to be φ's set of (enumerated) function occurrences. For example, for $\varphi =$ `(+ (* 1 1) (- 2 (* 5 2)))`, we have $F(\varphi) = \{+_1, *_1, -_1, *_2\}$. $\varphi[f_1/f_2]$ describes the substitution of function f_2 by f_1 in φ. Expressions and functions are typed. For example, `1` is of type *Int*, `1.0` is of type *Real*, `"foo"` is of type *String*. Similarly functions also have types. We denote the type of a function f by $f : A \rightarrow B$ where A is the type of its arguments and B its return type. We use Γ to denote the static typing environment of the SMT-LIB language. For example, we write $\Gamma \vdash Int \times Int \rightarrow Int$ for the type of function `mod` and $\Gamma \vdash Int \times \cdots \times Int \rightarrow Int$ for the function `div`. $Int \times \cdots \times Int$ means function `div` accepts more than one argument with type *Int*. Fig. 5 shows selected functions and their types considered in this paper. We emphasize that our theory is not restricted to the functions used in Fig. 5. It can be extended to a richer set of functions and types according to the SMT-LIB standard.

Similar to other programming languages with types, we can define a subtyping relation for the SMT-LIB language. We now formalize a fragment of the SMT-LIB's type system. We define type *Int* to be a subtype of *Real*, *i.e.*, $\Gamma \vdash Int <: Real$. Let A be an arbitrary type, then we define type $A \times A$ to be a subtype of $A \times \cdots \times A$, *i.e.*, $\Gamma, A <: \top \vdash A \times A <: A \times \cdots \times A$. For two functions $f_1 : A_1 \rightarrow B_1$ and $f_2 : A_2 \rightarrow B_2$ with $A_1 <: A_2$ and $B_2 <: B_1$ we have

$$\frac{A_1 <: A_2 \quad B_2 <: B_1}{f_2 : A_2 \rightarrow B_2 <: f_1 : A_1 \rightarrow B_1}$$

[3]We consider the SMT-LIB language in version 2.6.

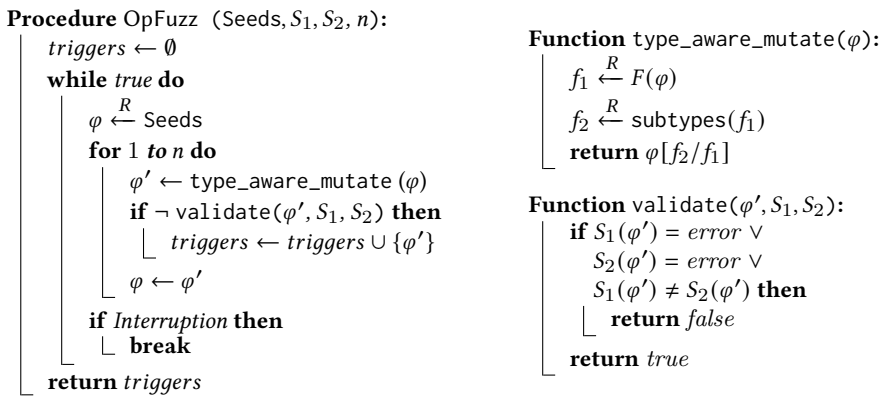

Fig. 6. Left: OpFuzz's main process. Right: Function `type_aware_mutate` realizes type-aware operator mutations and function `validate` differentially tests the SMT solvers S_1 and S_2.

For example, consider function `div` of type $Int \times \cdots \times Int \longrightarrow Int$ and function `mod` of type $Int \times Int \longrightarrow Int$ from Fig. 5. We can hence conclude that `div`'s type is a subtype of `mod`'s type:

$$\frac{\Gamma \vdash Int \times Int <: Int \times \cdots \times Int \quad \Gamma \vdash Int <: Int}{\Gamma \vdash Int \times \cdots \times Int \longrightarrow Int <: Int \times Int \longrightarrow Int}$$

We call a formula φ *well-typed* if it complies with the rules of SMT-LIB's typing system.

3.1 Type-Aware Operator Mutation

Having provided basic background, we present type-aware operator mutation, the key concept of this paper. We first introduce type-aware operator mutations and then show that type-aware operator mutants realize well-typed SMT-LIB programs.

Definition 3.1 (Type-aware operator mutation). Let φ be an SMT formula and let $f_1 : t_1$ and $f_2 : t_2$ be two of its functions. We say formula $\varphi' = \varphi[f_2/f_1]$ is a ***type-aware operator mutant*** of φ if $t2 <: t1$. Transforming φ to $\varphi[f_2/f_1]$ is called ***type-aware operator mutation***.

Proposition 3.2. *Type-aware operator mutants are well-typed.*

Proof. Let φ be a well-typed SMT formula and let φ' be a type-aware operator mutant of φ. According to Definition 3.1 we know that $\varphi' = \varphi[f_2/f_1]$ where $f_1 : t_1$ and $f_2 : t_2$ are two of φ's functions. By Definition 3.1, we also know $t2 <: t1$. This implies that all arguments of f_1 are also accepted by f_2 and all values returned by f_2 could be produced by f_1. Thus, f_2 accepts all the inputs provided by φ', and formula φ' accepts all the outputs of f_2. Therefore φ' is well-typed. □

Example 3.3. Consider φ = `(assert (= (mod 1 1) 1)` with $F(\varphi) = \{\texttt{mod}, =\}$. We randomly pick function `mod` from F, substitute it with a function that has its subtype, *e.g.*, function `div`. We get the following type-aware operator mutant φ' = `(assert (= (div 1 1) 1)`. As Proposition 3.2 shows, φ' is guaranteed to be well-typed. Thus, we can use φ' to test SMT solvers.

3.2 OpFuzz

We implemented OpFuzz, a type-aware operator mutation-based fuzzer, for stress-testing SMT solvers. OpFuzz leverages type-aware operator mutation to generate test inputs and validates the results of the SMT solvers via differential testing, *i.e.*, by comparing the results of two or more SMT

solvers and reporting their inconsistencies. Fig. 6 presents the algorithm of OpFuzz. OpFuzz takes a set of seed formulas `Seeds`, two SMT solvers S_1, S_2 and parameter n as its input. OpFuzz collects bug triggers in the set *triggers* which is initialized to the empty set. The main process runs inside a while loop until an interrupt is detected, *e.g.*, by the user or by a time or memory limit that is reached. We first choose a random formula φ from the set of formulas `Seeds` for initialization. In the while loop, we then perform a type-aware mutation on φ realized by the `type_aware_mutate` function. In the `type_aware_mutate` function, we first randomly pick a function f_1 from the set of functions in φ. Then, we randomly choose a function f_2 from the set of f_1's subtypes. The subtype function is realized based on Fig. 5. After we obtained $\varphi' = \varphi[f_2/f_1]$ by type-aware mutation on φ, we call the function `validate`. It tests two SMT solvers S_1 and S_2 via differential testing on the input formula φ'. First, it checks whether either of the solvers has produced an error on processing φ', e.g., the SMT solver did not terminate successfully, threw out an error message. We distinguish two cases: either φ' triggered an assertion violation or segmentation fault (crash), or a model validation error that occurs for solvers with model validation enabled (invalid model). In both cases, the function returns $false$. Otherwise, it checks whether the results of the solvers are different, and returns $false$ if so, else `validate` returns $true$ indicating that φ' has not exposed a bug trigger in neither of the solvers S_1 and S_2. OpFuzz realizes an n-times repeated type-aware operator mutation on every seed formula. The choice for parameter n is arbitrary but an n within 200 and 400 has worked well in practice.

OpFuzz is very light-weight. We realized OpFuzz in a total of only 212 lines of Python 3.7 code. OpFuzz can be run in parallel mode, which can significantly increase its throughput. Users can customize OpFuzz's command-line interface to test specific solvers and/or configurations. OpFuzz can be used with any SMT solver that takes SMT-LIB v2.6 files as its input. We have implemented the type-aware operator mutations *w.r.t.* the function symbols in Fig. 5.

4 EMPIRICAL EVALUATION

This section details our extensive evaluation with OpFuzz demonstrating the practical effectiveness of type-aware operator mutation for testing SMT solvers. Between September 2019 and September 2020, we were running OpFuzz to stress-test the SMT solvers Z3 [de Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011]. We have chosen Z3 and CVC4, since they (1) both are popular and widely used in academia and industry, (2) support a rich set of logics, and (3) adopt an open-source development model. During our testing period, we have filed numerous bugs on the issue trackers of Z3 and CVC4. This section describes the outcome of our efforts.

Result summary and highlights. In summary, OpFuzz is unusually effective.

