Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration

Peisen Yao
The Hong Kong University of Science and Technology
Hong Kong, China
pyao@cse.ust.hk

Heqing Huang
The Hong Kong University of Science and Technology
Hong Kong, China
hhuangaz@cse.ust.hk

Wensheng Tang
The Hong Kong University of Science and Technology
Hong Kong, China
wtangae@cse.ust.hk

Qingkai Shi
The Hong Kong University of Science and Technology
Hong Kong, China
qshiaa@cse.ust.hk

Rongxin Wu
Xiamen University
China
wurongxin@xmu.edu.cn

Charles Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
charlesz@cse.ust.hk

ABSTRACT

Satisfiability Modulo Theories (SMT) solvers serve as the core engine of many techniques, such as symbolic execution. Therefore, ensuring the robustness and correctness of SMT solvers is critical. While fuzzing is an efficient and effective method for validating the quality of SMT solvers, we observe that prior fuzzing work only focused on generating various first-order formulas as the inputs but neglected the algorithmic configuration space of an SMT solver, which leads to under-reporting many deeply-hidden bugs. In this paper, we present Falcon, a fuzzing technique that explores both the formula space and the configuration space. Combining the two spaces significantly enlarges the search space and makes it challenging to detect bugs efficiently. We solve this problem by utilizing the correlations between the two spaces to reduce the search space, and introducing an adaptive mutation strategy to boost the search efficiency. During six months of extensive testing, Falcon finds 518 confirmed bugs in CVC4 and Z3, two state-of-the-art SMT solvers, 469 of which have already been fixed. Compared to two state-of-the-art fuzzers, Falcon detects 38 and 44 more bugs and improves the coverage by a large margin in 24 hours of testing.

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation*;

KEYWORDS

Fuzz testing, SMT solvers

ACM Reference Format:
Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464803

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464803

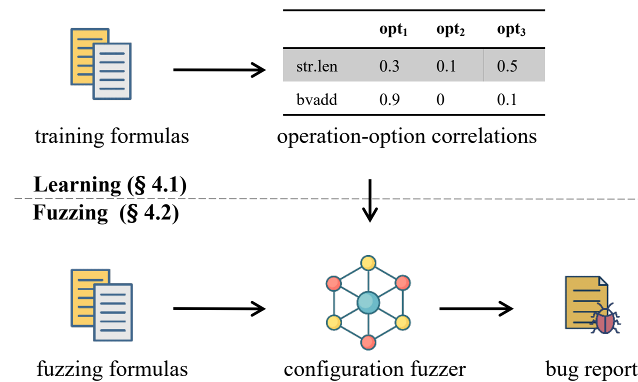
1 INTRODUCTION

Satisfiability Modulo Theories (SMT) solvers decide the satisfiability of formulas over some first-order theories, such as integers, reals, bit-vectors, strings, and their combinations [6, 15, 19, 22, 25, 30, 50]. Over the past two decades, SMT solvers have played critical roles in solving software engineering problems [9, 11, 17, 29, 31], such as finding zero-day software vulnerabilities via symbolic execution [3, 32], verifying the safety of radiotherapy machines [55], and enforcing the access control policies of Amazon Web Services [13, 21]. For example, to guarantee the security of a customer-facing service, it is reported that Amazon makes tens of millions of calls to SMT solvers per day [21].

Despite the tremendous research progress in SMT solving, many industrial-strength SMT solvers are still error-prone [16, 46, 66, 67]. For instance, in the last two years, thousands of issues have been reported, including crashes, correctness bugs, and others, for Z3 [22] and CVC4 [6], the two most prominent SMT solvers. Crash issues may lead to Denial-of-Service vulnerabilities, and correctness issues may lead to the escape of serious zero-day vulnerabilities from the inspection of code analyzers.

To ensure the quality of SMT solvers without diving into the details of their sophisticated internal logic, previous work has employed grammar-based black-box fuzzing to randomly generate syntactically valid SMT formulas [12, 14, 66]. BanditFuzz [60] extends this line of research by combining reinforcement learning for grammar-aware mutations. More recently, several techniques have emerged for detecting correctness bugs in SMT solvers [16, 46, 67]. These techniques generate formulas whose satisfiability is known by construction, and use such ground truth as the test oracle.

However, the above solutions still have many limitations. Specifically, in addition to the *formula space* that consists of first-order formulas, a critical space, referred to as the *algorithmic configuration space*, has been less-explored, which leads to under-reporting many deeply-hidden bugs in an SMT solver. The configuration space is critical because it determines the functionalities used by the



	opt_1	opt_2	opt_3
str.len	0.3	0.1	0.5
bvadd	0.9	0	0.1

Figure 1: Overview of Falcon workflow.

SMT solver among its complex and enormous algorithmic components [23]. If the configuration space is not taken into consideration, a fuzzer for testing SMT solvers has at least two weaknesses. First, the results of code coverage suffer, because many algorithmic components and their combinations may not be tested at all. Second, the fuzzer will miss bugs, as many bugs can only be discovered by using some specific formulas together with some specific solver configurations.

Unfortunately, extending existing fuzzers to explore both the formula space and the algorithmic configuration space would be stunningly challenging. Consider that the formula space has already been untraceable because of the semantic richness of the first-order formulas. The configuration space additionally creates a large space orthogonal to the formula space. For example, Z3 exposes more than 500 solver options, which result in millions of configurations by combining these options in different manners. Consequently, the space enlargement dramatically increases the search efforts, which can result in a sharp reduction in fuzzing efficiency.

Our idea to mitigate this space explosion issue is based on the observation that, many of the solver options (the configuration space) are often correlated with the operations in an SMT formula (the formula space). For example, the solver option `str.regex_automata` of Z3 only works for string formulas with regex expression operations and, thus, has no effects on integer or real formulas. While an SMT solver's configuration space is innately huge, the search space for the individual formula can be significantly reduced, if the correlation between the formula space and the configuration space can be automatically identified.

Based on the observation, we present Falcon, a feedback-driven fuzzing technique, which explores both the formula and the configuration space of an SMT solver. At a high level, Falcon works in two phases, as shown in Figure 1.

- The *learning phase* infers the correlations between operations in SMT formulas and the options of SMT solvers. Such information can be preserved to identify the relevant solver options for unseen formulas.
- The *fuzzing phase* utilizes a grammar-based generative fuzzer to sample diverse SMT formulas. Further, for each generated formula, Falcon explores the configuration space by adaptively mutating the solver options. Specifically, we utilize the pre-inferred correlation information to identify the relevant options, and design a feedback-driven mechanism for mutating the options.

During the six months of extensive testing, Falcon detects 105 and 413 confirmed bugs in CVC4 and Z3, respectively. These reported bugs cover correctness bugs, crash bugs, and performance bugs, demonstrating the promising capability of bug detection. By the time of writing, 469 of the bugs had been fixed, and our bug-revealing test cases had been added to the official regression test suites of the solvers. To understand the differences with other fuzzers, we also compare our approach to state-of-the-art tools, namely Storm [46] and OpFuzz [66]. In 24 hours of testing, Falcon detects 38 and 44 more bugs respectively, and, on average, improves the line coverage by 24.0% and 17.2% respectively. Besides, we apply Falcon to enhance Storm and OpFuzz by better exploring the configuration space, and demonstrate that Falcon can improve the code coverage and bug finding capability of the two fuzzers significantly.

In summary, we make the following contributions:

- We design and implement Falcon, a fuzzing framework that effectively explores the combined formula-configuration search space for fuzzing SMT solvers.
- We propose a method for learning the correlations between the formula space and the configuration space, and present a feedback-driven mechanism that significantly improves the fuzzing efficiency.
- We conduct a comprehensive evaluation, which demonstrates Falcon's capability of bug detection for different SMT solvers, and shows its outperformance over other existing fuzzers.

2 PRELIMINARIES

In this section, we present the basic notations and terminologies throughout the paper.

2.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula over some first-order theories. Examples include the theories of integer and real arithmetics, uninterpreted functions, bit-vectors, arrays, strings, and many more. In this paper, we assume the use of the SMT-LIB2 language [7], a standard for SMT formulas, which formally specifies the input format for theories that attract enough interest in the research community.

