Validating SMT Solvers via Skeleton Enumeration Empowered by Historical Bug-Triggering Inputs

Maolin Sun[1†], Yibiao Yang[2*], Ming Wen[1†*], Yongcong Wang[1†], Yuming Zhou[2], Hai Jin[3†]
[1]*School of Cyber Science and Engineering, Huazhong University of Science and Technology, China*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, China*
[3]*School of Computer Science and Technology, Huazhong University of Science and Technology, China*
Email: {merlinsun, mwenaa, m202071390, hjin}@hust.edu.cn, {yangyibiao, zhouyuming}@nju.edu.cn

***Abstract*—SMT solvers check the satisfiability of logic formulas over first-order theories, which have been utilized in a rich number of critical applications, such as software verification, test case generation, and program synthesis. Bugs hidden in SMT solvers would severely mislead those applications and further cause severe consequences. Therefore, ensuring the reliability and robustness of SMT solvers is of critical importance. Although many approaches have been proposed to test SMT solvers, it is still a challenge to discover bugs effectively. To tackle such a challenge, we conduct an empirical study on the historical bug-triggering formulas in SMT solvers' bug tracking systems. We observe that the historical bug-triggering formulas contain valuable skeletons (i.e., core structures of formulas) as well as associated atomic formulas which can cast significant impacts on formulas' ability in triggering bugs. Therefore, we propose a novel approach that utilizes the skeletons extracted from the historical bug-triggering formulas and enumerates atomic formulas under the guidance of association rules derived from historical formulas. In this study, we realized our approach as a practical fuzzing tool `HistFuzz` and conducted extensive testing on the well-known SMT solvers Z3 and cvc5. To date, `HistFuzz` has found 111 confirmed new bugs for Z3 and cvc5, of which 108 have been fixed by the developers. More notably, out of the confirmed bugs, 23 are soundness bugs and invalid model bugs found in the solvers' default mode, which are essential for SMT solvers. In addition, our experiments also demonstrate that `HistFuzz` outperforms the state-of-the-art SMT solver fuzzers in terms of achieved code coverage and effectiveness.**

***Index Terms*—SMT solver, fuzzing, skeleton enumeration, association rules, bug detection**

I. Introduction

Satisfiability Modulo Theory (SMT) solvers are used for checking the satisfiability of a logical formula with respect to first-order theories, such as arithmetic, arrays, floating points, and Unicode strings [1]. As one of the cornerstones in formal methods, they are widely used in program analysis [2], [3], software verification [4], software repair [5], [6], symbolic execution [7], [8], [9], [10], program synthesis [11], and vulnerability detection [12]. Therefore, bugs hidden in SMT solvers would inevitably mislead their client applications, thus causing serious consequences. For example, a symbolic execution engine would be misled to make opposite decisions over the feasibility of a program path, and further the code in the path will not be covered in testing. Thus, ensuring the reliability of SMT solvers is of critical importance.

To ensure the correctness and robustness of SMT solvers, many approaches have been proposed to validate SMT solvers in recent years. Specifically, YinYang [13] constructs two formulas and adds additional variables as constraints to stress-test SMT solvers. STORM [14] fragments a formula and reconstructs its sub-formulas to generate mutants with completely different Boolean structures. In addition, OpFuzz [15] mutates the operators in the formulas and performs differential testing. Furthermore, Sparrow [16] defines multiple approximation relations so that it can manipulate the operators or constants in a formula to obtain new SMT instances with test oracles. More recently, TypeFuzz [17] improves OpFuzz, combining mutation and generation, and thus more diverse formulas can be generated. The above approaches have achieved promising results as a total of thousands of bugs have been uncovered in the most widely-used solvers, Z3 [18] and cvc5 [19] (i.e., the successor of CVC4 [20]), in recent years. The above existing approaches mainly focus on generating formulas with diverse characteristics in different ways, such as mutating the variables, operators, structures, or their combinations. However, formulas with diverse characteristics do not necessarily mean that they are more likely to trigger bugs as an existing study [21] points out. Besides, they rarely consider the historical information (i.e., existing bug-triggering formulas), which has been found to be effective in the fuzzing of other domains [21], [22], [23], [24]. To further enhance the effectiveness of testing SMT solvers, in this study, we cast our attention on exploring the feasibility of utilizing the historical bug-triggering formulas to discover new SMT solvers' bugs. We first performed a preliminary investigation and made the following novel observations.

First, we observe that *the structures of formulas play an important role in triggering bugs*. Specifically, in terms of triggering bugs, even two semantic-equivalent formulas with different structures lead to opposite consequences. We use **skeleton** to represent a formula's structure [16], [17], which is merely composed of the logical connectives and quantifiers of the formula. For example, the skeleton of the formula φ in Figure 1a is "*forall* □ (*forall* □ (*and* □))". Note that the connectives should connect formulas while quantifiers are

† Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab.
* Corresponding authors

followed by variables. Interestingly, the formula φ triggered an assertion violation crash in Z3, however, the semantically equivalent formula φ' in Figure 1b cannot trigger any bug. We can find that φ and φ' consist of the same atomic formulas but with different connectives and quantifiers. An **atomic formula** (*a.k.a.* atom) is a formula that does not contain any logical connectives (e.g., `(= y (f (f y))` in formula φ). Motivated by this, we further analyzed the skeletons of existing bug-triggering inputs of two representative SMT solvers (i.e., Z3 and cvc5). Specifically, we collected 804 formulas that can reproduce the bugs in the default mode.[1] Next, we can apply certain tactics (e.g., `simplify`) in Z3 on those bug-triggering formulas to obtain semantically equivalent formulas with different skeletons. Then, we checked whether the equivalent variants can still trigger bugs. The results show that 252 (31%) semantic-equivalent variants transformed by the `simplify` tactic cannot trigger the corresponding bugs anymore. Such results indicate that the skeletons of formulas indeed play important roles in revealing bugs for SMT solvers.

