DIVER: Oracle-Guided SMT Solver Testing with Unrestricted Random Mutations

Jongwook Kim*
Korea University
Republic of Korea
jongwook123@korea.ac.kr

Sunbeom So*
Korea University
Republic of Korea
sunbeom_so@korea.ac.kr

Hakjoo Oh†
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

***Abstract*—We present DIVER, a novel technique for effectively finding critical bugs in SMT solvers. Ensuring the correctness of SMT solvers is becoming increasingly important as many applications use solvers as a foundational basis. In response, several approaches for testing SMT solvers, which are classified into differential testing and oracle-guided approaches, have been proposed until recently. However, they are still unsatisfactory in that (1) differential testing approaches cannot validate unique yet important features of solvers, and (2) oracle-guided approaches cannot generate diverse tests due to their reliance on limited mutation rules. DIVER aims to complement these shortcomings, particularly focusing on finding bugs that are missed by existing approaches. To this end, we present a new testing technique that performs oracle-guided yet unrestricted random mutations. We have used DIVER to validate the most recent versions of three popular SMT solvers: CVC5, Z3 and dReal. In total, DIVER found 25 new bugs, of which 21 are critical and directly affect the reliability of the solvers. We also empirically prove DIVER's own strength by showing that existing tools are unlikely to find the bugs discovered by DIVER.**

***Index Terms*—software testing; fuzzing; SMT solver**

I. INTRODUCTION

Ensuring the correctness of SMT solvers is of paramount importance in software engineering. SMT solvers are the cornerstone of many software-engineering applications, including, among others, program verification [1]–[7], symbolic execution [8]–[14], program repair [15]–[18], and program synthesis [19]–[21]. In these applications, correctness is a crucial success factor yet ultimately depends on the downstream SMT solvers. However, state-of-the-art SMT solvers are large and highly complex software systems; as a result, bugs are not uncommon even in mature, widely-used solvers [22]–[28] such as CVC5 [29] and Z3 [30], threatening the reliability and robustness of a wide range of tools based on SMT solvers.

Goal and Scope. In this paper, we tackle the problem of finding two important classes of bugs in SMT solvers: (1) refutational soundness bugs and (2) invalid-model bugs. A refutational soundness bug (called soundness bug in short hereafter) occurs when an SMT solver incorrectly returns `unsat` instead of `sat` for a satisfiable formula. An invalid-model bug, on the other hand, occurs when a solver correctly answers `sat` for a satisfiable formula but produces an unsatisfying model under which the formula evaluates to *false*.

These bugs critically affect solver-aided tools' reliability. For example, soundness bugs in SMT solvers may invalidate the results of program verifiers; when an SMT solver reports that a satisfiable Verification Condition (VC) is unsatisfiable, unsafe programs are erroneously proved to be safe, which can lead to disasters in safety-critical domains. Also, invalid-model bugs may hinder software testing techniques such as symbolic execution and concolic testing, because test cases obtained from invalid models would fail to explore intended program paths or reproduce detected bugs.

Limitations of Existing Approaches. Recently, a number of techniques have been proposed to test SMT solvers [22]–[28], but they have drawbacks in detecting various soundness and invalid-model bugs. The main challenge is the test oracle problem [31]; the bugs can be identified only when the tested formula is a priori known to be satisfiable. Yao et al. [22] classified existing techniques into two categories based on how they approach the oracle problem: (1) differential testing [26]–[28] and (2) oracle-guided [22]–[25] approaches.

Techniques based on differential testing [26]–[28] address the problem by using multiple solvers. They randomly generate syntactically valid formulas, and check whether all of the solvers agree with the satisfiability results. The strength of this approach is that it can test SMT solvers with diverse formulas generated via unrestricted random mutations. However, its use is fundamentally limited to testing features shared by multiple solvers. In particular, it cannot be used to find bugs in solver-specific yet important features.

On the other hand, oracle-guided techniques [22]–[25] find the bugs by generating formulas that are satisfiable by construction. To this end, they use a pre-defined set of mutation strategies, e.g., satisfiability-preserving transformations [22], [24]. The main strength of this approach is flexibility; it can be used when multiple solvers are unavailable. The downside, however, is that oracle-guided techniques are inherently limited to generating restricted forms of mutant formulas. They cannot detect bugs that are triggered by formulas whose syntax is outside the scope of the pre-defined mutation rules.

Our Approach. In this paper, we present DIVER, a novel technique that complements the shortcomings of existing approaches. Like oracle-guided techniques, DIVER can find bugs without relying on multiple solvers. DIVER, however, aims to

* The first two authors contributed equally to this work.
† Corresponding author

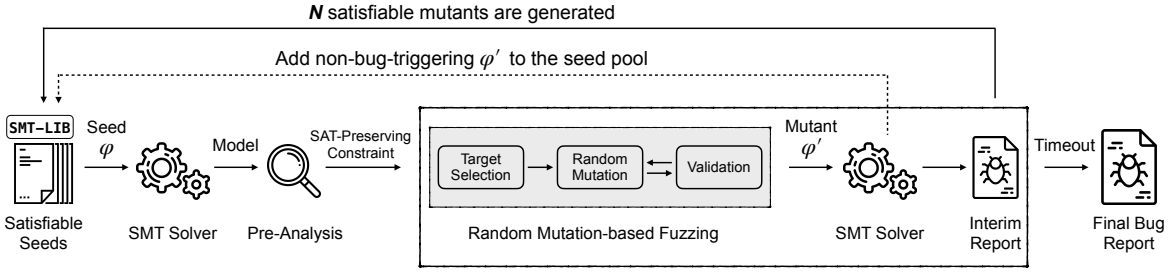

Fig. 1: Overview of DIVER. Input: a pool of satisfiable seed formulas in the SMT-LIB format [32]. Output: a bug report with bug-triggering mutant formulas for which an SMT solver under test produces `unsat` or invalid models.

do so without imposing pre-defined, syntactic restrictions on mutant formulas.

Our basic idea to achieve this goal is simple and intuitive; given a satisfiable seed formula F and its model M, we randomly mutate F until we find a mutant F' that still evaluates to *true* under M. Clearly, such a mutant F' is satisfiable (since M is a model of F') and we can use this information as oracle to validate bugs in SMT solvers. Note that, thanks to the use of unrestricted mutations, our approach has the potential to find bugs beyond the reach of existing oracle-guided techniques. However, naively applying this approach poses a significant performance challenge, since purely random mutations hardly succeed to satisfy the original seed's model.

Thus, DIVER accelerates the basic approach via weighted sampling, so that "easy-to-mutate" sub-terms of a formula are preferentially mutated. To this end, given a seed formula and its model, we perform a pre-analysis which computes, for each sub-term of the seed, the constraints that the mutated sub-terms must meet in order for the resulting mutant formula to be still satisfied by the given model. We then compute the sampling weights of sub-terms by quantifying how constrained the corresponding constraints are, where the intuition is that mutating less constrained sub-terms has higher chances of finding model-satisfying formulas.

Figure 1 depicts the workflow of DIVER based on these ideas. It first performs the pre-analysis to generate the constraints of each sub-term of the seed formula. Next, DIVER repeats the following until a satisfiable mutant formula is found: (1) it selects a sub-term to mutate via weighted sampling based on the pre-analysis results, (2) generates diverse forms of mutant formulas by randomly mutating the sub-term, (3) and validates their satisfiability against the model of the seed formula. Any satisfiable mutant found is used to test the SMT solver. When it does not trigger a bug, we augment the seed pool with the mutant and repeat the process described above.

Results. We demonstrate that DIVER is effective at revealing bugs that are difficult to find with previous approaches. We used DIVER to validate the most recent versions of CVC5 [29], Z3 [30], and dReal [33], which are widely-used and state-of-the-art SMT solvers. In particular, note that CVC and Z3 have been extensively tested over the past few years using previous approaches [22]–[27] and hundreds of bugs have already been fixed in the versions we used. Nonetheless, in the three solvers, DIVER discovered 25 new bugs, of which 21 are confirmed by developers. Notably, 17 out of confirmed 21 bugs are critical bugs (9 soundness bugs + 8 invalid-model bugs). We double-checked that existing approaches are unlikely to find those bugs by running five state-of-the-art tools: three oracle-guided tools (Storm [23], AutoString [24], Fusion [25]) and two differential testing tools (TypeFuzz [26], OpFuzz [27]). Given the seed formulas that DIVER used to find new 25 bugs, the five tools collectively detected 7 bugs only.

Contributions. Our contributions are as follows.

- We present a novel technique for testing SMT solvers, which specifically aims to find bugs that existing techniques are likely to miss. The main contribution is a new oracle-guided technique that does not rely on pre-defined mutation strategies.
- We make DIVER, the implementation of our approach, publicly available.
- We demonstrate the effectiveness of DIVER by finding 25 new bugs in the latest versions of three SMT solvers and comparing with five state-of-the-art fuzzers [23]–[27].

