Generative Type-Aware Mutation for Testing SMT Solvers

JIWON PARK*, LIX, École Polytechnique, France
DOMINIK WINTERER*, ETH Zurich, Switzerland
CHENGYU ZHANG, East China Normal University, China
ZHENDONG SU, ETH Zurich, Switzerland

We propose Generative Type-Aware Mutation, an effective approach for testing SMT solvers. The key idea is to realize generation through the mutation of expressions rooted with parametric operators from the SMT-LIB specification. Generative Type-Aware Mutation is a hybrid of mutation-based and grammar-based fuzzing and features an infinite mutation space—overcoming a major limitation of OpFuzz, the state-of-the-art fuzzer for SMT solvers. We have realized Generative Type-Aware Mutation in a practical SMT solver bug hunting tool, TypeFuzz. During our testing period with TypeFuzz, we reported over 237 bugs in the state-of-the-art SMT solvers Z3 and CVC4. Among these, 189 bugs were confirmed and 176 bugs were fixed. Most notably, we found 18 soundness bugs in CVC4's default mode alone. Several of them were two years latent (7/18). CVC4 has been proved to be a very stable SMT solver and has resisted several fuzzing campaigns.

CCS Concepts: • **Software and its engineering → Formal methods**; **Correctness**.

Additional Key Words and Phrases: SMT solvers, Fuzz testing, Generative type-aware mutation

ACM Reference Format:
Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 152 (October 2021), 20 pages. https://doi.org/10.1145/3485529

1 INTRODUCTION

Satisfiability modulo theory (SMT) solvers lie at the heart of many important tools for programming advances and applications [Cadar et al. 2008; DeLine and Leino 2005; Detlefs et al. 2005; Godefroid et al. 2005; Solar-Lezama 2008; Torlak and Bodik 2014]. Soundness bugs in SMT solvers shatter their user's trust and can have severe consequences in safety-critical and security-critical domains. Hence it is crucial that SMT solvers are sound. Recently, researchers devised several SMT solver fuzzers and a few large-scale fuzzing campaigns on SMT solvers are ongoing. [1] One such approach is OpFuzz [Winterer et al. 2020a] which found several hundreds of bugs in the state-of-the-art SMT solvers Z3 [de Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011]. OpFuzz generates formulas for stress-testing SMT solvers by mutating the operators within seed formulas. However, despite its effectiveness, OpFuzz has several limitations. First, OpFuzz is limited by its finite mutation space: the seed formulas have a fixed set of operators and for each of them there are often only 2-3

*Both authors contributed equally to this work.
[1]For citations and an in-depth discussion, we defer to the related work (Section 6).

Authors' addresses: Jiwon Park, LIX, École Polytechnique, France, jiwon.park@polytechnique.edu; 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.

2475-1421/2021/10-ART152
https://doi.org/10.1145/3485529

```
(declare-fun x () String)
(declare-fun y () String)
(declare-fun z () Int)
(assert (= "B" (str.replace (str.substr "A" 0 z) ""
        (str.replace "B" (str.substr "B" 0 0) (str.substr "A" 0 z)))))
(check-sat)
```

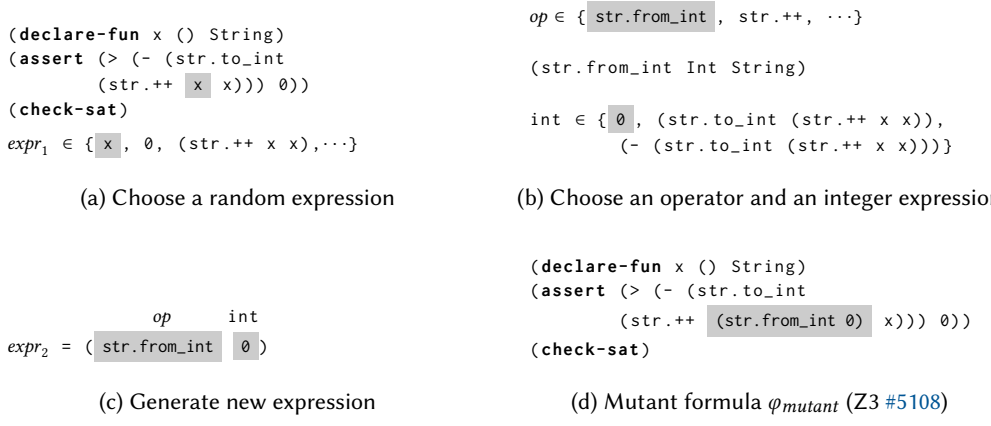
Fig. 1. Almost four-year latent soundness bug in CVC4's string logic.
https://github.com/cvc5/cvc5/issues/5940

type-conforming choices for mutation. Furthermore, as fuzzing campaigns and especially OpFuzz have led to hundreds of bug fixes in the SMT solvers Z3 and CVC4, the solvers have matured. Because of this saturation effect [Amalfitano et al. 2015], SMT solver fuzzers are finding fewer and fewer critical bugs. Yet, important bugs have been missed. Consider the formula in Fig. 1 which manifests a long-latent soundness bug in CVC4. The "declare-fun" statements specify two string variables and one integer variable respectively, the "assert" specifies the constraints, and the "check-sat" queries the solver. The formula is satisfiable which Z3 correctly reports. However, CVC4 returns `unsat` on this formula which indicates a soundness bug in CVC4. This bug is long-latent: it existed in CVC4 since at least CVC4 v1.5—for almost four years. [2] Moreover, since March 2019, several large-scale fuzzing campaigns have targeted string logic (some even exclusively), yet none of the other fuzzers found this bug. It is a refutational soundness bug—the most critical bug category.

Generative type-aware mutation. This paper introduces *Generative Type-Aware Mutation*, a novel, effective approach for testing SMT solvers, capable of finding many longstanding soundness bugs in both Z3 and CVC4. It has found the almost four-year latent soundness bug from Fig. 1. Moreover, with Generative Type-Aware Mutation, we reported 237 bugs in the state-of-the-art SMT solvers Z3 and CVC4, 189 bugs were confirmed and 176 bugs were fixed. Most notably, Generative Type-Aware Mutation found 18 soundness bugs in CVC4's default mode alone. Several of them (7/18) were at least 2 years latent and predated all previous SMT solver fuzzing campaigns. By comparison, prior approaches did not find any bugs in CVC4 [Bugariu and Müller 2020; Numair Mansur et al. 2020], others found similar numbers of soundness bugs during much longer time spans: YinYang [Winterer et al. 2021] found eight in nine months, and OpFuzz [Winterer et al. 2020a] found eleven in a year. All approaches were reportedly using the SMT-LIB seeds and similar resources as TypeFuzz did. TypeFuzz found these bugs despite robustified Z3 and CVC4 thanks to the bug fixes that resulted from prior fuzzing campaigns. The core idea behind *Generative Type-Aware Mutation* is simple: to combine mutational and grammar-based type-aware fuzzing. Given a seed formula φ, we first choose an expression *expr* within φ. Second, we pick an operator *op* of the same type as *expr* and fill *op*'s arguments with expressions from φ. The newly generated expression then replaces *expr* in φ. The resulting formula φ_{mutant} is then used to differential test SMT solvers.

Relationship to type-aware operator mutation and FuzzChick. Generative Type-Aware Mutation generalizes two fuzzing techniques: type-aware operator mutation [Winterer et al. 2020a], and FuzzChick [Lampropoulos et al. 2019]. Type-aware operator mutation mutates the operators within the SMT-LIB formulas making use of operators from the SMT-LIB specification. While unusually effective for testing SMT solvers, the mutation space is finite. FuzzChick has been used for testing Coq programs. Its mutation space is infinite for Coq. However, FuzzChick applied in the SMT setting limits the mutation of operators. Generative Type-Aware Mutation overcomes both limitations by combining mutation and grammar-based fuzzing.