Example 2.1. Figure 2 shows a formula specified in the SMT-LIB2 language. The first line declares the logic type of the formula (QF_LIA), which stands for quantifier-free linear integer arithmetic. The next two lines declare the variables, following which each line starting with "assert" is a constraint. Multiple asserts can be viewed as the conjunction of the constraints in each individual assert statement. The command "check-sat" instructs the solver to decide the satisfiability of the constraints. Finally, the command "get-unsat-core" requests for an unsat core, which is a subset of constraints that introduce unsatisfiability. [1]

[1]For example, the unsat core of the formula in Figure 2 is $\{x > 6, x < 4\}$. Usually, SMT solvers do not search for the unsat core by default.

```
1 (set-logic QF_LIA)
2 (declare-const x Int)
3 (declare-const y Int)
4 (assert (> x 6))
5 (assert (= y 3))
6 (assert (< x 4))
7 (check-sat)
8 (get-unsat-core)
```

Figure 2: An integer formula in the SMT-LIB2 language.

Table 1: A selected list of options in Z3. Each option has a default value, as well as two or more values to choose from.

Option	Type	Values	Default
rewriter.elim_and	Bool	{false, true}	false
smt.string_solver	String	{"seq", "z3str3"}	"seq"
smt.arith.solver	Int	{1, 2, . . . , 6}	6
sat.restart_factor	Double	[−∞, +∞]	1.5

In the remainder of the paper, we term the collection of all function symbols (e.g., integer addition), predicate symbols (e.g., <, =, >), Boolean connectives, and the set of commands { "get-model", "get-unsat-core", "get-unsat-assumptions", "get-proof", "push", "pop" } as the **operations** of an SMT formula.

2.2 Configuration Space of SMT Solvers

Modern SMT solvers are complex software systems with various algorithmic components, which either simplify a formula into a suitable representation or apply decision procedures to check for satisfiability. Simplifications like constant folding (e.g., $x + 0 \rightarrow x$) and bit-blasting [35] can put the problem in a form that is easier or suitable for solving. Decision procedures such as Simplex method [26] and CDCL algorithm [62] can solve linear arithmetic constraints and Boolean constraints, respectively.

SMT solvers often expose a collection of **options** to control certain behavioral aspects of their algorithmic components. The configuration of the solver is a subset of the options that are responsible for a user's preferences. It is often the case that an algorithmic component only gets executed when some options are set to specific values. For example, Table 1 lists a sampled list of options in the Z3 SMT solver. Consider the option `smt.string_solver`, which stands for the engine of string constraint solving. Z3 has two engines, namely seq [10] and z3str3 [8], which can be complementary in handling formulas of different characteristics.

As the use of SMT solvers continues to grow and diversify, the number of solver options has steadily risen in most solvers. For instance, the latest versions of Z3 and CVC4 expose 574 and 457 options, respectively. Thus, the configuration space of an SMT solver can be enormous. Note that, many of these options have been requested by end-users, who "*have a wide-spread wish for more methods to exert strategic control over the solver's reasoning*" [23].

2.3 Grammar-based Generative Fuzzing

Grammar-based generative fuzzers take a grammar as their input and generate test cases based on the grammar.

			$\theta(r)$	Meaning
S	$\rightarrow$	`arg` | `""`	0.006	string variables, literals
	|	(`str.replace` $S\ S\ S$)	0.012	string replacement
	|	(`str.++` $S\ S$)	0.231	string concatenation
	|	. . .	. . .	. . .

Figure 3: A simple PCFG for string expressions, where each production rule r has a probability $\theta(r)$.

Definition 2.1. A context-free grammar (CFG) is a quadruple $\mathcal{G} = (N, \Sigma, P, S)$, where N is a set of non-terminals, Σ is a set of terminals, $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of production rules, and $S \in N$ is the start symbol.

Suppose that we are given a language and its context-free grammar. To generate programs of the language, it is natural to walk over the grammar's productions, which typically starts from a unique start symbol and proceeds by applying left-right production rules from the grammar [34, 47, 58, 63].

Definition 2.2. A probabilistic context-free grammar (PCFG) is defined as a pair $(\mathcal{G}, \theta)$, where $\mathcal{G}$ is a context-free grammar, and $\theta : P \mapsto \mathbb{R}^+$ is a function, such that $\theta(r)$ is the production probability associated with a production rule $r \in P$.

Intuitively, the probabilities determine the relative occurrences of the constructs that appear in the grammar. Consequently, a PCFG defines a probability distribution on programs that can be generated from the grammar. For example, Figure 3 depicts a PCFG for string expressions, which is biased towards the string concatenation operation "str.++". In this paper, we leverage a probabilistic grammar model for the SMT-LIB2 language. We will detail its usage in the following sections.

3 OVERVIEW

In this section, we motivate the problem of exploring the combined formula-configuration search space, highlight its challenges, and sketch our solutions.

3.1 Motivation

An ideal fuzzer is desired to thoroughly exercise the algorithmic components of an SMT solver under test. To this end, as illustrated in § 2.2, it is often necessary to supply the solver with specific options to control the behavioral aspects of different algorithmic components. Thus, augmenting the fuzzing search space to include the SMT solver's configuration space has great potential for improving the fuzzing capability.

Intuitively, given an input formula and a solver under test, the fuzzer can try different solver configurations by mutating the solver's options. Unfortunately, randomly mutating the vast number of options is often ineffective, for the following three reasons:

- *Conflicting mutation*: The solver rejects the formula when some options are mutated improperly. For instance, if a formula contains the command "get-unsat-core", the option `produce-unsat-cores` in CVC4 should be set to true; otherwise, the formulas would fail to pass the option checking phase of CVC4.

- *Invalid mutation*: For the formula, different values of an option have no effects on the behavior of the solver. For example, the Z3 option `smt.str_theory` in Table 1 only controls the decision procedures for string formulas and, thus, does not affect real or bit-vector formulas.
- *Ineffective mutation*: An option can hold many different values, and it is possible that only under certain values can it trigger bugs in the solver. For example, the Z3 option `sat.restart_factor` in Table 1 is double-typed and has infinite values.

As a result, blindly exploring an SMT solver's configuration space can result in a sharp reduction in fuzzing efficiency and significantly discount the effectiveness.

3.2 Challenges and Approaches

At first glance, Falcon is a grammar-based generative fuzzer: it generates SMT formulas using a context-free grammar. For a generated formula, Falcon further attempts to explore the configuration space of a solver by *mutating the solver options*. The goal here is to exercise the relevant algorithmic components of the solver and find bug-revealing formula-configuration pairs. To improve the effectiveness of the mutation, our key idea is to exploit the semantic correlations between the formula operations and the solver options, as exemplified in § 3.1.

However, there are two major challenges for inferring and utilizing the operation-option correlations.

- *No prior domain knowledge*. First, we need to identify the options correlated to a formula, to reduce the number of conflicting or invalid mutations. Manually tracking the correlations between formula operations and solver options is tedious and often ineffective, because the number of operations and options can be enormous.
- *Mutation strategies for options*. Second, we need to fuzz the configuration space by mutating the correlated solver options, each of which can have many values to choose from. The values can be either discrete (e.g., {1, 2, …, 6 }) or continuous (e.g., $[-\infty, +\infty]$).

Falcon addresses the above challenges in two phases, as illustrated in Figure 1.

- *Operation-Option Correlation Learning (§ 4.1)*. The first phase uses a data-driven approach to infer the correlations between formula operations and solver options. The underlying intuition is: when solving the same formula, if different values of an option behave differently (e.g., in terms of the solving time, satisfiability results, or executed code), then the option is likely to be correlated with the operations in the formula. Our key idea is to iteratively sample formulas and solver configurations, during which we aggregate the evidence of correlations. As shown in Figure 1, the learned correlations can be preserved to handle *unseen* SMT formulas.
- *Adaptive Formula-Configuration Fuzzing (§ 4.2)*. The second phase aims to generate diverse formula-configuration pairs to test SMT solvers. Specifically, we present an approach that effectively mutates the solver options for the generated formulas. First, using the correlations learned in the first phase, Falcon can prioritize the set of options that are correlated to a formula. Second, by incorporating coverage information as the feedback, Falcon adaptively mutates the correlated solver options, further improving effectiveness.

4 METHODOLOGY

In this section, we explain the technical details of Falcon. We first present the procedure for learning the operation-option correlation. We then describe the key fuzzing engine that adaptively generates formula-configuration pairs to stress-test SMT solvers.

4.1 Operation-Option Correlation Learning

We first formally define the concept of operation-option correlation, and then detail the types of correlations. Following that, we present a random simulation based approach for automatically inferring the correlations. Finally, we illustrate the use of the correlations for fuzzing SMT solvers.