Second, *the atomic formulas in bug-triggering formulas contain specific patterns*. Naturally, whether a skeleton will trigger bugs is also determined by the filled atomic formulas. Moreover, we observe that formulas sharing the same skeleton can even reveal different bugs when filled with similar atomic formulas. Therefore, we are further motivated to investigate the hidden patterns of atomic formulas among existing bug-triggering formulas. Association rule learning is an effective approach to identifying the frequently occurring patterns among items in data, which has been applied to various domains [25], [26], [27]. We performed association rule learning on 2,898 bug-triggering instances. Consequently, we found that certain atomic formulas frequently appear collectively among various bug-triggering inputs, forming strong association rules. More importantly, those association rules analyzed from bug-triggering formulas are dramatically different from those mined from normal formulas, with less than 4% overlaps. Driven by this, we are further motivated to utilize the association rules among atomic formulas extracted from historical data to guide formula generation for more effective testing of SMT solvers.

Approach. Motivated by the above observations, in this paper, we propose a **Hist**ory-Driven Skeleton Enumeration **Fuzz**er (`HistFuzz`) to validate SMT solvers. Our core insight is that the historical bug-triggering formulas contain a wealth of domain knowledge that can guide effective testing. Concretely, we leverage historical information from two perspectives. First, we collect existing bug-triggering formulas and extract their skeletons as the backbone for generating new test formulas. Second, we gather the atomic formulas in those bug-triggering formulas into a candidate pool, and further learn association rules among them. We construct new formulas by enumerating the skeletons and instantiating them with the associated atomic formulas in the candidate pool. Note that, we organize those atomic formulas under the guidance of certain association rules extracted from the historical data rather than by randomness. These rules describe the dependencies of the different atomic formulas in the bug-triggering formula. As a result, more program logic in SMT solvers can be exercised extensively, and thus hidden bugs can be unveiled.

[1] SMT solvers usually provide many supplied options for additional functionalities. The default mode of SMT solvers is their default usage with those additional options disabled.

To evaluate `HistFuzz`, we applied it to two widely-used SMT solvers, Z3 and cvc5. A total of 149 bugs are reported for these two state-of-the-art SMT solvers, of which 111 were fixed or confirmed by the developers. It is worth noting that one of the cvc5 developers expressed great interest in our work and inspired us to report more significant bugs to improve the reliability of SMT solvers. Besides, `HistFuzz` can achieve higher code coverage and exhibit greater bug detection ability compared to other SMT solver testing tools. In this paper, we make the following main contributions:

- ***Originality:*** We propose a novel approach to extensively utilize historical bug-triggering formulas to generate well-structured formulas with associated atomic formulas for fuzzing SMT solvers.
- ***Significance:*** We aim to discover new and significant bugs in SMT solvers. Besides, the achieved results demonstrate that utilizing historical knowledge is promising, which points out new perspectives for SMT solver testing.
- ***Prototype:*** We implement our idea as a prototype, `HistFuzz`, a general and practical fuzzer for SMT solvers. Extensive evaluations have demonstrated the effectiveness of our approach. `HistFuzz` and our bug reports are available at: **https://github.com/CGCL-codes/HistFuzz**.
- ***Usefulness:*** We have reported 149 bugs for Z3 and cvc5, of which 111 are confirmed and 108 of those confirmed bugs are fixed by developers.

Paper Organization. The rest of this paper is structured as follows. Section 2 illustrates the high-level idea of our approach. Section 3 formalizes our approach and describes the implementation of `HistFuzz`. Next, we elaborate on our extensive evaluation in detail in Section 4. In Section 5, we formulate more discussions on our results. Finally, we survey related work in Section 6 and the conclusion is in Section 7.

II. Background and Motivation

This section first presents a brief introduction to the SMT-LIB language and the standard for describing SMT formulas. Then, we motivate and illustrate our technique using empirical study and real examples.

A. Background

SMT-LIB Language. The SMT-LIB language is the standard input language for SMT solvers [28], which is adopted by the majority of SMT solvers. The most basic commands in the SMT-LIB language are `declare-fun`, `assert`, and `check-sat`. The `declare-fun` command is used to declare new symbols. In the SMT-LIB language, variables and functions can be declared in a uniform way. The difference is

```
(declare-sort A)
(declare-fun f (A) A)
(assert (forall ((x A)) (forall ((y A)) (and (= y (f (f y)))))))
(check-sat)
```

(a) formula φ reduced from issue #5532 triggering a bug in Z3.

```
(declare-sort A)
(declare-fun f (A) A)
(assert (forall ((y A)) (= y (f (f y)))))
(check-sat)
```

(b) formula φ', semantic-equivalent to φ, cannot trigger the bug.

Fig. 1: A bug-triggering formula and its equivalent formula generated by Z3.

that variables are treated as functions without arguments. For example, the statement "(`declare-fun` x () Int)" declares the variable x as an integer type. The `assert` command adds constraints to the variables or functions that the formula should satisfy, such as the "(`assert` (`and` ($<$ x 3) ($>$ y 1)))" statement asserts that variable x is less than 3 and variable y is larger than 1. The `check-sat` statement queries the solver to decide on the satisfiability of a formula. Typically, an SMT instance often includes multiple `assert` statements. They can be seen as the conjunctions of multiple constraints. Therefore, an SMT solver will return `sat` only if the constraints can be satisfied simultaneously. Conversely, if no assignment can satisfy all the assertions, that instance is `unsat`.

B. Motivation

For the purpose of testing SMT solvers more effectively, we first conducted a preliminary study on historical bug-triggering formulas. Generally, we made the following major observations.

Skeletons of formulas play an important role in triggering bugs. For a first-order logic formula, we regard the sequence composed of its logical connectives (e.g., conjunctions, disjunctions, and negations) together with the quantifiers as its skeleton. For instance, the skeleton of formula φ in Figure 1 written in the SMT-LIB language is $(forall\ ((\Box_1))\ (forall\ ((\Box_2))\ (and\ \Box_3)))$. To investigate the significant role of skeletons in exposing different bugs, we performed the following empirical studies based on the bugs of Z3 and cvc5, which have been extensively tested by users and thus contain ampler targets available for study.