- *Many confirmed bugs:* In one year, we have reported 1,092 bugs, and 819 unique bugs in Z3 and CVC4 have been confirmed by the developers.
- *Many soundness bugs:* Among these, there were 184 soundness bugs in Z3 and CVC4. Most notably, we have found 27 in CVC4.
- *Most logics affected:* Our bug findings affect most SMT-LIB logics including strings, (non-)linear integer and real arithmetic, bit-vectors, uninterpreted functions, floating points, arrays, sets, sequences, horn, and combinations thereof.
- *Most bugs in default modes:* 489 out of our confirmed 819 bugs are in the default modes of the solvers, *i.e.*, without additionally supplied options.

4.1 Evaluation Setup

Hardware setup and test seeds. We have run OpFuzz on an AMD Ryzen Threadripper 2990WX processor with 32 cores and 32GB RAM on an Ubuntu 18.04 64-bit. As test seeds, we have mainly used the SMT-LIB benchmarks[SMT-LIB 2020]. We chose the SMT-LIB benchmarks as our test seeds since they make the largest collection of SMT formulas in the SMT-LIB 2.6 language. These SMT-LIB benchmarks are also used in the SMTComp, the annual SMT solver competition. Therefore, they are unlikely to trigger bugs in Z3 and CVC4 since they have already been run on them. In addition to the SMT-LIB benchmarks, we used the regression test suites of Z3[Z3 2020] and CVC4[CVC4 2020]. We show the seed formula counts categorized by logic and solving mode in the Appendix A. We treated all seed files equally during fuzzing. The effort spent on testing for a specific logic is therefore proportional to the number of its seed files within the overall seed set. Consequently, logics with a high seed count get tested more frequently as compared to others with a lower seed count. We regularly ran Z3 and CVC4 on all seed files and excluded bug triggering seeds but have very rarely encountered any bug triggering seed formulas.

Tested options and features. We mainly focused our testing efforts on the default modes of the solvers. For CVC4, this includes enabling the options `–produce-models`, `–incremental` and `–strings-exp` as needed to support all test seed formulas. To detect invalid model bugs, we have supplied `–check-models` to CVC4 and `model.validate=true` to Z3. We consider these to be part of the default mode for the two solvers Z3 and CVC4 if apart from these necessary options, no other options or tactics were used. Besides the default modes of Z3 and CVC4, we have considered many frequently used options and solver modes for Z3 and CVC4 of which we only detail a subset here. For Z3, we have stress-tested several tactics and several arithmetic solvers including `smt.arith_solver=x` with $x \in \{1, \cdots, 6\}$. We have also tested, among others, the string solver z3str3 by supplying `smt.string_solver=z3str3`. In CVC4 we have tested, among many other options, syntax-guided synthesis procedure [Reynolds et al. 2015] by specifying `–sygus-inference` and higher-order reasoning for uninterpreted functions by specifying `–uf-ho`.

Bug types. We have encountered many different kinds of bugs and issues while testing SMT solvers. We distinguish them by the following categories with two SMT solvers S_1 and S_2.

- *Soundness bug*: Formula φ triggers a soundness bug if solvers S_1 and S_2 both do not crash and give different satisfiabilities for φ.
- *Invalid model bug*: Formula φ triggers an invalid model bug if the model returned by the solver does not satisfy φ.
- *Crash bug*: Formula φ triggers a crash bug if the solver throws out an assertion violation or a segmentation fault while solving φ.

OpFuzz detects soundness bug triggers by comparing the standard outputs of the solvers S_1 and S_2. OpFuzz detects invalid model bug triggers by internal errors when using the SMT solver's model validation configuration. A crash bug trigger is detected whenever a solver returns a non-zero exit and no timeout occurred.

Bug trigger de-duplication. OpFuzz collects bug triggers that may stem from the same underlying bug. Hence, we de-duplicated the bug triggers after each fuzzing run with OpFuzz to avoid duplicate bug reports on the GitHub issue trackers. Crash bugs are either assertion violations or segmentation faults. We de-duplicate assertion violations via the location information (file name and line number) printed on standard output/error. We de-duplicate segmentation faults by comparing their ASAN traces. For soundness and invalid model bugs we used the following procedure. We first categorize the bug triggers by theory. We do this because bug triggers in different theories are likely to be

Status	Z3	CVC4	Total
Reported	811	281	1,092
Confirmed	578	241	819
Fixed	521	164	685
Duplicate	88	18	106
Won't fix	106	16	122

(a)

Type	Z3	CVC4	Total
Crash	316	185	501
Soundness	157	27	184
Invalid model	83	19	102
Others	22	10	32

(b)

#Options	Z3	CVC4	Total
default	388	101	489
1	109	67	176
2	45	31	76
3+	36	42	78

(c)

Fig. 7. a) Status of bugs found in Z3 and CVC4. b) Bug types among the confirmed bugs. c) Number of options supplied to the solvers among the confirmed bugs.

unique bugs. Then, we select one bug trigger per theory at a time for reporting. If the bug was fixed, we checked the remaining bug triggering formulas of the same theory. If either one of them still triggered a bug in the solver, we repeat this process until none of them triggers a bug anymore.

Bug reduction. When a bug trigger is selected in the trigger de-duplication, we reduce the bug-triggering formula to a small enough size for reporting. We use C-Reduce [Regehr et al. 2012], a C code reduction tool which also works for the SMT-LIB language.

4.2 Evaluation Results

Having defined the setup and bug types, we continue with the presentation of the evaluation results. The section is divided into three parts: (1) statistics on the bug findings by OpFuzz to assess its effectiveness, (2) coverage measurements of OpFuzz relative to the seed formulas (3) solver trace comparisons to gain further insights into the technique.

Bug findings. Fig. 7a shows the bug status counts. By "Reported", we refer to the unique bugs after bug trigger de-duplication that we posted on the GitHub issue trackers of the solvers; by "Confirmed", we refer to those posted bugs that were confirmed by the developers as unique bugs; by "Fixed", we refer to those posted bugs that were confirmed by the developers as unique bugs and addressed through at least one bug fixing commit; by "Duplicate", we refer to those bugs posted on GitHub that have been identified by the developers as duplicate to another bug report of ours or to a previously existing bug report; by "Won't fix", we refer to those posted bugs that were rejected by the developers, due to misconfigurations.

We have reported a total of 1,092 bugs on Z3's and CVC4's respective issue trackers. Among these, 819 unique bugs were confirmed and 685 were fixed. Although we devoted equal testing effort to both solvers, we found more than twice as many bugs in Z3 as in CVC4. Previous approaches made similar observations [Winterer et al. 2020]. Fig. 7b shows the bug types. Among the bug types of the confirmed bugs, crash bugs were most frequent (501), followed by soundness bugs (184) and invalid model bugs (102). The type "Others" refers to all other unexpected behaviors in SMT solvers such as rejecting syntax-correct formulas, alarming invalid models when generating a valid model. The large majority (489 out of 819) of bugs found by OpFuzz were found in the default modes of the solvers, *i.e.*, no additional options were supplied, some were found with one or two additional options enabled, and clearly less bugs with more than three options enabled (see Fig. 7c).

We have also examined the distribution of logics among the confirmed bugs of Z3 and CVC4 (see Fig. 8a and 8b). We observe that most soundness bugs in Z3 are in the string logics QF_S (42), QF_SLIA (14) and nonlinear logics NRA (20), QF_NRA (12). Notably, there are also a number of soundness bugs in bitvectors QF_BV (7) and linear real and integer arithmetic QF_LRA (4), LRA (4), LIA (4). Similar to Z3, most soundness bugs in CVC4 are also in the string logic QF_S (4) and

Logic	S	I	C	Total
QF_S	42	27	43	112
NRA	20	-	44	64
QF_SLIA	14	7	20	41
QF_NRA	12	9	18	39
QF_LIA	2	6	25	33
QF_NIA	13	3	16	32
UFLIA	4	6	15	25
UF	3	1	18	22
QF_BV	7	3	8	18
LIA	4	-	13	17
Uncategorized	5	-	10	15
QF_UF	1	6	8	15
NIA	5	-	9	14
QF_FP	1	3	9	13
QF_LRA	4	1	7	12
LRA	4	1	7	12
Horn	4	2	5	11
QF_AX	2	1	7	10
ALIA	1	-	8	9
BV	3	-	5	8
QF_UFLIA	-	2	4	6
Set	1	-	5	6
QF_NIRA	1	4	1	6
UFIDL	2	-	2	4
AUFNIRA	-	1	2	3
UFLRA	-	-	2	2
QF_ABVFP	-	-	2	2
QF_UFIDL	-	-	1	1
UFNIA	1	-	-	1
QF_ABV	-	-	1	1
FP	-	-	1	1
NIRA	1	-	-	1
Total	**157**	**83**	**316**	**556**