II. OVERVIEW

In this section, we illustrate DIVER using examples.

A. Motivating Example

We motivate our approach by illustrating a bug that is found by DIVER in a recent version[1] of CVC5 [29] and confirmed by the developers. The original and mutant formulas were minimized using a delta debugger [34] so that they only contain essential syntactic elements to trigger the bug.

Figure 2 shows a seed formula, written in the SMT-LIB format [32] and simplified from the SMT-LIB benchmarks [35]. The formula is expressed in `QF_SLIA` (the theory of quantifier-free string and linear integer arithmetic). At line 9, command `check-sat` asks whether the formula, which is the conjunction of the sub-formulas at lines 3–6, is satisfiable. The formula is satisfiable as there exists a model $[t \mapsto \text{“}-0\text{”}]$ that makes all sub-formulas at lines 3–6 evaluate to *true* (as explained in the comments).

Given this satisfiable seed formula, DIVER found a critical soundness bug in CVC5. DIVER generated the bug-triggering

[1] https://github.com/cvc5/cvc5/releases/tag/cvc5-1.0.0

```
1  (set-logic QF_SLIA); satisfiable when t = "-0"
2  (declare-fun t () String)
3  (assert (str.prefixof "-" (str.substr t 0 1))) ; (str.substr t 0 1) = "-"
4  (assert (> (str.len (str.substr t 0 2)) 1))    ; (str.substr t 0 2) = "-0"
5  (-)(assert (not (= (- 1) (str.to_int (str.substr t 1 1))))) ; (str.substr t 1 1) = "0"
6  (-)(assert (>= (+ 0 2) (str.len t)))                        ; (str.len t) = 2
7  (+)(assert (not (xor (str.< (str.update "-0" 0 t) "-0") false))) ; not (xor false false) = true
8  (+)(assert (str.suffixof (str.replace t "-0" "-") "-"))     ; (str.replace t "-0" "-") = "-"
9  (check-sat)
```

Fig. 2: A simplified formula in the SMT-LIB format. DIVER detected a critical soundness bug in CVC5 by replacing the original sub-formulas at lines 5–6 with the new sub-formulas at lines 7–8. The comments at lines 3–8 explain why each of the sub-formula evaluates to *true* under the seed model $[t \mapsto \text{“}-0\text{”}]$.

mutant formula by replacing the sub-formulas at lines 5–6 with the sub-formulas at lines 7–8. The resulting mutant formula (i.e., the conjunction of sub-formulas at lines 3, 4, 7, and 8) still evaluates to *true* under the same model $[t \mapsto \text{“}-0\text{”}]$, but CVC5 erroneously reported that it is unsatisfiable. Note that the mutant generated by DIVER is substantially different from the original. To trigger the bug, for example, DIVER introduced new functions such as `str.update` (line 7) and `str.suffixof` (line 8), which are absent in the original seed formula.

Discovering such a bug-triggering mutant is beyond the reach of existing testing techniques [22]–[28]. The oracle-guided approaches [22]–[25] are ineffective in this case because it is virtually impossible to devise general mutation rules that capture the seemingly unrelated changes from lines 5–6 to lines 7–8. The differential testing approaches [26], [27] are also not applicable since the newly introduced function `str.update` at line 7 is supported only by CVC5. Therefore, cross-checking using multiple solvers is not possible.

B. How DIVER Works

Suppose the formula φ is given as a satisfiable seed:

$$\varphi = \ ((0.0^{l_3} \leq y^{l_4})^{l_1} \wedge (x^{l_5} = (2.0^{l_7} + y^{l_8})^{l_6})^{l_2})^{l_0}$$

where all sub-terms of φ are annotated with labels l_0–l_8. Suppose also we are given a satisfying model M for φ:

$$M = [x \mapsto 3.0, y \mapsto 1.0]$$

which can be obtained by invoking an SMT solver on φ. The goal of DIVER is to generate a mutant formula φ' that is still satisfiable but causes soundness or invalid-model bugs.

Basic Approach. Given the seed formula φ and the satisfying model M, DIVER basically repeats (1) selecting a sub-term t^l in φ, (2) randomly generating a syntactically valid sub-term t', and (3) validating the resulting mutant $\varphi' = \varphi[t'/t^l]$ against M. When φ' is *true* under M, we invoke the SMT solver under test on φ' to see if φ' causes a bug (i.e., the solver produces `unsat` or an invalid model). For example, DIVER may select the term y^{l_4} and applies the following mutation at l_4 of φ:

$$y \rightarrow ((y - 1.0)/x)$$

which replaces the original term y at l_4 by $((y-1.0)/x)$. The resulting mutant formula φ' is as follows:

$$\varphi' = \ ((0.0^{l_3} \leq ((y - 1.0)/x)^{l_4})^{l_1} \wedge (x^{l_5} = (2.0^{l_7} + y^{l_8})^{l_6})^{l_2})^{l_0}$$

Note that the mutant formula φ' evaluates to *true* under the model M. In this case, we run the SMT solver to see if it returns `unsat` or an invalid model on φ'.

Acceleration via Weighted Sampling. The success of our approach depends on how frequently we can generate mutant formulas that are satisfied by the given model. The basic approach falls short in this aspect as it merely relies on purely random mutations.

DIVER addresses this issue by performing random mutations with weighted sampling, where weights are systematically computed from the constraints generated by a pre-analysis. The result of the pre-analysis is a set C of satisfiability-preserving constraints, which are sufficient conditions in order for mutants to be satisfied by the given model M. For the example seed formula and model, DIVER generates the following constraints:

$$C = \left\{ \begin{array}{llllll} l_0 & \mapsto & \widehat{true}, & l_1 & \mapsto & \widehat{true}, \\ l_2 & \mapsto & \widehat{true}, & l_3 & \mapsto & [-\infty, 1.0], \\ l_4 & \mapsto & [0.0, +\infty], & l_5 & \mapsto & [3.0, 3.0], \\ l_6 & \mapsto & [3.0, 3.0], & l_7 & \mapsto & [2.0, 2.0], \\ l_8 & \mapsto & [1.0, 1.0] & & & \end{array} \right\}$$

which maps each label l_i in φ to a constraint that can yield a satisfiable mutant by mutating a sub-term at l_i. For example, the constraint for l_4, i.e., $C(l_4) = [0.0, +\infty]$, indicates that we can freely mutate the term y at l_4 of φ as long as the resulting mutant at l_4 evaluates to a value within the range $[0.0, +\infty]$.

Next, DIVER computes the weight of a sub-term by quantifying how weakly the term is constrained in C. For example, we prefer the terms at $l_3 - l_4$ to the terms at $l_5 - l_8$, because $C(l_3)$ and $C(l_4)$ are less constrained than $C(l_5)$ and $C(l_8)$, and therefore mutating the terms at $l_3 - l_4$ would give a higher chance of producing satisfiable mutants. Concretely, according to the probability estimation method in Section III-B, we select the terms at $l_3 - l_4$ with the probability 0.2854 and the terms at $l_5 - l_8$ with the probability 0.0003.

III. THE DIVER ALGORITHM

In this section, we describe our approach in detail.

Notations. A term in a formula φ denotes a sub-expression of φ, such as a variable, a constant, and an application of a function or a predicate. We assume each term in a formula is associated with a unique label. For example, a formula $\varphi =$

Algorithm 1 DIVER Algorithm

Input: A set of seed formulas ($Seed$), an SMT solver (S)
Output: Sets of formulas that trigger soundness bugs (B_s), invalid-model bugs (B_i), and crash bugs (B_c)

1: $(B_s, B_i, B_c) \leftarrow (\emptyset, \emptyset, \emptyset)$
2: **repeat**
3: $\varphi \leftarrow$ randomly select a seed formula from $Seed$
4: $Seed \leftarrow Seed \setminus \{\varphi\}$
5: $M \leftarrow \mathsf{Model}_S(\varphi)$
6: $C \leftarrow$ PREANALYSIS(φ, M) ▷ § III-A
7: $i \leftarrow 0$ ▷ i: # of satisfiable mutants so far
8: **while** $i \leq n$ **do** ▷ n: pre-set mutation bound
9: $\varphi' \leftarrow$ GENMUTANT(φ, M, C) ▷ § III-B
10: **if** $\varphi' \neq \bot$ **then**
11: $i \leftarrow i + 1$
12: $res \leftarrow \mathsf{CheckSAT}_S(\varphi')$
13: **if** $res = \mathsf{UNSAT}$ **then**
14: $B_s \leftarrow B_s \cup \{\varphi'\}$ ▷ soundness bug
15: **else if** $res = \mathsf{SAT}$ and
16: $\mathsf{Model}_S(\varphi') \not\models \varphi'$ **then**
17: $B_i \leftarrow B_i \cup \{\varphi'\}$ ▷ invalid-model bug
18: **else if** $res = \mathsf{Crash}$ **then**
19: $B_c \leftarrow B_c \cup \{\varphi'\}$ ▷ crash bug
20: **else**
21: $Seed \leftarrow Seed \cup \{\varphi'\}$
22: **until** pre-determined time budget expires
23: **return** (B_s, B_i, B_c)

$(a^{l_1} \geq (b^{l_3} + c^{l_4})^{l_2})^{l_0}$ has terms $\{(a^{l_1} \geq (b^{l_3} + c^{l_4})^{l_2})^{l_0}, (b^{l_3} + c^{l_4})^{l_2}, a^{l_1}, b^{l_3}, c^{l_4}\}$. We write $M(t)$ for the evaluation result of term t under model M, which is inductively defined on the structure of terms. For example, given a formula φ and its satisfying model $M = [a \mapsto 2, b \mapsto 1, c \mapsto 1]$, $M(b + c) = 2$.