First, we investigate whether certain bugs can only be triggered by formulas with specific skeletons while the corresponding semantic-equivalent formulas with different skeletons cannot trigger the same ones. To achieve such a goal, we leveraged the `simplify` and `tseitin-cnf` tactic of Z3 to restructure formulas with multiple connectives or quantifiers and create semantic-equivalent formulas, respectively. Specifically, the `simplify` tactic can modify the targeted formulas, including the quantifiers and connectives, to a semantically equivalent variant, while `tseitin-cnf` ignores the quantifiers and converts the formulas to conjunctive normal form (CNF). In this investigation, we only take the formulas that can reproduce the bugs in the solvers' default mode into consideration since the default mode is more widely-used and significant. In addition, we ignore those formulas containing only one connective or quantifier because their skeletons are too simple to create diverse semantic-equivalent variants. Ultimately, 804 formulas (625 for Z3 and 179 for cvc5) are selected. After applying the above two tactics, we fed the variants (i.e., semantic-equivalent with the bug-triggering ones) to the corresponding version of solvers and observed if the bug can still be triggered. Table I shows the results. Specifically, out of the CNF variants, 103 (13%) cannot trigger the bugs as the original formulas. Moreover, for the variants altered by the `simplify` tactic, 252 (31%) cannot trigger the bugs anymore. Therefore, we can conclude that even if they are semantically equivalent, a number of variants cannot reveal the bugs as the original bug-triggering formulas. This also suggests that it is reasonable to consider quantifiers as part of the skeleton since they contribute to the triggering of bugs as well.

TABLE I: The number of bug-triggering formulas that can trigger bugs in the default mode of solvers and their variants modified by the `CNF` and `simplify` tactic that can also trigger the same bugs. The fifth column shows the number of variants that cannot trigger any bugs.

Category	Z3	cvc5	Total	Concealed
Valid	625	179	804	
CNF	552	149	701	103 (13%)
Simplify	440	112	552	252 (31%)

Second, we check the duplication ratios of skeletons extracted from bug-triggering formulas to investigate whether certain skeletons are continuously triggering diverse bugs over time. Specifically, for the 2,898 bug-triggering formulas collected from Z3 and cvc5's bug tracking systems, we extract 963 distinct skeletons in total, and 289 of them (30%) occur repeatedly (i.e., more than one time). Among those repeated skeletons, some uncommon or special forms have triggered a large number of bugs that can hardly be considered or tested exhaustively by developers. For instance, the skeleton "`(assert xor)`" appears multiple times in the bug-triggering formulas. In those circumstances, developers do not adequately take the unusual situation that the connective `xor` connects no formula into consideration (e.g., issue #5260). According to the code changes of the bug-fixing commit a61e9d6, we can find the developers reconsider this uncommon case to avoid the crash. Based on this skeleton, we reported a new bug in issue #5722, whose root cause also points to it. In this issue, the tactic "`aig`" in Z3, which aims to compress Boolean structures of formulas, mishandles this skeleton, resulting in incorrect outputs. Therefore, the skeletons in bug-triggering formulas, including those frequently repetitive ones, are of critical significance for exposing new bugs.

The patterns of the atomic formulas in bug-triggering formulas differ significantly from those in normal formulas. The above investigation reveals that skeletons are critical for exposing bugs in SMT solvers. Besides, we observe that

whether a skeleton will trigger bugs is also affected by the filled atomic formulas. In addition, formulas sharing the same skeletons and similar atomic formulas can even reveal different bugs. Figure 3 presents two formulas σ (Figure 3a) and σ' (Figure 3c) that share the same skeleton and contain only one different atomic formula (i.e., `(= v ((_ iand 1) 1 v))` and `(= v ((_ iand 1) v v))`). However, they trigger different bugs concerning two rewrite rules on the same operator "iand" in cvc5. Motivated by this, we conduct association rule learning on the atomic formulas in the bug-triggering formulas to investigate if there exist certain patterns with respect to their combinations. Note that, since atomic formulas contain various constants, variables, and functions, we first abstract each atomic formula and use its operator as well as operand types as its signature as follows:

$$\Sigma \triangleq \langle O, S_1, S_2, ..., S_n \rangle$$

The signature Σ is denoted as a sequence composed of O and S_i. O represents the operator of the formula, and S_i is the sort of a term, including the theory (e.g., integers, reals, and arrays) of it and whether it contains variables or only constants (e.g., S_i=`IntConst`). Next, we utilize the *Apriori* algorithm [29], an association rule mining algorithm, to analyze the association rules in the frequent itemsets composed of formula signatures. An association rule is an implication of the form $X \Rightarrow Y$, where X is antecedent and Y is consequent, and we consider both *Support* and *Confidence* for the items. *Support* is the proportion of atomic formula set $X \cup Y$ contained in all the formulas, and *Confidence* indicates the proportion of itemsets that also contain Y when X is included. Since the number of instances is fixed in this investigation, we use the quantity instead of the proportion as *Support* for convenience. To capture the association rules with a certain scale, we set the minimum *Support* from 5 to 10, which indicates the times of the items' occurrence, respectively. Besides, the minimum *Confidence* is 0.5.

Specifically, we conduct the investigation on the above-mentioned 2,898 bug-triggering instances as well. For comparison, we also perform association rule learning on the SMT-LIB official benchmarks [30]. They contain extensive SMT instances, which are mostly used for industry and generally do not trigger bugs. We first categorize those instances in benchmarks into five groups by vintage (i.e., from 2017 to 2021), which will be analyzed respectively to draw generalizable conclusions. We then sample 2,898 from each group at random and conduct the association rule learning, respectively. Next, we investigate how many rules in the bug-triggering formulas are also included in the whole sets of all rules learned from the formulas in different groups. Figure 2 shows the results and demonstrates that the rules mined from the bug-triggering formulas and benchmarks contain few overlaps. For example, when the minimum *Support* is 9, a total of 108 rules can be extracted from the bug-triggering formulas, while only 3 rules also exist in benchmarks. Such results indicate that