(a) Z3

Logic	S	I	C	Total
QF_NRA	3	4	20	27
Set	3	-	17	20
UFLIA	-	2	16	18
NRA	1	1	15	17
UF	1	-	16	17
QF_BV	1	-	15	16
Uncategorized	2	2	8	12
QF_LIA	2	-	8	10
QF_S	4	-	6	10
LIA	1	-	8	9
QF_UF	1	-	8	9
LRA	-	-	9	9
BV	-	-	8	8
QF_LRA	1	-	5	6
QF_NIA	2	-	3	5
QF_AX	-	-	5	5
QF_FP	-	-	4	4
QF_AUFBVLIA	-	2	1	3
QF_AUFLIA	-	3	-	3
QF_UFLIA	1	2	-	3
QF_ABVFP	-	-	3	3
NIA	1	-	2	3
QF_ABV	-	1	1	2
QF_SLIA	1	-	1	2
UFNIRA	-	-	1	1
NIRA	-	-	1	1
AUFNIRA	-	1	-	1
UFBV	-	-	1	1
QF_UFNRA	-	1	-	1
QF_UFLRA	-	-	1	1
QF_UFIDL	-	-	1	1
QF_NIRA	1	-	-	1
QF_ALIA	1	-	-	1
UFNRA	-	-	1	1
Total	**27**	**19**	**185**	**231**

(b) CVC4

Fig. 8. Logic distribution of the confirmed bugs: (S) soundness bugs, (I) invalid model bugs, (C) crash bugs. "Uncategorized" refers to bugs that could not be associated with either of SMT-LIB's logics.

	Z3			CVC4		
	lines	*functions*	*branches*	*lines*	*functions*	*branches*
$\texttt{Seeds}_{1000}$	33.2%	36.2%	13.7%	28.5%	47.1%	14.3%
`Type-aware Mutation`	**33.5%**	**36.4%**	**13.8%**	**28.8%**	**47.4%**	**14.4%**

Fig. 9. Line, function and branch coverage achieved by the baseline $\texttt{Seeds}_{1000}$ versus OpFuzz on Z3 and CVC4's respective source codes.

nonlinear arithmetic QF_NRA (3). Moreover, there are three soundness bugs in set logics. However, in difference to Z3 there are almost no soundness bugs in bitvectors and only a single soundness bug in the linear arithmetic QF_UFLIA.

Code coverage of OpFuzz's mutations. Code coverage is a reference for the sufficiency of software testing. This experiment aims to answer whether the mutants generated by OpFuzz can achieve higher coverage than the seed formulas. We randomly sampled 1000 formulas ($\texttt{Seeds}_{1000}$) from all formulas that we used for stress-testing SMT solvers. We instantiated OpFuzz with $n = 300$, run

OpFuzz on the seeds $\texttt{Seeds}_{1000}$ and then measure the cumulative line/function/branch coverage over all formulas and runs.[4] For all coverage measurements, we used Gcov[5] from the GCC suite.

The results show that OpFuzz increases the code coverage upon $\texttt{Seeds}_{1000}$ (Fig. 9). Z3 and CVC4 have over 436K LoC and 238k LoC respectively, so that 0.1% improvement already translate to hundreds of additionally covered lines. However, although noticeable, the coverage increments are not significant ($\leq 0.5\%$). A partial explanation is that decision procedures of Z3 and CVC4 are highly recursive. This leads to many calls of the same functions with different arguments. Hence the difference in line/function/branch coverage achieved by different formulas of the same theory, may not be as significant. This experiment also provides further evidence that standard coverage metrics (*e.g.*, statement and branch coverages), although useful, are insufficient for measuring the thoroughness of testing.

Execution trace comparison. Since code coverage could not thoroughly explain the effectiveness of OpFuzz, we also examine the internals of the solvers by investigating the similarity of their execution traces upon type-aware operator mutations. *What is the relative similarity of the execution traces with respect to the seed?* In the following experiment, we approach this question. In Z3 and CVC4, we can obtain an execution trace by setting the flags `TRACE=True` and `-trace-theory` respectively. Before describing this experiment, we first show the format of Z3's and CVC4's respective traces via an example. Consider formula φ and its type-aware operator mutation φ_{mutant} (see Fig. 10a and 10d). Fig. 10c and 10e shows Z3's and CVC4's traces on solving φ respectively, Fig. 10d and 10f show Z3's and CVC4's traces on solving φ_{mutant} respectively.

Having obtained an intuition of the execution traces, we now get to the actual experiment. Our aim is to measure the relative change in the execution traces of Z3 and CVC4. We therefore perform 40 mutation steps for every formula in $\texttt{Seeds}_{1000}$ and record the execution trace triggered in each step. To quantify the similarity of two traces t_1 and t_2, we compute a metric $sim(t_1, t_2)$ with

$$sim(t_1, t_2) = \frac{2 \cdot |LCS(t_1, t_2)|}{\#lines(t_1) + \#lines(t_2)}$$

where $LCS(t_1, t_2)$ corresponds to the longest common subsequence of t_1 and t_2; $\#lines(t_1)$ and $\#lines(t_2)$ are the number of lines in t_1 and t_2 respectively. As an example, re-consider Fig. 10. The differing lines of φ's trace and φ_{mutant}'s trace are shaded. φ's Z3 trace and φ_{mutant}'s Z3 trace match in 10 out of 11 lines and therefore their similarity score is $\frac{10}{11}$. For the trace pair of CVC4, the number of longest common subsquence is of length 3 and hence the similarity of Z3's trace is $\frac{1}{2}$. To compute the longest common subsequence, we used the `difflib`[6] package from python's standard library. Note that type-aware operator mutation may rename the AST node identifiers of Z3's trace. Here, we under-approximate the similarity of Z3 traces by considering the identifier renaming as the change of the trace.

In our experiment, we fix the trace of the original formula to be t_1, and t_2 corresponds to the trace triggered of the mutant. Fig. 11 shows the similarity of the corresponding mutation step averaged over all formulas in $\texttt{Seeds}_{1000}$. The results of Z3 and CVC4 consistently show that along with a gradual mutation step increase, the similarity between the traces triggered by the mutant and the original formula gradually decreases. The result indicates that OpFuzz can generate diverse test cases that trigger different execution traces via type-aware operator mutation.

Takeaways. We designed three quantitative evaluations to measure and gain an intuition about the effectiveness of OpFuzz. First, we observe that OpFuzz can find a significant number of bugs in

[4]This makes a total of 300k runs.
[5]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[6]https://docs.python.org/3/library/difflib.html

```
;phi
(declare-fun a () Real)
(declare-fun b () Real)
(assert ( < a 0))
(assert (< b 0))
(check-sat)

```

(a)

```
;phi_mutant
(declare-fun a () Real)
(declare-fun b () Real)
(assert ( > a 0))
(assert (< b 0))
(check-sat)

```

(b)

```
[mk-app] #23 a
[mk-app] #24 Int
[attach-meaning] #24 arith 0
[mk-app] #25 to_real #24
[mk-app] #26 < #23 #25
[mk-app] #27 Real
[attach-meaning] #27 arith 0
[inst-discovered] theory-solving 0
arith# ; #25
[mk-app] #28 = #25 #27
[instance] 0 #28
[attach-enode] #28 0

```

(c)

```
[mk-app] #23 a
[mk-app] #24 Int
[attach-meaning] #24 arith 0
[mk-app] #25 to_real #24
[mk-app] #26 > #23 #25
[mk-app] #27 Real
[attach-meaning] #27 arith 0
[inst-discovered] theory-solving 0
arith# ; #25
[mk-app] #28 = #25 #27
[instance] 0 #28
[attach-enode] #28 0

```

(d)

```
TheoryEngine::assertFact
  (not (>= b 0.0)) (0 left)
Theory<THEORY_ARITH>::assertFact[1]
  ((not (>= a 0.0)), false)
TheoryEngine::assertFact
  ((not (>= b 0.0)))
Theory<THEORY_ARITH>::assertFact[1]
  ((not (>= b 0.0)), false)
Theory::get() =>
  (not (>= a 0.0))(1 left)
Theory::get() =>
  (not (>= b 0.0)) (0 left)

```

(e)

```
TheoryEngine::assertFact
  ((not (>= (* (- 1.0) a) 0.0)))
Theory<THEORY_ARITH>::assertFact[1]
  ((not (>= (* (- 1.0) a) 0.0)), false)
TheoryEngine::assertFact
  ((not (>= b 0.0)))
Theory<THEORY_ARITH>::assertFact[1]
  ((not (>= b 0.0)), false)
Theory::get() =>
  (not (>= (* (- 1.0) a) 0.0)) (1 left)
Theory::get() =>
  (not (>= b 0.0)) (0 left)

```

(f)

Fig. 10. Left column: (a) seed formula φ (b) Z3 trace snippet of φ and (c) CVC4 trace snippet of φ. Right column: (d) type-aware operator mutant φ_{mutant} (e) Z3 trace snippet of φ_{mutant} (f) CVC4 trace snippet of φ_{mutant} of φ. Differences in the execution traces snippets are shaded.

various logics, solver configurations, most of which are in default mode. Second, to understand why OpFuzz can find so many bugs, we designed a coverage evaluation. The evaluation result shows that OpFuzz can increase coverage, but the increment is minor. As the coverage evaluation did not answer why OpFuzz is effective, we further designed the third evaluation investigating the similarity of execution traces. The trace evaluation shows that OpFuzz can gradually change the execution traces of the solvers, which partially explains the effectiveness of OpFuzz.

