Bounded-Exhaustive Subspace Diversification for SMT Solver Testing

Junda Zheng and Peisen Yao✉

The State Key Laboratory of Blockchain and Data Security, Zhejiang University
{zhengjd04, pyaoaa}@zju.edu.cn

Abstract. SMT solvers form critical infrastructure for many verification and program analysis systems. Recent fuzzing efforts since 2019 have significantly improved solver robustness, yet these approaches often fall short of systematically probing the diverse semantic subspaces within a formula's satisfiability domain. This paper introduces subspace diversification, which systematically partitions the solution space of seed formulas to guide solvers into exploring different behavioral regions. We instantiate the idea using three general, bounded, and efficient mutation strategies that confine the space with cubes, numerical domains, and quantifiers. An extensive evaluation on Z3 and CVC4 demonstrates the effectiveness of our implementation, Canary, which uncovered 108 confirmed bugs across multiple theories and bug types.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers determine the satisfiability of formulas over first-order theories, such as integers, reals, bit-vectors, and strings. SMT solvers have been widely used in various techniques such as symbolic execution [1–3], program verification [4–6], program synthesis [7, 8], program repair [9, 10], refinement types [11, 12], among others. SMT solvers have also been successfully deployed in the industry to address practical software engineering programs, such as finding zero-day software vulnerabilities [13], verifying the safety of radiotherapy machines [14], and enforcing the access control policies of Amazon Web Services [15, 16].

Despite substantial advances, modern SMT solvers remain susceptible to bugs [17–20], such as soundness issues, invalid models, and runtime crashes. These faults compromise the reliability of tools that rely on SMT solvers, particularly in safety-critical software systems. Ensuring the correctness of these solvers is a persistent challenge, particularly in developing effective testing methodologies. In particular, it is challenging to generate test formulas that thoroughly exercise the solver's diverse components, such as preprocessing routines, general frameworks, and theory-specific engines.

Existing approaches to test formula generation generally fall into two categories: generative and mutational. Generative approaches create formulas from scratch—typically by randomly assembling syntactically valid expressions according to target theory grammars [21, 18, 22, 23]—and can produce many

variants. However, it is often hard to steer them toward testing specific solver features. In contrast, mutational approaches systematically transform existing formulas [24, 20, 17, 19, 25]. By starting from known "seed" formulas, these techniques generate small changes that preserve much of the original structure while exploring new solver behaviors.

Yet, even sophisticated mutation strategies tend to focus on localized edits—altering a constraint, flipping a logical operator, or adding redundant terms. Although such transformations can explore a range of syntactic variations, they often fail to expose deeper semantic differences in solver behavior. In particular, they lack mechanisms for deliberately probing distinct logical regions within a formula's satisfiability space. As a result, significant portions of the solver's decision-making paths may remain untested, leaving certain bugs hidden.

This paper presents a novel mutational testing approach for SMT solvers, termed *subspace diversification*. The key insight is that, for a given formula φ, many SMT solvers exhibit deterministic search behaviors that focus on narrow regions of the solution space, potentially overlooking significant unexplored areas. These underexamined sub-regions may conceal latent bugs that evade detection by conventional tests. Subspace diversification addresses this limitation by systematically partitioning the solution space of a formula φ using a set of constraints $M_1, \ldots, M_n$, where each M_i restricts φ to a distinct subspace of the solution space. By guiding the solver to explore these regions, our approach can potentially expose unexpected behaviors or faults.

We evaluate Canary on two widely used SMT solvers—Z3 and CVC4—and demonstrate its effectiveness by uncovering 108 confirmed bugs, many of which have since been fixed. These bugs span multiple theories and encompass critical issues, including soundness violations, invalid model generation, and crashes. Our evaluation also shows that Canary can improve the code coverage and bug-detection efficiency of baseline fuzzers. Furthermore, our head-to-head comparison with state-of-the-art fuzzers (HistFuzz and Yinyang) shows that Canary can discover bugs that existing techniques miss. When integrated with these baseline fuzzers, Canary improves bug detection across all categories, demonstrating that subspace diversification complements existing approaches and can uncover deeper semantic errors.

In summary, our contributions are as follows:

- We introduce subspace diversification, a novel mutation-based testing methodology for exploring under-tested regions of SMT solver behavior.
- We instantiate the methodology by proposing three mutation strategies focused on cubes, numerical domains, and quantifiers.

2 Overview

SMT-LIB2 Language. SMT extends the classical Boolean satisfiability (SAT) problem by incorporating reasoning over first-order theories, such as linear integer arithmetic, real numbers, and strings. The SMT-LIB2 format has become the de facto standard for expressing SMT constraints, offering a unified language

for defining variables, asserting conditions, and invoking satisfiability checks. For instance, the following SMT-LIB2 code snippet defines two integer variables and asserts a constraint that both must satisfy:

```
1  (set-logic QF_LIA)
2  (declare-const x Int)
3  (declare-const y Int)
4  (assert (and (> x 1) (< y 3)))
5  (check-sat)
```

2.1 SMT Formula Generation

The development of rigorous test formulas is central to evaluating the correctness and performance of SMT solvers. These formulas aim to explore solver behavior under diverse and challenging conditions. Two prevailing methodologies exist in the literature: generative and mutational approaches.

Generative Approach. Generative methods produce entirely new formulas from scratch, often guided by the grammar and semantics of the target logic. Several notable strategies include:

- FuzzSMT [21] is the first, grammar-based blackbox fuzzing tool developed to validate SMT solvers.
- StringFuzz [22] uses grammar-based generation to construct formulas tailored to specific theories systematically.
- BanditFuzz [24] uses reinforcement learning-based generation, which adapts its generation policy based on feedback from solver performance.
- Falcon [23] focuses on mutating solver configuration options using a feedback-driven mechanism to test solver behaviors.
- ET [26] is a grammar-based enumerator for systematically validating the correctness and performance of SMT solvers.

Mutational Approach. Mutational approaches modify existing seed formulas to generate new test inputs. This category is further divided into subcategories based on whether the oracle information preserves or evaluates the satisfiability status.

Oracle-Guided Mutations: These techniques rely on known solver outputs or structural transformations that maintain satisfiability. For instance:

- Bugariu and Müller [18] present formula transformations that preserve satisfiability and create increasingly complex formulas to test string solvers.
- Storm [19] generates satisfiable formulas that are structurally different from the original seeds.
- Yiyang [20] combines formulas with identical satisfiability outcomes to create new variants.
- Sparrow [25]: generates formulas using approximation strategies, enabling the construction of test oracles.