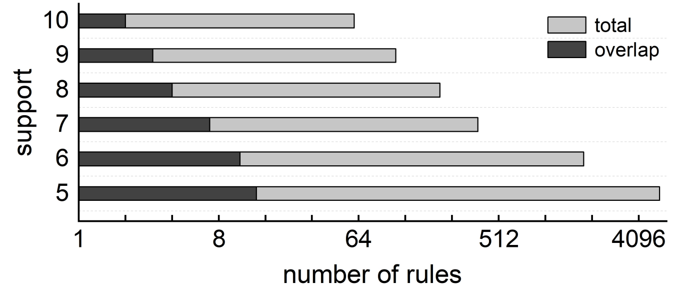


Fig. 2: The number of association rules analyzed from bug-triggering formulas and the overlap with the rules extracted from the benchmark formulas.

```
(assert (exists ((v Int)) (and (= 2 v) (= v ((_ iand 1) 1 v)))))
(check-sat)

skeleton = (assert (exists ☐ (and ☐ ☐)))
```

(a) formula σ simplified from cvc5 #8016 and its skeleton

```
1) Choose atomic formulas from the candidate pool
atoms ∈ { (= 2 v) ,(= v ((_ iand 1) 1 v)),(= v ((_ iand 1) v v)),...}
skeleton = (assert (exists ☐ (and (= 2 v) ☐)))

2) Fill formulas guided by association rules
rules = {(= IntConst IntVar) ⇒ (= IntVar IntVar), ...}
atoms ∈ {(= 2 v),(= v ((_ iand 1) 1 v)), (= v ((_ iand 1) v v)) ,...}
skeleton = (assert (exists ☐ (and (= 2 v) (= v ((_ iand 1) v v)))))
```

(b) construction with the skeleton extracted from σ

```
(assert (exists ((v Int)) (and (= 2 v) (= v ((_ iand 1) v v)))))
(check-sat)
```

(c) mutant formula σ' simplified from cvc5 #8052

Fig. 3: Two formulas revealing different bugs in cvc5 and the construction process.

the patterns of formula combinations between bug-triggering formulas and benchmarks are essentially different.

In fact, the existence of special association patterns among atomic formulas in bug-triggering formulas is intuitive. According to the literature [15], SMT solvers are more error-prone in some specific logics (e.g., (QF_) NRA in Z3 and CVC4), and the signature we define implies the theories of formulas, whose combination leads to different logics. Therefore, under the guidance of those association rules, more test inputs conforming to those logics can be derived and the error-prone features will be validated more intensively.

Illustrative Example. We now use a concrete example to illustrate our observations and the insights of our approach. Specifically, we extract skeletons from historical bug-triggering formulas and then enumerate those skeletons to generate new formulas to test SMT solvers. Assuming we have obtained the formula σ from solvers' bug trackers in advance, we can extract its skeleton as shown in Figure 3a. Once the skeleton is available, we randomly select atomic formulas,

which can either be obtained from the same formula or different formulas, for each blank in the skeleton. Therefore, as shown in Figure 3b, we can first choose `(= 2 v)`, an expression composed of an integer variable and constant in σ, as the candidate for the blank connected by `and`. If we further know the rule that formulas whose signature is `(= IntConst IntVar)` always accompany the formulas whose skeleton is `(= IntVar IntVar)` in the bug-triggering formulas, as implied by σ, we will favor those atomic formulas of this form to fill the other blanks in the skeleton. Therefore, we can select the atomic formula `(= v ((_ iand 1) v v))`. To generate valid input, for skeletons containing quantifiers like this one, we can further select the variables contained in the corresponding sub-formulas as the bound variables (i.e., the variables bound to a specific formula by quantifiers). Ultimately, it can be instantiated as an SMT instance as the formula σ' in Figure 3c. Actually, the formula σ' generated by `HistFuzz` trigger a real bug in cvc5 (i.e., issue #8052). According to the bug-fixing modifications, we discover that these two issues #8016 and #8052, which are fixed on commits 1d4b79e and d036ca9, respectively, are related. They cater to two situations in terms of the rewrite rules (i.e.,'iand' operates on one constant and one variable as well as the case of two variables).

III. Approach

Motivated by our observations, we proposed an effective testing technique, `HistFuzz`, which adequately utilizes valuable historical information to test SMT solvers. Generally, the core insight of our approach is that the skeletons and associated atomic formulas extracted from the historical bug-triggering formulas are effective for detecting new bugs in SMT solvers. In the following, we will specify how to leverage our observations to fuzz the solvers.

A. Skeleton and Atomic Formula Extraction

Our approach starts with a set of historical bug-triggering formulas in SMT solvers' bug tracking systems. Particularly, `HistFuzz` automates the collection of formulas in the resolved issues of two well-known open-source SMT solvers, Z3 and cvc5. Both of them are highly mature SMT solvers and extensively tested by developers, users, and other fuzzers. Therefore, sufficient historical bug-triggering formulas can be easily obtained. Besides, the reason why we only consider the resolved issues is that the other issues are unconfirmed or unfixed by developers. Thus, the skeletons and atomic formulas extracted from the formulas of those unconfirmed or unfixed issues may still trigger the same bugs. In other words, disregarding the formulas exhibited in the pending reports can reduce the workload of deduplication.

1) Skeleton Extraction: With the historical bug-triggering formulas, `HistFuzz` first extracts the skeletons from these formulas and saves them as a configuration file. Specifically, for a quantifier-free formula, the sequence of connectives, including conjunction, disjunction, negation, implication, and exclusive or (i.e., *and*, *or*, *not*, *implies*, and *xor*) in the bug-triggering formulas are extracted and preserved as skeletons, while the connected atomic formulas are replaced by blanks. As for a quantified formula, the quantifiers, including *universal quantifier* and *existential quantifier* (i.e., *forall* and *exists*), are also retained. The corresponding bound variables are removed apart from the formula and prepared for subsequent mutation. Note that a bug-triggering input usually contains multiple `assert` commands. Consequently, the constraint in each `assert` command will be processed separately to extract the quantifier and connective sequence as a skeleton. Ultimately, all the skeletons in the bug-triggering formulas are saved as a configuration file for later use.