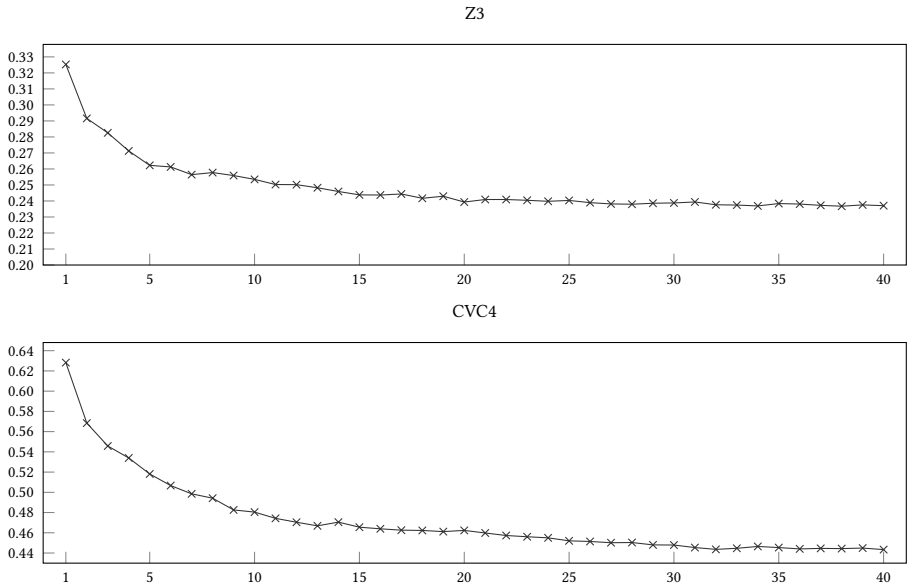

Fig. 11. Average similarity of consecutively generated mutants (y-axis) per mutation step (x-axis).

5 IN-DEPTH BUG ANALYSIS

This section presents an in-depth study on OpFuzz's bug findings in which we (1) quantify the fixing efforts for Z3's and CVC4's developers (2) identify weak components in Z3 and CVC4 and (3) examine the file sizes of bug triggering SMT formulas. We summarize the insights gained and then present selected bug samples, examine their root causes, and the developer's fixes.

5.1 Quantitative Analysis

We collected all GitHub bug reports that we have filed in our extensive evaluation of OpFuzz. This data serves as the basis for our analysis. We guide our analysis with three research questions.

RQ1: How much effort had the developers taken with fixing the bugs found by OpFuzz? To approach this question, we consider two metrics: the files affected by a bug fix and the number of lines of code (LoC) for fixing. If a bug causes many lines of code and/or file changes, this may indicate a high fixing effort necessary by the developers. To examine the LoC and file changes for the bugs found by OpFuzz, we collected 377 bug fixing commits in Z3 and 101 bug fixing commits in CVC4. We solely considered commits that could be one-to-one matched to their bug reports *i.e.*, the commit log mentions the issue id of our bug reports and no other issue ids. Fig. 12 shows the distributions of file changes for bug-fixing commits in Z3 (left) and CVC4 (right). We observe that, in both solvers, most bug-fixing commits change less than five files, and the single file fixes are the majority. However, a few commits affected many files. We have manually examined the right tail of the distribution. We specifically present the top-2 file changing commits in both Z3 and CVC4 individually to demonstrate exemplary reasons for major changes in Z3 and CVC4. We begin with Z3. The highest-ranked bug-fixing commit in Z3 triggered 65 changes. The main part of this fix was in `"smt/theory_bv.cpp"` which is the implementation of bit-vector logic and also serves as the low-level implementation for floating-point logic. The developers' fix resulted in many function name updates and added checkpoints and additional 64 file changes. Another bug-fixing commit in Z3

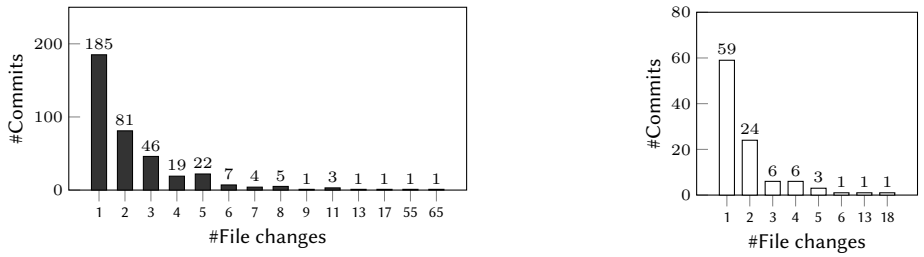

Fig. 12. Distributions of the file changes for a single bug-fixing commit in Z3 (left) and CVC4 (right).

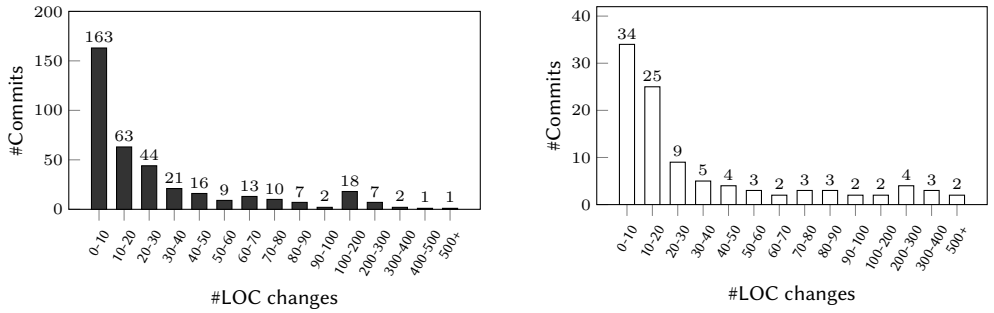

Fig. 13. Distributions of the LoC changed in one commit in Z3 (left) and CVC4 (right).

that affected 55 files is a crash. It was caused by an issue in Z3's abstract syntax tree. The core issue addressed by this fix was in `"ast/rewriter/rewriter_def.h"` and `"ast/rewriter/th_rewriter.cpp"`. Reorganizations of the assertion checks triggered additional 54 file changes. In CVC4, the top-2 fixes with most file changes (18 and 13) are both caused by the crash bugs affecting string operators. The first bug is due to the unsound variable elimination that triggered the assertion violation. The fix refactored the variable elimination with 295 LoC changed. For fixing the second bug, the developers added support for the regex operators `re.loop` and `re.^` that were recently added to the theory of strings. As an intermediate conclusion we observe: Although a relatively high numbers of file changes indicate extensive revisions in the SMT solvers Z3 and CVC4, their root cause are often rather simple fixes such as updating function names, adding assertions, *etc.* We therefore also investigate the LoC changes for each bug fixing commit. Many simple fixes, on the other hand, exhibit subtle missed corner cases.