– Diver [27]: uses random mutations and assignment-based oracles to test SMT solvers, focusing on satisfiable formulas.

Oracle-Less Mutations: These methods operate without oracle feedback, often using syntactic heuristics or probabilistic models:

– OpFuzz [20] uses a type-aware operator mutation technique targeting first-order logic formulas.
– HistFuzz [28] leverages historical bug-triggering inputs. The method extracts skeletons (core structures) and atomic formulas from past bug reports, then uses association rule mining to guide the generation of new test formulas.

Limitations. Although these techniques can explore nuanced input variations, they often fall short in systematically directing solvers toward semantically distinct regions within a seed formula's satisfiability space. SMT solvers typically employ deterministic search strategies that concentrate on narrow solution regions, neglecting potentially significant unexplored areas. Mutation-based approaches may generate syntactically different formulas without challenging solvers to navigate fundamentally different search spaces. As a result, substantial portions of a solver's decision space may remain untested, allowing subtle bugs to persist undetected.

2.2 Subspace Diversification

To bridge this gap, we propose a new mutation strategy called *subspace diversification*. Rather than relying on random edits or broad syntactic changes, our approach systematically explores unexplored logical sub-regions within a base formula. The central idea is rooted in the observation that SMT solvers often follow deterministic paths during satisfiability checking. As a result, large portions of the formula's solution space may remain unvisited.

Subspace diversification aims to expose these latent execution paths by selectively constraining or activating specific subformulas. This targeted perturbation increases the likelihood of triggering divergent solver behaviors.

To illustrate the approach, consider the following seed formula, which we use for triggering a confirmed bug in CVC4:

```
1  (set-logic QF_NIA)
2  (declare-const a Int)
3  (declare-const b Int)
4  (declare-const c Int)
5  (declare-const d Bool)
6  (declare-const e Int)
7  (assert (or (= (* (+ 0 0 e 0 888) c b a) 0) d))
8  (check-sat)
```

This formula is satisfiable if either disjunct holds. After we replace "check-sat" with "(check-sat-assuming (d))", CVC4 can solve the formula instantly. However, if we replace "check-sat" with "(check-sat-assuming ((= (* (+ 0 0 e 0 888) c b a) 0)))", CVC4 experiences significant performance degradation. Both cases correspond to distinct branches of the disjunction, yet they elicit dramatically different solver behaviors.

The discrepancy was traced to a bug in CVC4's branching heuristic. The developers acknowledged the issue and subsequently fixed it. This example demonstrates the importance of systematically probing alternative semantic paths within a formula—an objective directly addressed by our work.

3 Approach

This section presents our methodology for systematically uncovering bugs in SMT solvers by decomposing the original formula into logically distinct subspaces. We begin by formalizing the solution space partitioning problem (§ 3.1). We then describe three complementary strategies for generating sub-formulas that explore these subspaces (§ 3.2). Finally, we describe how our method integrates with a differential testing workflow to identify and categorize solver inconsistencies (§ 3.3).

3.1 Solution Space Partition

Given an SMT formula φ, its solution space consists of all interpretations (or models) under which φ evaluates to true. We formalize this as follows:

Definition 1. *(Solution Space) The solution space $S(\varphi)$ of a formula φ comprises all interpretations under which φ evaluates to true:*

$$S(\varphi) = \{I \mid I \vdash \varphi\}$$

Example 1. Let $\varphi \equiv p \vee q$, where p and q are Boolean variables. The formula evaluates to `true` under the following interpretations:

$$\begin{aligned} S(p \vee q) = \{&\{p \mapsto true,\ q \mapsto false\},\\ &\{p \mapsto false,\ q \mapsto true\},\\ &\{p \mapsto true,\ q \mapsto true\}\} \end{aligned}$$

Modern SMT solvers typically follow deterministic search trajectories when exploring solution spaces.[1] While this determinism benefits reproducibility and robustness, it creates a critical limitation: during any single execution, large portions of the solution space may remain entirely unexplored.

[1] We can set the random seed and other parameters to diversify the runtime behavior to a certain degree.

To address this, we design a method to produce syntactic mutations of φ that apply targeted constraints, effectively dividing the solution space into multiple, non-overlapping sub-regions. Each resulting sub-formula explores a distinct portion of $S(\varphi)$, encouraging solvers to exercise different decision paths.

Problem Statement. Given a formula φ, our objective is to generate a collection of formulas $\varphi_1, \ldots, \varphi_n$ such that:

- Each φ_i restricts φ to a specific segment of its solution space.
- Each sub-formula φ_i preserves the syntactic and semantic properties necessary for meaningful SMT solver testing.

The key challenge is how to construct informative sub-formulas that prompt diverse solver behavior, without relying on exhaustive enumeration, which is infeasible for formulas with combinatorial or infinite solution spaces, common in theories involving integers, reals, arrays, or strings.

3.2 Partition Strategies

We introduce three strategies for constructing sub-formulas that partition the solution space of a given formula φ: (i) numerical domain constraints, (ii) Boolean cubes, and (iii) quantifier-based transformations. Each strategy is designed to satisfy the following criteria:

- *Diversification* – Each strategy helps target distinct logical regions of the solution space.
- *Generality* – The strategies should be broadly applicable.
- *Efficiency* – The mutations should be fast to enable testing throughput.

Partition via Numerical Domains. First, for formulas over numeric theories, we partition the solution space by constraining variables to lie within specific abstract domains [29, 30]. This strategy draws on techniques from abstract interpretation, using domains such as intervals, zones, and octagons to define semantically meaningful subregions.

Example 2. For an integer formula $\varphi(x, y)$, we can partition its solution space into $\varphi \wedge x > a$ and $\varphi \wedge x \leq a$, where a represents a randomly-generated constant.

For arithmetic-heavy theories, such as nonlinear arithmetic and floating-point logic, solver behavior is often sensitive to the magnitude of numeric values. Partitioning by value range may expose corner cases in arithmetic reasoning, overflow handling, or rounding behavior.

Besides, this approach generalizes naturally to multiple variables and more expressive domains. For instance, we can partition using relational constraints such as $x - y \leq c$ (zones) or $x + y \leq c$ and $x - y \leq c$ (octagons). These abstractions allow us to define subregions that are both expressive and tractable.

Partition via Cubes. Second, inspired by cube-and-conquer approaches in parallel SAT solving [31, 32], we define Boolean cubes as conjunctions of literals (atoms or their negations) to form partitions.