Defining the Operation-Option Correlations. To identify the correlations among formula operations and solver options, we need a measurement to examine the effects of an option. To this end, we define the *observable output* of a solver as any quantitative and measurable information when solving an SMT formula. Specifically, we use the solving time, satisfiability results, and the SMT solver's code coverage as the observable outputs, because intuitively, they reflect the differences of the solver configurations. Then, we can define the concept of an operation-option correlation as below.

Definition 4.1. Let $S = \{s_1, s_2, \ldots, s_m\}$ and $O = \{o_1, o_2, \ldots, o_n\}$ be two sets of formula operations and solver options, respectively. We say that o_i correlates with s_j, if when solving formulas containing s_j, different values of o_i have a high probability of resulting in different observable outputs.

As illustrated in § 2, SMT solver options have diverse functionalities, which can control the behavior aspects of the simplifications and decision procedures. We observe that the options are typically correlated with the syntactic features of the formulas. More concretely, we differentiate three categories of correlations between a pair (s_j, o_i) of operation s_j and option o_i:

- *Conflict*: Mutating the option can lead to the rejection of a formula. For example, if a formula contains the "push/pop" commands, then the option `sygus-rr` in CVC4 should not be set to true, because otherwise, the formula would fail to pass the option checking phase of the solver.
- *Irrelevancy*: The option does not affect the observable outputs. For example, the option `rewriter.elim_rem` in Z3 has no effects on formulas without "rem" function symbols, because it controls a simplification algorithm that works only for the "rem" function.
- *Relevancy*: The option can affect the observable outputs. For example, if there are "get-unsat-core" commands in a formula, then the option `smt.core.minimize` in Z3 is relevant to the formula, because the option controls the algorithms for generating unsat cores. [2]

In practice, we only adopt the "Relevancy" among the three kinds, because "Conflict" options cannot pass the option checking, i.e.,

[2] If `smt.core.minimize` is true, Z3 will attempt to generate the minimal unsat core.

Algorithm 1: Random simulation for learning the operation-option correlations.

Input: Formula operations $S = \{s_1, s_2, \ldots, s_m\}$ and solver options $O = \{o_1, o_2, \ldots, o_n\}$
Output: Probabilities of correlations between the operations in S and the options in O

```
1  ∀(s_i ∈ S ∧ o_j ∈ O). p_(s_i,o_j) ← 0;
2  while time limit is not reached do
3      S_k ← select k elements from S;     /* restrict the
       operation kinds in the tranining formulas */
4      Π_{S_k} ← generate a set of formulas that use only S_k;
5      foreach formula φ ∈ Π_{S_k} do
6          foreach option o ∈ O do
7              solve φ by changing the values of o;
8              if o results in no conflict and affects the observable
               outputs then
9                  foreach operation s ∈ S_k do
10                     increase the value of p_(s,o);
```

the SMT solving results change from "sat/unsat" to "error". Besides, irrelevant options make no contributions to the exploration of the wider search space, as it does not affect the internal solving process.

Learning the Operation-Option Correlations. Identifying the correlations requires domain knowledge about the SMT solvers' implementations. A user could read the source code of an SMT solver to try to understand what solver options and formula operations can be correlated. However, the manual approach could require a significant investment of time.

To automatize this process, we utilize the idea of random simulation [45] to infer the operation-option correlations. Our intuition is: given the same SMT formula, if different values of an option cause the solver to behave differently, then the option is likely to correlate with the operations in the formula. Therefore, the overall method is to repeatedly sample SMT formulas with restricted formula operations, solve them under different configurations, and gather evidence of the correlations.

Algorithm 1 outlines the procedure, which does not assume any prior knowledge about the solver implementations. The algorithm tracks a probability of correlation for each operation-option pair (s_i, o_j), denoted as $p_{(s_i,o_j)}$. Initially, we assume there is no prior knowledge of the correlations, so we set each probability to 0 (line 1). Next, we attempt to learn the correlations directly from the solving process feedback. In each round of the iteration, we first generate a group of training formulas, each of which has only k kinds of operations (lines 3-4). We then hand each formula to the SMT solver, which solves the formula under different configurations. If the outputs contain option conflicts, we do not increase the probabilities of correlations. Once the observable output emerges,[3] the probabilities of correlations are updated for each formula (line 10).

[3]The differences in satisfiability results and code coverage can be measured easily. For the solving time, we solve each formula 20 times per configuration, and conduct the Mann-Whitney U-test to examine whether the solving time of two configurations significantly differs (with p-values <0.01)

```
(set-logic QF_NIA)
(declare-const x Int)
(declare-const y Int)
(assert (> x 6))
(push 1)
(check-sat)
(get-unsat-core)
```

(a) An integer formula.

	rwriter. elim_rem	smt.core. minimize	blast_ euality
push	0	0	0
get-unsat-core	0	1	0
rem	1	0	0

(b) Pre-inferred correlations.

Figure 4: An example of using pre-inferred correlations to process an unseen formula.

There are two noteworthy points about Algorithm 1. First, since different formula operations may interfere with each other when learning the correlations, we restrict the number of operation kinds (k in the algorithm) in the training formulas (line 3).[4] This is done by tuning the PCFG-based formula generator (§ 2.3), i.e., setting the probabilities of some production rules to zero. Second, when changing the solver configurations for a formula, we only mutate one option (line 7). This is because our goal is to learn the correlations in Definition. 4.1, which concerns the pairing of a formula operation and a solver option.

Example 4.1. In Z3, the option `blast_equality` is irrelevant to formulas containing "push/pop" operations, because (1) the presence of "push/pop" commands means that the solver is in the incremental solving mode, and (2) the incremental solver of Z3 never calls the bit-blasting procedure [35], which is affected by `blast_equality`. Therefore, when we generate a set of SMT formulas that contain "push/pop", different values of the option are very likely to have no effects on the observable outputs. Consequently, the probabilities $p_{(push,\texttt{blast_equality})}$ and $p_{(pop,\texttt{blast_equality})}$ are not increased by Algorithm 1.

Implications for Fuzzing. The correlation information has several benefits for the subsequent fuzzing process. Specifically, it enables the fuzzer to exploit the semantic information of an *unseen* formula when knowing only its syntactic features. More concretely, given an SMT formula, we first identify the correlated solver options by leveraging the pre-inferred operation-option correlations. We can then explore the algorithmic components related to the formula, by mutating the relevant solver options.

Example 4.2. Suppose the fuzzer generates a formula shown in Figure 4(a). Instead of just feeding the formula to an SMT solver under test, we would like to explore the configuration space of the solver by mutating its options, in an attempt to exercise more algorithmic components with the same formula.

First, we compute the features of the formula as the set of operations it contains, which is { ">", "push", "get-unsat-core" }. Second, we combine the features and the pre-inferred correlations (shown in Figure 4(b)) to obtain the formula's relevant options. Specifically, we have (1) the option `rewriter.elim_rem` is irrelevant since the formula contains no "rem" functions, (2) the option `blast_equality` is irrelevant since "push" is in the set of operations, and (3) the option `smt.core.minimize` is relevant because of

[4]In the implementation, we set k to 4 based on our experience.

the "get-unsat-core" command. Therefore, we can mutate the value of `smt.core.minimize` to obtain different solver configurations. As a result, we reduce the configuration space with respect to a particular formula.

4.2 Adaptive Formula-Configuration Fuzzing

With the inferred operation-option correlations, our fuzzing engine consists of two sub-components, which generates SMT formulas and explores solver configurations, respectively.

Overview of Falcon. Our fuzzer can find the following three types of bugs in SMT solvers:

- *Correctness bugs*: By "correctness", we refer to three kinds of bugs in a solver's results:
  1. Refutation correctness: a formula is satisfiable, but the solver yields "unsat";
  2. Solution correctness: a formula is unsatisfiable, but the solver yields "sat";
  3. Invalid model : a formula is satisfiable and the solver yields "sat", but the solver gives an infeasible model that falsifies the formula. [5]
- *Crash bugs*: The solver terminates abnormally when solving a formula, which can be caused by assertion failures or memory safety problems such as buffer overflow.
- *Performance bugs*: The solver cannot terminate on a formula for a long time. We regard such cases as bugs only when the developers confirm that implementation issues cause them.

Algorithm 2 presents the general fuzzing process. The three sets *soundbugs*, *invalid_models*, and *crashes* are used to collect soundness, invalid model, and crash bugs, respectively (line 1). All of them are initialized to the empty set.