Fig. 13 presents the distributions of the LoC changes for each bug-fixing commit. For both Z3 and CVC4, we observe that most commits have less than 100 LoC changes and many bugs fixes only involve a 0-10 LoC change. We have manually inspected all 0-10 LoC fixes and observed the majority of them are subtle corner cases. Again we examine top-2 commits for each solver. In Z3 these have 572 and 332 LoC changes respectively. The 572 LoC change commit is a fix for a soundness bug in string logic. It leads to an extensive change in the rewriter of the sequential solver. The 481 LoC changes commit is a fix for a soundness bug in non-linear arithmetic logic. The fix was systematically revamping the decoupling of monomials in non-linear arithmetic logic. For CVC4, the top-2 commits have 1162 and 588 LoC changes respectively. The 1162 LoC changes in CVC4 commit fixes a crash bug by systematically removing the instantiation propagator infrastructure of CVC4. The developer commented that they will redesign this infrastructure in the future. The bugfix with 588 LoC changes is fixing a soundness bug which is labeled as "major" in CVC4's issue tracker. The bug is due to a buggy ad-hoc rewriter that was incorporated into CVC4's extended

File	#Commits
smt/theory_seq.cpp	33
smt/smt_context.cpp	30
smt/theory_lra.cpp	25
qe/qsat.cpp	16
ast/ast.cpp	16
smt/theory_arith_nl.h	15
ast/rewriter/seq_rewriter.cpp	14
ast/rewriter/rewriter_def.h	12
tactic/arith/purify_arith_tactic.cpp	11
smt/theory_seq.h	11

(a)

Filename	#LoC changes
smt/theory_seq.cpp	1082
ast/rewriter/seq_rewriter.cpp	837
smt/theory_arith_nl.h	637
smt/theory_lra.cpp	434
tactic/ufbv/ufbv_rewriter.cpp	375
math/lp/emonics.cpp	333
smt/smt_context.cpp	265
smt/theory_recfun.cpp	247
tactic/core/dom_simplify_tactic.cpp	229
tactic/arith/purify_arith_tactic.cpp	224

(b)

File	#Commits
theory/arith/nonlinear_extension.cpp	7
theory/strings/theory_strings.cpp	6
preprocessing/passes/unconstrained_simplifier.cpp	5
theory/arith/nl_model.cpp	5
smt/smt_engine.cpp	5
theory/quantifiers/extended_rewriter.cpp	4
theory/quantifiers/quantifiers_rewriter.cpp	4
theory/quantifiers_engine.cpp	3
theory/quantifiers/instantiate.cpp	3
theory/arith/nonlinear_extension.h	3

(c)

Filename	#LoC changes
theory/quantifiers/inst_propagator.cpp	864
theory/quantifiers/quantifiers_rewriter.cpp	611
theory/arith/nonlinear_extension.cpp	519
theory/strings/regexp_operation.cpp	292
theory/quantifiers/local_theory_ext.cpp	270
theory/strings/theory_strings.cpp	250
theory/arith/nonlinear_extension.h	212
preprocessing/passes/int_to_bv.cpp	201
theory/quantifiers/inst_propagator.h	194
smt/smt_engine.cpp	130

(d)

Fig. 14. Top-10 (a) files affected by bug fixing commits in Z3. (b) LoC changes per file in Z3 (c) files affected by bug fixing commits in CVC4. (d) LoC changes per file in CVC4.

quantifier rewriting module. The fix deleted the previous buggy rewriting steps and re-implemented an alternative rewriter. Compared to the analysis of file changes, commits with high LoC have a stronger correlation with interesting and systematic fixes in the SMT solvers. On average, the bugs found by OpFuzz lead to 34 and 63 LOC changes for each commit in Z3 and CVC4 respectively.

RQ2: Which parts/files of Z3's and CVC4's codebases are most affected by the fixes? In this research question, we investigate the influence of OpFuzz's bug findings on the respective codebases of Z3 and CVC4. For this purpose, we use two metrics. First, the number of bug-fixing commits that changed a specific file f in either Z3's or CVC4's codebase, *i.e.*, in how many bug-fixing commits file g was included. The second metric is the cumulative number of LoC changes for a file f caused by fixes in either Z3's or CVC4's codebase. For each file f we add up additions and deletions based on GitHub's changeset.

In general, there are 103 files in CVC4 and 348 files in Z3 are affected by the fixes of our bugs. Fig. 14 shows a top-10 ranking of files in Z3's (top row) and CVC4's codebase (bottom row) with respect to the two metrics. We observe that in both Fig. 14a and Fig. 14b, most files belong to the `"smt"` directory which contains the core implementations of Z3. Strikingly, the files `"smt/theory_seq.cpp"` (Z3's sequence and string solvers), `"smt/theory_arith_nl.h"` (Z3's non-linear arithmetic solver) and `"smt/theory_lra.cpp"` (Z3's linear arithmetic solver) are ranked in the top-6 in both #Commits and #LoC changes rankings. This suggests that many of OpFuzz bug findings lead to the fixes in the core components of Z3. Besides files from the `"smt"` directory, the remaining files are mostly part of Z3's `"tactic"` and `"ast"` directories. These contain the

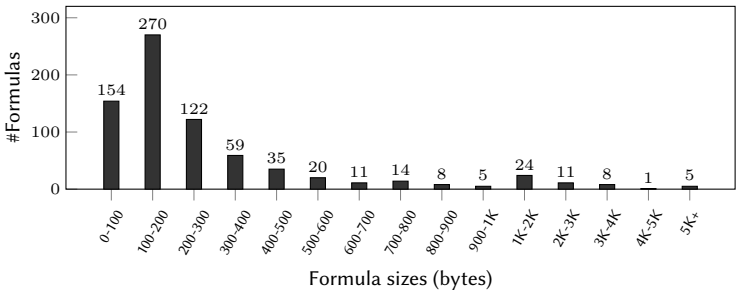

Fig. 15. File-size distribution of reduced bug-triggering formulas.

implementations of solver front-end and Z3's solving tactics. Note, several formulas rewriters related files such as files `"ast/rewriter/seq_rewriter.cpp"`, `"ast/rewriter/rewriter_def.h"` and `"tactic/ufbv/ufbv_rewriter.cpp"` are also highly ranked in the top-10 files affected by the fixes.

We now turn our attention to CVC4. Consider the bottom row of Fig. 14 that presents file and LoC rankings in CVC4. The files that are related to quantifiers (under the path `"theory/quantifiers"`) are the majority in both rankings. Besides, the files `"nonlinear_extension.cpp"`, `"theory_strings.cpp"` and `"quantifiers_rewriter.cpp"` are listed in both rankings. The file `"nonlinear_extension.cpp"` was the implementation of non-linear arithmetic solver, and a recent pull request moved the core of the non-linear arithmetic solver elsewhere. The file `"quantifiers_rewriter.cpp"` contains the implementations of quantifier rewriters that caused soundness bugs, as RQ1 revealed. The file `"theory_strings.cpp"` contains the decision procedures for string logic in CVC4. Moreover the model generator of non-linear arithmetic (`"nl_model.cpp"`) and the pre-processor (`"int_to_bv.cpp"`, `"unconstrained_simplifier.cpp"`) are also heavily influenced by bug fixes.

RQ3: What is the file-size distribution of the bug-triggering formulas? In this research question, we investigate the file-size distribution of reduced bug-triggering formulas. We collected the bug-triggering formulas from all confirmed and fixed Z3 and CVC4 bug reports we filed. Fig. 15 presents the distribution of bug-triggering formulas collectively for Z3 and CVC4. According to Fig. 15, most formulas have less than 600 bytes, while the range of 100-200 bytes has the highest formula count. Among all bugs-triggering formula we reported, there are three formulas to have more than 10,000 bytes *i.e.*, 23,770 bytes, 19,473 bytes, and 10,562 bytes. All of these are in bit-vector logic. The formula with 23,770 bytes and 10 562 bytes triggered an invalid model bug and a crash bug respectively in Z3, and both of them take the developers a half month to fix. The formula with 19 473 bytes triggered a crash bug in CVC4.

The top-3 smallest bug-triggering formulas have 21, 30, and 34 bytes respectively. The 21-byte formula is an invalid formula that triggers a crash bug in Z3. The 30 bytes and 34 bytes formulas are a crash-triggering formula for Z3 and CVC4 respectively, both point to the corner cases. These three bugs were all fixed promptly, i.e., in less than one day, which is significantly faster than the bugs triggered by the top-3 largest formulas. In general, the average size of the bug-triggering formulas reported by us is 426 bytes, which is usually sufficiently small for the developers.

5.2 Insights

Insight 1: OpFuzz's Bugs Are of High Quality. RQ1 and RQ2 have shown that OpFuzz's bug findings have not only led to non-trivial file and LoC changes in both CVC4 and Z3, but also motivated the developers to reorganize and redesign some parts of the solvers. Systematic infrastructure changes such as the decoupling of the monomial instantiation propagator show this. Furthermore, OpFuzz's bugs affected core implementations of the SMT solvers Z3 and CVC4. As

RQ2 presented, the `"smt"` directory in Z3 and `"theory"` directory in CVC4, solvers are among the most affected. Besides, the bugs also affected various pre-processors and rewriters components. Third, the bug-triggering formulas that OpFuzz could be reduced to reasonable sizes (cf. RQ3).