Overall Algorithm. Algorithm 1 shows the overall workflow of DIVER. The inputs of the algorithm are a set $Seed$ of satisfiable seed formulas in the SMT-LIB format [32] and an SMT solver S to test. The outputs are three kinds of bug-triggering formulas, which are mutants of $Seed$: B_s (soundness bugs), B_i (invalid-model bugs), and B_c (crash bugs). Note that, while our main focus is to find soundness and invalid-model bugs, our approach can find crash bugs too as a side effect. The outer fuzzing-loop (lines 2–22) consists of the preparation step (lines 3–7) and the inner while-loop (lines 8–21). The preparation step generates useful information (explained shortly) to guide the while-loop. The while-loop iteratively searches for satisfiable mutant formulas, which can be used to automatically identify critical bugs in SMT solvers.

At line 1, we initialize B_s, B_i, and B_c with the empty set, and enter the fuzzing-loop. We then randomly select a seed formula φ from $Seed$ (line 3) and remove it from $Seed$ (line 4). At line 5, we obtain a satisfying model M for φ by invoking the SMT solver S. At line 6, we generate a set C of satisfiability-preserving (SAT-preserving) constraints. Specifically, C contains (mostly complete, see Section III-A) semantic specifications that represent the space of possible mutant formulas. At line 7, we initialize i, which stores the number of satisfiable mutants generated so far, with 0.

Now we enter the while-loop (lines 8–21) that repeats until n satisfiable mutant formulas are generated (we set n to $1,000$ in experiments). We first try to generate a satisfiable formula φ' (line 9) that is the mutant of the seed φ. If we fail to produce φ' ($\varphi' = \bot$), we move on to the next iteration. If we succeed to find φ' ($\varphi' \neq \bot$, line 10), we increase i by 1 (line 11) and check the satisfiability result of φ' with the SMT solver S (line 12). If S incorrectly answers that φ' is unsatisfiable (line 13), we add φ' to B_s (line 14 – we found a soundness bug). If S correctly answers φ' is satisfiable (line 15) but provides a buggy (unsatisfying) model (line 16), we add φ' to B_i (line 17 – we found an invalid-model bug). If a crash occurs when invoking S on φ' (line 18), we add φ' to B_c (line 19 – we found a crash bug). If φ' does not trigger any types of bugs (line 20), we add it to the seed pool (line 21); this way, we can generate another mutant formula that has multiple mutant terms. The entire fuzzing-loop continues until a pre-determined time limit is reached.

A. Generating Constraints via Pre-Analysis

We explain our pre-analysis for obtaining the set of SAT-preserving constraints C, which are used to accelerate our random mutation-based fuzzing procedure (Section III-B). More specifically, by computing the range of possible values that each mutant term can have, the pre-analysis aims to identify easy-to-mutate terms for which random mutations are likely to be successful (as the ranges are wider, random mutations are more likely to succeed in producing satisfiable mutants).

SAT-Preserving Constraint. We first describe the notion of SAT-preserving constraints C. Suppose we have a seed formula and its satisfying model M. The set C is a mapping from labels to abstract values ($\widehat{\mathbb{V}}$). An abstract value for each label can be viewed as the range of possible values of a mutant term to produce a formula satisfiable by M. We consider the following three kinds of abstract values ($\widehat{\mathbb{V}} = \widehat{\mathbb{B}} \cup \mathbb{I} \cup \widehat{\mathbb{S}}$), because, in this paper, we focus on testing formulas in the theories of (non-)linear integer arithmetic, (non-)linear real arithmetic, and string, and their combinations; we discuss the generality of our approach in Section V.

- $\widehat{\mathbb{B}} = \{\top_{\widehat{\mathbb{B}}}, \widehat{true}, \widehat{false}\}$ denotes abstract boolean values with a partial order $b_1 \sqsubseteq_{\widehat{\mathbb{B}}} b_2 \iff (b_1 = b_2) \vee (b_2 = \top_{\widehat{\mathbb{B}}})$. For example, for some label l, the constraint $C(l) = \top_{\widehat{\mathbb{B}}}$ means: if the value of a mutant term at l is either $true$ or $false$ under M, a resultant mutant formula will be satisfiable by M. Also, $C(l) = \widehat{true}$ (resp., $C(l) = \widehat{false}$) means: if the value of a mutant term at l is *true* (resp., *false*), a resulting formula must be *true* under M.
- $\mathbb{I} = \{[l, u] \mid l, u \in \mathbb{Z} \cup \mathbb{R} \cup \{-\infty, +\infty\}, l \leq u\}$ denotes interval values that abstract integers ($\mathbb{Z}$) and real numbers ($\mathbb{R}$) with a partial order $[a, b] \sqsubseteq_{\mathbb{I}} [c, d] \iff (c = -\infty \wedge d = +\infty) \vee (c \leq a \wedge b \leq d)$. An example is in Section II-B.

- $\widehat{\mathbb{S}}$ denotes abstract string values (regular expressions) with a partial order $r_1 \sqsubseteq_{\widehat{\mathbb{S}}} r_2 \iff r_1 = \{s\} \wedge s \in r_2$ where s is a string constant. For example, when Σ is a set of unicode characters and Σ^* is the Kleene closure of Σ, $C(l) = a \cdot \Sigma^*$ means: if a mutant term evaluates to a string that begins with a under M, a resulting formula must be satisfiable by M.

***The Procedure* PREANALYSIS.** Alogrithm 2 shows the workflow of PREANALYSIS (line 6 in Algorithm 1) that aims to produce SAT-preserving constraints C. At line 1, we initialize T with φ, where T is a workset for maintaining terms that need to be explored for collecting C. At line 2, we initialize C with $[l_0 \mapsto \widehat{true}]$ where l_0 is the outermost label of φ and $\widehat{true}$ is an abstract boolean value for *true*; this constraint means we enforce any mutant formula to be *true* under M. We enter the loop (lines 3–8). We choose a labeled term t^l from T (line 4) and remove it from T (line 5). At line 6, we invoke $\mathcal{A}_{M,C}$ (explained shortly) to obtain SAT-preserving constraints for sub-terms of t^l and we add the result to C ($\sqcup$ denotes a standard map-join operator). At line 7, we produce $\mathsf{Sub}(t^l)$, a set of immediate sub-terms of t^l, and add it to T. For example, given a term $t^{l_1} = (a^{l_2} \geq (b^{l_4} + c^{l_5})^{l_3})^{l_1}$, $\mathsf{Sub}(t^{l_1}) = \{a^{l_2}, (b^{l_4} + c^{l_5})^{l_3}\}$. The algorithm repeats until T becomes empty (line 8), i.e., until constraints for all terms of φ are obtained.

Constraint Generation Rules. The function $\mathcal{A}_{M,C}$ at line 6 in Algorithm 2, which extracts SAT-preserving constraints for all immediate sub-terms of t^l, is defined as follows:

$$\mathcal{A}_{M,C}(t^l) = \begin{cases} \mathcal{B}_{M,C}(t^l) \text{ if } \mathsf{logic}(t) = \mathsf{Bool}, \mathcal{I}_{M,C}(t^l) \text{ if } \mathsf{logic}(t) = \mathsf{Int}, \\ \mathcal{R}_{M,C}(t^l) \text{ if } \mathsf{logic}(t) = \mathsf{Real}, \mathcal{S}_{M,C}(t^l) \text{ if } \mathsf{logic}(t) = \mathsf{String}. \end{cases}$$

$\mathcal{B}_{M,C}$, $\mathcal{I}_{M,C}$, $\mathcal{R}_{M,C}$, and $\mathcal{S}_{M,C}$ are functions that return abstract boolean, integer, real, and string values for sub-terms of t^l, respectively. Table I and II show a subset of the rules for generating semantic constraints for each logic.