[2] CVC4 1.5 was released on July 10, 2017. The bug was hidden in CVC4 1.6 but present in all other releases.



(a) Choose a random expression

(b) Choose an operator and an integer expression

(c) Generate new expression

(d) Mutant formula φ_{mutant} (Z3 #5108)

Fig. 2. Generative Type-Aware Mutation illustrated.

Main contributions. We make the following contributions:

- We introduce *Generative Type-Aware Mutation*, a novel, effective approach for stress-testing SMT solvers; Generative Type-Aware Mutation is a hybrid of mutation-based and grammar-based fuzzing and has an infinite space, overcoming one of OpFuzz's key limitations.
- Based on Generative Type-Aware Mutation, we have realized TypeFuzz, a highly practical testing tool customizable by a configuration file of SMT-LIB's theory specification language.
- From end of January 2021 to mid September 2021, we reported 237 bugs in the state-of-the-art SMT solvers Z3 and CVC4 with TypeFuzz. Among these, 189 bugs were confirmed and 176 bugs were fixed. Most notably, Generative Type-Aware Mutation found 18 soundness bugs in CVC4's default mode alone. Several of them (7/18) are at least 2 years latent and predated all previous SMT solver fuzzing campaigns.
- We examine Generative Type-Aware Mutation in comparison to the two closely related approaches, type-aware operator mutation and FuzzChick, theoretically and practically through code coverage comparisons.

Organization of the paper. Section 2 describes Generative Type-Aware Mutation using an illustrative example. Section 3 introduces Generative Type-Aware Mutation formally and describes the implementation of TypeFuzz which realizes Generative Type-Aware Mutation. Section 4 presents the evaluation of our approach. In Section 5, we present several bug samples and their root causes. Section 6 surveys related work, and Section 7 concludes the paper.

2 ILLUSTRATIVE EXAMPLE

This section gives a brief introduction to the SMT-LIB language [Barrett et al. 2010] and illustrates Generative Type-Aware Mutation on an example.

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 `"(declare-fun x () Real)"` declares a variable of type `Real` with name `x`. 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 `Bool` 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.

2.1 Generative Type-Aware Mutation in Steps

The key idea of Generative Type-Aware Mutation is mutating expressions in the AST of an SMT-LIB script by newly generated expressions of the same type. Let φ be a seed formula (see Fig. 2).

Step 1 Choose a random expression: We first choose a random expression $expr_1$ from the set of φ's expressions $expr(\varphi)$. Say we have picked the $expr_1$ = `x`. The expression is of type `String` and will serve as the replacee (*i.e.*, the candidate to be replaced) for the newly generated expression.

Step 2 Choose a random operator: Next, we choose a suitable random operator. Such an operator should have the return type `String` and for all of its arguments there should be at least one expression of conforming type in $expr(\varphi)$. Since φ contains terms of type `Bool`, `Int`, and `String`, the operator's arguments should be one of those types. Candidates are the string to integer conversion function `str.from_int`, the string concatenation `str.++`, and all other operators taking `Bool`, `String` as arguments and returning `Bool`. For the complete list of possible candidates that we use, we refer the reader to Section 4. Assume we have chosen `str.from_int`.

Step 3 Generate new expression: Then, we generate an expression $expr_2$ with respect to the signature of the chosen operator. The signature for the operator `str.from_int` is defined as

```
(str.from_int Int String)
```

Hence, we select an `Int` expression from $expr(\varphi)$. For the single parameter of type `Int`, we choose `0`. Then, with the chosen operator and expression, we construct the following new expression:

$$expr_2 = \texttt{(str.from\_int 0)}$$

Step 4 Substitution: Finally, we substitute $expr_1$ by $expr_2$ in φ which results in the formula φ_{mutant}. We feed φ_{mutant} to two or more SMT solvers and compare their results.

The formula φ_{mutant} is a real case. Z3 and CVC4 give different results on φ_{mutant}. CVC4 correctly reported `sat` on it, while Z3 incorrectly reported `unsat`. We have filed this bug on the Z3 issue tracker. The developers promptly fixed this soundness issue in the trunk version of Z3. As we will show (Section 3), Generative Type-Aware Mutation is a powerful generalization of type-aware operator mutation [Winterer et al. 2020a] and FuzzChick [Lampropoulos et al. 2019]. Neither approach could have generated this bug-triggering formula from the seed φ.

3 APPROACH

This section (1) formally introduces generative type-aware mutation, (2) shows the conditions under which generative type-aware mutation produces type-correct formulas, (3) clarifies the relationships to type-aware operator mutation and FuzzChick, and (4) proposes TypeFuzz, a practical fuzzing tool for stress-testing SMT solvers.

Definitions. 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. Formulas are realized by SMT-LIB programs [Barrett et al. 2019]. We use standard notions of typed higher-order logic, such as term, quantifier, function, *etc.*, and write expression for term occurrences. We view formulas as abstract syntax trees of typed expressions. Such an expression *expr* has an associated type $type(expr)$ and the set of all types is $types = \{\texttt{Bool}, \texttt{Int}, \texttt{Real}, \texttt{String}, \texttt{RegLan}, \texttt{A}\}$ where `A` is a generic supertype of all the other types. For an expression *expr* in φ, we define $locals(expr, \varphi)$ to be the set of local

variable occurrences in *expr*. When φ is clear from the context, we simply write $locals(expr)$. Within a formula φ, local variables can be defined by quantifiers, let expressions, *etc.* By $expr(\varphi)$, we denote φ's (enumerated) expression occurrences. We write $\varphi[expr_2/expr_1]$ for the substitution of expression $expr_1$ by expression $expr_2$ in φ. An operator *op* has the attributes $rtype(op)$ and the tuple $arg\_types(op)$ for *op*'s return type and the types of *op*'s arguments respectively. We denote the set of operators by *ops* and write ops_{type} for all operators of return type *type*. The type skeleton $skeleton(\varphi)$ of φ is a tree where each expression *expr* in φ is represented by its type.

3.1 Generative Type-Aware Mutation

We first introduce Generative Type-Aware Mutation as regular tree grammar and then show under which conditions generative type-aware mutants are well-typed.

Regular tree grammar. A *regular tree grammar* $G = (N, \Sigma, S, P)$ consists of a finite set of nonterminals N, a ranked alphabet Σ, a starting nonterminal S in N, and a finite set of productions P. Each symbol in Σ has an associated arity, and the productions in P are of the form $A \rightarrow t$ where $A \in N$ and $t \in T_\Sigma$ with T_Σ being the set of all trees composable from symbols $\Sigma \cup N$. The language $L(G)$ generated by G describes any tree that can be derived from S using the rule set P.