Definition 2. *(Partition Cube) Given a formula φ and a set S of partition predicates, a k-dimensional partition cube C is defined as the conjunction $l_1 \wedge \cdots \wedge l_k$, where each l_i is either a predicate or the negation of a predicate $p \in S$.*

Example 3. Consider the formula $\varphi \equiv p \wedge (q \vee \neg s) \wedge (r \vee s)$, with atoms p, q, r, s. Possible cubes include:

- 1-dimensional: p, $\neg p$, r;
- 2-dimensional: $p \wedge q$, $p \wedge \neg s$;
- 3-dimensional: $p \wedge q \wedge r$, $p \wedge \neg q \wedge \neg s$.

By forcing the solver to commit to specific cubes, we can guide it toward decision paths that may otherwise remain unexplored. Cubes can be constructed over both propositional and theory-level atoms. For instance, in string logic, predicates may include $x =$ "alice" or $z = \texttt{str.++}(x, y)$. In array logic, predicates may include $a[i] = v$ or $select(a, i) = v$. Because cube construction operates over literals, it generalizes across theories without requiring theory-specific reasoning.

This strategy offers two key advantages. First, it provides bounded combinatorial complexity: for k predicates, there are at most 2^k cubes. This allows controlled exploration of the solution space by adjusting k. Second, it is syntactically lightweight: cube construction requires only syntactic analysis of φ, without invoking expensive semantic reasoning.

Partition via Quantifiers. Finally, quantifiers significantly enhance the expressive power of SMT formulas and are essential for modeling systems with variable scope or data abstraction. However, supporting quantifiers introduces significant complexity into SMT solving and is often a source of incompleteness or performance degradation, such as incomplete instantiation strategies that miss relevant ground terms and improper handling of quantifier alternation or variable shadowing.

To further diversify test instances, we apply transformations that inject quantifiers into seed formulas.

Example 4. Given the quantifier-free formula $\varphi(x, y) \equiv x + y < 1$, we may construct quantified variants as follows:

- Universal quantification: $\forall z.\varphi(x, z)$;
- Existential quantification: $\exists z.\varphi(z, y)$;
- Quantifier alternation: $\forall z.\exists w.\forall v.\varphi(z, w, v)$.

Notably, this strategy can also indirectly stress the quantifier-free reasoning engines, since quantifier-handling algorithms, such as MBQI (model-based quantifier instantiation) [33] and E-matching [34], typically employ quantifier-free decision procedures as fundamental subroutines.

3.3 Testing Workflows

Our testing framework utilizes a multi-strategy partitioning scheme, combined with differential testing, to identify discrepancies and crashes in SMT solvers. By

Algorithm 1: Search Space Partition-Based Differential Testing

Input: $solver1, solver2$: SMT solvers under test
Input: Φ: a set of input seed formulas
Input: N: number of mutations per seed formula
Output: $bugs$: set of bug-triggering formulas

```
1  bugs ← ∅;
2  foreach φ ∈ Φ do
3      if φ is unsat then
4          φ ← ¬φ;                          /* negation an unsatisfiable seed */
5      for i = 1 to N do
6          s ← randomly select a mutation strategy;
7          ψ ← apply the strategy to φ;
8          r1 ← solver1.solve(ψ);
9          r2 ← solver2.solve(ψ);
10         if r1 = crash or r2 = crash or r1 ≠ r2 then
11             bugs ← bugs ∪ {ψ};
12 return bugs
```

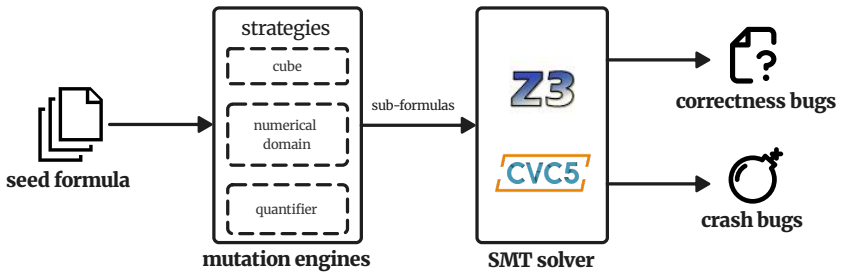


Fig. 1: Overall workflow of Canary

generating logically distinct sub-formulas from an original formula φ, we drive solvers into varied execution paths.

Overall Workflow. Algorithm 1 presents the complete testing pipeline. The algorithm accepts two SMT solvers, a set of seed formulas, and a mutation count per seed. For each seed formula, it first checks satisfiability; if the formula is unsatisfiable, it is negated to ensure a satisfiable starting point. Given a satisfiable seed φ, the algorithm generates N test cases by applying randomly selected mutation strategies. Each mutation yields a variant formula ψ, which is evaluated by both solvers. A discrepancy—either a crash or a disagreement in satisfiability results—signals a potential bug.

It is worth noting that while the quantifier partition strategy can be applied to unsatisfiable seeds, the current implementation focuses on satisfiable formulas by negating unsatisfiable seeds to align with the other mutation strategies.

Bug Categorization. For each subspace ψ, the workflow performs *differential testing*, comparing the results of multiple SMT solvers. This involves running ψ across all solvers and checking for:

- Correctness bugs: Cases where solvers disagree on the satisfiability of ψ (e.g., one returns SAT while another returns UNSAT).
- Crash bugs: Instances where a solver terminates unexpectedly due to internal errors or resource exhaustion.

Any sub-formula that triggers a crash or disagreement is added to the candidate bug set, denoted as `bugs` in Algorithm 1. To streamline debugging and facilitate triage, we apply automated formula minimization tools such as ddSMT. Crashing instances are grouped by failure trace (e.g., assertion location or memory errors), while correctness bugs are categorized and reported to developers by theory (e.g., linear arithmetic, strings).

While our approach is inspired by the idea of bounded-exhaustive testing, our implementation does not attempt to exhaustively enumerate all possible subspaces. Instead, we sample a representative subset of partitions within a given resource budget to balance coverage and efficiency.

4 Evaluation

This section presents a comprehensive assessment of Canary, aiming to answer the following research questions:

- **RQ1**: How effective is Canary in uncovering previously unknown bugs in state-of-the-art SMT solvers (§ 4.1)?
- **RQ2**: How does Canary impact code coverage, and can it enhance the effectiveness of existing SMT fuzzers (§ 4.2)?
- **RQ3**: Can Canary detect bugs effectively compared to existing fuzzers (§ 4.3)?