The basic principle in designing the rules is to compute the complete range of possible values that a mutant term can have, as long as allowed by (1) the model M, (2) the root SAT-preserving constraint $C(l_0) = \widehat{true}$ (line 2 in Algorithm 2), and (3) higher-level constraints generated so far. Here, trying to compute the complete range is intended to identify "easy-to-mutate" terms as accurately as possible. The relevant rules are shown in Table I. For example, in the second rule for theory of integers, if $C(l_3) = \widehat{true}$, we produce the constraints so that the value of a mutant term at l_1 (resp., l_2) is always greater than or equal to (resp., less than) the value of its neighbor term y^{l_2} (resp., x^{l_1}) under M.

As special cases, when it is hard to compute complete ranges, we simply produce constraints using the current values under M. The relevant rules are shown in Table II. For example, in the rule for string logics, the most general constraint for l_1 is a regular expression X that satisfies $X \cdot M(b) = C(l_3)$. Since finding such a regular expression X that meets the condition can be challenging, we simply produce the constraint $M(a)$ for l_1. Note that a mutant that satisfies a constraint generated by this way still guarantees to preserve the satisfiability by M, because any term in the formula will have the same value under M.

Algorithm 2 The procedure PREANALYSIS

Input: A formula φ, a satisfying model M for φ
Output: A set of SAT-preserving constraints C

1: $T \leftarrow \{\varphi\}$
2: $C \leftarrow [l_0 \mapsto \widehat{true}]$ ▷ l_0: the outermost label of φ
3: **repeat**
4: $\quad t^l \leftarrow$ select a labeled term from T
5: $\quad T \leftarrow T \setminus \{t^l\}$
6: $\quad C \leftarrow C \sqcup \mathcal{A}_{M,C}(t^l)$
7: $\quad T \leftarrow T \cup \mathsf{Sub}(t^l)$
8: **until** $T = \emptyset$
9: **return** C

Property of SAT-Preserving Constraints. Given a term t^l, let us write $\alpha(M(t)) \sqsubseteq \widehat{v}$ iff if the value of t^l under M is within the range specified by the abstract value $\widehat{v} \in \widehat{\mathbb{V}}$, where α is the abstraction function that transforms concrete values to abstract values:

$$\alpha(M(t)) = \begin{cases} \widehat{true} & \text{if } \mathsf{sort}(t) = \mathsf{Bool} \text{ and } M(t) = true \\ \widehat{false} & \text{if } \mathsf{sort}(t) = \mathsf{Bool} \text{ and } M(t) = false \\ [n, n] & \text{if } \mathsf{sort}(t) = \mathsf{Int} \text{ and } M(t) = n \\ [n, n] & \text{if } \mathsf{sort}(t) = \mathsf{Real} \text{ and } M(t) = n \\ \{s\} & \text{if } \mathsf{sort}(t) = \mathsf{String} \text{ and } M(t) = s. \end{cases}$$

The following proposition states that the set of SAT-preserving constraints C produced by Algorithm 2 *soundly captures* (i.e., under-approximates) the space of SAT-preserving mutant formulas; a mutant formula that meets C must be satisfiable by the model M of a seed formula.

Proposition 1. *Let φ be a seed formula satisfiable by a model M and L be the set of all labels in φ. Let C be the set of SAT-preserving constraints obtained by running Algorithm 2. For any target term annotated with a label $l \in L$, if the value of a mutant term q is within the range of $C(l)$, the resulting mutant formula should be satisfiable by the seed's model M:*

$$\forall l \in L.\ \alpha(M(q)) \sqsubseteq C(l) \implies M \models \varphi[q^l/p^l].$$

The proposition is proved by structural induction on our constraint generation rules. The converse, however, is not always true as we have incomplete rules such as the ones in Table II. Inspired by Proposition 1, we can systematically enhance our random mutation procedure (Section III-B).

B. Generating Satisfiable Mutant Formulas

Now, we explain how to use the pre-analysis results to effectively perform unrestricted random mutations. The pre-analysis results are used to sample a sub-term to mutate.

***The Procedure* GENMUTANT.** Algorithm 3 shows the procedure GENMUTANT (line 9 in Algorithm 1) that generates a satisfiable mutant formula φ'. At line 1, according to the

TABLE I: Basic rules (selected) for generating SAT-preserving constraints. These rules try to extract the most general condition for each mutant term without violating existing SAT-preserving constraints. In the rule for string logics, Σ is a set of unicode characters. $X = \{c \mid c \in \Sigma, M(y)[0] >_s c\}$ and $Y = \{c \mid c \in \Sigma, c >_s M(y)[0]\}$ are characters smaller and greater than $M(y)[0]$ in lexicographic ordering ($>_s$), respectively.

Term Logic	Input (Term)	Output (Constraint)
Bool ($\mathcal{B}_{M,C}$)	$(x^{l_1} \wedge y^{l_2})^{l_3}$	$[l_1 \mapsto \top_{\widehat{\mathbb{B}}},\ l_2 \mapsto \top_{\widehat{\mathbb{B}}}] \quad \cdots$ **if** $C(l_3) = \top_{\widehat{\mathbb{B}}}$, $\quad [l_1 \mapsto \widehat{true},\ l_2 \mapsto \widehat{true}] \cdots$ **if** $C(l_3) = \widehat{true}$
		$[l_1 \mapsto \top_{\widehat{\mathbb{B}}},\ l_2 \mapsto \widehat{false}] \cdots$ **if** $C(l_3) = \widehat{false}, M(x) = true, M(y) = false$
		$[l_1 \mapsto \widehat{false},\ l_2 \mapsto \top_{\widehat{\mathbb{B}}}] \cdots$ **if** $C(l_3) = \widehat{false}, M(x) = false, M(y) = true$
		$[l_1 \mapsto \top_{\widehat{\mathbb{B}}},\ l_2 \mapsto \top_{\widehat{\mathbb{B}}}] \quad \cdots$ **if** $C(l_3) = \widehat{false}, M(x) = false, M(y) = false$
Int ($\mathcal{I}_{M,C}$), Real ($\mathcal{R}_{M,C}$)	$(x^{l_1} \geq y^{l_2})^{l_3}$	$[l_1 \mapsto [-\infty, +\infty],\ l_2 \mapsto [-\infty, +\infty]] \quad \cdots$ **if** $C(l_3) = \top_{\widehat{\mathbb{B}}}$
		$[l_1 \mapsto [M(y), +\infty],\ l_2 \mapsto [-\infty, M(x)]] \quad \cdots$ **if** $C(l_3) = \widehat{true}$
		$[l_1 \mapsto [-\infty, M(y) - n],\ l_2 \mapsto [M(x) + n, +\infty]] \cdots$ **if** $C(l_3) = \widehat{false}$ ($n = 1$ for Int, $n = 10^{-6}$ for Real)
	$(x^{l_1} + y^{l_2})^{l_3}$	$[l_1 \mapsto C(l_3)\widehat{-}[M(y), M(y)],\ l_2 \mapsto C(l_3)\widehat{-}[M(x), M(x)]]$ where $[a, b]\widehat{-}[c, d] = [a - d, b - c]$
String ($\mathcal{S}_{M,C}$)	$(x^{l_1}\ \texttt{str.} < y^{l_2})^{l_3}$	$[l_1 \mapsto \Sigma^*, l_2 \mapsto \Sigma^*] \quad \cdots$ **if** $C(l_3) = \top_{\widehat{\mathbb{B}}}$
		$[l_1 \mapsto X \cdot \Sigma^*, l_2 \mapsto Y \cdot \Sigma^*] \cdots$ **if** $C(l_3) = \widehat{true}$
		$[l_1 \mapsto Y \cdot \Sigma^*, l_2 \mapsto X \cdot \Sigma^*] \cdots$ **if** $C(l_3) = \widehat{false}$

TABLE II: Constraint extraction rules (selected) for terms whose complete ranges of possible values are hard to be expressed or computed.

Term Logic	Input (Term)	Output (Constraint)
Int ($\mathcal{I}_{M,C}$)	$(x^{l_1}\ \texttt{mod}\ y^{l_2})^{l_3}$	$[l_1 \mapsto [M(x), M(x)],$ $l_2 \mapsto [M(y), M(y)]]$
String ($\mathcal{S}_{M,C}$)	$(x^{l_1} \mathbin{++} y^{l_2})^{l_3}$	$[l_1 \mapsto M(x),\ l_2 \mapsto M(y)]$

probability distribution P (explained shortly), we select a target term p^l to mutate. We enter the loop that iteratively generates satisfiable mutant formulas via randomized mutations and validates the mutants (lines 2–7). At line 3, we randomly generate a mutant term q^l whose type equals to the type of p^l. Specifically, we iteratively and randomly choose an operator (e.g., $\wedge$, $>$, $+$), a variable, or a constant while satisfying the well-typed property. We repeat this process until a complete mutant term q^l is generated or a pre-set iteration bound (5 in the implementation) is reached; for the latter case, we terminate by replacing unfilled operands with random variables (from a seed formula φ) or random constants (from φ or its satisfying model M).