Definition 1 (Generative type-aware mutation). *Let $G_{\text{GTA}} = (N, \Sigma, S, P)$ be a regular tree grammar and φ a formula:*

- $N = types$
- $\Sigma = expr(\varphi)$
- $S = skeleton(\varphi)$
- $P = P_{expr} \cup P_{gen}$ *where*
  - $P_{expr} = \{type \rightarrow expr \mid expr \in expr(\varphi) \wedge type(expr) = type\}$
  - $P_{gen} = \{type \rightarrow (op\ arg\_types(op)) \mid op \in ops_{type}\}$

We say a formula φ_{mutant} is a **generative type-aware mutant** *of φ if φ_{mutant} is in $L(G_{\text{GTA}})$.*

A generative type-aware mutant φ_{mutant} of φ can be conveniently fabricated by replacing an expression $expr_1$ within φ with an expression $expr_2$ which is either another type-conforming expression from $expr(\varphi)$, or rooted with a new operator of type-conforming return type and type-conforming arguments from $expr(\varphi)$. Generative type-aware mutations will by design not lead to type-incorrect replacements, *e.g.*, replacing an integer expression by a string expression. However, as the following example illustrates, they also do not guarantee well-typed mutants. Consider the following formula in SMT-LIB language:

$$\varphi = \texttt{(and (> x 10) (forall ((z Int)) (< z y)))}$$

We choose $expr_1 = \texttt{(> x 10)}$, $expr_2 = \texttt{(< z y)}$ and replace $expr_1$ by $expr_2$ in φ. The resulting formula φ_{mutant} is by definition a generative type-aware mutant of φ:

$$\varphi_{mutant} = \texttt{(and (< z y) (forall ((z Int)) (< z y)))}$$

However, φ_{mutant} is not well-typed since the variable `z` is out of the scope of the quantifier. We address this issue by the following definition.

Definition 2 (Local compatibility). *Let $expr_1$ and $expr_2$ be two expressions of the same type. We say $expr_2$ is* **locally compatible** *with $expr_1$ if $locals(expr_2) \subseteq locals(expr_1)$.*

Checking for local compatibility avoids the above issue: $locals(expr_1) = \emptyset$ and $locals(expr_2) = \{\texttt{z}\}$ and hence $expr_2$ would not be locally compatible with $expr_1$.

Proposition 3.1. *Generative type-aware mutants are well-typed if for every substitution local compatibility is ensured.*

```
(declare-fun x () String)
(declare-fun y () String)
(assert
  (= (str.replace x "B" (str.++ "B" "B"))
    (str.++ y "B")))
(check-sat)
```

(a) Seed formula φ

```
(declare-fun x () String)
(declare-fun y () String)
(assert
  (= ( str.replace
       (str.replace x "B" (str.++ "B" "B"))
       "B" (str.++ y "B"))
    (str.++ y "B")))
(check-sat)
```

(b) Mutant formula φ_{mutant} (CVC4 #5915)

Fig. 3. Generative Type-Aware mutation illustrated.

3.2 Relationship to FuzzChick and Type-aware Operator Mutation

This section clarifies the relationships between generative type-aware mutation and two related techniques that have been used for stress-testing software, FuzzChick and type-aware operator mutation. We adapt them to the formal setting of this paper.

FuzzChick's Mutator [Lampropoulos et al. 2019]: FuzzChick tests Coq programs using grammar-based generators and coverage feedback. Adapted to our setting, this corresponds to $\varphi_{mutant} = \varphi[expr_2/expr_1]$ where $expr_1, expr_2$ are expressions from $expr(\varphi)$.

Type-Aware Operator Mutation [Winterer et al. 2020a]: OpFuzz embodies type-aware operator mutation and is an effective technique for SMT solver testing. The key idea is to mutate operators of the same type. Let φ be an SMT formula and let op_1 of type $type_1$ be one of its operator and op_2 of type $type_2$ be an operator of the SMT-LIB specification. $\varphi_{mutant} = \varphi[op_2/op_1]$ is a *type-aware operator mutant* if $type_2$ is a subtype of $type_1$.

Let formula φ be a formula and $G_{\text{GTA}} = (N, \Sigma, S, P_{gen} \cup P_{expr})$ the regular tree grammar specifying generative type-aware mutations for φ. FuzzChick's mutator can be described by the grammar $G_{\text{FC}} = (N, \Sigma, S, P_{expr})$. Since the grammar G_{FC} is identical to G_{GTA} without generation rules, $L(G_{\text{FC}})$ is a subset of $L(G_{\text{GTA}})$. Every type-aware operator mutation $\varphi_{mutant} = \varphi[op_2/op_1]$ with op_1, op_2 from *ops* can be imitated by G_{GTA} by starting from the $skeleton(\varphi)$ and applying the productions P_{expr} to generate φ except for op_1. Then, we apply the production $op_1.type \rightarrow (op_2\ type_1, \cdots, type_m)$ from P_{gen} and again rules from P_{expr} to recover the former arguments of op_1.

Corollary 3.2. *Generative type-aware mutation generalizes FuzzChick's mutator.*

Corollary 3.3. *Generative type-aware mutation generalizes type-aware operator mutation.*

The following example is a real case that constructively shows the strict dominance of Generative Type-Aware Mutation over the type-aware operator mutation, the state-of-the-art approach.

Example 3.4. Consider formula φ from Fig. 3a. Assume, we picked the expression $expr$ = `(str.replace x "B" (str.++ "B" "B"))` from $expr(\varphi)$. The expression is of type `String` and will serve as the replacee for the newly generated expression. Next, we choose the operator `str.replace` that takes three strings as its arguments and returns a string. Then, we generate an expression:

$$expr_2 = \texttt{(str.replace String String String String)}$$

by substituting the function arguments with type-aware expressions from $expr(\varphi)$ such as

$$expr_2 = \texttt{(str.replace (str.replace x "B" (str.++ "B" "B")) "B" (str.++ y "B"))}$$

Finally, we substitute $expr_1$ by $expr_2$ in φ which results in the formula φ_{mutant} (Fig. 3b). We feed φ_{mutant} to two or more SMT solvers and compare their results. Z3 and CVC4 give different results on φ_{mutant}. Z3 correctly returned `unsat` on it while CVC4 returned `sat` on it. We have reported this bug to the CVC4 issue tracker. The developers promptly fixed this four-year longstanding soundness issue in CVC4. Type-aware operator mutation cannot generate this bug since the number of operators has increased from φ to φ_{mutant}.

Differences in practice. As a key difference to FuzzChick, generative type-aware mutation can leverage all available operators from the SMT-LIB specification for the types in its skeleton. In contrast, FuzzChick is limited to the operators occurring in the seed formula. If the seed formula contains all operators from the SMT-LIB specification for all the types in its skeleton, the two techniques are identical, i.e, the grammars $L(G_{\text{GTA}})$ and $L(G_{\text{FC}})$ would induce the same language. However, in practice, this is rarely the case. In contrast to type-aware operator mutation, generative type-aware mutation can generate expressions rooted by operators from the SMT specification with type-conforming terms from the seed file as its arguments while type-aware operator mutation can only mutate operators. Hence, generative type-aware mutation's additional expressive power translates to practical advantages over both techniques.

3.3 TypeFuzz

Based on Generative Type-Aware Mutation, we have engineered TypeFuzz, a bug hunting tool for SMT solvers. This subsection details the underlying procedures of TypeFuzz.

Main process. Algorithm 1 presents the parameterized pseudocode of TypeFuzz. The main process takes a set of seed formulas, *seeds*, and a set of SMT solvers, *solvers* under test. First, the algorithm reads a configuration file (Line 2). The configuration file (see Fig. 4) contains all signatures of SMT-LIB operators and can be customized by the user. The list *triggers* is used for collecting the bug triggers and is initialized to the empty list (Line 3). The body of the while loop is executed until a termination criterion is met. This could be a timeout or an interruption by the user. We first randomly chose a formula φ from *seeds* (Line 5). Then, we call the function GetTypedExpressions (Line 6) which returns the list *expressions*. The body of the for loop (Line 7) realizes n consecutive generative type-aware mutations. At the end of each iteration (Line 13), we reset φ to the previously mutated formula φ_{mutant} to realize the mutation chain. For parameter n, we have used values in the range of 10 to 100. Smaller n help traverse the seed set faster, larger n lead to deeper mutations. Inside the for-loop body, we first call the function GenerativeTypeAwareMutate which returns a boolean *success* whether the function successfully generated a mutant formula φ_{mutant}. If the mutation attempt was unsuccessful, we continue with the next iteration. Otherwise, we call the function validate with the mutant formula φ_{mutant} and the set of solvers, *solvers*. validate sequentially executes each solver on φ and checks for (1) crashes, *i.e.*, segmentation faults, assertion violations, *etc.* by matching standard output to a known list of errors, (2) soundness issues by comparing the satisfiability results of the solvers, and (3) invalid models where the solver returns an incorrect model on a satisfiable formula. In any of the three cases, the function returns *false* and we add φ_{mutant} to the set of candidate bugs *triggers*.