2) Association Rule Learning: With those skeletons, we can complete the generation of new formulas via enumerating atomic formulas. To better structure atomic formulas that are more likely to trigger bugs, we mine association rules among atomic formulas from historical formulas. Here, we leverage the *Apriori* algorithm [29] to perform association rule learning for those atomic formulas inside historical bug-triggering formulas. The *Apriori* algorithm can uncover the hidden relations that exist in the data. In particular, association rules are acquired by searching frequent item sets (i.e., relation patterns) in data and using a specific criterion under minimum *Support* and *Confidence* to define the important relationships. However, the atomic formulas in first-order logic formulas are diverse due to the different variables and constants, and thus it is difficult to discover strong rules among them. Consequently, we will abstract the atomic formulas inside the formulas as we describe in Section II. More specifically, an atomic formula is abstracted as a form of the operator as well as the sorts of operation objects. The abstract form of the atomic formulas in each formula constitutes a set, and these item sets will be analyzed by association rule learning. Finally, multiple association rules in the bug-triggering formulas can be obtained. The mined association rules will be also stored as an initial configuration file and used to guide the following enumeration process, which will be elaborated on later.

3) Formula Refactor: To generate valid test inputs, `HistFuzz` will first preprocess the collected bug-triggering formulas. Specifically, it renames the variables and functions in the bug-triggering formulas uniformly. For example, variable x is an integer in the collected bug-triggering formula f_a while it is a string in another formula f_b. `HistFuzz` will rename the former x to $var0$ and the latter to $var1$. This can avoid the variables or functions with the same names but different types, which could lead to generating invalid instances during mutation. Besides, we will exclude those redundant atomic formulas (i.e., completely the same formulas). Finally, these normalized atomic formulas will be saved as seed files along with their corresponding abstract forms obtained previously. In this way, we do not need to parse and process these formulas repeatedly during mutation, thus enhancing efficiency and throughput. Those processed atomic formulas will be the basic building blocks for generating new test formulas.

B. Skeleton-preserving Mutation

With the initialized configuration file and atomic formulas as mentioned above, `HistFuzz` can construct new formulas to test SMT solvers. The procedure `Fuzz` as shown in Algorithm 1 illustrates the main process of `HistFuzz`.

Algorithm 1: Main procedure of `HistFuzz`

```
1  Procedure Fuzz(Solvers)
2      Atoms ← ReadSeed()
3      SkeletonList, Rules ← ReadConfig()
4      Bugs ← [ ]
5      while no termination criterion met do
6          Skeleton ← RandomSelect(SkeletonList)
7          Formula ← Populate(Skeleton, Rules, Atoms)
8          if Detect(Formula, Solvers) then
9              Bugs ← Append(Bug, Formula)
10 Procedure Populate(Skeleton, Rules, Atoms)
11     AbstractForm ← [ ]
12     while isIncomplete(Skeleton) do
13         if MeetRule(AbstractForm, Rules) then
14             rule ← SelectRule(AbstractForm, Rules)
15             candidate ← Select(rule, Atoms)
16         else
17             candidate ← RandomSelect(Atoms)
18         AbstractForm ← Add(Abstract(candidate))
19         Skeleton ← Construct(Skeleton, candidate)
20     if isQuantified(Formula) then
21         Formula ← PopulateBoundVar(Formula)
22     return Skeleton
```

For this procedure, multiple solvers *Solvers* are pending to be validated. Then, `HistFuzz` will parse the configuration files and the seed file, which are acquired from historical bug-triggering formulas in the preprocessing phase (Lines 2 - 3). Concretely, *Atoms* can be regarded as a list that contains all of the processed atomic formulas with their corresponding abstract forms. The skeletons and association rules are also stored in *SkeletonList* and *Rules*, respectively. In addition, the formulas generated by `HistFuzz` that can reveal new potential bugs will be added to the list *Bugs*.

Subsequently, `HistFuzz` will continuously generate mutants unless users choose to interrupt. At the beginning, `HistFuzz` randomly selects skeletons from *SkeletonList* (Line 6). Next, the skeleton will be filled by the atomic formulas in *Atoms* under the guidance of the association rules in *Rules*. The constructed formulas will be fed to different solvers to detect crashes or correctness bugs via cross-checking (Line 8).

For the procedure `Populate`, it uses the given skeleton and atomic formulas to generate mutants. The list *AbstractForm* will store the abstract forms of the filled atomic formulas. The while loop will fill all the blanks connected by logical connectives in the skeleton. First, before selecting atomic formulas to construct, this procedure determines whether the abstract form of the currently filled formula conforms to a certain rule's antecedent in *Rules* (Line 13). If the antecedent of any rule is met, the formula with the abstract form of the corresponding consequent will be selected according to it (Line 15). Note that, once a rule has been considered, it will be ignored for the subsequent substantiation of the currently selected skeleton. As a consequence, it avoids superfluous atomic formulas with the same abstract form to be filled into the same skeleton. Conversely, if no rule matches, a random candidate formula will be chosen (Line 17).

These candidate formulas will be filled into the skeleton until all the blanks connected by logical connectives are complete. Since all the seeds have been processed before the construction, there is no variable with the same name but different sorts. Note that, in order to generate more diverse mutants, `HistFuzz` will perform *variable translocation* for the constructed formulas. Concretely, variables of the same type in different atomic formulas can be substituted for each other. In this way, a finer-grained mutation is achieved, and thus the diversity of the generated formulas is strengthened. In addition, if the skeleton contains quantifiers, further manipulation of it is required. To be specific, the free variables in the sub-formulas will be randomly selected and transformed to bound variables specified by the quantifiers (Line 21), or the bound variables that do not exist in the sub-formula can be generated arbitrarily from scratch. This design is motivated by our preliminary study (e.g., formula φ in Figure 3a), which illustrates that even if a bound variable does not appear in the sub-formulas, it can affect the triggering of a bug.

C. Implementation

We implemented `HistFuzz` in 2,711 lines of Python code. In our implementation, the following aspects have been carefully handled.

Formula Collection. For most cases, `HistFuzz` can acquire formulas from the bug tracking systems of SMT solvers automatically using the GitHub API. First, `HistFuzz` collects the issues that report soundness bugs, invalid model bugs, and crashes, which can be inferred by the titles of issues. Then, it gathers the formulas from the issue bodies or comments based on predefined matching rules, since most formulas are in the fenced code blocks of Markdown. However, bug reports are not always in a uniform format, and thus `HistFuzz` cannot easily access certain bug-triggering formulas. For instance, some reports may not explicitly present the formula that caused the bug, so these formulas will be ignored. Finally, we obtain 2,193 and 705 potential bug-triggering formulas for Z3 and cvc5, respectively. `HistFuzz` automates the creation and updating of configuration files based on the collected formulas.