In each round of the main loop, we first generate a formula φ using a context-free grammar, and then hand it to the SMT solver (lines 3-4). We then leverage the inferred operation-option correlations to identify the correlated solver options for the formula (line 8). Based on the correlation information, we run the configuration fuzzer to mutate the solver options (line 11), and instruct the solver to solve φ under a new configuration (line 12). Then, we check whether the solver gives a consistent answer or not. If not, we have observed a soundness bug (line 14). If the answer is consistent and "sat", but the solver returns a model that falsifies φ', we have found an invalid model bug (line 18). Finally, if the solver crashes on the seed or the mutant, we have found a candidate crash bug (line 6 and line 20).

In what follows, we first briefly describe the formula generator. We then focus on presenting the configuration fuzzer, including the termination condition (line 10) and the concrete strategies for mutating solver options (line 11).

Sampling SMT Formulas. Similar to conventional generative fuzzers for structural inputs [28, 38, 43, 63], we can generate SMT formulas from scratch, by using the context-free grammar for the SMT-LIB2 language. However, without an extra mechanism to control the generation strategy, the exhaustive test production is explosive, and the testing coverage is often unbalanced [34, 47].

[5]A model for a formula φ is a function that maps all variables in $vars(\varphi)$ to values in their respective domains such that φ evaluates to `true`.

Algorithm 2: Main procedure of the fuzzing phase.

```
Input: An SMT solver under test and a grammar G
Output: The set of candidate bugs
1  crashes ← ∅, soundbugs ← ∅, invalid_models ← ∅;
2  while time limit is not reached do
3      φ ← generate a formula with the grammar G;
4      r ← solve φ using the SMT solver;
5      if r == "crash" then
6          crashes ← crashes ∪ {φ};
7          continue;
8      opts ← identify the correlated solver options for φ;
9      /* start mutating the solver options for φ   */
10     while no termination criterion met do
11         c ← a solver configuration by mutating opts;
12         r_c ← solve φ under the configuration c;
13         if r_c ≠ r then
14             soundbugs ← soundbugs ∪ {(φ, c)};
15         else if r_c == "sat" then
16             M ← a model returned by S;
17             if M does not satisfy φ then
18                 invaid_models ← invaid_models ∪ {φ′};
19         else if r_c == "crash" then
20             crashes ← crashes ∪ {(φ, c)};
```

To better explore the formula space, we utilize a probabilistic context-free grammar (PCFG) model (§ 2.3). The probabilities determine the relative occurrences of the constructs that appear in the grammar. Given pre-assigned probabilities, our algorithm enforces the sampled SMT formulas to be uniformly distributed over the formula space described by the grammar. Thus, diversifying the probabilities enables us to generate formulas with diverse syntactic features. Following the idea of swarm testing [18, 33], [6] we randomly assign several groups of probabilities, which naturally yield different variants of PCFGs, and then sample SMT formulas from the grammars.

Fuzzing Solver Configurations. As illustrated in § 3.1, in the interests of maximizing coverage and fault detection, we propose to augment the fuzzing search space to include SMT solvers' configuration space. Prior work [12, 14, 16, 46, 60, 67] has not emphasized the exploration of the configuration space, thus losing many optimization opportunities. Specifically, given a formula, our configuration fuzzer is responsible for searching solver configurations by mutating a vast set of options.

First, as discussed in § 4.1 and illustrated in Algorithm 2, we utilize the learned operation-option correlations to identify the conflicting and irrelevant options. If the number of those options are large enough, the correlation information can produce a dramatic size reduction of the search space of the solver configurations.

[6] In Swarm testing, a test configuration determines the behaviors of the input generator, such as the generation probability of various kinds of statements for a C program generator. Swarm testing aims to generate diverse test cases by randomizing configurations of the input generator.

Second, it remains vital to mutate the values of the relevant options, each of which can have two or more values. To this end, we adopt a genetic algorithm to optimize the mutation strategies. The overall idea is to incorporate branch coverage as feedback to guide the evolutionary search. Algorithm 3 outlines the procedure, which randomly initializes the option value vectors V_0. The algorithm terminates when the code coverage remains unchanged during ten consecutive iterations, or the number of offsprings reaches an upper bound (100) (line 3). In what follows, we detail the genetic operators. For ease of presentation, we assume that the relevant options are specified in Table 1.

Search Space. The search space is represented by the possible values of the relevant solver options. As exemplified in Table 1, the option values can be in discrete or continuous space. For instance, the fourth option `sat.restart.refactor` has a default value 1.5, yet it accepts floating-point numbers with infinite ranges. Our goal is to better mutate the option value vector $\boldsymbol{v} = \{v_1, v_2, v_3, v_4\}$, by attempting to cover more branches. Accordingly, the algorithmic components in the SMT solver can be explored more thoroughly.

Fitness. The fitness in our setting is to evaluate the possibility of options' values in exploring more algorithmic components. To this end, we adopt the branch coverage as the fitness function. Algorithm 3 runs such a fitness function for each configuration associated with their values (line 4). After that, the n top configurations with the highest fitness are chosen for producing the next generation (line 5). This process ensures that the configurations selected by the fitness can produce better coverage results.

Crossover Operator. The crossover operator is a vital means that generates offsprings by composing parent solutions. In our setting, the solution is a vector of option values. Therefore, the crossover operator has to generate two offsprings $\boldsymbol{p}$ and $\boldsymbol{q}$ by shuffling the values in parent solutions $\boldsymbol{u}$ and $\boldsymbol{v}$ (line 7). This step first generates a random crossover point μ to split the parent solutions for crossover. Then, the first offspring $\boldsymbol{p}$ inherits the first μ option values from parent $\boldsymbol{u}$, while the rest are inherited from parent $\boldsymbol{v}$. On the contrary, the second offspring inherits the leftover of what the first offspring inherits from the two parents. For example, we choose a separator between the 2nd and the 3th options. Let two parents be $\{u_1, u_2, u_3, u_4\}$ and $\{v_1, v_2, v_3, v_4\}$, two offspring value vectors $\{u_1, u_2, v_3, v_4\}$ and $\{v_1, v_2, u_3, u_4\}$ are generated.

Mutation Operator. When a new offspring $\boldsymbol{v}$ is generated, there is a probability $1/|\boldsymbol{v}|$ for each option value v_j to be mutated (line 8). Options are mutated by their types: booleans are mutated by negating their values, e.g., from true to false; integers are mutated by adding a random delta value; floating-point numbers are mutated using polynomial mutation [24], the standard real number mutation technique; strings are mutated by shuffling the list of available values for that option, e.g., from "seq" to "z3str3" for the Z3 option `smt.string_solver`.

5 IMPLEMENTATION

We have implemented Falcon in a total of 13,415 lines of Python code. Falcon can be used with any SMT solver that takes as input the SMT-LIB2 formulas, and it supports most of the first-order theories within the SMT-LIB 2.6 standard [7].

Algorithm 3: Genetic algorithm for optimizing the strategy of mutating solver options.

Input: Initial option value vectors V_0
Output: Optimized option value vectors V_k

1 $i \leftarrow 0$; // i^{th} generation;
2 $V_i \leftarrow$ initial option value vectors;
3 **while** *no termination criterion met* **do**
4   fitness(V_i);
5   *values* ← top n option value vectors with highest fitness;
6   **foreach** $\boldsymbol{u}, \boldsymbol{v} \in values$ **do**
7         $\boldsymbol{p}, \boldsymbol{q} \leftarrow$ crossover($\boldsymbol{u}, \boldsymbol{v}$);
8         $\boldsymbol{p}', \boldsymbol{q}' \leftarrow$ mutate($\boldsymbol{p}, \boldsymbol{q}$);
9         $V_{i+1} \leftarrow V_{i+1} \cup \{\boldsymbol{p}', \boldsymbol{q}'\}$;

Formula Generation. Falcon can generate syntactically valid SMT formulas from scratch, by using a probabilistic context-free grammar (§ 2.3). Recall that both the learning phase (§ 4.1) and the fuzzing phase (§ 4.2) need to generate formulas. In the learning phase, each formula should have a small number of operation kinds, to reduce the interference among different operations. This is done by setting some probabilities in the grammar to zero. In the fuzzing phase, we do not restrict operation kinds in the formulas, but attempt to test different operation combinations. Thus, we use multiple groups of probabilities to generate formulas with diversified syntactic features.

Test Case Reduction. When a bug is revealed, we need to reduce the size of the bug-triggering formula by removing the bug-irrelevant fragments. A reduced version of the formula can help the developers diagnose the bug. To automate the test case reduction, we implement a test case minimizer on top of ddSMT [49], a delta debugger tailored for the SMT-LIB2 language. ddSMT was introduced in 2013 and is only compatible with limited theories. We have extended ddSMT to support more theories, such as floating points, strings, sets, recursive functions, and abstract data types, to name just a few.