Generative type-aware mutation. Algorithm 2 presents the implementation of a generative type-aware mutation. The function GenerativeTypeAwareMutate (Line 1) takes a formula and *expressions* as its inputs. We first retrieve the set of unique expressions from the list *expressions*. The list of expressions may contain duplicates since syntactically equivalent expressions can occur multiple times in φ (Line 2). In the for loop (Line 3), we first choose a random expression $expr_1$ from the list of expression *expressions*. Then, we call the function GetReplacee to obtain an expression

Algorithm 1 TypeFuzz's pseudocode

```
1:  procedure TypeFuzz(solvers, seeds)
2:      all_ops ← ReadConfigfile()
3:      triggers ← []
4:      while no termination criterion is met do
5:          φ ← random.choice(seeds)
6:          expressions ← GetTypedExpressions(φ)
7:          for i to n do
8:              φ_mutant, success ← GenerativeTypeAwareMutate(φ, expressions)
9:              if not success then
10:                 continue
11:             if not Validate(φ_mutant, solvers) then
12:                 triggers ← triggers.append(φ_mutant)
13:             φ ← φ_mutant
```

Algorithm 2 Generative Type-Aware Mutation's pseudocode

```
1:  function GenerativeTypeAwareMutate(φ, expressions)
2:      unique_expr ← GetUniqueExpressions(expressions)
3:      for j to len(expressions) do
4:          expr_1 ← random.choice(expressions)
5:          expr_2 ← GetReplacee(expr_1, unique_expr)
6:          if expr_2 ≠ None then
7:              return φ[expr_2/expr_1], true
8:      return None, false

9:  function GetReplacee(expr, unique_expr)
10:     ops ← {op ∈ all_ops | rtype(op) = expr.type}
11:     op ← random.choice(ops)
12:     args ← []
13:     for type in op.arg_types do
14:         choices ← {e ∈ unique_expr |e ≠ expr ∧ e.type = type ∧ local_compatible(e, expr)}
15:         if choices = ∅ then
16:             return None
17:         arg ← random.choice(choices)
18:         args.append(arg)
19:     return make_expr(op, args)
```

for replacing $expr_1$. If the function was unsuccessful, it returns *None*. If successful, we return a formula in which $expr_1$ is replaced by $expr_2$ and *true* (Line 7), indicating that the mutation attempt was successful. Otherwise, if after *len*(*expressions*) tries no replacee has been found, we return *None* and *false* (Line 8) indicating that the mutation attempt was unsuccessful. The GetReplacee function (Line 9) realizes a greedy algorithm for finding a suitable replacee expression for a given expression *expr*. First, we collect a set of operators of conforming return type with *expr* (Line 10) and randomly choose one of them (Line 11). We then iterate through the argument types of the chosen operator (Line 13). For each argument type, we compute the set *choices* (Line 14) representing the type matching expressions e distinct from *expr* that are locally compatible with *expr*. If *choices* is empty, we return *None* (Line 14) to indicate that the mutation attempt was unsuccessful, i.e., there is no expression e of the same type that is locally compatible with *expr* and syntactically different.

Otherwise, we randomly choose an argument from *choices* and add it to the list of arguments for the chosen operator (Line 17 + 18). In Line 19, we then instantiate the chosen operator with the selected arguments and return them.

Computing local compatibility. To realize local compatibility (see Line 14), we use a recursive procedure. For an expression *expr*, we recursively collect the local variables defined by its parent and upward. The reason for collecting local variables from the parent onwards is when the expression declaring the local variable, which itself contains the local variable, is substituted by one of its child expressions with the local variable it declared, the mutant will be faulty as the declaration is lost and the local variable becomes undefined.

Implementation. We have implemented TypeFuzz on top of the SMT solver fuzzer yinyang [Winterer et al. 2021]. For that matter, we implemented the generative type-aware mutation as a mutation strategy (260 LoC) and augment the yinyang framework by a type-checker (790 LoC). TypeFuzz's mutations can be customized with a configuration file. We have used the configuration file from Fig. 4. Its syntax is similar to the meta-language of SMT-LIB theory specifications. [3] We hope this will facilitate SMT developers and practitioners to run customized configurations and have released our tool on GitHub [4]. The tool can be installed via `pip install yinyang`.

4 EMPIRICAL EVALUATION

This section details our extensive evaluation with TypeFuzz demonstrating the practical effectiveness of Generative Type-Aware Mutation for testing SMT solvers. Between end of January 2021 and mid September 2021, we were running TypeFuzz to stress-test the state-of-the-art SMT solvers Z3 [de Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011]. During our testing period, we have filed numerous bugs on the issue trackers of Z3 and CVC4.

Result summary.

- *Many bugs in a short time:* In eight months, we found 237 bugs, 177 in Z3 and 60 in CVC4. Among these, 176 were already fixed by the developers.
- *Many longstanding soundness bugs in CVC4:* We found 18 soundness bugs alone in CVC4's default mode. Many of them (7/18) are at least 2 years latent and pre-date any previous SMT solver fuzzing campaign.

4.1 Evaluation Setup

We have run TypeFuzz on a machine equipped with an AMD Ryzen Threadripper 3990X with 64 cores and 32GB RAM. We occupied half of its cores. Additionally, we ran another machine equipped with an Intel Core i7-8700 CPU with 6 CPU cores of which we used full cores. Both machines were running Ubuntu 18.04 (64-bit).

Test seeds & options. As the test seeds, we used non-incremental formulas from the linear and nonlinear reals and integer arithmetic, their combinations (LIA, LRA, NIA, NRA, QF_LIA, QF_LIRA, QF_LRA, QF_NIA, QF_NIRA, QF_NRA) and the string logics QF_S and QF_SLIA. All seeds were taken from the GitLab repositories provided by the SMT-LIB initiative.[5] Since their creation, the following minor modifications were made on these files: (1) README updates and satisfiability status labels, (2) removal of a few incorrectly assigned instances to QF_LIA, and (3) several updates in the QF_S and QF_SLIA seeds changing string operator labels from "-" to underscore, *etc.* to