Skeletons. Totally, we extract 963 skeletons from the collected bug-triggering formulas. In the generation process of `HistFuzz`, more than one skeleton can be selected simultaneously and filled as assertions to construct an input instance. To determine the number of skeletons each mutant contains, we also investigated and counted the number of `assert` commands in the bug-triggering formulas. We found that about 80% of instances contain no more than five `assert` commands. Besides, on average, each instance contains 2.8

assert commands. Therefore, we randomly select one to five skeletons at a time to construct the input, and most of the generated instances are composed of three assert commands. This ensures the mutants generated by HistFuzz generally small, which facilitates the reduction before reporting as well as debugging by developers.

Association Rules. As previously elaborated, we utilize the *Apriori* Algorithm to learn the association between different abstract atomic formulas. Obviously, the rules with higher values of *Support* and *Confidence* are more significant. However, a higher *Support* or *Confidence* value will also reduce the number of the compliant rules. Therefore, we need to determine an appropriate threshold. In our implementation, those associated atomic formulas whose *Confidence* is greater than 0.5, and the item sets that appear more than nine times are considered as an association rule. Eventually, 108 association rules are mined to guide the mutation.

IV. Empirical Evaluation

In this section, we conduct extensive evaluations to investigate the effectiveness of our tool. Specifically, the experiments aim to answer the following research questions:

- **RQ1:** How many bugs can HistFuzz find? (Section IV-B)
- **RQ2:** Does HistFuzz improve on the state of the arts? (Section IV-C)
- **RQ3:** How is the effectiveness of HistFuzz's major components? (Section IV-D)

A. Evaluation Setup

Bug Types. In order to ensure the reliability and robustness of SMT solvers, in our evaluation, we mainly aim at the following three kinds of bugs as prior work does [13], [15], [16], [31].

• Soundness bugs: If two solvers give different results (i.e., one yields sat while the other yields unsat) for a formula, it is regarded as a soundness bug.

• Invalid model bugs: A solver throws sat correctly, but gives an unsatisfiable model for a formula.

• Crashes: When solving a formula, a solver aborts exceptionally, e.g., assertion violations and segmentation faults.

Targeted Solvers. We select Z3 and cvc5 for our experimental evaluation for the following reasons. First, they are widely used in many popular frameworks (e.g., KLEE can enable Z3 support), which are of significance. Second, Z3 and cvc5 support various logics extensively and their superior performance in each logic has been widely witnessed. Finally, as open-source software, Z3 and cvc5 have become increasingly sound thanks to the continuous efforts made by both developers and researchers. Consequently, they are not only suitable for evaluating our methodology, but it is also significantly challenging to detect their potential bugs. Besides, the developing teams of Z3 and cvc5 are also active, and thus they can deal with the reported issues and provide us feedbacks timely.

Selected Options. We mainly focus on spotting the bugs in the default mode of SMT solvers. The reason is that the default option is the most commonly used, and the combination of too many options is neither general nor meaningful. However, some options are necessary. For example, the "model_validate=true" in Z3 and "–check-models" in cvc5 are enabled to detect invalid model bugs, and "–incremental" in cvc5 should be used to support incremental constraint solving. Those options are considered to be part of the default mode of the solvers [15]. In addition, some important options are also tested by HistFuzz, e.g., Z3's new core ("tactic.default_tactic=smt, sat.euf=true"), which is supposed to become the default mode of Z3 as it matures [17]. Moreover, the developers of cvc5 provide a list of options that deserves to be tested, which serves for proper guidelines for fuzzing [32]. Thereby, we also conduct stress-testing on those options in our experiments.

Bug Inspection and Reduction. When two solvers produce inconsistent results (i.e., a potential soundness bug), we will manually inspect them to identify which solver is buggy before reporting. Concretely, assuming Z3 returns *sat* while cvc5 returns *unsat* for the same formula f, we can utilize Z3 to obtain a model m for f. Then, we substitute m into f to check whether f is indeed satisfiable. In this way, we can identify the buggy solver. For invalid model bugs and crashes, we can determine the buggy solver directly based on their outputs.

Additionally, it is necessary to de-duplicate and reduce the bug-triggering instances, which makes it easier for developers to check, confirm and fix the bugs. To simplify the bug-triggering formulas, we employ ddSMT [33], a delta debugger that is tailored for the SMT-LIB2 language and recommended by developers. When crashes are revealed, we will consider the crashes pointing to the same file and code line as the same bug and search the exception information thrown by the solvers in the bug tracking systems to de-duplicate. For soundness bugs and invalid model bugs, we will report one reduced instance at a time according to their theory categories following prior work [13]. If there are multiple formulas with the same theory that reveal the same kind of bug, we will report one of them first. After it has been fixed, we check if the remaining formulas can still trigger bugs and determine whether to report them or not. Otherwise, if the report has not been addressed, we will group the possible relevant bug reports as a consolidated issue, as a developer suggested. In this way, most of the duplicate reports can be avoided, thus reducing the workload of developers.

Environment. Our tool tested SMT solvers on a machine with an Intel Xeon CPU Gold-6230 processor (40 cores and 128GB RAM). The operating system on the machine is Ubuntu 20.04 64-bit. We build Z3 and cvc5 with AddressSantizer [34] to facilitate identifying and de-duplicating crashes. Besides, in line with prior work [16], we set the time limit of solving query for each test formula to 10 seconds in our experiments.

B. Bug Detection

We extensively tested Z3 and cvc5 with HistFuzz from December 2021 to August 2022. To ensure that all detected bugs are new, we always run HistFuzz on the latest commits of the solvers for testing.

Bug Count. Overall, HistFuzz produces more than 4,000 instances that trigger potential bugs. Based on the aforementioned bug deduplication principle, we reported 149 bug issues for Z3 and cvc5 in total, of which 111 bugs have been confirmed or fixed by developers as shown in Table II. Note that all the confirmed bugs are previously unknown and cannot be triggered by any historical bug-triggering formulas. Of all the confirmed bugs, most (72 out of 111) of them are crashes. In addition, HistFuzz exposes 9 confirmed soundness bugs in Z3 and 6 soundness bugs in cvc5, which are the most significant bugs in SMT solvers. The bugs that the developers claimed "won't fix" are mainly due to the lack of bandwidth to address them currently.