Testing Process. Our testing process is automated and runs continuously, which involves little human intervention. At each iteration, Falcon tests SMT solvers by randomly generating an SMT formula and mutating solver options for the formula. The manual effort mainly comes from confirming that a reduced SMT formula is valid, and checking whether the new bug is duplicate to any existing bugs in the solvers' GitHub issue trackers.

6 EVALUATION

In this section, we design a series of experiments to investigate the following research questions:

- **RQ1**: How effective is Falcon in detecting bugs in state-the-art SMT solvers?
- **RQ2**: How precise are the operation-option correlations learned by Falcon?
- **RQ3**: How do the key components of Falcon contribute to its effectiveness?

- **RQ4**: How does Falcon compare against existing fuzzers? Can Falcon enhance existing fuzzers, by utilizing the learned operation-option correlations and the mutation strategies for solver options?

6.1 Experimental Setup

Targeted Solvers. We select CVC4 [6] and Z3 [22] for the experimental evaluation, which have 457 and 574 options, respectively. We choose the solvers by following four criteria. First, they are widely used in academia and/or industry. Second, they support most of the features and logics in the SMT-LIB2 standard and have state-of-the-art performance. Third, they have open issue trackers on GitHub, and their developers are responsive to bug reports. Fourth, they are mature and have been extensively tested by previous efforts [12, 14, 46, 60, 66, 67].

We detect candidate soundness bugs via differential testing: If different configurations of a solver yield different solving results for the same formula, then the solver may have a bug (lines 13-14, Algorithm 2). We find invalid model bugs using the CVC4 option `check-models` and the Z3 option `model_validate`. We compile CVC4 and Z3 with assertions and Addressantizer [61] enabled, to detect crash bugs. After collecting and reducing the bug candidates, we then contact the developers to confirm the bugs.

Methodology. To answer RQ1, we report all the correctness bugs, crash bugs, and performance bugs found by Falcon. When studying RQ3-RQ4, we only report the detected correctness bugs and crash bugs, because a performance bug requires a much longer time to verify. Following the commonly-used guidelines [42] for evaluating fuzzing techniques, we run the studied fuzzers with a time budget of twenty-four hours, using the latest Git commits of the SMT solvers at the time of writing. Besides, we run each experiment ten times and use the average data as the final results.

We count the number of unique crash bugs as follows. If a crash is caused by assertion failures, the SMT solver can print the assertion line. We compare the line information to prune duplicate bugs. [7] If a crash is due to memory safety problems such as buffer overflow, we use the stack trace provided by the AddressSanitizer for pruning the duplicates.

Environment. We conduct all the experiments on a Linux workstation with an 80-core Intel (R) Xeon CPU@2.42 GHz and 256 GB RAM, running Ubuntu 16.04 operation system. We use the GNU coverage tool gcov to measure the code coverage. All the tools are set to run in single-threaded mode.

6.2 Discovering Bugs

Summary of the Results. From March 2020 to August 2020, we were running Falcon to test the latest GitHub commits of CVC4 and Z3. The bug reports are publicly available at the site below for independent validation.

https://smtfuzz.github.io/

Enormous Number of Bugs. Table 2 summarizes the status of the reports. "Reported" represents the unique bugs we report after de-duplicating by ourselves; "Confirmed" represents the bugs that are confirmed by the developers as real and unique bugs; "Fixed" represents a subset of the confirmed bugs that have been fixed by the developers through at least one bug fixing commit; "Duplicate" represents those bugs that are identified by the developers as duplicate; "Invalid" represents the bugs that are rejected by the developers (e.g., due to misconfigurations). As can be seen, Falcon finds more than 500 confirmed bugs in the two SMT solvers. All the bugs are previously unknown, and the majority of the bugs are long latent despite extensive prior testing efforts.

[7] It is possible that two formulas trigger the same assertion failures, but they have different root causes. In this experiment, we regard them as one bug, because the root causes need to be confirmed by the developers.

Table 2: Status the bugs found by Falcon.

Status	CVC4	Z3	Total
Reported	164	534	698
Confirmed	105	413	518
Fixed	80	389	469
Duplicate	34	53	87
Invalid	12	65	77

Table 3: Types of the bugs among the confirmed bugs.

Type	CVC4	Z3	Total
Correctness	19	85	104
Crash	80	324	404
Performance	6	4	10

Table 4: Number of supplied solver options among the confirmed bugs.

#Options	CVC4	Z3	Total
0	21	19	40
1	34	92	126
2	16	96	112
3+	34	206	240

Majority of the Theories Affected. Falcon has found bugs in almost all the SMT-LIB2 theories, such as (non)linear integer and real arithmetics, bit-vectors, uninterpreted functions, arrays, floating points, strings, sets, and the combinations of these theories. At the time of writing, 469 of the bugs have already been fixed by the solver developers.

Diverse Bug Types. As shown in Table 3, the bugs detected by Falcon have a wide range of types. Among the confirmed ones, crash bugs are most frequent (404), followed by correctness bugs (104), and performance issues (10). Notably, *Falcon finds 19 and 80 correctness bugs in CVC4 and Z3, respectively.* Most of the correctness bugs were fixed promptly, which makes the solvers more reliable.

Impact of Solver Options. We have also examined the impact of exploring SMT solvers' configuration spaces. Table 4 summarizes the number of involved solver options among the confirmed bugs. As can be seen, more than 90% of the bugs are due to configuration fuzzing, and 46% of them involve more than 3 solver options. The

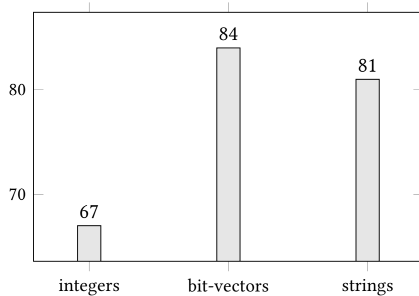


Figure 5: Precision of the sampled correlation pairs for three selected domains.

results clearly demonstrate the importance of fuzzing the configuration space for bug finding.

Feedback from the Developers. We have received very positive feedback from the solver developers. As an example, Falcon detects dozens of bugs in a new decision procedure for integer and real arithmetics in Z3, which has replaced the previously default arithmetic solver. To quote the responses of a Z3's developer, "*Many of these bug reports contribute to making Z3 solid for unsuspecting and innocent users.*" " *There are many good things coming out: proof objects are getting scrutinized, and smt.arith.solver=6 is exercised much more than simply running on benchmarks the SMT-LIB2 repository* [8] *provides.*"

Many solver developers maintain their own regression test suites to validate their solvers. During the six months of extensive testing, the test cases produced by Falcon were continually added to the SMT solvers' official regression test suits.

Takeaways. Falcon is highly effective in finding a large number of diverse bugs that affect most of the SMT-LIB2 theories. Besides, its findings are significant and well-recognized by the developers.

6.3 Precision of the Learned Correlations

We ran Algorithm 1 for roughly two weeks to learn the operation-option correlations in CVC4 and Z3. Note that, it is non-trivial to evaluate the precision and recall of the learned correlations because there is no existing oracle. As an SMT solver can support hundreds of formula operations and hundreds of algorithmic options, manually verifying all the results requires domain knowledge and an immense amount of time.

Thus, to provide a sense of the observed precision, we select three domains for the study, including integers, bit-vectors, and strings. We sample 100 learned correlation pairs for each of the domain, We then conduct a cross-checking manual process to verify the precision of the sampled data.

Figure 5 summaries the true positive rates. In general, the learned correlations are mode effective for bit-vectors and strings, because they typically have many kinds of formula operations. Besides, there are many algorithms specialized for the operations, such as different rewriting rules. In particular, formula rewriters can be as important as the decision procedure for bit-vectors and strings [52].

[8] http://smtlib.cs.uiowa.edu/benchmarks.shtml

Table 5: Line (denoted "L"), function (denoted "F"), and branch (denoted "B") coverage achieved by the four variants of our approach.

		Falcon	Falcon-Search	Falcon-Learn	Falcon-NoOpt
CVC4	L	40.5%	27.9%	29.9%	20.4%
	F	51.7%	37.2%	40.2%	29.4%
	B	14.8%	8.5%	9.5%	6.9%
Z3	L	42.9%	28.8%	34.8%	19.2%
	F	45.6%	32.7%	38.6%	20.3%
	B	19.2%	11.6%	14.3%	5.0%

6.4 Evaluation of Key Components

In this section, we present the results of an ablation study, which aims to investigate the contributions of the components in our approach to its overall effectiveness.