[3] http://smtlib.cs.uiowa.edu/theories.shtml
[4] https://github.com/testsmt/yinyang
[5] https://smtlib.cs.uiowa.edu/benchmarks.shtml

```
;;; Functions from the core theory
(not Bool Bool)
(=> Bool Bool Bool :right-assoc)
(and Bool Bool Bool :left-assoc)
(or Bool Bool Bool :left-assoc)
(xor Bool Bool Bool :left-assoc)
(par (A) (= A A Bool :chainable))
(par (A) (distinct A A Bool :pairwise))
(par (A) (ite Bool A A A))

;;; Functions from Ints
(- Int Int)
(- Int Int Int :left-assoc)
(+ Int Int Int :left-assoc)
(* Int Int Int :left-assoc)
(div Int Int Int :left-assoc)
(mod Int Int Int)
(abs Int Int)
(<= Int Int Bool :chainable)
(<  Int Int Bool :chainable)
(>= Int Int Bool :chainable)
(>  Int Int Bool :chainable)

;;; Functions from Reals
(- Real Real)
(- Real Real Real :left-assoc)
(+ Real Real Real :left-assoc)
(* Real Real Real :left-assoc)
(/ Real Real Real :left-assoc)
(<= Real Real Bool :chainable)
(<  Real Real Bool :chainable)
(>= Real Real Bool :chainable)
(>  Real Real Bool :chainable)

;;; Functions from Real\_Ints
(- Int Int Int :left-assoc)
(+ Int Int Int :left-assoc)
(* Int Int Int :left-assoc)
(div Int Int Int :left-assoc)
(mod Int Int Int)
(abs Int Int)
(<= Int Int Bool :chainable)
(<  Int Int Bool :chainable)
(>= Int Int Bool :chainable)
(>  Int Int Bool :chainable)
(- Real Real)
(- Real Real Real :left-assoc)
(+ Real Real Real :left-assoc)
(* Real Real Real :left-assoc)
(/ Real Real Real :left-assoc)
(<= Real Real Bool :chainable)
(<  Real Real Bool :chainable)
(>= Real Real Bool :chainable)
(>  Real Real Bool :chainable)
(to_real Int Real)
(to_int Real Int)
(is_int Real Bool)

;;; Functions from Strings
;
; Core string functions
(str.++ String String String :left-assoc)
(str.len String Int)
(str.< String String Bool :chainable)

; Regular expression functions
(str.to_re String RegLan)
(str.in_re String RegLan Bool)
(re.none RegLan)
(re.all RegLan)
(re.allchar RegLan)
(re.++ RegLan RegLan RegLan :left-assoc)
(re.union RegLan RegLan RegLan :left-assoc)
(re.inter RegLan RegLan RegLan :left-assoc)
(re.* RegLan RegLan)
(re.comp RegLan RegLan)
(re.diff RegLan RegLan RegLan :left-assoc)
(re.+ RegLan RegLan)
(re.opt RegLan RegLan)
(re.range String String RegLan)

; Misc string functions
(str.<= String String Bool :chainable)
(str.at String Int String)
(str.substr String Int Int String)
(str.prefixof String String Bool)
(str.suffixof String String Bool)
(str.contains String String Bool)
(str.indexof String String Int Int)
(str.replace String String String String)
(str.replace_all String String String String)
(str.replace_re String RegLan String String)
(str.replace_re_all String RegLan String String)

; Maps to and from integers
(str.is_digit String Bool)
(str.to_code String Int)
(str.from_code Int String)
(str.to_int String Int)
(str.from_int Int String)
```

Fig. 4. TypeFuzz's configuration file. The syntax is purposefully adapted to the SMT-LIB theory specifications. The configuration is tailored to the theories Core, Reals, Ints, RealInts and Strings.

ensure compliance with the evolving standard. Therefore, we can safely assume that previous approaches [Numair Mansur et al. 2020; Winterer et al. 2020a,b] used the same seeds. The benchmarks range from verification of systems, proofs, synthesized programs to symbolic execution runs and randomly generated formulas. A subset of the formulas is used by the annual SMT solver competition. We mainly focused our testing efforts on the default modes of the solvers. We consider CVC4

to be in default mode, if apart from options to support SMT-LIB seeds such as `--produce-models` and `--strings-exp`, no further options are enabled. For Z3, the option `unicode=true` was necessary during the first month of the testing period to guarantee Z3's compliance with the specification of the string theory. Apart from the default mode, we focused on a few popular pre-processing options and rewriter options. These configurations are interesting since bugs in them are likely to cause undetectable soundness issues. We selected the options as per the developer's priorities. Furthermore, upon request of Z3's main developer, we have tested Z3's new core (`tactic.default_tactic=smt`, `sat.euf=true`), which is supposed to become Z3's default mode once stable. For CVC4, we experimented with the lazy preprocessing options `--no-strings-lazy-pp` and `--strings-lazy-pp`. For Z3, we used `rewriter.cache_all=true`, `rewriter.eq2ineq=true`, `rewriter.hoist_mul=true`, `rewriter.pull_cheap_ite=true`, and `rewriter.flat=false`.

Bug types. During testing, we encountered many different kinds of bugs. We distinguish them by the following categories.

- *Soundness bug*: Formula φ triggers a soundness bug if solvers S_1 and S_2 both do not crash and give different satisfiability results.
- *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 an assertion violation or a segmentation fault.

TypeFuzz detects soundness bug triggers by comparing the standard outputs of the solvers. TypeFuzz detects invalid model bug triggers by internal errors using the model of the SMT solver. Crash bug triggers are detected whenever a solver returns a non-zero exit code and no timeout occurs.

Bug triggers. Dozens of sizable bug triggers usually point to the same underlying bug. Hence, we need to de-duplicate and reduce the bug triggers. For bug reduction, we use two reducers with complementary strengths: pydelta [Kremer 2021], a domain-specific reducer for the SMT-LIB language and C-Reduce [Regehr et al. 2012], a C code reduction tool that also works for the SMT-LIB language. TypeFuzz collects bug triggers that may stem from the same underlying bug. Hence, we de-duplicated the bug triggers after each fuzzing run with TypeFuzz 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. For soundness and invalid model bugs, we first categorize the bug triggers by theory. We do this because bug triggers in different theories are likely to be unique bugs. Then, we select one bug trigger per theory at a time for reporting. If the bug was fixed, we check the remaining bug triggering formulas of the same theory. If one of them still triggers a bug in the solver, we repeat this process until none of them triggers a bug anymore. We have evaluated 897 bug trigger-seed pairs found by TypeFuzz. The number is much larger than the number of our reported bugs because a bug could often be triggered multiple times. The average seed size is 2,023 bytes, the average bug trigger size is 1,776 bytes. Bug triggers are in most cases not significantly larger than the seeds: 80.7% of the bug triggers are smaller than the seed, while 19.3% of the bug triggers are larger than the seed.

RQ1: How effective is Generative Type-Aware Mutation?

From end of January 2021 to mid September 2021, we have extensively stress-tested the SMT solvers Z3 and CVC4 with TypeFuzz. From the 237 reported bugs, 189 were confirmed, 176 were fixed, 14 were categorized as duplicates, and 8 were won't fixes (see Fig. 5a). As for the duplicates in Z3, their main developer followed a rather aggressive approach by categorizing every bug as duplicate

Status	Z3	CVC4	Total
Reported	177	60	237
Confirmed	135	54	189
Fixed	132	44	176
Duplicate	9	5	14
Won't fix	8	0	8

(a)

Type	Z3	CVC4	Total
Soundness	49	24	73
Crash	47	20	67
Invalid model	39	10	49

(b)

#Options	Z3	CVC4	Total
default	55	28	83
1	12	16	28
2	64	10	74
3+	5	0	5

(c)

Fig. 5. (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.

for which a syntactically similar-looking formula in an open issue existed. The few won't fixes were caused by bugs which the developers confirmed. They were either viewed as not worth fixing or could not be reproduced. Among the confirmed bug (Fig. 5b), the most frequent category are soundness bugs (73 out of 189) followed by invalid model bugs (49 out of 189) and crash bugs (67 out of 189). Most (83 out of 189) of confirmed bugs occurred the defaults modes of the solvers and only 28 out of 189 were with one additional option (see Fig. 5c). For the confirmed bugs with two options in Z3, 63 out of 64 were related to the new core of Z3, which is supposed to replace Z3's default mode, once stable enough. In fact, Z3's main developer appreciated our fuzzing efforts. After dozens of bug fixes for the new core, he wrote:

> *Thanks for targeting the new code. It is a very good use of the fuzzing facilities and helps reaching a more solid state for this so-far not exercised code. All bugs reported in this thread have now been fixed.*

We have also examined the logic distribution of the confirmed bugs. Most confirmed bugs in Z3 were in the QF_S (38 out of 135), followed by the QF_SLIA (32 out of 135) and the QF_NIA (9 out of 135). For CVC4, the top-3 logics are the same: QF_S logic (36 out of 54) followed by the QF_SLIA (13 out of 54). Strikingly, TypeFuzz found 18 bugs in CVC4's default mode. Most prior approaches did not find any bugs in CVC4 [Bugariu and Müller 2020; Numair Mansur et al. 2020], YinYang [Winterer et al. 2021] found eight in nine months and OpFuzz [Winterer et al. 2020a] found eleven in a year. All approaches were reportedly using the SMT-LIB seeds and similar resources as TypeFuzz did. TypeFuzz found these bugs despite the robustified Z3 and CVC4 and all the bug fixes caused by prior fuzzing campaigns.