At line 4, we check if a mutant formula is satisfiable by the model M of the seed formula. If so, we randomly choose a final mutant formula φ' out of the two (line 5): the formula where p^l is replaced by q^l (left side), or the formula obtained by a procedure Complicate for adding more complexity (right side, explained shortly). We return the chosen φ' (line 6). If we failed to find a mutant during a given time limit (15m in the implementation) or the pre-determined number of loop iterations (50 in the implementation), we return $\perp$ (line 8).

Weighted Term Selection. The most important step in our approach is selecting the target term at line 1 of Algorithm 3. If the selected term p^l has no chance of having a solution

Algorithm 3 The procedure GENMUTANT

Input: A formula φ, a satisfying model M for φ, a set of SAT-preserving constraints C
Output: A mutant formula φ' or $\perp$
1: Pick a term p^l in φ with a probability distribution P
2: **repeat**
3: Randomly generate q^l such that $\mathsf{sort}(p) = \mathsf{sort}(q)$
4: **if** $M(\varphi[q^l/p^l]) = true$ **then**
5: $\varphi' \leftarrow \mathsf{Rand}(\varphi[q^l/p^l], \mathsf{Complicate}(\varphi, p^l, q^l, M, C))$
6: **return** φ'
7: **until** timeout or pre-set loop-bound exceeded
8: **return** $\perp$

(i.e., satisfying the condition at line 4), all repeated random mutations we try at line 3 are doomed to failure, incurring significant overhead of the overall algorithm.

Our key idea to address this performance issue is to derive a probability distribution P from the SAT-preserving constraints C, and prioritize terms that are likely to have solutions (i.e., satisfying the condition at line 4 of Algorithm 3). The intuition is simple and natural: as constraints in C are more relaxed (more general), the associated sub-terms have larger solution spaces and therefore random mutations are more likely to yield satisfiable mutants. Note that we can design such a probability distribution P, because C completely captures the range of possible values in most cases (Table I) and soundly captures them in the remaining cases (Table II, Proposition 1).

Based on this intuition, we compute the sampling probability $P(t^l)$ for term t^l as follows:

$$P(t^l) = \frac{\mathsf{score}(t^l)}{\sum_{t'^{l'} \in T_\varphi} \mathsf{score}(t'^{l'})}$$

where T_φ denotes the set of all terms in the seed formula φ and

the function score quantifies how much terms are constrained:

$$\mathsf{score}(t^l) = \begin{cases} 1.0 & \text{if } C(l) = \top_{\widehat{\mathbb{B}}} \\ 0.5 & \text{if } C(l) = \widehat{true} \ \vee \ C(l) = \widehat{false} \\ 1.0 & \text{if } C(l) = [-\infty, -] \ \vee \ C(l) = [-, +\infty] \\ 1.0 & \text{if } C(l) = [a, b] \wedge a, b \in \mathbb{Z} \cup \mathbb{R} \wedge b - a \geq 1000 \\ \frac{(b-a)+1}{1000} & \text{if } C(l) = [a, b] \wedge a, b \in \mathbb{Z} \cup \mathbb{R} \wedge b - a < 1000 \\ 1.0 & \text{if } C(l) \in \widehat{\mathbb{S}} \ \wedge \ C(l) \text{ contains } (-)^* \\ 0.001 & \text{if } C(l) \in \widehat{\mathbb{S}} \ \wedge \ C(l) \text{ does not contain } (-)^* \end{cases}$$

Basically, the scores reflect the degree of "weakness" of the constraints (higher is less constrained). For each term, we set the maximum and minimum scores to 1 and 0.001, respectively. Given this, we set the score as 0.5 for the second case to model the half possibility of the first case. In the third and fourth cases, we assign the score 1.0; interval values whose differences are larger than 1000 rarely appeared and therefore we treat them as if they are effectively infinitely large. In the fifth case, $+1$ in the numerator is intended to avoid zero probability. In the sixth and the last cases, we assign the maximum and the minimum scores, because we observed mutating terms whose abstract values with (resp., without) Kleene stars are likely (resp., unlikely) to yield satisfiable mutants in practice.

Adding Complexity. To find bugs more effectively, at line 5 in Algorithm 3, we may complicate a final mutant via the procedure Complicate. Specifically, Complicate tries to add complexity by introducing fresh variables that are absent in the seed formula φ. We design Complicate to work in three steps (for brevity, we assume we introduce one fresh variable, but the extension for multiple variables is straightforward).

- First, from a mutant term q^l, we select a variable or a constant annotated with a label k, denoted m^k. m^k will be replaced by a fresh variable n^k.
- Second, we obtain the SAT-preserving constraints $C' = \textsc{PreAnalysis}(\varphi[q^l/p^l], M)$ in order to identify the constraint for the label k.
- Lastly, we produce a final mutant formula $\varphi' = \psi_1 \wedge \psi_2$ where $\psi_1 = \varphi[q^l[n^k/m^k]/p^l]$ and $\psi_2 = \mathsf{convert}(C'(k))$. ψ_1 means a formula where the target term is replaced by the mutant term in which a variable m^k is also replaced by a fresh variable n^k. ψ_2 is the formula obtained by converting the constraint $C'(k)$ into a corresponding first-order logic formula. For example, if a fresh variable is n^k and $C'(k) = [2, +\infty]$, $\psi_2 = \mathsf{convert}(C'(k)) = (n^{l_b} \geq 2^{l_c})^{l_a}$ where l_a, l_b, and l_c are unique labels.

Note that a final mutant φ' produced by Complicate is still satisfiable, because replacing a target term with a fresh variable does not break the satisfiability and conjoining the sound computation result ($\mathsf{convert}(C'(k))$) for the fresh variable also does not.

IV. Experiments

In this section, we evaluate DIVER to answer the following three major research questions:

- **Effectiveness of DIVER**: How effective is DIVER at finding critical bugs in state-of-the-art SMT solvers? How impactful are the found bugs? (Section IV-A)
- **Comparison with existing tools**: How does DIVER compare to state-of-the-art tools for testing SMT solvers? Can DIVER complement the shortcomings of existing oracle-guided techniques and differential testing-based techniques? (Section IV-B)
- **Utility of our techniques**: What are the benefits of using weighted sampling and Complicate in Section III-B? Are they essential for enhancing DIVER's practicality? (Section IV-C)

Implementation. We implemented DIVER in about 5,000 lines of Python code. We made our own parser for processing the SMT-LIB language [32] using the pyparsing library [36]. To generate constraints for functions in logics related to integers and reals, we used pyinterval [37]. We also used the re module [38] to generate constraints for string logics. The current implementation of the procedure Complicate replaces int and real-typed variables/constants (denoted m^k in Section III-B) only, but it can be extended to replace other types (e.g., string) of variables/constants as well.

Hardware Setup. We performed experiments on four machines: two Linux machines equipped with AMD Ryzen Threadripper 3970X with 64 cores and 64GB and 128GB RAM, a Linux machine with Intel Xeon Processors E5-2630 with 32 cores and 192GB RAM, and a Linux machine with Intel Xeon Silver 4214 CPU with 48 cores and 128GB RAM.

A. Effectiveness of DIVER

Target SMT Solvers. We evaluate the bug-finding capability of DIVER by using it for testing the recent versions of the three SMT solvers CVC5 [29], Z3 [30] and dReal [33], which have been widely used both in academia and industry.

- For CVC5, we tested v.1.0.0 and v.1.0.1 (released on April 2022 and July 2022, respectively), and night versions after v.1.0.0.
- For Z3, we tested v.4.8.14 – 4.11.0 (released on December 2021–August 2022) and night versions after v.4.8.14.
- For dReal, we tested v.4.21.06.2 (released on June 2021 – the latest version). To our knowledge, our work is the first to automatically test dReal.

To validate the three solvers against generated mutant formulas, we provide options necessary to obtain satisfying models (lines 5 and 16 in Algorithm 1), which we consider as configurations for testing their default modes: `--check-sat` for CVC5 and `model_validate=true` for Z3. To test the solvers more extensively, for each mutant formula, we tested the solvers with default modes or by providing up to two additional options. For example, for CVC5, we provided one of the options (e.g., `--sygus-rr-synth-input`) for supporting SyGuS competitions [39] or the string-theory related options (e.g., `--no-strings-lazy-pp`), because bugs related to these options have been reported in the recent issue trackers. We set the timeout for the solvers to 10 seconds.