Setup. We compare the following four different variants of our fuzzer, in terms of code coverage and bug detection.

- Falcon, the standard strategy presented in this paper, which applies all the optimizations in § 4.
- Falcon-Search leverages the genetic algorithm (§ 4.2) to mutates all solver options, without running the learning phase to infer the operation-option correlations.
- Falcon-Learn learns the operation-option correlations (§ 4.1), but does not apply the genetic algorithm to guide the mutation. Instead, we configure it to mutate the relevant solver options randomly.
- Falcon-NoOpt only feeds a generated formula to the SMT solver once, without further exploring the configuration space by mutating the solver's options.

Results. Table 5 summarizes the code coverage achieved by the four variants of our fuzzer, where the numbers represent the line (L), function (F), and branch (B) coverage, respectively. Table 6 compares the number of detected bugs. A column "correctness" represents the number of instances that trigger correctness bugs, and a column "crash" represents the number of unique crashes. Figure 6 shows the Venn-diagrams that depict the relationships between the bugs found by Falcon, Falcon-Search, and Falcon-Learn. Briefly, we make the following major observations.

First, the impact of the configuration space is clearly visible for bug hunting. Specifically, Falcon detects 24 and 23 bugs in CVC4 and Z3, respectively. At the other extreme, Falcon-NoOpt only finds 4 and 2 crashes in CVC4 and Z3 respectively, and fails to detect any correctness issues in the solvers. In total, Falcon, Falcon-Search, and Falcon-Learn can detect 41, 7, and 23 more bugs than Falcon-NoOpt, respectively.

Second, Falcon-Learn performs better than Falcon-Search on both of the solvers, in terms of code coverage and bug finding. Especially, Falcon-Learn can find 9 and 7 more bugs than Falcon-Search in CVC4 and Z3, respectively. We notice that the learning phase can often prune a large number of options for the fuzzing phase. Therefore, the number of options for which Falcon-Learn mutate can be much smaller than that of Falcon-Search. Despite the fact that the mutation in Falcon-Learn is random, the search space is smaller than that of Falcon-Search.

Table 6: Bugs detected by the four variants of Falcon. The "correctness" columns represent the number of SMT instances triggering correctness bugs. The "crash" columns represent the number of unique crashes.

	Falcon		Falcon-Search		Falcon-Learn		Falcon-NoOpt	
	correctness	crash	correctness	crash	correctness	crash	correctness	crash
CVC4	2	22	0	8	2	15	0	4
Z3	9	14	1	4	3	9	0	2

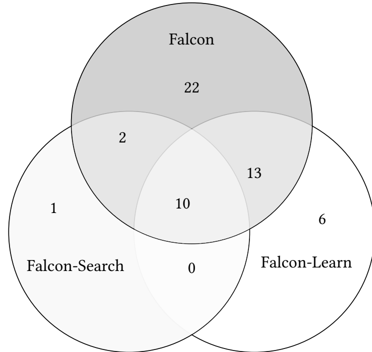


Figure 6: Venn-diagrams depicting the sets of bugs found by Falcon, Falcon-Search, and Falcon-Learn.

As illustrated in § 2, an SMT solver can consist of many algorithmic components. To thoroughly exercise these components, we can explore the solver's configuration space by mutating the exposed options. Without attempting to mutate the options, Falcon-NoOpt can lose many fuzzing opportunities. Without optimizing the mutation strategy, the explorations can also be less effective than Falcon, as in the cases of Falcon-Search and Falcon-Learn.

6.5 Comparison with Existing Fuzzers

In this section, we compare Falcon against two state-of-the-art fuzzers for testing SMT solvers:

- Storm [46] is a mutational fuzzer that mutates seed formulas with fixed rules, which can generate satisfiable instances from any given seed. Storm builds on Z3's Python API, and, thus, can fuzz the logics supported by Z3.
- YinYang [66, 67] is a mutational fuzzer that embodies two engines, the semantic fusion strategy [67] and the type-aware operation mutation strategy [66].

Since Winterer et al. [66] have shown that type-aware operation mutation outperforms semantic fusion in bug finding and supports much more logics, [9] we exclude semantic fusion from this study. We denote YinYang that uses the type-aware operation mutation strategy as OpFuzz.

Setup. To answer RQ4, we have designed two experiments.

- First, we compare Falcon against the default settings of Storm and OpFuzz. By default, Storm does not fuzz the solvers' configuration spaces, while OpFuzz can fuzz them by mutating the solver options randomly.

[9] Semantic fusion only supports integer, reals, and string formulas, while type-aware mutation further supports bit-vectors, floats, etc.

Table 7: Line, function, and branch coverage achieved by Falcon, Storm, OpFuzz, Storm-Opt, and OpFuzz-Opt.

		Falcon	Storm	OpFuzz	Storm-Opt	OpFuzz-Opt
CVC4	L	40.5%	18.6%	25.3%	33.8%	37.9%
	F	51.7%	23.3%	33.6%	36.7%	43.9%
	B	14.8%	6.4%	8.3%	9.8%	11.4%
Z3	L	42.9%	16.7%	23.7%	32.5%	38.6%
	F	45.6%	17.1%	26.1%	38.4%	41.8%
	B	19.2%	4.8%	9.6%	14.5%	18.6%

- Second, we use Falcon to enhance Storm and OpFuzz, by leveraging the learned operation-option correlations (§ 4.1) and the genetic algorithm (§ 4.2) to better fuzz the configuration spaces. We denote the obtained variants of Storm and OpFuzz as Storm-Opt and OpFuzz-Opt, respectively.

We run the tools using the same settings of § 6.4. For Storm and OpFuzz, we collect the seed formulas by following their original papers [46, 66], and use their default running parameters. For Storm-Opt and OpFuzz-Opt, we configure them to use the same set of seeds as Storm and OpFuzz. That is, we do not change the formula mutation strategies of OpFuzz-Opt and Storm-Opt, but only enhance them by mutating the solver options.

Results. Table 7 summarizes the results of code coverage. Table 8 shows the comparison of bug finding.

Comparison with Vanilla Storm and OpFuzz. We can observe that Falcon consistently achieves higher coverage for CVC4 and Z3 by a large margin. Compared with the best fuzzer between Storm and OpFuzz, on average, Falcon increases the line, function, and branch coverage of the two solvers by 17.2%, 18.8%, and 8.1%, respectively. It is noteworthy that CVC4 and Z3 have near 248 KLoC and 459 KLoC, respectively. Therefore, 1% improvement of line coverage already translates to thousands of additionally covered lines.

Falcon detects 22 and 14 unique crash bugs in CVC4 and Z3, respectively. Besides, it generates 2 and 9 instances that trigger correctness bugs in CVC4 and Z3, respectively. In comparison, Storm generates 3 formulas triggering correctness issues in Z3 but does not find crash bugs. OpFuzz uncovers 5 and 3 unique crashes in CVC4 and Z3 respectively, and reveals one correctness bug in Z3. We observe that all the crashes detected by OpFuzz are included in the bugs found by Falcon.

Using Falcon to Enhance Storm and OpFuzz. In Table 7 and Table 8, the columns "Storm-Opt" and "OpFuzz-Opt" show the results of applying Falcon to enhance Storm and OpFuzz. Using the same set of seed formulas, Storm-Opt and OpFuzz-Opt can achieve much

Table 8: Bugs detected by Falcon, Storm, OpFuzz, Storm-Opt, and OpFuzz-Opt.

	Falcon		**Storm**		**OpFuzz**		**Storm-Opt**		**OpFuzz-Opt**	
	correctness	crash	correctness	crash	correctness	crash	correctness	crash	correctness	crash
CVC4	2	22	0	0	0	5	0	11	1	13
Z3	9	14	3	0	1	3	6	4	4	8

higher coverage and detect more bugs than their unoptimized counterparts. Take OpFuzz as an example. By default, it can explore the configuration space by randomly mutating solver options. After using Falcon to better explore the configuration spaces of CVC4 and Z3, OpFuzz-Opt improves the line coverage over OpFuzz by 12.6% for CVC4 and 14.9% for Z3. Besides, OpFuzz-Opt can find 9 and 8 more bugs in CVC4 and Z3, respectively.

To summarize, the results demonstrate that (1) compared with Storm and OpFuzz, Falcon can result in a noticeable coverage increase and find more bugs within the time limit, and (2) Falcon can significantly increase the code coverage and bug finding capability of Storm and OpFuzz, by optimizing the explorations of the solvers' configuration spaces.