RQ2: How significant are TypeFuzz's findings?

To understand the significance of our bug findings, we have studied the influence on historic Z3 and CVC4 releases that supported the tested logics. For CVC4, we consider all official releases versions from 1.5 (released on July 10, 2017) and later. For Z3, we consider, versions 4.5.0 (released on Nov 11, 2016) and later. Fig. 6 shows the cumulative bug counts in the different release versions of Z3 and CVC4 respectively. In Z3, TypeFuzz found 4 bugs in the 4.5.0 release. Among these, two were invalid model bugs in QF_SLIA and QF_NIA respectively a segmentation fault in Z3's rewriter flat configuration and an assertion violation which were consecutively baked into later release versions. The first soundness bug occurs at version 4.8.8 (released on Apr 9, 2020). Two more occurred at version and nine more in 4.8.10. For CVC4, one refutational soundness bug in the default mode affects the 1.5 release, which was also baked in the 1.6 release. Two additional soundness bugs are affecting CVC4 1.6, both in the default mode. This makes 3 confirmed bugs affecting the 1.6 release.

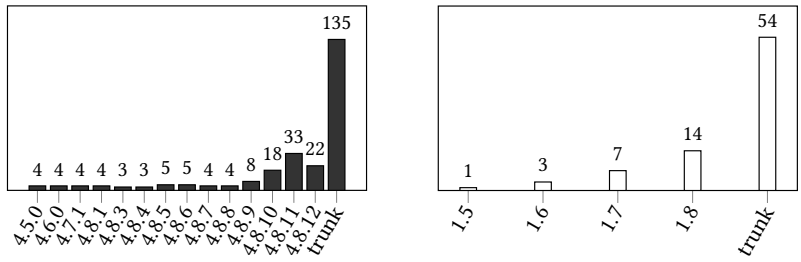


Fig. 6. Confirmed bugs that affect corresponding release versions of Z3 (left) and CVC4 (right).

	Z3			CVC4		
	lines	*functions*	*branches*	*lines*	*functions*	*branches*
Seeds	17.2%	16.5%	10.6%	21.5%	39.7%	8.0%
OpFuzz	17.8%	16.8%	11.1%	22.3%	40.9%	8.4%
TypeFuzz	19.4%	18.7%	11.9%	22.2%	40.7%	8.3%
TypeFuzz + OpFuzz	19.7%	18.8%	12.2%	22.7%	40.9%	8.5%

Fig. 7. Line, function and branch coverage achieved by the baseline seeds, OpFuzz, TypeFuzz and their combination on Z3 and CVC4's respective source codes.

RQ3: Are Generative Type-Aware Mutation and operator mutations orthogonal?

To answer this research question, we have run an experiment to measure the code coverage of TypeFuzz compared to its seeds, the state-of-the-art fuzzer for SMT solvers OpFuzz [Winterer et al. 2020a]. We have sampled 100 files from all test seeds and then ran the following four configurations: A run on each seed from the chosen set with Z3 and CVC4 (Seeds), the state-of-the-art fuzzer OpFuzz, our tool TypeFuzz, and the sequential combination of OpFuzz and TypeFuzz, all with the initial set of seeds. The number of mutating iterations for each seed is 10 and the timeout for each solving query is 8 seconds. For all coverage measurements we used gcov [6] from the GCC suite.

Fig. 7 shows the cumulative coverage data. We first observe that both OpFuzz and TypeFuzz cover strictly more code than the seed set on Z3 and CVC4 respectively. From the first three rows Seeds, OpFuzz, and TypeFuzz, we deduce that both OpFuzz and TypeFuzz can cover additional code as compared to the seeds. For OpFuzz, this increase is rather low (+0.6% LoC in Z3 and +0.8% LoC in CVC4) confirming previous experiments [Winterer et al. 2020a]. For TypeFuzz, the increase is significantly higher in Z3 (+2.2% LoC) and slightly lower in CVC4 (+0.7% LoC). Looking again at the first three rows, we can also deduce that TypeFuzz covers code that OpFuzz does not, since TypeFuzz's percentage is higher than OpFuzz's (17.8% vs 19.4%). From the third and fourth rows, we deduce that OpFuzz also covers different code regions than Typefuzz since there is an increase in code coverage, i.e., 19.7% for TypeFuzz + OpFuzz versus 19.4% for TypeFuzz alone. This indicates that OpFuzz and TypeFuzz are complementary in terms of code coverage.

Limitations. Generative type-aware mutation has been demonstrated to be effective for SMT solver testing. Naturally, it also comes with some limitations. First, generative type-aware mutation cannot add new assertions to the seed formula. Second, it cannot mutate unseen constants. For example, if a bug would be triggered by a term `(= (str.len x) 5)` and all but the constant `"5"` would occur in the seed formula, generative type-aware mutation could not generate the term and may miss the bug. Both type-aware operator mutation and FuzzChick share these limitations.

[6] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

```
(declare-fun x () String)
(declare-fun y () String)
(assert (str.< x (str.replace ""
(str.++ (str.replace "B" x "")
(str.replace "B"
(str.replace "B" x "") "")) y)))
(check-sat)
```

(a) Long-latent solution soundness bug in CVC4 undetected by model validator.
https://github.com/CVC4/CVC4/issues/6075

```
(declare-fun x () String)
(declare-fun y () String)
(declare-fun z () Int)
(assert (not (= (str.substr "B" z
(str.indexof x "" (str.len x)))
(str.substr "B" z (str.len x)))))
(check-sat)
```

(b) Refutation soundness bug in Z3's QF_SLIA logic
https://github.com/Z3Prover/z3/issues/5074

```
(declare-fun a () Bool)
(declare-fun b () Int)
(declare-fun c () String)
(declare-fun d () String)
(assert (= c (str.++ (str.replace d
(str.substr (ite a c d) 0 b) c) d)))
(check-sat)
```

(c) Invalid model bug in Z3's QF_SLIA logic.
https://github.com/Z3Prover/z3/issues/5140

```
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(assert (and (= 0 (- (div 0 0) a))
(= 0 (+ b 1 b (* c
(mod (* a (- 1)) 0))))))
(check-sat)
```

(d) Invalid model bug in Z3's QF_NIA logic.
https://github.com/Z3Prover/z3/issues/5136

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

(e) Crash bug in CVC4 on a QF_NRA formula.
https://github.com/CVC4/CVC4/issues/6228

```
(declare-fun a () String)
(assert (str.< a "ar"))
(assert (str.prefixof "ar"
(str.replace a "ar" "")))
(check-sat)
```

(f) Z3 refutation soundness bug in QF_SLIA logic.
https://github.com/Z3Prover/z3/issues/5117

```
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(declare-fun d () Int)
(declare-fun e () Int)
(assert (and (>= b 0) (<= b 3)
(>= c 2 d) (= c (* 2 a) d)
(= (- (- e c d)) 0) (= e 1)))
(check-sat)
```

(g) Segmentation fault on QF_LIA formula in Z3.
https://github.com/Z3Prover/z3/issues/5035

```
(declare-fun x () String)
(declare-fun y () String)
(assert (= (str.replace
(str.replace x "B" (str.++
"B" "B")) "B" (str.++ y "B"))
(str.++ y "B")))
(check-sat)
```

(h) Refutation soundness bug in CVC4's QF_S logic.
https://github.com/CVC4/CVC4/issues/5915

Fig. 8. Selected bug samples in Z3 and CVC4.