Insight 2: weak components in Z3 and CVC4. From the rankings in RQ2, we identify several "weak" components in Z3 and CVC4. First, in both Z3 and CVC4, source files for the non-linear arithmetic solvers rank high. This indicates: decision procedures for non-linear arithmetic are among the weak components in SMT solvers. Apart from these, rewriters are weak components as well. Z3's `"rewriter_def.h"`, `"ufbv_rewriter.cpp"` and `"seq_rewriter.cpp"` are among the top-10 in LoC changes. In CVC4, the quantifier rewriter `"quantifiers_rewriter.cpp"` is ranked high (5th and 2nd in Fig. 14c and Fig. 14d respectively). In Z3, we identified the tactics to be a weak component. Among the filed bug reports, there are 126 including reports related to tactics. In Fig. 14, these are `"purify_arith_tactic.cpp"` and `"dom_simplify_tactic.cpp"` which are ranked 9th or 10th in both Fig. 14a and Fig. 14b.

Insight 3: bugs found by OpFuzz can usually be reduced to small-sized formulas but bug reduction can be challenging. As we have observed (c.f. Fig. 15), 90% of all bugs found by OpFuzz are triggered by formulas of less than 600 bytes. Small-sized formulas facilitate the bug fixing efforts significantly. As we observed in RQ3, the top-3 largest formulas took the developers around half a month while the top-3 smallest formulas have been fixed very fast, usually within just a few hours. However, reducing SMT formulas to such small sizes can be challenging. ddSMT [Niemetz and Biere 2013] is the only existing specialized SMT formula reducer for that purpose which does however not fully support the SMT-LIB 2.6 standard and formulas in string logic. We therefore preferred C-Reduce, a C code reducer to reduce SMT formulas. While creduce worked well in practice, bug reduction is often challenging especially if the time for solving the formula is high.

5.3 Assorted Bug Samples

This subsection details multiple bug samples from our extensive bug hunting campaign of the SMT solvers Z3 and CVC4 and inspects the root causes. The bugs shown are reduced by C-Reduce, since the unreduced formulas are too large to be presented.

Fig. 16a. shows a soundness bug in Z3's bit-vector logic. The formula is clearly unsatisfiable as the nested `bvxnor` expression equals the unnested `bvxnor` expression. However, Z3 reports `unsat` on it, which is incorrect. The root cause for this bug is an incorrect handling of the ternary `bvxnor` in Z3's bitvector rewriter `"bv_rewriter.cpp"`. The `bvxnor` was implemented as the negation of the `bvxor` operator. This is correct in the binary case, however incorrect for the n-ary case. To see this consider, *e.g.*, `(bvxnor a b c) = (bvxnor (bvxnor a b) c) = true ≠ (not (bvxor a b c)) = false` for `a = b = c = true`. In the fix, Z3's main developer recursively reduces n-ary case `bvxnor` expression to the binary case. The fix lead to a 17 LoC change in `ast/rewriter/bv_rewriter.cpp`.

Fig. 16b. shows a soundness bug in the implementation of the symbolic square root in CVC4. The formula can be satisfied by assigning an arbitrary negative real to variable `x`. CVC4 incorrectly reported `unsat` on this formula. The root cause for this bug is an inadmissible reduction of the square root expression `(sqrt x)` to "choice real `y` s.t. $x = y \cdot y$". For negative `x`, there is no `y` to satisfy the equation. However, square roots of negative numbers are permitted by the SMT-LIB standard. CVC4's developers fixed this bug by interpreting square roots of negative numbers as an undefined value that can be chosen arbitrarily. For the formula in Fig. 16b, the term `(/ (sqrt x) (sqrt x))` can be arbitrarily chosen, as the second assert demands `x` to be negative. Therefore, the formula in Fig. 16b is satisfiable. The bug-fixing pull request was labeled as "major" which reveals that this issue was of high importance to the CVC4 developers. The fix lead to a 126 LoC change on 5 files.

```
(declare-const a (_ BitVec 8))
(declare-const b (_ BitVec 8))
(declare-const c (_ BitVec 8))
(assert (=  (bvxnor a b c)
    (bvxnor (bvxnor a b) c)))
(check-sat)
```

(a) Soundness bug in Z3 caused by a logic in the handling of the ternary bvxnor operator.
https://github.com/Z3Prover/z3/issues/2832

```
(set-logic ALL)
(declare-fun x () Real)
(assert (< x 0))
(assert (not (=
    (/ (sqrt x) (sqrt x)) x)))
(check-sat)
```

(b) Soundness bug in CVC4 caused by an inadmissble reduction of the square root operator.
https://github.com/CVC4/CVC4/issues/3475

```
(declare-fun a () Int)
(declare-fun b (Int) Bool)
(assert (b 0)) (push)
(assert (distinct true
    (= a 0) (not (b 0))))
(check-sat)
```

(c) Soundness bug in Z3 in the boolean rewriter handling the distinct operator.
https://github.com/Z3Prover/z3/issues/2830

```
(set-logic QF_AUFBVLIA)
(declare-fun a () Int)
(declare-fun b (Int) Int)
(assert (distinct (b a)
    (b (b a))))
(check-sat)
```

(d) Soundness bug in CVC4 due to a variable reuse in a simplification.
https://github.com/CVC4/CVC4/issues/4469

```
(declare-fun a () String)
(declare-fun b () Int)
(assert (> b 0))
(assert (= (int.to.str b)
    (str.++ "0" a)))
(check-sat)
```

(e) Soundness bug in Z3 due to a missing axiom in the integer to string conversion function.
https://github.com/Z3Prover/z3/issues/2721

```
(declare-fun x () String)
(declare-fun y () String)
(assert (= (str.indexof x y 1)
    (str.len x)))
(assert (str.contains x y))
(check-sat)
```

(f) Soundness bug in CVC4 due to an invalid indexof range lemma.
https://github.com/CVC4/CVC4/issues/3497

```
(declare-fun a () Real)
(assert (forall ((b Real))
    (= (= a b) (= b 0))))
(check-sat-using qe)
```

(g) Longstanding soundness bug in Z3's qe tactic (since version 4.8.5).
https://github.com/Z3Prover/z3/issues/4175

```
(declare-fun a () Real)
(assert (= (* 4 a a) 9))
(check-sat)
(get-model)
```

(h) Invalid model bug in CVC4 caused by an incorrect implementation of the square root.
https://github.com/CVC4/CVC4/issues/3719

```
(declare-fun a () Int)
(declare-fun b () Real)
(declare-fun c () Real)
(assert (> a 0))
(assert (= (* (/ b b) c) 2.0))
(check-sat)
(check-sat)
(get-model)
```

(i) Invalid model bug in Z3. For the second check-sat query, Z3 returns an invalid model.
https://github.com/Z3Prover/z3/issues/3118

```
(declare-fun d () Int)
(declare-fun b () (Set Int))
(declare-fun c () (Set Int))
(declare-fun e () (Set Int))
(assert (subset b e))
(assert (= (card b) d))
(assert (= (card c) 0 (mod 0 d)))
(assert (> (card (setminus e
    (intersection (intersection e b)
    (setminus e c)))) 0))
(check-sat)
```

(j) Soundness bug in CVC4's set logic caused by an incorrectly implemented cardinality rule.
https://github.com/CVC4/CVC4/issues/4391

Fig. 16. Selected bug samples in Z3 and CVC4.

Fig. 16c. is a soundness bugs in Z3. Although the second assert is unsatisfiable (as `true` cannot be distinct with `(not (b 0))`, Z3 reported `sat` on this formula. The bug is caused by a logic error in a loop condition of a rewriting rule for the `distinct` operator. An incorrect index condition accidentally skips the last argument in an n-ary distinct. The `push` command is necessary for triggering the bug, as it actives the rewriter for `distinct`. The developer has fixed this bug by correcting the index condition. Hence, his fix consisted of only two character deletes in `ast/rewriter/bool_rewriter.cpp`.

Fig. 16d. shows a soundness bug in CVC4's logic of uninterpreted functions In default mode, CVC4 incorrectly reports `unsat` on this satisfiable formula. If we disable unconstrained simplification (`–no-unconstrained-simp`), CVC4 correctly reports `unsat`. The bug is caused by an unsound variable reuse. Our bug report got a "major" label by CVC4's developers and was promptly fixed. The core fix consisted of three LoC deletions in file the unconstrained simplifier implementation "`preprocessing/passes/unconstrained_simplifier.cpp`".

Fig. 16e. depicts a soundness bug in Z3's QF_SLIA logic. The formulas is unsatisfiable, since if assertion `b > 0` holds, there does not exist an `a` starting with "`0`". However, Z3 reports `sat` on this formula. The developers fixed this issue by adding an axiom to the `smt/theory_seq.cpp` adding two additional LoC to this file.