6.6 Case Studies on Sample Bugs

In this section, we select and discuss four reported bugs in CVC4 and Z3. Figure 7 describes the corresponding reduced SMT queries with bug classifications and GitHub issue IDs.

Figure 7(a) shows a heap-use-after-free bug in Z3. Note that, the bug occurs only when Z3 is called with the command-line options `rewriter.push_ite_arith=true` and `smt.string_solver=seq`. The bug is related to the formula simplifier affected by the option `rewriter.push_ite_arith` and the string solver controlled by the option `smt.string_solver`.

Figure 7(b) depicts an invalid model bug in CVC4, which is related to a new algorithmic component affected by the option `cbqi-prereg-ins`. The component named "counterexample-guided quantifier instantiation" is recently introduced for optimizing the solving of quantified formulas [57].

Figure 7(c) presents an invalid model bug in Z3. The bug is related to Z3's arithmetic solver and formula rewriters. The bug is triggered by the command-line options `smt.arith.solver=2`, `rewriter.flat=false`, and `rewriter.push_ite_arith=true`.

Figure 7(d) is a performance bug in CVC4's bit-vector solver. Z3 solves this formula within 0.5 seconds, but CVC4 does not terminate in 10 minutes. To address the issue, the developers add a new formula transformation rule, after which CVC can solve the formula instantly.

```
1  (declare-const i1 Int)
2  (declare-const s1 String)
3  (assert (>= (str.len s1) i1))
4  (push 1)
5  (assert (>= 0 (abs i1)))
6  (pop 1)
7  (assert  (>= 0 (* i1 135
8     (mod i1 i1) i1 i1)))
9  (push 1)
10 (check-sat)
```

(a) Z3 use-after-free bug 4427

```
1  (declare-const v1 Bool)
2  (declare-const v2 Bool)
3  (assert (exists (
4     (q1 (_ BitVec 12))
5     (q2 Bool)
6     (q3 (_ BitVec 12)))
7        (xor v1 v2)))
8  (assert (forall
9     ((q4 (_ BitVec 6))) v2))
10 (check-sat)
```

(b) CVC4 invalid model bug 4243

```
1  (declare-const i1 Int)
2  (assert ( >= (* 495 17
3     (abs i1)) 287))
4  (check-sat)
```

(c) Z3 invalid model bug 3910

```
1  (declare-const b (_ BitVec 40))
2  (assert (bvugt (bvurem
3     ((_ rotate_right 6) b) b) b))
4  (check-sat)
```

(d) CVC4 performance bug 4936

Figure 7: Selected bug samples in CVC4 and Z3.

6.7 Discussion

Threats to Validity. First, we validate Falcon over CVC4 and Z3, which have also been chosen in several previous works [46, 66, 67]. However, the two SMT solvers are not necessarily representative of other tools. In the future, we will further apply our approach to other solvers. Second, Storm and OpFuzz are mutational fuzzers and require seed formulas as input, which can affect the evaluation results. To reduce the threat, we have followed the settings of the authors' papers to collect the seeds. Third, to mitigate the threat brought by randomness in fuzzing, we run each experiment ten times and use the average data, following the evaluation instructions in the prior work [42].

Limitations. The studies demonstrate the effectiveness of our approach, but Falcon has some limitations. First, a solver option may only be valid for a combination of formula operations, and therefore the learning phase could miss certain correlations that are beyond Definition 4.1. Second, the learning phase does not explicitly consider the interactions of different solver options. Nonetheless, when adaptively mutating the set of correlated options in the fuzzing phase, Falcon can exploit the interactions implicitly.

Future Work. There are several avenues for further improving the effectiveness of our approach. First, the interplay of the sheer number of formula operations and solver options can be very complicated, and, thus, it is stunningly challenging to consider all of them. A possible future direction is to consider the correlations formed by t operators and u options in the learning phase, where t and u are parametric. Second, currently, we follow the idea of swarm testing to diverse the probabilities of Falcon's formulas generator. Another future direction is using some feedback information to optimize the probabilities, such that the generated test formulas may be more likely to trigger exceptional behavior [27, 64].

7 RELATED WORK

Fuzzing SMT Solvers. FuzzSMT [14] and StringFuzz [12] use grammar-based blackbox fuzzing to generate syntactically valid SMT formulas. BanditFuzz [60] extends this line of work by using reinforcement learning to learn the grammatical constructs that are likely the cause of performance issues. BtorMBT [51] uses model-based testing to generate API calling sequences, according to manually constructed models. Winterer et al. [66] introduce type-aware mutations that mutate the operators in a seed formula. More recently, several techniques have emerged for detecting soundness bugs in SMT solvers [16, 46, 67]. They employ various strategies to generate SMT formulas whose satisfiability is known by construction. Such ground truth acts as the test oracle, which is compared against the actual SMT solving results to detect soundness bugs.

In this work, we introduce a framework to better explore the combined formula-configuration space. Specifically, we propose a data-driven approach for inferring the operation-option correlations, and design a feedback-driven mechanism that adaptively mutates solver options. Our work shows a promising direction for fuzzing SMT solvers that can continually benefit the community. For example, the developers can use our approach to test new solver features, which often expose new formula operations and algorithmic options [23].

Combinatorial Interaction Testing. There is a huge amount of literature on combinatorial interaction testing [5, 20, 36, 41, 44, 48, 53, 56, 65, 68], which aims to test configurable systems with large configuration spaces. Given an interaction strength t, conventional approaches compute a set of configurations such that all possible t-way combinations of option settings appear in at least one configuration. The subject program is then tested under each configuration in the covering array. However, prior work mainly focuses on exploiting the correlations within the configuration space. For example, several approaches encode the relations of different options as a Boolean formula, and then sample configurations by generating solutions of the formula [5, 53, 56]. In comparison, our approach rests on utilizing the correlations between the formula space and the configuration space. By identifying such correlations, we can reduce the configuration space *w.r.t* a concrete formula.

Grammar-based Fuzzing. Grammar-aware fuzzing is a popular direction in the state of the arts. Recently, there have been growing interests in satisfying both syntax and semantic correctness. Nautuils [2] combines the execution feedback such as coverage with tree-based mutations to prioritize the seeds with higher potential to detect target program behaviors. CodeAlchemist [37] preserves the semantic requirement, e.g., type correlation, as the constraint during input generation to ensure semantic correctness. Zest [54] combines the coverage feedback with property-based testing to provide better guidance for seed prioritization.

Similar to conventional generative fuzzers [28, 38, 63], Falcon constructs syntactically correct formulas from scratch. Besides, we observe that it is insufficient to detect program flaws in SMT solvers if unaware of the configuration space. By mutating the correlated solver options for a test formula, we can better exploit the semantic information of the formula, as more relevant algorithmic components in the solver would be exercised.

Hyperparameter Optimization for SMT. There have been several approaches for automatically tuning the parameters, i.e., options, of SAT/SMT solvers [1, 4, 39, 40, 59]. Given a set of formulas, their goal is to find proper solver configurations from a vast space of discrete and continuous options, so that the solver can solve the formulas faster. In comparison, we address a different problem—learn how to mutate the options to find good solver configurations for fuzz testing. The genetic algorithm in StratEVO [59] is closely related to our method for mutating solver options. However, StratEVO does not leverage the actual features of a formula, e.g., the operations the formula contains. In contrast, Falcon can utilize the inferred correlations and the features of the formula to identify the relevant options, thereby reducing the genetic algorithm's search space. Besides, StratEVO uses the solving time as feedback, while Falcon can leverage code coverage as its feedback information.

8 CONCLUSION

SMT solvers serve as the substrate for many techniques in software engineering research and have found many practical applications in the industry. In this paper, we have presented Falcon, which explores the two-dimensional input space for fuzzing SMT solvers. In six months of extensive testing, Falcon discovered 518 confirmed bugs in CVC4 and Z3, almost all of which have been fixed, clearly demonstrating its effectiveness.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. We also appreciate the developers of CVC4 and Z3 for discussing and addressing our reported bugs. Rongxin Wu is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001) and NSFC61902329. Other authors are supported by the RGC16206517 and ITS/440/18FP grants from the Hong Kong Research Grant Council, and the donations from Microsoft and Huawei. Rongxin Wu is the corresponding author.