5 BUG SAMPLES

This section details multiple bug samples from our extensive bug hunting campaign of the SMT solvers Z3 and CVC4 and inspects their root causes. The reports shown are reduced bug triggers after bug reduction with pydelta and C-Reduce.

Fig. 8a. shows a solution soundness bug in CVC4. The bug has existed since CVC4 1.7 released on Apr 9, 2019 and pre-dates any fuzzing campaign. The bug is due to an inadmissible rewrite and not detected by the model validator.

```
// (= "" (str.replace x "A" "")) ---> (str.prefix x "A")
- if (StringsEntail::checkLengthOne(ne[1]) && ne[2] == empty)
+ if (StringsEntail::checkLengthOne(ne[1], true) && ne[2] == empty)
{
  Node ret = nm->mkNode(STRING_PREFIX, ne[0], ne[1]);
  return returnRewrite(node, ret, Rewrite::STR_EMP_REPL_EMP);
}
```

src/theory/strings/sequences_rewriter.cpp (132504c)

Method `bool checkLengthOne(Node s, bool strict=false)` checks whether a string expression `s` has length one. The comparison is exact if `strict=true` and otherwise requires `s` to have at most length one. The precondition for the rewrite in the above listing is to check whether `str.replace`'s second argument is a string of length one. The developer fixed the bug by enforcing the strict case.

Fig. 8b. shows a refutational soundness bug in Z3's QF_SLIA logic. The bug was caused by an incorrect sequences axiom (`src/ast/rewriter/seq_axioms.cpp`). The bug trigger has one assert with a negated binary equation. This format has inspired Z3's main developer to add the following rewrite along with the bugfix:

```
    indexof("", b, r) -> if b = "" and r = 0 then 0 else -1
+   indexof(a, "", x) -> if 0 <= x <= len(a) then x else - 1
    indexof(unit(x)+a, b, r+1) -> indexof(a, b, r)
```

src/ast/rewriter/seq_rewriter.cpp (e83f319)

Accordingly, Z3 will rewrite `indexof(a, "", x)` to `x` if index `x` is in the range of the string and to constant `-1` otherwise.

Fig. 8c. shows an invalid model in Z3's QF_SLIA logic. The formula is satisfiable but Z3 returns an invalid model on it. The bug is still pending on Z3's issue tracker.

Fig. 8d. depicts an invalid model bug in Z3's QF_NIA logic. The formula is satisfiable but Z3 reports an invalid model on it. The issue was that Z3 did not match the integer division by zero `(div 0 0)` as an uninterpreted constant as mandated by the SMT-LIB standard. Z3's main developer fixed this regression by adding a matching case for integer division and for modulo, remainder, and division.

```
+     MATCH_BINARY(is_mod0);
+     MATCH_BINARY(is_rem0);
+     MATCH_BINARY(is_div0);
+     MATCH_BINARY(is_idiv0);
```

src/ast/arith_decl_plugin.h (c71bbb6)

Fig. 8e. presents a crash bug in CVC4 triggered by a QF_NRA formula. The pull request in response to this bug got a *major* label. According to a CVC4 developer, CVC4 was incorrectly trying to repair a model when one is not guaranteed to exist, leading to a spurious assertion failure.

Fig. 8f. shows a refutational soundness bug in Z3's string logic QF_SLIA. The formula is satisfiable, but Z3 returns `unsat` on it. A model is realized by `a = "aarr"`. The first assert is satisfied since `"aarr"` is lexicographically smaller than `"ar"`. The second assert is also satisfied by this model: if we replace the `"ar"` within `"aarr"` by the empty string, we obtain `"ar"` which is a prefix of itself. The root cause for this bug was an incorrect rewrite rule for the case when the third argument `str.replace` is empty; in Z3's sequential rewriter. Z3's main developer fixed this bug by removing the faulty rewrite rule and replacing it with a correct one.

Fig. 8g. depicts a segfault in Z3's `rewriter.flat=false` configuration. The segfault is caused by an issue with inconsistent assignments in the lia2pb tactic (`src/tactic/arith/lia2pb_tactic.cpp`) This issue is longstanding: it existed since Z3 4.5.0 which was released on Nov 8, 2016.

Fig. 8h. depicts a refutational soundness bug in CVC4's string logic. Similar to the bug in Fig. 8f, the bug occurs in the sequences rewriter. The logic of the rewrite rule is detailed in the following code snippet:

```
2204      // (str.contains (str.replace x y z) w) --->
2205      //   (str.contains (str.replace x y "") w)
2206      // if (str.contains z w) ---> false and (str.len w) = 1
2207      if (StringsEntail::checkLengthOne(node[1]))
2208      {
2209  -   Node ctn = d_stringsEntail.checkContains(node[1], node[0][2]);
2210  +   Node ctn = d_stringsEntail.checkContains(node[0][2], node[1]);
```

src/theory/strings/sequences_rewriter.cpp (48047e8)

The method `bool checkContains(Node z, Node w)` decides for two string nodes/expressions whether `z` is contained in `w`. `node[0][2]` corresponds to `z` (third child of the `str.replace` expression) and `node[1]` to `w`. The bug occurred since two arguments were reversed which lead to an incorrect precondition for the rewrite rule. The CVC4 team fixed this bug adding it to the regression tests.

6 RELATED WORK

We first discuss related work on SMT solver robustness and performance testing. Then, as generative type-aware mutation is a hybrid of mutation-based and grammar-based fuzzing, we discuss related approaches on mutation-based and grammar-based fuzzing.

SMT solver robustness and performance testing. Our approach is particularly related to the prior works on SMT solver testing. The first approach on testing SMT solvers was the fuzzing tool FuzzSMT [Brummayer and Biere 2009b] which is based on differential testing and targets bit-vector logic. Unlike generative type-aware mutation, FuzzSMT was entirely based on grammar-based fuzzing without a mutational component. FuzzSMT totally found 16 solver defects in five older 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. Thanks to the SMT-LIB initiative [Barrett et al. 2019], SMT theories have been formalized and common input/output file formats have been devised. In addition, the yearly solver competition SMT-COMP [Competition. 2021] heavily penalizes solvers with soundness issues. As a result, the SMT solvers Z3 and CVC4 have robustified and were believed to be quasi-stable. In fact, until October 2019 there were less than 50 soundness bugs reported during eight years of development of CVC4 and around 150 in Z3 in 3 years [Winterer et al. 2020a]. 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 for the unicode string theory in mature solvers such as Z3 and CVC4. Semantic fusion [Winterer et al. 2020b] 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 Mansur et al. 2020], another recent mutation-based SMT solver testing approach, found 27 bugs in Z3, however none in CVC4. Later, type-aware operator mutation [Winterer et al. 2020a] has found several hundreds of bugs in the SMT solvers Z3 and CVC4. However, recently, previous approaches have experienced the saturation effect. Generative type-aware mutation has overcome the shortcomings of previous approaches by combining mutation-based and grammar-based fuzzing. TypeFuzz is a highly practical tool which SMT solver developers can use to stress-test new features conveniently.