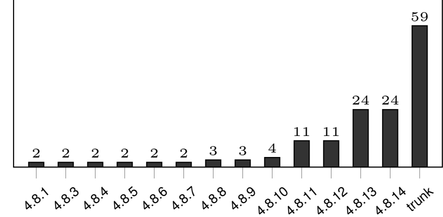
Of all the confirmed bugs, around half (55 out of 111) of them are related to the default mode of the solvers. Particularly, 8 soundness bugs (5 in Z3 and 3 in cvc5) and 14 invalid model bugs (6 in Z3 and 8 in cvc5) are detected in the default mode of the solvers, which are the most critical kinds of bugs for SMT solvers. The rest are almost associated to the new core option of Z3 (26 out of 111) and the individual options suggested by the developers to fuzz in cvc5 (17 out of 111). Therefore, the significance of the bugs detected by HistFuzz is remarkable. In addition, we also investigate the logic distribution of those confirmed bugs. We find that the bugs discovered by HistFuzz touch a number of logics. Specifically, the Top-3 logics of the confirmed bugs in Z3 are QF_SLIA (8/59), LRA (5/59), and QF_BV (5/59). As for cvc5, most of the confirmed bugs are in the QF_NRAT (10/52) followed by the QF_SLIA (6/52) and the NIA (4/52).

Significance. Many of the bugs found by our tool have attracted a lot of discussion of developers. For example, a soundness bug of Z3 detected by HistFuzz is separately maintained by the developers and serves as the "starting assumption" to inspire them to reconsider the code logic in the solver.[2] In addition, to better understand the significance of the identified bugs, we conduct a study on how the historical releases of Z3 and cvc5 are affected by those confirmed bugs found by HistFuzz. Specifically, we feed each of the formulas to a range of Z3 and cvc5's official released versions and observed whether abnormal results occur. We choose the official released versions from 4.8.1 (released on Oct 16, 2018) to 4.8.14 (released on Dec 24, 2021) for Z3, which was released when we started our bug hunting campaign. Similarly, with respect to cvc5, including its predecessor CVC4, we select the versions from CVC4-1.7 (released on Apr 4, 2019) to cvc5-0.0.5 (released on Jan 8, 2022). These versions have been extensively tested by different tools, such as YinYang, STORM, OpFuzz, and TypeFuzz. As Figure 4 shows, we found two bugs that originally occurred at version 4.8.1 of Z3. In other words, they escaped other fuzzers' detection and thus have been latent in Z3 for more than three years. In addition, one soundness bug appear at version 4.8.8 which was released in Apr 2019, which means it has also lurked for about two years. For cvc5/CVC4, four bugs were introduced at CVC4's version 1.7, which have also been latent for about three years. Such results illustrate that HistFuzz is capable of finding not only new bugs but also those that are missed by regression tests and other fuzzers.

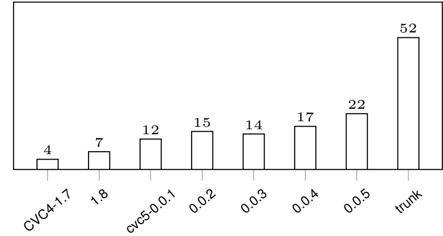
[2] https://github.com/Z3Prover/z3/issues/6078

TABLE II: Bugs found by HistFuzz in Z3 and cvc5. *S* denotes soundness bug, *I* is invalid model bug, and *C* is crash.

Status	**Z3**			**cvc5**			**Total**
	S	*I*	*C*	*S*	*I*	*C*	
Reported	17	18	55	6	14	39	149
Confirmed	9	11	39	6	13	33	111
Fixed	8	11	38	6	13	32	108
Duplicate	4	0	3	0	0	4	11
Won't fix	2	2	10	0	0	0	14



(a) Z3



(b) cvc5

Fig. 4: Confirmed bugs that affect the corresponding release versions of Z3 and cvc5.

C. Improvement on the state of the arts

In this RQ, we investigate whether our approach advances the existing techniques, including STORM, YinYang, OpFuzz, and TypeFuzz. The results can reflect the effectiveness of our proposed strategy of skeleton enumeration with atomic formulas compared with conventional mutation strategies as adopted by other fuzzers. Here, we mainly focus on the comparison in terms of the bug-finding capability and achieved code coverage of different testing tools.

*1) **Bug-finding Capability**:*

To evaluate HistFuzz's bug-finding capability, we re-run HistFuzz and the baselines. Since most bugs found by the baselines have been fixed, it is unfair to use the latest version to compare HistFuzz with baselines. Thus, we select Z3 4.8.5 and CVC4 1.7 as the subjects, which were released more than three years ago. At that time, little research attentions have been paid on testing SMT solvers, and the baselines have found

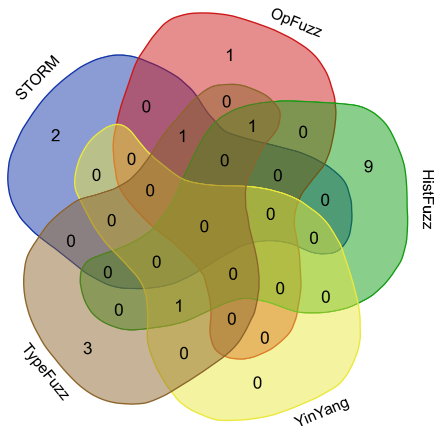


Fig. 5: The distribution of bugs detected by `HistFuzz` and the baselines on previous versions of solvers.

many bugs for these two releases and subsequent versions. To have a fair comparison, `HistFuzz` only uses the historical bug-triggering formulas in the issues resolved before the time these two versions are released. As a result, 139 historical bug-triggering formulas (116 in Z3 and 23 in cvc5) are processed and 56 skeletons are extracted from them by `HistFuzz`. For the baselines, we also utilize them to mutate those formulas to detect bugs. Particularly, since YinYang needs to obtain the satisfiability as ground truth, we classify these formulas according to their satisfiability in advance. Note that we also conduct this experiment on the default mode of SMT solvers and use the default configurations of the testing tools. We run each tool for 120 hours, which follows the existing fuzzing study [35]. Then we will examine the number of known unique bugs. Specifically, for crashes, we identify the unique bugs by examining the exception messages as adopted in Section IV-A. With respect to the correctness bugs, we leverage the Correcting Commit technique [36] to check the unique ones. Concretely, we employ binary search to find which commit in the solvers corrects the bug. Bugs fixed on different commits are considered as different bugs.