Fig. 16f. shows a soundness in CVC4's string logic. The intuition behind this formula is the following. The index of string `y` in `x` after position 1 should be equal to the length of string `x`. Furthermore `x` should contain `y`. The formula can be satisfied by setting `y` to the empty string and `x` to a string of length 1. However, CVC4 incorrectly reports `unsat`. The root cause was a logic error in `theory/strings/theory_strings.cpp` The developer's fix changed three characters in `theory/strings/theory_strings.cpp`. The fix is labelled as "major".

Fig. 16g. presents a long-standing soundness bug in Z3's `qe` tactic. It affects z3 release from version `4.8.5` to `4.8.7`. The `qe` tactic is an equisatisfiable transform for eliminating quantifiers. Hence, the satisfiability should not be changed by using the `qe` tactic. The formula is satisfiable by assigning `a` to `0`, while Z3's `qe` tactic reports `unsat`. The bug has been confirmed by Z3's developers but has not been fixed yet.

Fig. 16h. shows an invalid model bug in CVC4. CVC4 correctly reports `sat` but generates the model $\{a \mapsto \frac{-9}{2}\}$ which does not satisfy the formula. The bug is caused in CVC4's implementation of the square root. A logic error assigns the result of the square root to be the square root's argument. The fix is labeled as "major" by the developers, and promptly fixed only with a two LoC change in file `theory/arith/nl_model.cpp`.

Fig 16i. shows an invalid model bug in Z3. The `(check-sat)` command appears twice in the formula. This means that Z3 is queried twice for solving. Z3 reports `unknown` for the first query and `sat` for the second. In the second query, Z3 gives the following invalid model $\{a \mapsto 0, b \mapsto 0.0, c \mapsto 16.0, \frac{0}{0} \mapsto \frac{1}{8}\}$ violating `(> a 0)`. The developers fixed this bug through three LoC changes in file `solver/tactic2solver.cpp`.

Fig. 16j. presents a soundness bug in CVC4's set logic. CVC4 returns `unsat` on this satisfiable formula. The root cause is an incorrectly implemented set cardinality rule in the cardinality extension of CVC4. CVC4's set solver uses lemmas to guess the equalities for terms by identifying cycles of terms $e_1 = \cdots = e_2 = \cdots = e_2$. CVC4 has incorrectly assumed that these cycles are loops and in that case would conclude $e_1 = \cdots = e_2$. However, the cycles could have a lasso form which was triggered by our formula. The developers fixed this issue, included the formula to CVC4's regression test suite and marked the pull request to be critical for CVC4's 1.8 release. The fix was labeled as "major" and made 9 LoC changes to `theory/sets/cardinality_extension.cpp`.

6 RELATED WORK

Testing SMT solvers. This paper is not the first work on testing SMT solvers. Roughly ten years ago, the fuzzing tool FuzzSMT [Brummayer and Biere 2009b] has been proposed, which is based on differential testing and targeted bit-vector logic. FuzzSMT uses a grammar for generating the SMT formula. FuzzSMT totally found 16 solver defects in five solvers, however, none in Z3. BtorMBT [Niemetz et al. 2017] is a testing tool for Boolector [Brummayer and Biere 2009a], an SMT solver for the bit-vector theory. BtorMBT tests Boolector by generating random valid API call sequences. However, BtorMBT did not find any bugs in a real setting.

The efforts of the SMT-LIB initiative [Barrett et al. 2020] have resulted in formalized SMT theories and common input/output file formats. In addition, the yearly solver competition SMT-COMP [Competition. 2020] heavily penalized solvers with soundness issues. Consequently, SMT solvers have robustified and finding bugs in SMT solvers became more difficult. Researchers have hence targeted the less mature logics such as the recently proposed theory of strings. Blotsky *et al.* [Blotsky et al. 2018] proposed StringFuzz which focuses on performance issues in string logic. StringFuzz generates test cases in two ways, one is mutating and transforming the benchmarks, another one is randomly generating formulas from a grammar. StringFuzz found 2 performance bugs and 1 implementation bug in z3str3. Bugariu and Müller [Bugariu and Müller 2020] proposed a formula synthesizer for String formulas that are by construction satisfiable or unsatisfiable. They showed that their approach can detect many existing bugs in String solvers and they found 5 new soundness/incorrect model bugs in z3 and z3str3. However, it remained an open question whether automated testing tools could find bugs in theories except the unicode string theory in Z3 and CVC4. Recently, semantic fusion [Winterer et al. 2020] has been proposed which is an approach to stress-test SMT solvers by fusing formula pairs that are by construction either satisfiable or unsatisfiable. Winterer et al.'s tool YinYang found 39 bugs in Z3 and 9 in CVC4. STORM [Numair et al. 2020], another recent mutation-based SMT solver testing approach, found 27 bugs in Z3, however none in CVC4. Another related approach is BanditFuzz [Scott et al. 2020], a reinforcement learning-based fuzzer to detect SMT solver performance issues.

Compared to previous work, type-aware operator mutation is the simplest, while it has also demonstrated to be the most effective technique for testing SMT solvers. Type-aware operator mutations show a promising direction for testing SMT solvers which can benefit the whole community. For example, OpFuzz can be used for the solver developers to stress-test new features conveniently.

Testing program analyzers. SMT solvers are fundamental tools for various program analyzers. Hence, bugs in SMT solvers may affect the reliability of program analyzers. Especially because program analyzers have become mature for practical use in recent years, ensuring the reliability of program analyzers is crucial [Cadar and Donaldson 2016]. There are several works on program analyzer's robustness. For example, Zhang *et al.* [Zhang et al. 2019] tested software model checkers via reachability queries, Bugariu *et al.* [Bugariu et al. 2018] found soundness and precision bugs in numerical abstract domains, Qiu *et al.* [Qiu et al. 2018] and Pauck *et al.* [Pauck et al. 2018] found bugs in the analyzers of Android apps. Type-aware operator mutation contributes to testing program analyzers by finding bugs in SMT solvers. Differential testing based approaches have been effective in finding bugs in program analyzers. For example, Klinger *et al.* [Klinger et al. 2019] and Kapus *et al.* [Klinger et al. 2019] proposed the approaches for testing software model checkers and symbolic executors respectively using differential testing, Wu *et al.* [Wu et al. 2013] tested alias analyzers by cross-checking the dynamic aliasing information. In this work, OpFuzz leverages differential testing to detect soundness bugs in SMT solvers.

Mutation-based testing. Type-aware operator mutation belongs to the family of mutation-based testing techniques. The closest work is skeletal program enumeration (SPE) [Zhang et al. 2017], an approach for validating compilers. Similar to type-aware mutation testing, program skeletons are generated from a set of seed programs. The holes in these skeletons are then systematically filled by exhaustive enumeration. However, unlike type-aware operator mutation, SPE focuses on program variables and not on functions. SPE provides relative guarantees with respect to the input seed programs. Type-aware operator mutation is also related to FuzzChick [Lampropoulos et al. 2019], a coverage-guided fuzzer for Coq programs. FuzzChick generates test cases by semantic mutations at type-level. FuzzChick is aware of parameter types and generates new values for the parameters while preserving type-correctness. Type-aware operator mutation, on the other hand, is focusing on the operators' types and to generate highly diverse SMT formulas.

Type-aware operator mutation also belongs to black-box fuzzing techniques. The black-box fuzzing techniques, such as SYMFUZZ [Cha et al. 2015], leverage user-provided seeds and generate new mutated inputs to uncover security issues. Grey-box fuzzing enhances black-box fuzzing by code coverage guidance and has been successfully applied to software testing recently. AFL [Zalewski 2020] is a popular tool for binary grey-box fuzzing. Follow-up works, such as FairFuzz [Lemieux and Sen 2018] and Steelix [Li et al. 2017], improved the performance of AFL on the binary level. However, binary level fuzzing is ineffective on programs with highly structured inputs (*e.g.*. PDF viewers, programming language engines *etc.*) because of the many syntactically invalid inputs being generated. To generate valid test inputs, grammar-aware grey-box fuzzers were proposed. AFLSmart [Pham et al. 2019], Superion [Wang et al. 2019] and Nautilus [Aschermann et al. 2019] are general grammar-aware grey-box fuzzers targeting programming language engines. They use code coverage to guide the grammar-aware mutations. As a key difference to type-aware operator mutation, they both need to fully parse the program and work on the abstract syntax tree level, which may lead to a higher computational cost during fuzzing. Type-aware operator mutation, on the other hand, works on the token level and without fully parsing the formula.