Mutation-based fuzzing. Mutation-based fuzzing techniques leverage user-provided test seeds and generate new mutated inputs to uncover bugs in programs. The two closest works from the family of mutation-based testing techniques are skeletal program enumeration (SPE) for testing C compilers [Zhang et al. 2017], and FuzzChick [Lampropoulos et al. 2019] an approach to test Coq programs. Similar to generative type-aware mutation, SPE also performs random type-aware mutations. However, in contrast to generative type-aware mutation, SPE is limited to variables and is not generative. FuzzChick generates test cases by grammar-based generators. FuzzChick stores parameter types and generates new values for the parameters while preserving type-correctness. However, unlike generative type-aware mutation, FuzzChick uses coverage feedback to guide its mutations (grey-box fuzzing) while generative type-aware mutation is a black-box fuzzing technique. Grey-box fuzzing enhances black-box fuzzing by coverage information. The most prominent tool for binary grey-box fuzzing is AFL [Zalewski 2021]. Given a set of test seeds, AFL performs mutations at the binary level, such as bit-shifts, *etc.* 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. Thus, towards structured test inputs, grammar-based grey-box fuzzers were proposed. To generate valid test inputs, grammar-based grey-box fuzzers were proposed. AFLSmart [Pham et al. 2019], Superion [Wang et al. 2019] and Nautilus [Aschermann et al. 2019] are general grammar-based grey-box fuzzers targeting PL engines. They use code coverage to guide the grammar-based mutations.

Generative fuzzing. Generative fuzzers [Hanford 1970] synthesize test inputs from scratch using a language grammar or a (language) model. Csmith [Yang et al. 2011] generates random C programs through repeated application of rules from the C grammar. Similar to generative type-aware mutation, Csmith relies on differential testing to cross-check the generated seeds. Csmith has found 300+ bugs in the compilers GCC and LLVM. A recent follow-up work to Csmith is YarpGen [Livinskii et al. 2020] which additionally prevents generating C programs with undefined behavior. Another recent generative fuzzing approach is pivot query synthesis (PQS) [Rigger and Su 2020]. It synthesizes specific SQL queries on random databases. Unlike Csmith and YarpGen, PQS's is a metamorphic testing approach. Moreover, researchers have adapted generative language models [Godefroid et al. 2017] to generate and guide input generation. As a key difference to all the above approaches, generative type-aware mutation does not generate inputs from scratch but instead supports generation through the substitution of expressions.

7 CONCLUSION

We introduced generative type-aware mutation, a novel and effective approach for testing SMT solvers. Generative type-aware mutation is a powerful generalization of type-aware operator mutation and FuzzChick whose limitations it overcomes. Furthermore, generative type-aware mutation supports generation and can be seen as a hybrid of mutation-based and grammar-based fuzzing. We have realized generative type-aware mutation in the testing tool TypeFuzz with which we ran a bug hunting campaign of the state-of-the-art SMT solvers Z3 and CVC4. Based

on TypeFuzz's findings, we reported 237 bugs out of which 189 were confirmed (176 fixed) on the respective issue trackers of Z3 and CVC4. Many of these bugs were longstanding soundness bugs missed by previous approaches. We are currently working on extending TypeFuzz to SMT theories other than strings and (non-)linear arithmetic.

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, Andrew Reynolds, Haniel Barbosa, Andres Nötzli for useful information and for addressing our bug reports. Chengyu Zhang was partially supported by the NSFC Projects No. 61632005 and No. 61532019.

REFERENCES

Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. 2015. Exploiting the Saturation Effect in Automatic Random Testing of Android Applications. In *MOBILESoft '15*. 33–43. https://doi.org/10.1109/MobileSoft.2015.11

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS '19*. https://doi.org/10.14722/ndss.2019.23412

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV '11*. 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2019. *The Satisfiability Modulo Theories Library (SMT-LIB)*. Retrieved 2021-04-16 from www.SMT-LIB.org

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *SMT '10*.

Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *CAV '18*. 45–51. https://doi.org/10.1007/978-3-319-96142-2_6

Robert Brummayer and Armin Biere. 2009a. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS '09*. 174–177. https://doi.org/10.1007/978-3-642-00768-2_16

Robert Brummayer and Armin Biere. 2009b. Fuzzing and delta-debugging SMT solvers. In *SMT '09*. 1–5. https://doi.org/10.1145/1670412.1670413

Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE '20*. https://doi.org/10.1145/3377811.3380398

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI '08*. 209–224. https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems

The International SMT Competition. 2021. *SMT-COMP*. Retrieved 2021-04-16 from https://smt-comp.github.io/2021/

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS '08*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

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. https://doi.org/10.1145/1066100.1066102

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI '05*. 213–223. https://doi.org/10.1145/1064978.1065036

Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn Fuzz: Machine learning for input fuzzing. In *ASE '17*. 50–59. https://doi.org/10.1109/ASE.2017.8115618

K. V. Hanford. 1970. Automatic generation of test cases. *IBM Systems Journal* 9, 4 (1970), 242–257. https://doi.org/10.1147/sj.94.0242

Gereon Kremer. 2021. *pyDelta: delta debugging for SMT-LIB*. Retrieved 2021-04-16 from https://github.com/nafur/pydelta

Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. In *OOPSLA '19*. 1–29. https://doi.org/10.1145/3360607

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. In *OOPSLA '20*. 1–25. https://doi.org/10.1145/3428264

Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Model-based API testing for SMT solvers. In *SMT '17*. 10.

Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *FSE '20*. 701–712. https://doi.org/10.1145/3368089.3409763

Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *TSE '19* (2019). https://doi.org/10.1109/TSE.2019.2941681

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *PLDI '12*. 335–346. https://doi.org/10.1145/2345156.2254104

Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *OSDI '20*. 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Advisor(s) Bodik, Rastislav. https://dl.acm.org/doi/10.5555/1714168

Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI '14*. 530–541. https://doi.org/10.1145/2666356.2594340

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *ICSE '19*. 724–735. https://doi.org/10.1109/ICSE.2019.00081

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020a. On the Unusal Effectiveness of Type-Aware Operator Mutation. *OOPSLA '20*. https://doi.org/10.1145/3428261

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020b. Validating SMT Solvers via Semantic Fusion. *PLDI '20*. https://doi.org/10.1145/3385412.3385985

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. *yinyang: a fuzzer for SMT solvers*. Retrieved 2021-04-16 from https://github.com/testsmt/yinyang

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI '11*. 283–294. https://doi.org/10.1145/1993316.1993532

Michal Zalewski. 2021. *american fuzzy lop*. Retrieved 2021-04-16 from https://lcamtuf.coredump.cx/afl/

Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *PLDI '17*. 347–361. https://doi.org/10.1145/3140587.3062379

A SOLVER RELEASE DATES

Solver	version	release date
Z3	4.5	08/11/2016
Z3	4.6	18/12/2017
Z3	4.7.1	23/05/2018
Z3	4.8.1	16/10/2018
Z3	4.8.3	20/11/2018
Z3	4.8.4	20/12/2018
Z3	4.8.5	03/06/2019
Z3	4.8.6	20/09/2019
Z3	4.8.7	19/11/2019
Z3	4.8.8	09/05/2020
Z3	4.8.9	11/09/2020
Z3	4.8.10	20/01/2021
Z3	4.8.11	11/07/2021
Z3	4.8.12	13/07/2021
CVC4	1.5	10/07/2017
CVC4	1.6	26/06/2018
CVC4	1.7	09/04/2019
CVC4	1.8	19/06/2020

Table 1. Release dates of Z3 and CVC4.