TABLE III: Statistics on bugs found by DIVER.

Status	Z3	CVC5	dReal	Total
Reported	12	15	2	29
Duplicate	4	0	0	4
New	8	15	2	25
-Confirmed	4	15	2	21
-Fixed	0	14	0	14
-Won't Fix	0	0	0	0

(a) Status of the found bugs. Duplicate: # of bugs that were confirmed as duplicated ones by developers. New: # of bugs after excluding duplicated bugs from Reported. Confirmed: # of bugs that were confirmed to be real by developers out of New. Fixed: # of bugs fixed by developers out of New. Won't Fix: # of bugs that developers won't fix.

Bug Type	Z3	CVC5	dReal	Total
Soundness	6	4	2	12
Invalid-Model	2	7	0	9
Crash	0	4	0	4

(b) Statistics on types of 25 new bugs.

Bug Type	Unique Issues	DIVER	$\frac{\text{DIVER}}{\text{Unique Issues}}$
Soundness	7	4	57.1%
Invalid-Model	18	7	38.9%
Total	25	11	44.0%

(c) Statistics of bugs for CVC5 v.1.0.0 and v.1.0.1 (released on April and July 2022).

Seed Formulas. As a pool of satisfiable seed formulas, we used satisfiable formulas from the benchmarks provided by the SMT-LIB initiative.[2] During the test period that ranges from 4 to 5 months for each tool, we validated the solvers against more than 9,000 seed formulas in total. We used the non-incremental benchmarks that contain quantifier-free formulas over logics for integers, reals, and strings, and their combinations (the full list is in Section V). To improve the testing efficiency, we excluded benchmarks if: (1) the size of a seed formula is too large ($>$ 200KB), or (2) an SMT solver fails to check the satisfiability of a seed formula within 15 seconds. Before providing seed formulas as inputs to DIVER, to ease the implementation in the constraint generation step (Section III-A), we desugared `let`-binding and invocations of defined functions, in a way that preserves the original semantics of a seed formula. For example, given a seed formula that contains a term $let\ x = f(a)$ in $p(x, y)$, our algorithm works on the preprocessed seed formula where the term is transformed into $p(f(a), y)$.

Bug-Reporting Method. When bugs are found by DIVER, in order to meaningfully help developers as much as possible, we did our best to report bugs after deduplicating the found bugs (e.g., deduplicating syntactically different bug-triggering formulas that result in the same error messages). For example, even though DIVER generated 223 bug-triggering formulas during 2 weeks, we reported only 4 bugs after deduplication.

Bug-Finding Results. Table III provides various statistics on the bugs found by DIVER. As shown in Table III(a), we reported 29 bugs to the developers of SMT solvers, of which 25 are new bugs after excluding 4 bugs confirmed to be duplicated by the developers. Out of 25 new bugs, 21 were confirmed to be real and unique by the developers. Out of 25 new bugs, 8 bugs were detected with default modes, 16 bugs with one option, and 1 bug with two options, respectively.

Table III(b) shows the distribution of types of the 25 new bugs in Table III(a). The majority of bugs found by DIVER are critical ones; 84.0% ($\frac{21}{25}$) are either soundness bugs or invalid-model bugs.

Notably, as shown in Table III(c), out of the bugs reported for the very recent versions (v.1.0.0 and v.1.0.1) of CVC5, DIVER found almost half them (44.0%). Specifically, 7 reputational soundness bugs and 18 invalid-model bugs have been reported (and fixed) for these versions, of which 4 soundness bugs (57.1%) and 7 invalid-model bugs (38.9%) were found by DIVER.

In summary, DIVER demonstrated its usefulness by finding critical bugs in the three popular SMT solvers. In particular, considering that CVC5 has been extensively tested in prior works [22]–[28], [40] and therefore detecting bugs in CVC5 is becoming increasingly difficult, our result shows that DIVER is effective at finding bugs potentially missed by existing testing techniques.

Positive Responses from Developers. We found that the bugs found by DIVER were highly useful for enhancing the robustness of the SMT solvers. For example, after fixing the 14 bugs found by DIVER (Table III(a)), all bug-triggering mutant formulas that were reported together have been added to the regression test suites of the CVC5 developers.

Moreover, from the CVC5 developer's comments, we observed that DIVER revealed a fundamental issue and a hard-to-detect error: *"This lemma is unsound, since it assumes the length of a skolem, which can be introduced for other reasons. In other words, the original form of the lemma is not valid."*,[3] and *"This PR fixes a subtle corner case in the generalization within the coverings solver."*.[4] In particular, regarding the first comment, it is interesting that DIVER was able to detect the impactful bug, which was not just an implementation error (it is a logic error) and had lurked in CVC for the past two years.

B. Comparison with Existing Tools

To see whether DIVER indeed complements the major drawbacks of existing approaches [22]–[28], we conducted a comparative experiment. Specifically, we would like to check

[2]https://smtlib.cs.uiowa.edu/benchmarks.shtml

[3]https://github.com/cvc5/cvc5/pull/9014

[4]https://github.com/cvc5/cvc5/pull/8662

TABLE IV: Comparison with existing tools. Bug Type: types of bugs found by DIVER. Solver: the latest versions of SMT solvers in which bugs found by DIVER are reproduced. Theory: background theory for each benchmark formula. Confirmed: whether developers confirmed a bug or not (■: confirmed, □: not confirmed yet). Specific: whether generating mutants with new solver-specific functions is necessary or not for bug-detection (■: necessary – when solver-specific functions are excluded from the search space, DIVER failed to find a bug even in separate runs (5 hours for each trial × 30 trials), □: unnecessary – DIVER found a bug without solver-specific functions). ✓: a tool found a bug at least once during 30 repetitions. ✗: a tool failed to find a bug during 30 repetitions. N/A: a tool does not support the theory of a benchmark formula.

Bug Type	no	Solver	Theory	Confirmed	Specific	Oracle-Guided Tools			Differential Testing-based Tools	
						Storm [23]	AutoString [24]	Fusion [25]	TypeFuzz [26]	OpFuzz [27]
Soundness Bug	1	CVC5 (v.1.0.1)	QF_SLIA	■	■	✗	✗	✗	✗	✗
	2	CVC5 (v.1.0.1)	QF_S	■	□	✗	✗	✗	✗	✗
	3	CVC5 (v.1.0.0)	QF_SLIA	■	■	✗	✗	✗	✗	✗
	4	CVC5 (v.1.0.0)	QF_NRA	■	□	✗	N/A	✗	✗	✗
	5	Z3 (v.4.11.0)	QF_SLIA	■	□	✗	✗	✗	✗	✗
	6	Z3 (v.4.11.0)	QF_SLIA	■	□	✗	✗	✗	✗	✗
	7	Z3 (v.4.11.0)	QF_SLIA	■	□	✗	✗	✗	✗	✗
	8	Z3 (v.4.11.0)	QF_NRA	□	□	✗	N/A	✗	✗	✗
	9	Z3 (v.4.11.0)	QF_NRA	□	□	✗	N/A	✗	✗	✗
	10	Z3 (v.4.11.0)	QF_NRA	□	□	✗	N/A	✗	✗	✗
	11	dReal (v.4.21.06.2)	QF_NRA	■	■	✗	N/A	✗	✗	✗
	12	dReal (v.4.21.06.2)	QF_NRA	■	■	✗	N/A	✗	✗	✗
Invalid-model Bug	13	CVC5 (commit 8311316)	QF_SLIA	■	□	✗	✗	✗	✓	✗
	14	CVC5 (v.1.0.1)	QF_SLIA	■	□	✓	✗	✓	✗	✗
	15	CVC5 (v.1.0.0)	QF_SLIA	■	□	✗	✗	✗	✗	✗
	16	CVC5 (v.1.0.0)	QF_SLIA	■	■	✗	✗	✗	✗	✗
	17	CVC5 (commit bf53190)	QF_SLIA	■	□	✗	✗	✓	✗	✗
	18	CVC5 (commit 0bf059f)	QF_SLIA	■	□	✗	✗	✗	✓	✓
	19	CVC5 (v.1.0.0)	QF_LIA	■	■	✗	N/A	✗	✗	✗
	20	Z3 (v.4.11.0)	QF_S	■	□	✗	✗	✗	✓	✗
	21	Z3 (v.4.11.0)	QF_NIA	□	□	✗	N/A	✗	✓	✗
Crash Bug	22	CVC5 (v.1.0.1)	QF_LIA	■	□	✗	N/A	✗	✗	✗
	23	CVC5 (v.1.0.1)	QF_SLIA	■	□	✓	✗	✓	✗	✗
	24	CVC5 (v.1.0.1)	QF_S	■	■	✗	✗	✗	✗	✗
	25	CVC5 (commit 0bf059f)	QF_SLIA	■	□	✗	✗	✗	✗	✗
Total				■:21 □:4	■:7 □:18	✓: 2 ✗: 23	✓: 0 ✗: 16	✓: 3 ✗: 22	✓: 4 ✗: 21	✓: 1 ✗: 24

whether the 25 new bugs found by DIVER (Table III(a)) can also be detected by existing tools or not.