Tested Solvers. We have selected Z3 and CVC4, the two most popular SMT solvers, for the experimental evaluation because they are popular and widely used in academia and industry, support most of the SMT-LIB2 theories [35], and have been extensively tested by previous efforts [21, 22, 17, 20, 19, 24, 23]. We primarily focus on the solvers' default modes. For CVC4, we enable options such as produce-models, incremental, and strings-exp to support all the seed formulas. We use the check-models option for CVC4 and model.validate=true for Z3 to detect invalid model bugs. Additionally, we test a new SMT core of Z3, which can be activated via the options tactic.default_tactic=smt sat.euf=true.

Baselines. We compare Canary against three state-of-the-art and mutational SMT fuzzing techniques, including HistFuzz [28] and Yinyang [20].

Environment. All experiments are conducted on a Linux workstation equipped with an 80-core Intel(R) Xeon(R) 2.2 GHz processor and 256 GB of RAM. We compile Z3 and CVC4 using GCC 5.4.0, with assertions and AddressSanitizer [36] enabled. We use Gcov [37] to measure the code coverage.

4.1 Analysis of the Discovered Bugs

This section summarizes the bugs identified during testing with Canary and categorizes them based on their nature and severity.

Number of the Bugs. Over a nine-month evaluation period from June 2021 to March 2022, Canary uncovered 123 unique bugs across Z3 and CVC4. Table 1 summarizes the status of these bugs. Out of the total, 108 were confirmed by solver developers, and 107 have already been addressed through patches. These outcomes demonstrate Canary's capacity to surface substantive issues in mature, production-grade solvers.

Types of the Bugs. Table 2 categorizes the confirmed bugs according to their type. Crash bugs form the majority (71 out of 108), followed by invalid model bugs (28) and soundness errors (9). Invalid model bugs indicate situations where the solver produces a model that does not satisfy the input formula, while soundness bugs involve incorrect satisfiability results. Notably, correctness issues (invalid models and soundness bugs) account for over 35% of the confirmed bugs, emphasizing Canary's strength in detecting deep semantic flaws.

Table 1: Summary of the bugs found by Canary.

Status	Z3	CVC4	Total
Reported	83	40	123
Confirmed	75	33	108
Fixed	75	32	107
Duplicate	1	5	6
Won't fix	7	2	9

Table 2: Bug types among the confirmed bugs.

Type	Z3	CVC4	Total
Soundness	6	3	9
Invalid model	21	7	28
Crash	48	23	71

Diversity of the Theories. Figure 2 depicts the distribution of theories among bug-triggering formulas. While AUFLIA, QF_BV, and AUFBV were the most frequent, a broad range of logics were represented, showcasing Canary's generality across multiple theory combinations. Notably, QF_BV—one of the most mature and widely adopted SMT theories—accounts for a substantial portion of the triggered bugs, underscoring the practical relevance of Canary's findings.

Impact on Solvers' Codebase. To assess the effort required to resolve identified bugs, we analyzed the commits associated with them. Specifically, we examined the number of files and lines of code modified in each commit. As illustrated in Figure 3, most fixes altered fewer than five files, suggesting localized issues, yet some required broader changes, highlighting how a single bug may uncover structural weaknesses. In total, 196 files and 5,669 lines of code were modified to fix bugs discovered by Canary.

In Figure 4, we studied the most changed files in these commits fixing bugs found by Canary. In Z3, the most modified files concern the array theory solver and EUF solver, among others. In contrast, the most frequently fixed files in

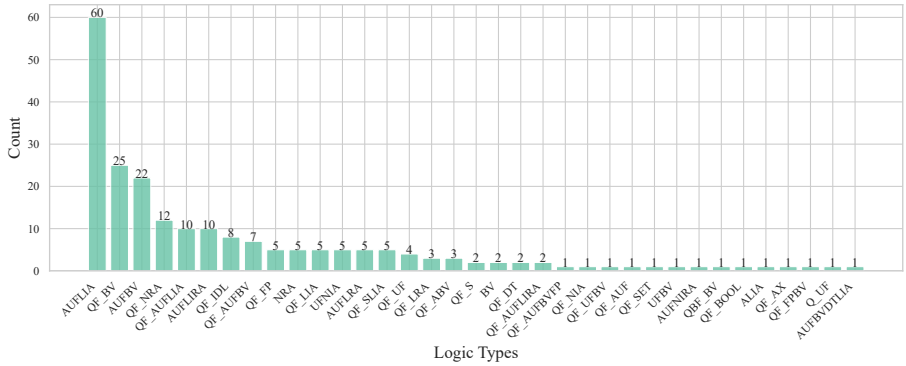


Fig. 2: The number of bug-triggering formulas from different theories.

CVC4 are the theory model builder, the quantifiers rewriter, and the sequences rewriter. These results provide insight into the solver components most susceptible to errors uncovered by Canary.

Summary. Our evaluation demonstrates that Canary is highly effective in testing SMT solvers. Here are the key highlights:

- Canary identified 123 bugs, with 108 confirmed and 107 already addressed.
- Bugs span a diverse set of SMT-LIB2 theories, including strings, (non-)linear arithmetic, bit-vectors, uninterpreted functions, floating-point operations, and combinations thereof.
- Many confirmed bugs were triggered in the solvers' default configurations, demonstrating the tool's ability to expose real-world failures without relying on exotic options.

4.2 Code Coverage

In addition to bug discovery, we evaluated Canary's ability to increase code coverage in SMT solvers.

Experimental Design. We used the seed formulas from HistFuzz, sampled from the SMT-LIB2 library, to represent a diverse range of theories and complexity levels. We generated 100 mutants for each seed using Canary's partition-based strategies. First, we ran the baseline fuzzers to collect their coverage results. Then, we layered Canary's partitioning techniques onto these fuzzers to measure the incremental improvement in coverage.