REFERENCES

[1] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. 2009. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings (Lecture Notes in Computer Science, Vol. 5732)*, Ian P. Gent (Ed.). Springer, 142–157. https://doi.org/10.1007/978-3-642-04244-7_14

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 1083–1094. https://doi.org/10.1145/2568225.2568293

[4] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. 2018. Learning to Solve SMT Formulas. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 10338–10349. https://proceedings.neurips.cc/paper/2018/hash/68331ff0427b551b68e911eebe35233b-Abstract.html

[5] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: an adaptive weighted sampling approach for improved t-wise coverage. In *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,*

Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1114–1126. https://doi.org/10.1145/3368089.3409744

[6] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177. http://dl.acm.org/citation.cfm?id=2032305.2032319

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

[8] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A String Solver with Theory-Aware Heuristics. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design* (Vienna, Austria) *(FMCAD '17)*. FMCAD Inc, Austin, Texas, 55–59. https://doi.org/10.5555/3168451.3168468

[9] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.1

[10] Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. 2012. An SMT-LIB format for sequences and regular expressions. *SMT* 12 (2012), 76–86.

[11] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko

[12] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 45–51. https://doi.org/10.1007/978-3-319-96142-2_6

[13] 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. 2020. Block public access: trust safety verification of access control policies. In *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*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 281–291. https://doi.org/10.1145/3368089.3409728

[14] Robert Brummayer and Armin Biere. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories* (Montreal, Canada) *(SMT '09)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1670412.1670413

[15] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The OpenSMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6015)*, Javier Esparza and Rupak Majumdar (Eds.). Springer, 150–153. https://doi.org/10.1007/978-3-642-12002-2_12

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

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. 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* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[18] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 305–316. https://doi.org/10.1109/ASE.2019.00037

[19] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7385)*, Alastair F. Donaldson and David Parker (Eds.). Springer, 248–254. https://doi.org/10.1007/978-3-642-31759-0_19

[20] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2006. Coverage and adequacy in software product line testing. In *Proceedings of the 2006 Workshop on Role of Software Architecture for Testing and Analysis, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), ROSATEA 2006, Portland, Maine, USA, July 17-20, 2006*, Robert M. Hierons and Henry Muccini (Eds.). ACM, 53–63. https://doi.org/10.1145/1147249.1147257

[21] Byron Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. 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* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[23] Leonardo de Moura and Grant Olney Passmore. 2013. The Strategy Challenge in SMT Solving. (2013), 15–44.

[24] Kalyanmoy Deb and Debayan Deb. 2014. Analysing Mutation Schemes for Real-Parameter Genetic Algorithms. *Int. J. Artif. Intell. Soft Comput.* 4, 1 (Feb. 2014), 1–28. https://doi.org/10.1504/IJAISC.2014.059280

[25] Bruno Dutertre. 2014. Yices2.2. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

[26] Bruno Dutertre and Leonardo Mendonça de Moura. 2006. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4144)*, Thomas Ball and Robert B. Jones (Eds.). Springer, 81–94. https://doi.org/10.1007/11817963_11

[27] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary Grammar-Based Fuzzing. In *Search-Based Software Engineering*, Aldeida Aleti and Annibale Panichella (Eds.). Springer International Publishing, Cham, 105–120. https://doi.org/10.1007/978-3-030-59762-7_8

[28] M Eddington. 2008. Peach fuzzer. *URl: http://www. peachfuzzer. com/(visited on 06/21/2017)* (2008).

[29] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. SMT-Based Verification of Software Countermeasures against Side-Channel Attacks. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 62–77. https://doi.org/10.1007/978-3-642-54862-8_5

[30] Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531. http://dl.acm.org/citation.cfm?id=1770351.1770421

[31] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[32] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[33] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, Mats Per Erik Heimdahl and Zhendong Su (Eds.). ACM, 78–88. https://doi.org/10.1145/2338965.2336763

[34] Hai-Feng Guo and Zongyan Qiu. 2013. Automatic Grammar-Based Test Generation. In *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8254)*, Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich (Eds.). Springer, 17–32. https://doi.org/10.1007/978-3-642-41707-8_2

[35] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. 2014. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 680–695. https://doi.org/10.1007/978-3-319-08867-9_45

[36] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empir. Softw. Eng.* 24, 2 (2019), 674–717. https://doi.org/10.1007/s10664-018-9635-4

[37] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/codealchemist-semantics-aware-code-generation-to-find-vulnerabilities-in-javascript-engines/

[38] Mark Hennessy and James F. Power. 2005. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005),*

November 7-11, 2005, Long Beach, CA, USA, David F. Redmiles, Thomas Ellman, and Andrea Zisman (Eds.). ACM, 104–113. https://doi.org/10.1145/1101908.1101926

[39] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers (Lecture Notes in Computer Science, Vol. 6683)*, Carlos A. Coello Coello (Ed.). Springer, 507–523. https://doi.org/10.1007/978-3-642-25566-3_40

[40] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res.* 36 (2009), 267–306. https://doi.org/10.1613/jair.2861

[41] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based sampling of software configuration spaces. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1084–1094. https://doi.org/10.1109/ICSE.2019.00112

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[43] Ralf Lämmel and Wolfram Schulte. 2006. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3964)*, M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko (Eds.). Springer, 19–38. https://doi.org/10.1007/11754008_2

[44] Roberto E. Lopez-Herrejon, Francisco Chicano, Javier Ferrer, Alexander Egyed, and Enrique Alba. 2013. Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 404–407. https://doi.org/10.1109/ICSM.2013.58

[45] Feng Lu, Li-C. Wang, Kwang-Ting Cheng, and Ric C.-Y. Huang. 2003. A Circuit SAT Solver With Signal Correlation Guided Learning. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*. IEEE Computer Society, 10892–10897. https://doi.org/10.1109/DATE.2003.10018

[46] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *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*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 701–712. https://doi.org/10.1145/3368089.3409763

[47] Peter M. Maurer. 1990. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Softw.* 7, 4 (1990), 50–55. https://doi.org/10.1109/52.56422

[48] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Comput. Surv.* 43, 2 (2011), 11:1–11:29. https://doi.org/10.1145/1883612.1883618

[49] Aina Niemetz and Armin Biere. 2013. ddSMT: a delta debugger for the SMT-LIB v2 format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories (SMT'13), affiliated to SAT*, Vol. 13. 36–45. https://doi.org/doi=10.1.1.380.134

[50] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *J. Satisf. Boolean Model. Comput.* 9, 1 (2014), 53–58. https://doi.org/10.3233/sat190101

[51] Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Model-based API testing for SMT solvers. In *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT*.

[52] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2019*, Mikoláš Janota and Inês Lynce (Eds.). Springer International Publishing, Cham, 279–297. https://doi.org/10.1007/978-3-030-24258-9_20

[53] Jeho Oh, Don S. Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 61–71. https://doi.org/10.1145/3106237.3106273

[54] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 329–340. https://doi.org/10.1145/3293882.3330576

[55] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 23–41. https://doi.org/10.1007/978-3-319-41540-6_2

[56] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 240–251. https://doi.org/10.1109/ICST.2019.00032

[57] Mathias Preiner, Aina Niemetz, and Armin Biere. 2017. Counterexample-Guided Model Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 264–280. https://doi.org/10.1007/978-3-662-54577-5_15

[58] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (1972), 366–375. https://doi.org/10.1007/BF01932308

[59] Nicolás Gálvez Ramírez, Youssef Hamadi, Éric Monfroy, and Frédéric Saubion. 2016. Evolving SMT Strategies. In *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016, San Jose, CA, USA, November 6-8, 2016*. IEEE Computer Society, 247–254. https://doi.org/10.1109/ICTAI.2016.0046

[60] Joseph Scott, Federico Mora, and Vijay Ganesh. 2020. 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 (Lecture Notes in Computer Science, Vol. 12549)*, Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel (Eds.). Springer, 68–86. https://doi.org/10.1007/978-3-030-63618-0_5

[61] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[62] João P. Marques Silva and Karem A. Sakallah. 1996. GRASP - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, Rob A. Rutenbar and Ralph H. J. M. Otten (Eds.). IEEE Computer Society / ACM, 220–227. https://doi.org/10.1109/ICCAD.1996.569607

[63] Emin Gün Sirer and Brian N. Bershad. 1999. Using production grammars in software testing. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99), Austin, Texas, USA, October 3-5, 1999*, Thomas Ball (Ed.). ACM, 1–13. https://doi.org/10.1145/331960.331965

[64] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.3013716

[65] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45. https://doi.org/10.1145/2580950

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

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

[68] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. 2016. Greedy combinatorial test case generation using unsatisfiable cores. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 614–624. https://doi.org/10.1145/2970276.2970335