Before conducting the experiment, we confidently expected that existing oracle-guided tools, which rely on pre-defined transformation rules, would mostly fail to detect the bugs found by DIVER, because mutated parts in all the bug-triggering formulas generated by DIVER are seemingly substantially different from the original parts in the seed formulas. Moreover, we expected all existing tools would fail to find at least 7 bugs, as DIVER could detect them only by generating mutant formulas that newly contain solver-specific functions (marked with ■ in the column Specific of Table IV); differential testing approaches cannot output such mutants as cross-checking is not possible in these cases, and relevant pre-defined mutation rules are highly unlikely to exist in existing oracle-guided tools. We hoped to confirm our plausible conjectures through a real experiment.

Setup. To answer the above research question, we selected the five tools as comparison targets: three oracle-guided tools (Storm [23], AutoString [24], Fusion [25]), and two differential testing-based tools (TypeFuzz [26], OpFuzz [27]). Out of the 7 tools whose main focus is to detect soundness and invalid-model bugs [22]–[28], we could not compare with Sparrow [22] as its implementation is not publicly available, and we did not consider FuzzSMT [28] as it is a generation-based fuzzing tool that does not require seed formulas (i.e., not a mutation-based tool like DIVER) and therefore a direct comparison is non-trivial.

For the experiment, we used the latest versions (as of August 2022) of the 5 tools [23]–[27] from their public GitHub repositories. When running the 5 tools, we provided the latest versions of SMT solvers in which each of the bugs found by DIVER is still reproduced; when invoking SMT solvers on mutant formulas generated by the tools, we supplied the same options where DIVER was able to find the bugs to the solvers. To run Fusion [25] that takes two satisfiable seed formulas as inputs, we simply provided the same seed formulas for each benchmark test. When running the two differential testing tools (TypeFuzz [26], OpFuzz [27]), as comparison solvers for cross-checking results, we used Z3 (v.4.11.0) to test CVC5, and used CVC5 (v.1.0.1) to test Z3 and dReal.

For each seed formula, we ran each tool for 1 hour and repeated the experiments 30 times (using 30 parallel executions), considering their internal randomness. The experiments in this section were conducted on a machine equipped with an AMD Ryzen Threadripper 3970X with 64 cores and 128GB RAM.

Results. Table IV shows the experimental results for the 5 compared tools. The column Solver shows the names and the versions of the SMT solvers, in which the 25 new bugs are found by DIVER. For each tool, which we classified into the oracle-guided tools and differential testing-based tools, ✓ indicates that a tool generated a bug-triggering mutant formula at least once during 30 trials, whereas ✗ means a tool failed to find a bug during 30 trials. In cases where a tool cannot be executed for a seed formula due to unsupported theories, we marked "N/A".

The results prove that, as expected, DIVER can successfully complement the limitations of existing techniques. In summary, 72% ($\frac{18}{25}$) of the bugs could not be found by the 5 existing tools. Specifically, the three oracle-guided tools (Storm [23], AutoString [24], Fusion [25]) and the two differential testing-based tools (TypeFuzz [26], OpFuzz [27]) collectively detected 3 and 4 bugs respectively. We note that, while TypeFuzz reported potential bugs for #11 and #12 in Table IV, we manually checked that they are false positives (hence marked with ✗) raised due to the reliance on cross-checking between solvers; although different results were obtained by each solver, they were all correct behaviors intended by each solver.

We also inspected why TypeFuzz and OpFuzz failed on the 14 and 17 seed formulas, respectively, for which triggering bugs is theoretically possible by the tools that use unrestricted random mutations and differential testing (marked with □ in the column Specific of Table IV). Based on their papers [26], [27], we conjecture that they failed because their mutation strategies are rather restricted in practice. For example, OpFuzz only attempts to mutate function operators [27].

***Robustness of* DIVER.** Note that DIVER is based on random mutation-based fuzzing. Thus, to evaluate the robustness of DIVER under the randomness, we ran DIVER on 25 seed formulas again. The results show that the randomness does not significantly harm the usefulness of DIVER in practice; during the 30 trials, DIVER was still able to find 22 bugs. DIVER still demonstrated its unique strength over the existing tools. From the results, we learned that the unique ability of DIVER, which is able to generate mutant formulas that are satisfiable by construction and substantially different from seed formulas, is crucial for revealing tricky bugs in modern SMT solvers that are likely to be missed by existing testing techniques.

Finding Bugs Detected by Existing Work. We also investigated whether DIVER is able to find bugs that can be detected by existing techniques. We tried hard to obtain specific seed formulas that existing tools had used to find bugs, but we could collect only two seed formulas from the public repository[5] of Storm [23]. Specifically, out of the 20 benchmark seeds from the repository, we were able to use two (bug10 and bug17 in the repository), excluding 18 benchmarks that are currently beyond the scope of DIVER (e.g., unsupported logics).

Given the two seeds used by Storm to find two bugs in the previous versions of SMT solvers, DIVER successfully detected both of the bugs in those solvers.

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

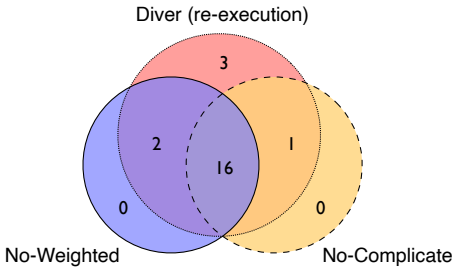

Fig. 3: Bug-finding results of DIVER and its variants.

C. Utility of Our Techniques

To inspect the impacts of weighted sampling and the procedure Complicate in Section III-B, we implemented two variants of DIVER: No-Weighted and No-Complicate. No-Weighted is a variant with Complicate and uniform sampling (rather than weighted sampling), where the probability distribution P at line 1 in Algorithm 3 is the uniform distribution. No-Complicate is a variant with weighted sampling and without Complicate, where line 5 in Algorithm 3 is replaced by $\varphi' \leftarrow \varphi[q^l/p^l]$. We ran No-Weighted and No-Complicate on each of the 25 benchmarks in Table IV, under the same setting used to rerun DIVER (with weighted sampling and Complicate) in Section IV-B.

In summary, the results show that both weighted sampling and Complicate are essential for greatly improving the practicality of DIVER.

Impact of Weighted Sampling. We found that weighted sampling is critical for boosting the performance of DIVER in terms of generating satisfiable mutant formulas; for the 25 seed formulas, DIVER generated 1.6 times more satisfiable mutants than No-Weighted: No-Weighted (3,679,241) vs. DIVER (5,856,673).

We also found that weighted sampling allowed to find more bugs (Figure 3). While DIVER found 22 bugs during re-executions (Section IV-B), No-Weighted detected 18 bugs, the subset of 22 bugs. Moreover, weighted sampling helped to reproduce bugs more stably by 1.5 times; for the 18 bugs detected by both No-Weighted and DIVER, the total numbers of bug-triggering mutant formulas generated by each mode are: No-Weighted (41,734) vs. DIVER (63,559).

Impact of Complicate. Figure 3 also shows that Complicate can enhance the bug-finding ability of DIVER by adding complexity to mutant formulas. DIVER detected 22 bugs, but No-Complicate detected 17 bugs only, the subset of 22 bugs.

V. LIMITATIONS AND FUTURE WORK

We discuss the limitations of DIVER and potential extensions. First, DIVER currently does not support some logics that are supported by existing tools, e.g., bit-vectors and arrays. In this paper, we only focused on testing quantifier-free formulas over the theories of (non-)linear arithmetic for integers and real numbers, strings, and their combined theories,

which are the fundamental logics for reasoning on programs: QF_IDL, QF_LIA, QF_NIA, QF_RDL, QF_LRA, QF_NRA, QF_LIRA, QF_NIRA, QF_S, and QF_SLIA. Note that, however, the underlying principle (i.e., abstraction-based constraint generation rules) of our approach can be generally applicable to other theories. For example, given a term $(x^{l_1}\ \texttt{bvuge}\ y^{l_2})^{l_3}$ in the theory of fixed-sized bit-vectors whose size is n, if $C(l_3) = \widehat{true}$, we can design a rule that generates the constraint $[l_1 \mapsto [M(y), 2^n - 1],\ l_2 \mapsto [0, M(x)]]$, where `bvuge` is the greater-than-or-equal-to operator for unsigned numbers.