To ensure fair and reproducible comparisons, we set the solver timeout to two seconds per formula—enough time to process most formulas without excessive resource consumption. We compiled both Z3 and CVC4 in debug mode without optimizations to enable accurate coverage measurement. We used `gcov` to record three key coverage metrics: line coverage (l), function coverage (f), and branch

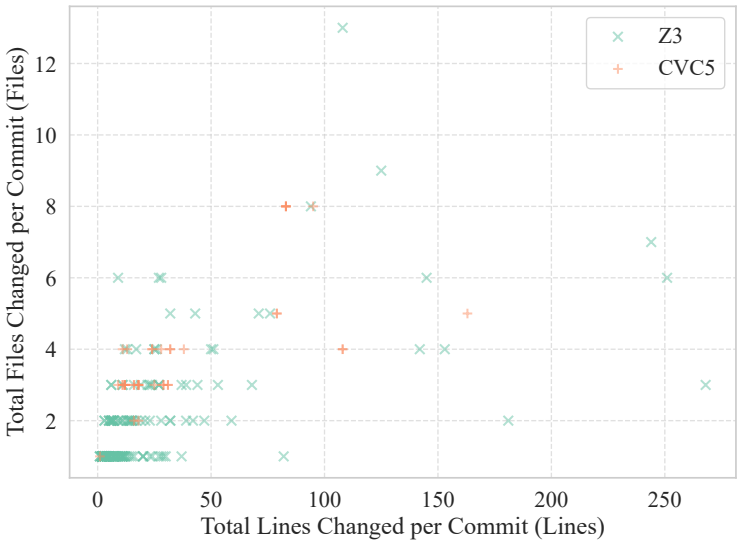


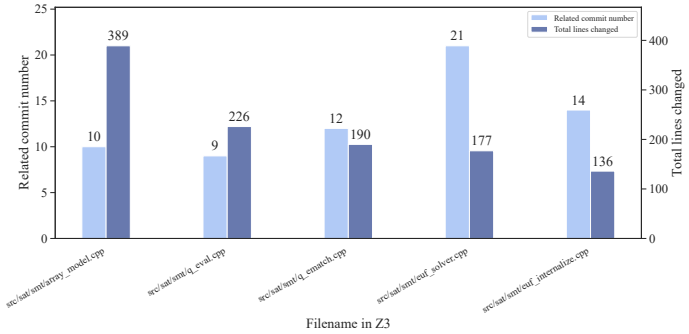
Fig. 3: Distribution of file changes and lines of code changes for Z3 and CVC4 bug-fixing commits.

coverage (b). Each experiment was run three times, and we report the average results to account for any variability.

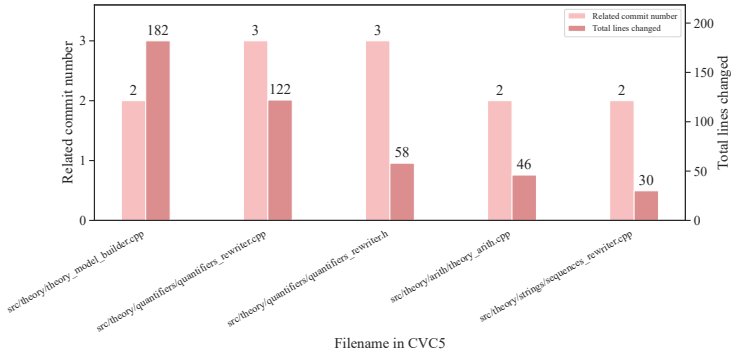
Integration with Other Fuzzers. To assess the compatibility and impact of Canary's core partitioning strategies when integrated with existing fuzzing frameworks, we combined Canary with two leading fuzzers, HistFuzz and Yinyang. Table 3 presents the resulting coverage metrics, illustrating the benefits of augmenting these state-of-the-art tools with our approach.

The results show that Canary's partitioning techniques are effective, modular enhancements to existing approaches. When integrated with other fuzzing strategies, our approach yields consistent improvements across all coverage metrics. For Z3, combining Canary with HistFuzz increased line coverage by 0.7 percentage points, function coverage by 0.7 percentage points, and branch coverage by 0.6 percentage points. Similarly, the Yinyang integration showed improvements of 0.6, 0.5, and 0.6 percentage points, respectively. For CVC4, both combinations demonstrated notable improvements, with HistFuzz+Canary achieving particularly strong results: 33.1% line coverage, 46.9% function coverage, and 26.9% branch coverage. The consistency of these improvements across both solvers indicates that Canary's partitioning strategy offers fundamental benefits that transcend specific solver implementations.

These results are particularly significant because they demonstrate that Canary can enhance even the most sophisticated existing fuzzers. The improve-



(a) Most frequently fixed files in Z3.



(b) Most frequently fixed files in CVC4.

Fig. 4: Number of commits and changed lines of Top 5 fixed files in Z3 and CVC4.

ments stem from Canary's ability to generate semantically diverse yet valid formulas that exercise different solver components through targeted partitioning.

Ablation Study. We conducted an ablation study to understand the contribution of each component of Canary to overall coverage improvement. We created three variants by selectively removing one core component at a time:

- Canary-NoCube: Excludes cube-based partitioning;
- Canary-NoDomain: Excludes numerical domain constraints;
- Canary-NoQuant: Excludes quantifier manipulations.

Table 4 presents the coverage results for each variant across different baseline fuzzers. The patterns revealed by this ablation study are nuanced and informative. While removing any component generally reduced overall coverage, certain variants performed better in specific contexts, indicating that the optimal configuration may depend on both the target solver and the baseline fuzzer.

For instance, for HistFuzz, Canary-NoQuant unexpectedly achieved better coverage compared to Canary-NoCube. This suggests that cube-based parti-

Table 3: Line coverage (l), function coverage (f), and branch coverage (b) results of using Canary to enhance existing fuzzers.

Tool	**Z3** (l/f/b)	**CVC4** (l/f/b)
Canary	24.4% / 27.0% / 20.6%	27.9% / 43.5% / 22.5%
HistFuzz	32.4% / 32.9% / 28.8%	32.1% / 46.2% / 25.9%
HistFuzz+Canary	33.1% / 33.6% / 29.4%	33.1% / 46.9% / 26.9%
Yinyang	27.0% / 28.8% / 24.0%	27.9% / 43.8% / 22.6%
Yinyang+Canary	27.6% / 29.3% / 24.6%	28.8% / 44.3% / 23.5%

Table 4: Line coverage (l), function coverage (f), and branch coverage (b) results for different variants of Canary.