Besides general black-box and grey-box fuzzing, various domain-specific fuzzing approaches exist, e.g., for testing compilers [Cummins et al. 2018; Le et al. 2014; Lidbury et al. 2015; Yang et al. 2011; Zhang et al. 2017], testing database management systems [Jung et al. 2019; Mishra et al. 2008; Rigger and Su 2020; Seltenreich 2020], and testing OS kernel [Corina et al. 2017; Han and Cha 2017; Schumilo et al. 2017]. Type-aware operator mutation is also a domain-specific fuzzing technique which is unusually effective for testing SMT solvers.

7 CONCLUSION

We introduced type-aware operator mutation, a simple and effective approach for stress-testing SMT solvers. We realized type-aware operator mutation in our testing tool OpFuzz in little more than 200 LoC, supporting only the most basic operators of the SMT-LIB language. Despite this, OpFuzz found 819 confirmed unique bugs (685 fixed) in Z3 and CVC4. These bug findings are highly diverse, ranging over various types, logics and solver configurations in both state-of-the-art SMT solvers. Among these ones, there were many critical bugs. Type-aware operator mutation has found many more bugs than previous approaches by a large margin. Our bug findings show that SMT solvers are not yet reliable enough, even the most popular and stable, such as Z3 and CVC4. Our highly practical tool OpFuzz can help SMT solver developers making their solvers more reliable. For future work, we want to explore the full potential of type-aware operator mutation by invoking more sophisticated type-aware mutations.

ACKNOWLEDGMENTS

We thank the anonymous SPLASH/OOPSLA reviewers for their valuable feedback. Our special thanks go to the Z3 and CVC4 developers, especially Nikolaj Bjørner, Lev Nachmanson, Christoph M. Wintersteiger, Murphy Berzish, Arie Gurfinkel, Andrew Reynolds, Andres Nötzli, Haniel Barbosa, Clark Barrett, *etc.*, for useful information and addressing our bug reports. Chengyu Zhang was partially supported by the China Scholarship Council, NSFC Projects No. 61632005 and No. 61532019.

A TEST SEED FORMULAS

Logic	# non-inc	# inc	# total
QF_SLIA	67,584	-	67,584
QF_FP	40,318	2	40,320
QF_NIA	23,876	10	23,886
AUFLIRA	20,011	-	20,011
QF_ABVFP	18,093	69	18,162
QF_BVFP	17,231	182	17,413
QF_ABV	15,084	1,272	16,356
UFNIA	13,509	-	13,509
QF_NRA	11,489	-	11,489
UFLIA	10,137	-	10,137
QF_UF	7,457	766	8,223
QF_DT	8,000	-	8,000
UF	7,668	-	7,668
QF_LIA	6,947	69	7,016
BV	5,846	18	5,864
UFDT	4,527	-	4,527
NRA	3,813	-	3,813
QF_UFBV	1,234	2,330	3,564
AUFLIA	3,276	-	3,276
FP	2,484	-	2,484
LRA	2,419	5	2,424
QF_S	2,319	-	2,319
QF_IDL	2,193	-	2,193
UFLRA	15	1,870	1,885
AUFBVDTLIA	1,708	-	1,708
QF_LRA	1,648	10	1,658
AUFNIRA	1,480	165	1,645
QF_AUFLIA	1,303	72	1,375

Logic	# non-inc	# inc	# total
QF_UFLIA	583	773	1,356
QF_UFLRA	1,284	-	1,284
AUFDTLIA	728	-	728
LIA	607	6	613
QF_AX	551	-	551
QF_UFNIA	478	1	479
QF_UFIDL	428	-	428
UFDTLIA	327	-	327
QF_RDL	255	-	255
BVFP	224	10	234
QF_ALIA	126	44	170
QF_BVFPLRA	168	-	168
UFBV	121	-	121
QF_ANIA	95	5	100
QF_AUFBV	56	31	87
UFIDL	68	-	68
ALIA	42	24	66
QF_FPLRA	57	-	57
QF_UFNRA	37	-	37
ABVFP	30	4	34
NIA	20	-	20
QF_AUFNIA	17	-	17
QF_LIRA	7	-	7
AUFNIA	3	-	3
QF_NIRA	3	-	3
UFDTNIA	1	-	1
cvc4regr	176	1,594	1,770
z3test	479	841	1,320
Total	308,640	10,173	318,813

Fig. 17. Formula counts for the respective benchmarks sets. Colum #non-inc refers to the count of non-incremental SMT-LIB files, colum #inc refers to the count of incremental SMT-LIB files. *z3test* and *cvc4regr* refer to CVC4's and Z3's respective regression test suites.

REFERENCES

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars. In *NDSS*.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV*. 171–177.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2020. *The satisfiability modulo theories library (SMT-LIB).* Retrieved 2020-05-15 from www.SMT-LIB.org

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB standard: Version 2.0. In *SMT*.

Murphy Berzish, Yunhui Zheng, and Vijay Ganesh. 2017. Z3str3: A string solver with theory-aware branching. *FMCAD* (2017), 55–59.

Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A fuzzer for string solvers. In *CAV*. 45–51.

Robert Brummayer and Armin Biere. 2009a. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*. 174–177.

Robert Brummayer and Armin Biere. 2009b. Fuzzing and delta-debugging SMT solvers. In *SMT*. 1–5.
Alexandra Bugariu and Peter Müller. 2020. Automatically testing string solvers. In *ICSE*. 459–1470.
Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically testing implementations of numerical abstract domains. In *ASE*. 768–778.
Cristian Cadar and Alastair Donaldson. 2016. Analysing the program analyser. In *ICSE*. 765–768.
Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*. 209–224.
Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *SP*. 725–741.
Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT solver. In *TACAS*. 93–107.
The International SMT Competition. 2020. *SMT-COMP*. Retrieved 2020-05-15 from https://smt-comp.github.io/2019/index.html
Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *CCS*. 2123–2138.
Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *ISSTA*. 95–105.
CVC4. 2020. *CVC4 Regression Test Suite*. Retrieved 2020-05-15 from https://github.com/CVC4/CVC4/tree/master/test/regress
Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*. 337–340.
Rob DeLine and Rustan Leino. 2005. *BoogiePL: A typed procedural language for checking object-oriented programs*. Technical Report.
David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A theorem prover for program checking. *JACM* (2005), 365–473.
Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *PLDI*. 213–223.
HyungSeok Han and Sang Kil Cha. 2017. Imf: Inferred model-based fuzzer. In *CCS*. 2345–2358.
Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic detection and diagnosis of performance regressions in database systems. In *VLDB*. 57–70.
Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *ISSTA*. 239–250.
Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. In *OOPSLA*. 181:1–181:29.
Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. 216–226.
Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *ASE*.
Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state based binary fuzzing. In *ESEC/FSE*.
Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. In *PLDI*. 65–76.
Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating targeted queries for database testing. In *SIGMOD*. 499–510.
Aina Niemetz and Armin Biere. 2013. ddSMT: A delta debugger for the SMT-LIB v2 format. In *SMT*. 36–45.
Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Model-based API testing for SMT solvers. In *SMT*. 10.
Mansur Numair, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. *arXiv e-prints* (April 2020), arXiv:2004.05934.
Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android taint analysis tools keep their promises?. In *ESEC/FSE*. 331–341.
Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *TSE* (2019).
Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *ISSTA*. 176–186.
John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI*. 335–346.
Andrew Reynolds, Morgan Deters, Viktor Kuncak, Clark W. Barrett, and Cesare Tinelli. 2015. On counterexample guided quantifier instantiation for synthesis in CVC4. In *CAV*.
Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference Engine Construction. In *OOPSLA*.
Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security*. 167–182.

Joseph Scott, Federico Mora, and Vijay Ganesh. 2020. BanditFuzz: Fuzzing SMT solvers with reinforcement learning. In *CAV*.

Andreas Seltenreich. 2020. *SQLSmith*. Retrieved 2020-08-13 from https://github.com/anse1/sqlsmith

SMT-LIB. 2020. *SMT-LIB Benchmarks*. Retrieved 2020-05-15 from http://smtlib.cs.uiowa.edu/benchmarks.shtml

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation. UC Berkeley. https://people.csail.mit.edu/asolar/papers/thesis.pdf

Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*. 530–541.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *ICSE*. 724–735.

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *PLDI*. 718–730.

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

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

Z3. 2020. *Z3 Regression Test Suite*. Retrieved 2020-05-15 from https://github.com/Z3Prover/z3test

Michal Zalewski. 2020. *american fuzzy lop*. Retrieved 2020-08-12 from https://lcamtuf.coredump.cx/afl/

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*. 763–773.

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