Second, DIVER cannot be used to find solution soundness bugs (reporting `sat` instead of `unsat`). In this paper, we focused on finding refutational soundness bugs (reporting `unsat` instead of `sat`) because they are arguably more critical than solution soundness bugs [23], [41]. It would be interesting to extend our work to support solution soundness bugs as well.

VI. RELATED WORK

Testing SMT Solvers. Compared to existing techniques [22]–[28] for testing SMT solvers, DIVER is the first oracle-guided technique that does not resort to pre-defined transformation rules.

One dominant approach for testing SMT solvers is differential testing [26]–[28], which mitigates the test oracle problem [31] by comparing results from multiple solvers. Unlike DIVER, they cannot support cases where cross-checking between solvers is not possible, e.g., testing solver-specific features or all compared solvers produce incorrect results [25]. Falcon [42] can be viewed as a variant of differential testing; given an SMT solver, it detects soundness and invalid-model bugs by treating the solver with different options as different solvers. Falcon inherits limitations of existing differential testing techniques (e.g., when results obtained with different options are all incorrect).

The other approach is the oracle-guided technique [22]–[25], which complements the shortcoming of differential testing-based techniques by leveraging pre-defined mutation rules to generate satisfiable mutants by construction. Their major drawback is the inability to generate diverse forms of mutants, due to the reliance on restricted syntactic mutation rules. DIVER supplements this shortcoming with a new oracle-guided technique that performs unrestricted random mutations.

Other than the above approaches whose main focus is on finding correctness bugs (soundness and invalid-model bugs) in SMT solvers, there are techniques that mainly target performance issues [40], [43], [44]. Unlike ours, these approaches are based on differential testing.

Fuzz Testing in Other Domains. Existing fuzzing techniques can be classified into two groups, depending on how they generate test cases [45]: generation-based fuzzing that synthesizes inputs according to a model (e.g., grammar) without seed inputs, and mutation-based fuzzing that generates mutants by changing some parts of seed inputs. Most of the testing tools [22]–[28] for SMT solvers, including DIVER, are based on mutation-based fuzzing. Although existing mutation-based approaches have shown to be effective by finding numerous bugs in other types of programs such as Linux utilities (e.g., [46]–[49]), they are not directly applicable to our purpose. In particular, for example, existing techniques in other domains do not provide means for resolving the test oracle problem [31] in the context of testing SMT solvers.

Several mutation-based fuzzers proposed methods for carefully selecting mutation locations of a given seed input like ours, but they have different goals. For example, FAIRFUZZ [49] and VUzzer [50] are designed to achieve high code coverage, and TaintScope [51] is designed to improve security vulnerability-finding ability. By contrast, our weighted term selection technique (Section III-B) aims to effectively find soundness and invalid-model bugs in SMT solvers.

VII. CONCLUSION

As SMT solvers are the keystone of many useful software-engineering applications, rigorously validating the correctness of SMT solvers is critically important. In this paper, we presented DIVER, a new oracle-guided technique that addresses the major shortcoming of the existing approaches. We demonstrated its effectiveness by showing that DIVER found 25 new bugs, which are hardly detected by existing testing techniques, in three popular SMT solvers.

DATA AVAILABILITY

The source code of DIVER and the package for replicating the experimental results are available in the public repository: https://github.com/kupl/Diver-Artifact.

ACKNOWLEDGMENT

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair). This research was also supported by the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program(IITP-2023-2020-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation). This work was also supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944). This work was also supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-01198, Convergence security core talent training business(Korea University)).

REFERENCES

[1] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: Safety verification by interactive generalization," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 614–630. [Online]. Available: https://doi.org/10.1145/2908080.2908118

[2] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 184–190.

[3] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370.

[4] W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Program analysis via satisfiability modulo path programs," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 71–82. [Online]. Available: https://doi.org/10.1145/1706299.1706309

[5] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *J. ACM*, vol. 52, no. 3, p. 365–473, may 2005. [Online]. Available: https://doi.org/10.1145/1066100.1066102

[6] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 1678–1694.

[7] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 159–169. [Online]. Available: https://doi.org/10.1145/1375581.1375602

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 322–335. [Online]. Available: https://doi.org/10.1145/1180405.1180445

[10] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 213–223. [Online]. Available: https://doi.org/10.1145/1065010.1065036

[11] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 109–119. [Online]. Available: https://doi.org/10.1145/3238147.3238172

[12] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. USA: IEEE Computer Society, 2008, p. 443–446. [Online]. Available: https://doi.org/10.1109/ASE.2008.69

[13] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[14] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005, p. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[15] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "Jfix: Semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 376–379. [Online]. Available: https://doi.org/10.1145/3092703.3098225

[16] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 691–701. [Online]. Available: https://doi.org/10.1145/2884781.2884807

[17] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 295–306. [Online]. Available: https://doi.org/10.1109/ASE.2015.60

[18] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 772–781.

[19] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 643–659. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko

[20] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 441–452. [Online]. Available: https://doi.org/10.1145/2254064.2254116

[21] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 62–73. [Online]. Available: https://doi.org/10.1145/1993498.1993506

[22] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "Skeletal approximation enumeration for smt solver testing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1141–1153. [Online]. Available: https://doi.org/10.1145/3468264.3468540

[23] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in smt solvers using blackbox mutational fuzzing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 701–712. [Online]. Available: https://doi.org/10.1145/3368089.3409763

[24] A. Bugariu and P. Müller, "Automatically testing string solvers," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1459–1470. [Online]. Available: https://doi.org/10.1145/3377811.3380398

[25] D. Winterer, C. Zhang, and Z. Su, "Validating smt solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 718–730. [Online]. Available: https://doi.org/10.1145/3385412.3385985

[26] J. Park, D. Winterer, C. Zhang, and Z. Su, "Generative type-aware mutation for testing smt solvers," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485529

[27] D. Winterer, C. Zhang, and Z. Su, "On the unusual effectiveness of type-aware operator mutations for testing smt solvers," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428261

[28] R. Brummayer and A. Biere, "Fuzzing and delta-debugging smt solvers," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ser. SMT '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1–5. [Online]. Available: https://doi.org/10.1145/1670412.1670413

[29] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24

[30] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis*

of Systems, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[31] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[32] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[33] S. Gao, S. Kong, and E. M. Clarke, "Dreal: An smt solver for nonlinear theories over the reals," in *Proceedings of the 24th International Conference on Automated Deduction*, ser. CADE'13. Berlin, Heidelberg: Springer-Verlag, 2013, p. 208–214. [Online]. Available: https://doi.org/10.1007/978-3-642-38574-2_14

[34] G. Kremer, A. Niemetz, and M. Preiner, "ddsmt 2.0: Better delta debugging for the smt-libv2 language and friends," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12760. Springer, 2021, pp. 231–242.

[35] "The smt-lib benchmark used in motivating example 1," https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA/-/blob/master/2019-full_str_int/py-conbyte_trauc/lib_int-datetime__parse_hh_mm_ss_ff/27.smt2, accessed: August 2022.

[36] "The github page for pyparsing library," https://github.com/pyparsing/pyparsing, accessed: August 2022.

[37] "The github page for pyinterval library," https://github.com/taschini/pyinterval, accessed: August 2022.

[38] "The webpage for python re module," https://docs.python.org/3/library/re.html, accessed: August 2022.

[39] "The webpage for sygus competition," https://sygus.org, accessed: August 2022.

[40] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, "Stringfuzz: A fuzzer for string solvers," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 45–51.

[41] "The guidelines for fuzzing cvc5," https://github.com/cvc5/cvc5/wiki/Fuzzing-cvc5, accessed: August 2022.

[42] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "Fuzzing smt solvers via two-dimensional input space exploration," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 322–335. [Online]. Available: https://doi.org/10.1145/3460319.3464803

[43] J. Scott, F. Mora, and V. Ganesh, "Banditfuzz: A reinforcement-learning based performance fuzzer for smt solvers," in *Software Verification: 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20–21, 2020, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 68–86. [Online]. Available: https://doi.org/10.1007/978-3-030-63618-0_5

[44] Y. Zhang, X. Xie, Y. Li, Y. Lin, S. Chen, Y. Liu, and X. Li, "Demystifying performance regressions in string solvers," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[45] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[46] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 725–741.

[47] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[48] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2329–2344. [Online]. Available: https://doi.org/10.1145/3133956.3134020

[49] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 475–485. [Online]. Available: https://doi.org/10.1145/3238147.3238176

[50] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[51] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 497–512.