Baseline	**Canary variant**	**Z3** (l/f/b)	**CVC4** (l/f/b)
HistFuzz	Canary-NoCube	33.1% / 33.6% / 29.6%	32.2% / 45.6% / 26.4%
	Canary-NoDomain	32.8% / 33.4% / 29.4%	32.4% / 45.9% / 26.5%
	Canary-NoQuant	33.3% / 33.8% / 29.8%	32.8% / 45.9% / 26.8%
Yinyang	Canary-NoCube	25.0% / 27.2% / 21.3%	28.5% / 43.7% / 23.3%
	Canary-NoDomain	27.9% / 29.5% / 24.6%	28.5% / 43.7% / 23.3%
	Canary-NoQuant	27.4% / 28.9% / 24.4%	28.0% / 43.5% / 22.6%

tioning may be more important than quantifier manipulations when starting from HistFuzz's mutation strategy. The most interesting variation appeared with Yinyang as the baseline. For Z3, Canary-NoQuant achieved notably higher coverage than Canary-NoCube, while for CVC4, the opposite was true—Canary-NoCube performed better than Canary-NoQuant.

These results have important implications for Canary's design and deployment. They suggest that: (1) No single component is universally dominant; the effectiveness depends on the context; (2) Different solvers benefit from different aspects of Canary's partitioning strategy; (3) There may be interaction effects between Canary's components and the baseline fuzzer's mutation strategy.

Summary. These results indicate that Canary can help improve code coverage across multiple dimensions—line, function, and branch—when used alone or in conjunction with existing fuzzers. While code coverage is a useful metric, it has its limitations. The relative coverage gains shown in our results may appear modest, as the baseline fuzzers are already effective at exploring many of the simpler execution paths in the solvers. However, the small percentage improvements still represent meaningful absolute increases in the number of lines, functions, and branches of code being tested. The coverage also provides a complementary set of stress-testing mechanisms that can uncover new and interesting solver behaviors.

Table 5: Comparison of bugs found by different tools within one week.

Tool	Z3			CVC4		
	Soundness	Invalid Model	Crash	Soundness	Invalid Model	Crash
HistFuzz	2	2	5	1	1	3
HistFuzz+Canary	4	2	6	3	1	4
Yinyang	4	0	3	2	1	1
Yinyang+Canary	4	1	5	3	1	3

4.3 Controlled Experiments for Bug Detection

To further evaluate the effectiveness of Canary, we conducted a head-to-head comparison with two state-of-the-art SMT solver fuzzing tools, including HistFuzz [28] and Yinyang [20].

Experimental Design. All tools were evaluated under identical experimental conditions: each was run for 1 week on the same hardware platform, using the same set of seed formulas. During this period, we systematically recorded both the number and types of unique bugs uncovered in Z3-4.8.7 and CVC4-1.8. The results of this comparative study are presented in Table 5.

Quantitative Analysis. The integration of Canary with baseline fuzzers shows consistent improvements. For HistFuzz, the combination with Canary increased the total number of bugs found from 14 to 20, with notable gains in soundness bugs and crash bugs. Similarly, for Yinyang, the integration improved the total bug count from 11 to 17, with significant improvements in crash bugs and the discovery of invalid model bugs.

Qualitative Analysis. The results reveal several important patterns. First, Canary demonstrates particular strength in detecting soundness bugs—the most critical type of correctness issues. When combined with HistFuzz, it doubled the number of soundness bugs found, and when combined with Yinyang, it maintained the high detection rate. This suggests that subspace diversification is particularly effective at uncovering deep semantic errors that affect solver correctness. Second, the integration shows consistent improvements in crash detection. Both HistFuzz+Canary and Yinyang+Canary found more crash bugs than their baseline counterparts, indicating that our partitioning strategies can expose execution paths that lead to unexpected termination. Third, the discovery of invalid model bugs by Yinyang+Canary — where the baseline Yinyang found none — demonstrates that Canary's approach can uncover bugs missed by existing techniques.

Cross-Solver Analysis. The results also show interesting patterns across different SMT solvers. For Z3, the improvements are more pronounced, with HistFuzz+Canary finding 12 bugs compared to HistFuzz's 9, and Yinyang+Canary finding 10 bugs compared to Yinyang's 7. For CVC4, the improvements are more modest but still consistent, with both combinations outperforming their baselines. This cross-solver analysis suggests that Canary's effectiveness may be

influenced by the specific characteristics of each solver's implementation. Z3's more complex architecture and broader theory support may provide more opportunities for subspace diversification to uncover bugs, while CVC4's more focused design may require more targeted partitioning strategies.

Complementarity Analysis. Notably, Canary can detect bugs that other state-of-the-art fuzzers miss, indicating that our approach can uncover unique issues missed by existing techniques. This confirms that subspace diversification is an effective strategy for uncovering deep and subtle semantic errors that are often overlooked by existing fuzzing techniques. The fact that different combinations (HistFuzz+Canary vs. Yinyang+Canary) find different sets of bugs suggests that Canary's partitioning strategies can adapt to and enhance different baseline approaches.

Summary. These results confirm that Canary is not only effective in finding a large number of bugs, but also excels at uncovering critical correctness issues in SMT solvers compared to state-of-the-art fuzzers under the same testing budget. We believe that combining complementary testing techniques within a unified system represents a promising direction. Such a system would leverage the strengths of individual methods while mitigating their weaknesses, yielding a more robust and comprehensive testing pipeline for SMT solvers.

4.4 Assorted Sample Bugs

In this section, we select and discuss six reported Z3 and CVC4 bugs. classifications and status.

Figure 5a shows a refutation soundness bug in Z3. The reason is that models are prematurely reported invalid when the EUF (Equality with Uninterpreted Functions) solver is active. The developers fixed the issue by deferring certain validation checks until after the EUF-specific model construction is completed.

Figure 5b shows a solution to a soundness bug in CVC4's quantifier instantiation engines. It is marked as "major", which is the highest severity in CVC4's bug tracking system. The bug is caused by allowing ineligible terms to appear in the instantiations.

Figure 5c shows a crash bug in CVC4 caused by an assertion that requires the model to always have a shared term for the real term in the conversion from real to floating-point value.

Figure 5d shows an invalid model bug in Z3's bit-vector solver. This bug is triggered by a corner case in the signed arithmetic solver: the input has a bit width of 1.

Figure 5e shows an invalid model bug in CVC4's ALIRA logic. This is because the sanity check for integer models in linear arithmetic was too strict when the linear solver had assigned a real value to an integer variable.

Figure 5f shows a crash bug in Z3's eager generation of axioms. This is due to the unconditional eager generation of axioms for arithmetic disequalities. The developers fixed the issue by controlling the generation of this axiom.

```
(declare-fun x () Real)
(assert (and (> 0.0 x) (= 0.0
    (/ 0.0 x))))
(check-sat-using (then
    add-bounds
    propagate-ineqs
    purify-arith uflra))
```

(a) A refutation soundness bug in Z3.

```
(declare-datatypes ((E 0))
    (((c (a Bool)))))
(assert (forall ((v E)) (and
    (a v))))
(check-sat)
```

(b) A solution soundness bug in CVC4.

```
(declare-const X (_
    FloatingPoint 8 24))
(declare-const R Real)
(assert (= X ((_ to_fp 8 24)
    RTZ (- R))))
(assert (= X ((_ to_fp 8 24)
    RTZ 0)))
(check-sat)
```

(c) A crash bug in CVC4.

```
(set-option :model_validate
    true)
(declare-fun bv_4-0 () (_
    BitVec 1))
(assert (not (bvsmul_noovfl
    bv_4-0 bv_4-0)))
(check-sat)
```

(d) An invalid model bug in Z3.

```
(set-logic ALIRA)
(declare-const x Real)
(declare-fun i () Int)
(declare-fun i1 () Int)
(push)
(assert (< 1 (- i)))
(check-sat)
(pop)
(push)
(assert (or (>= i1 (* 5 (- i)
    ))))
(check-sat)
(pop)
(assert (or (> i1 1) (= x (
    to_real i))))
(check-sat)
(assert (not (is_int x)))
(check-sat)
```

(e) An invalid model bug in CVC4

```
(set-option :smt.arith.
    eager_eq_axioms false)
(declare-fun z () Int)
(declare-fun y () Int)
(declare-fun x () Int)
(declare-fun named3 () Bool)
(declare-fun named5 () Bool)
(declare-fun named6 () Bool)
(declare-fun named7 () Bool)
(assert (and (= y (+ x 1)) (=
    1 (* z z))))
(assert (or named6 (not
    named3)))
(assert (or (not named5) (= y
    0)))
(assert (or named7 (= z y)))
(get-consequences (named5) (
    named3 named7))
```

(f) A crash bug in Z3.

Fig. 5: Sampled bugs detected by Canary.

4.5 Discussions

Limitations. While Canary has demonstrated effectiveness in uncovering bugs and improving solver coverage, several limitations merit discussion. First, the efficacy of partition-based mutation is theory-dependent; certain SMT fragments

may not benefit uniformly from the same mutation strategies. Second, our evaluation assumes deterministic solver behavior. Solvers employing randomized heuristics may exhibit variability across runs, complicating reproducibility and analysis. Third, as solvers evolve and adapt to current mutation patterns, the marginal utility of existing strategies may diminish, necessitating continual refinement of mutation operators to maintain effectiveness.

Adaptivity and Mutation Strategy Evolution. To sustain long-term effectiveness, mutation strategies must evolve. One avenue is to monitor metrics such as coverage growth or behavioral divergence over time; a plateau in these metrics may signal the need to revise the strategy. Canary's modular design facilitates the integration of adaptive mechanisms, such as online bandit algorithms that reallocate probability mass among operators based on empirical utility. Additionally, synthesizing new mutation operators—e.g., via SyGuS or learned models—offers a principled path to expanding the mutation space.

Program Analysis for Targeted Mutation. Incorporating program analysis—both static and dynamic—can potentially improve the relevance and effectiveness of mutations. Static analysis of seed formulas can uncover logical dependencies among variables or subformulas, enabling the generation of non-trivial subspace partitions, such as those that preclude trivially inconsistent cubes. Complementarily, dynamic analysis (e.g., runtime coverage profiling, SMT solver logs) can identify under-tested components of SMT solvers, guiding the mutation engine to generate inputs that stress these components, thereby improving bug exposure and coverage.

Applications Beyond Solver Testing. The partition-based mutation approach introduced in Canary has potential applications beyond SMT solver testing. Similar principles could be adapted to test other formal reasoning tools, such as theorem provers and program verifiers. Additionally, the generated diverse yet semantically connected formula sets could serve as benchmarks for evaluating solver performance or as training data for machine learning models to predict solver behavior [38]. The semantic partitioning approach might also inform strategies for distributed solving [32, 31], where formula space is partitioned intelligently among parallel solver instances.

5 Related Work

SMT Solver Testing. FuzzSMT [21] introduced grammar-based fuzzing to evaluate SMT solvers, marking one of the earliest efforts in this domain. Subsequent tools such as StringFuzz [22] and Winterer et al. [20] contributed a type-aware mutation strategy to improve the generation of diverse SMT formulas. However, these methods primarily rely on differential testing—comparing the outputs of multiple solvers—to detect inconsistencies. To address this issue, more recent approaches have focused on generating formulas with known satisfiability outcomes. Bugariu et al.[18] proposed constructing increasingly complex string formulas through satisfiability-preserving transformations. Winterer

et al.[17] introduced semantic fusion to obtain mutants of formulas whose satisfiability status remains unchanged. Existing mutation techniques either focus on localized syntactic changes or on changes that are vastly different from the seed, but the seed formula's original search space may not be thoroughly tested. Subspace diversification explicitly partitions a formula's solution space into disjoint regions using additional constraints.

Bounded-Exhaustive Testing. The principle behind bounded-exhaustive testing (BET) [39–42] is to systematically explore all inputs up to a predefined size or complexity. The underlying assumption is that many software defects are exposed by relatively small inputs, making exhaustive testing over these inputs an effective strategy for bug discovery. There are two prominent approaches in this space: declarative and imperative enumeration. Declarative methods [39] leverage logical invariants to constrain the space of valid inputs, while imperative strategies [43] construct inputs procedurally based on specific structural specifications. While BET offers thorough coverage, it faces challenges related to scalability and efficiency. To mitigate these issues, several optimization techniques have been proposed, such as sparse test generation [44] and structural test merging [45]. Our work draws inspiration from BET by employing small, bounded syntactical modifications. However, a key distinction is that our mutations also aim to constrain the search space semantically, guiding the generation towards more meaningful variations beyond purely structural exploration.

6 Conclusion

We have presented bounded-exhaustive subspace diversification, a principled approach to testing SMT solvers that targets underexplored regions of the solution space. Our implementation, Canary, uncovered 108 previously unknown bugs in Z3 and CVC4. These bugs span a wide range of bug types, input logics, and solver configurations, demonstrating the generality and effectiveness of our approach.

Acknowledgements

We would like to thank the reviewers for their helpful feedback. This work is supported by the National Key R&D Program of China (2023YFB3106000), the National Natural Science Foundation of China (62302434, U2341212, 62302442), and ZJU-China Unicom Digital Security Joint Laboratory. Peisen Yao is the corresponding author.

Bibliography

[1] Koushik Sen, Darko Marinov, and Gul 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*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM. doi: 10.1145/1081706.1081750.

[2] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. pages 322–335, 2006. doi: 10.1145/1180405.1180445.

[3] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1613–1627. IEEE, 2020. doi: 10.1109/SP40000.2020.00063.

[4] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti EJ Hyvärinen, and Natasha Sharygina. Hifrog: Smt-based function summarization for software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 207–213. Springer, 2017.

[5] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. Program analysis via efficient symbolic abstraction. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021.

[6] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *International Conference on Computer Aided Verification*, pages 36–52. Springer, 2013.

[7] Armando Solar-Lezama and Rastislav Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[8] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 643–659. USENIX Association, 2017.

[9] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM. doi: 10.1145/2884781.2884807.

[10] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[11] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 269–282. ACM, 2014. doi: 10.1145/2628136.2628161.

[12] A. Champion, T. Chiba, N. Kobayashi, and R. Sato. Ice-based refinement type discovery for higher-order functional programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10805 of *Lecture Notes in Computer Science*, pages 365–383. Springer, 2018. doi: 10.1007/978-3-319-89960-2_20.

[13] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM. doi: 10.1145/2568225.2568293.

[14] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. Investigating safety of a radiotherapy machine using system models with pluggable checkers. In *Computer Aided Verification (CAV 2016), Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 23–41. Springer, 2016. doi: 10.1007/978-3-319-41540-6_2.

[15] Byron Cook. Formal reasoning about the security of amazon web services. In *International Conference on Computer Aided Verification*, pages 38–47. Springer, 2018.

[16] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Daniel Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. Block public access: trust safety verification of access control policies. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 281–291. ACM, 2020. doi: 10.1145/3368089.3409728.

[17] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 718–730. ACM, 2020. doi: 10.1145/3385412.3385985. URL `https://doi.org/10.1145/3385412.3385985`.

[18] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1459–1470. ACM, 2020. doi: 10.1145/3377811.3380398. URL `https://doi.org/10.1145/3377811.3380398`.

[19] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1–12, New York, NY, USA, 2020. ACM. doi: 10.1145/3368089.3409736. URL `https://doi.org/10.1145/3368089.3409736`.

[20] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proc. ACM Program. Lang.*, 4(OOPSLA):193:1–193:25, 2020. doi: 10.1145/3428261. URL `https://doi.org/10.1145/3428261`.

[21] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, page 1–5, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584843. doi: 10.1145/1670412.1670413. URL `https://doi.org/10.1145/1670412.1670413`.

[22] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. Stringfuzz: A fuzzer for string solvers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Ox-*

ford, UK, July 14-17, 2018, Proceedings, Part II, volume 10982 of *Lecture Notes in Computer Science*, pages 45–51. Springer, 2018. doi: 10.1007/978-3-319-96142-2_6. URL https://doi.org/10.1007/978-3-319-96142-2_6.

[23] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing smt solvers via two-dimensional input space exploration. In *ISSTA'21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA*, 2021. doi: 10.1145/3460319.3464803. URL https://doi.org/10.1145/3460319.3464803.

[24] Joseph Scott, Federico Mora, and Vijay Ganesh. Banditfuzz: A reinforcement-learning based performance fuzzer for SMT solvers. In Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel, editors, *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*, volume 12549 of *Lecture Notes in Computer Science*, pages 68–86. Springer, 2020. doi: 10.1007/978-3-030-63618-0_5. URL https://doi.org/10.1007/978-3-030-63618-0_5.

[25] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Skeletal approximation enumeration for smt solver testing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021. Association for Computing Machinery, 2021.

[26] Dominik Winterer and Zhendong Su. Validating smt solvers for correctness and performance via grammar-based enumeration. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):355:1–355:24, 2024. doi: 10.1145/3689795.

[27] Jongwook Kim, Sunbeom So, and Hakjoo Oh. Diver: Oracle-guided smt solver testing with unrestricted random mutations. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*, pages 2224–2236. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00187. URL https://doi.org/10.1109/ICSE48619.2023.00187.

[28] Maolin Sun, Yibiao Yang, Ming Wen, Yongcong Wang, Yuming Zhou, and Hai Jin. Validating smt solvers via skeleton enumeration empowered by historical bug-triggering inputs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 69–81. IEEE, 2023. ISBN 9798350323701. doi: 10.1109/ICSE48619.2023.00020.

[29] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[30] Guangsheng Fan, Liqian Chen, Banghu Yin, Wenyu Zhang, Peisen Yao, and Ji Wang. Program analysis combining generalized bit-level and word-level abstractions. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '25. ACM, 2025.

[31] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference (HVC)*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2012. doi: 10.1007/978-3-642-34188-5_8.

[32] Peter van der Tak, Marijn J. H. Heule, and Armin Biere. Concurrent cube-and-conquer. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science*, pages 475–476. Springer, 2012. doi: 10.1007/978-3-642-31612-8_42.

[33] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Computer Aided Verification, 21st International Conference, CAV*, pages 306–320, 2009.

[34] K Rustan M Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *Computer Aided Verification: 28th International Conference (CAV'16)*.

[35] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). www. *SMT-LIB. org*, 15:18–52, 2010.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012.

[37] G Team. Gcov-using the gnu compiler collection (gcc). *Online, disponıvel em http://gcc. gnu. org/onlinedocs/gcc/Gcov. html-Ultimo acesso em*, 26(02):2015, 2014.

[38] Mislav Balunovic, Pavol Bielik, and Martin Vechev. Learning to solve smt formulas. In *Advances in Neural Information Processing Systems*, pages 10317–10328, 2018.

[39] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, pages 22–31. IEEE Computer Society, 2001. doi: 10.1109/ASE.2001.989792.

[40] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 133–142, 2004.

[41] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361, 2017.

[42] Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. Testmc: Testing model counters using differential and metamorphic testing. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 709–721. IEEE, 2020. doi: 10.1145/3324884.3416563.

[43] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133. ACM, 2002. doi: 10.1145/566172.566191.

[44] Yunho Kim and Shin Hong. Deminer: test generation for high test coverage through mutant exploration. *Software Testing, Verification and Reliability*, 31 (1-2):e1715, 2021. doi: 10.1002/stvr.1715.

[45] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003. doi: 10.1002/stvr.264.