Figure 5 shows the results. According to the results, we can find that `HistFuzz` still identifies the most bugs, whose number is 11 in total, compared to other tools. In addition, we observe that there are few overlaps between the bugs found by `HistFuzz` and other tools. For example, `HistFuzz` and TypeFuzz contain the most overlaps, whose number is 2, and there is no overlap between the bugs found by `HistFuzz` and STORM. It illustrates that `HistFuzz` is rather complementary to the existing techniques and also exhibits a stronger bug-finding capability to a certain extent.

*2) **Code Coverage**:*

Experiment Setting. In this experiment, we sample 100 formulas at random, consistent with previous work [17], from the aforementioned bug-triggering formulas we collected as the baseline seeds. We feed those formulas to Z3 and cvc5 and observe the code coverage achieved by them, respectively.

TABLE III: Code coverage achieved by bug-triggering formulas and the mutants derived from different fuzzers. The highest coverage is highlighted. The coverage achieved by `HistFuzz` with significant difference is bolded.

	Z3		**cvc5**	
	line	*function*	*line*	*function*
Seeds	26.6%	29.8%	28.3%	43.3%
OpFuzz	27.5%	30.5%	29.7%	45.4%
TypeFuzz	27.2%	30.2%	28.6%	43.5%
STORM	27.4%	30.8%	28.8%	43.7%
YinYang	27.6%	30.5%	28.6%	43.8%
`HistFuzz`$_n$	30.9%	32.1%	29.3%	44.4%
`HistFuzz`$_r$	32.5%	33.7%	30.3%	45.7%
`HistFuzz`	**33.9%**	**35.3%**	**31.2%**	**47.3%**

Then, `HistFuzz` will mine skeletons and atomic formulas from those selected baseline seeds while the association rules are still those learned from the whole bug-triggering formulas since the selected seeds are too few to learn. After that, we employ OpFuzz, TypeFuzz, STORM, and YinYang to mutate those selected formulas. All of them are open-source fuzzers that can support multiple logics and detect a large number of bugs in SMT solvers. We use the latest versions of those fuzzers provided by the developers to conduct the experiment. The number of mutants for each seed is set to 10 following the existing study [17], and thus each baseline will generate 1,000 formulas in total. Therefore, `HistFuzz` also utilizes the mined skeletons and atomic formulas to generate 1,000 instances to conduct fair comparison. Note that we will feed the solvers the mutants generated by each of the five tools together with the 100 selected baseline seeds and use gcov [37] to obtain the code coverage, respectively. Here, we choose Z3 4.8.14 and cvc5 0.0.5 as the solvers under test. To reduce the impact of randomness, we repeat the mutation of different tools 20 times and take the average as the final result.

Result. Table III presents the results we obtained (be noted that `HistFuzz`$_n$ and `HistFuzz`$_r$ in the Table are the variants of `HistFuzz`, which will be introduced in RQ3). According to the results, we observe that the mutants generated by `HistFuzz` can further cover more code compared to the baseline seeds (+7.3% line coverage in Z3 and +2.9% line coverage in cvc5), which is also the best among all the existing techniques. Besides, we further perform the Mann-Whitney U Test [38] to examine whether the superiority of `HistFuzz` over the others is significant. The coverage achieved by `HistFuzz` with the significant improvement compared to all the baselines is **bolded**. Therefore, we can deduce that `HistFuzz` is effective in fuzzing SMT solvers in terms of achieved code coverage, and also outperforms existing state-of-the-art fuzzers significantly.

Overall, we can observe that the code coverage of solvers that the fuzzers can achieve is limited, which are generally less than 40%. The reasons lie in the following aspects. First, considering

that Z3 and cvc5 are very large systems that contain over 400k and 200k lines of code respectively, even 1% increase will lead to thousands of extra code lines being covered. Second, we conduct this experiment on the default mode of solvers (i.e., no extra options applied), which is the most commonly used by users, and thus many features of the solvers are not enabled, which prevents substantial code from being covered. In addition, we also study the code coverage achieved by the formulas in the SMT-LIB official benchmarks. We find the line coverage reached by 100 formulas randomly sampled from SMT-LIB benchmarks (20.4% in Z3 and 24.7% in cvc5) is less than that of bug-triggering formulas. Furthermore, we also tried to utilize the default mode of the solvers to solve around 400,000 instances in the benchmarks, and found that they cannot achieve more than 40% line coverage (37.0% in Z3 and 37.6% in cvc5), which reveals the upperbound of code coverage under the default mode. In general, we can conclude that bug-triggering formulas are of substantial value, and `HistFuzz` can exploit them to achieve more code coverage compared to the state of the arts, which is beneficial for thoroughly validating the SMT solvers.

To conclude, in the experiments, we discover that `HistFuzz` improve on the state-of-the-art fuzzing techniques in terms of bug-finding capability and code coverage.

D. Sensitivity Analysis

We are also curious towards the contribution of *historical skeleton enumeration* and *association rule mining*, which are the two major components of our devised approach. Therefore, we design the following two variants of `HistFuzz`.

- **$\texttt{HistFuzz}_n$**: it utilizes the skeletons mined from normal formulas instead of bug-triggering inputs and substantiate them with randomly selected atomic formulas derived from bug-triggering inputs.
- **$\texttt{HistFuzz}_r$**: it utilizes the mined skeletons from bug-triggering inputs while instantiating them with randomly selected atomic formulas derived from bug-triggering inputs.

When comparing the variants, `HistFuzz` will construct atomic formulas under the guidance of association rules. Therefore, by comparing $\texttt{HistFuzz}_n$ and $\texttt{HistFuzz}_r$, we can understand the effectiveness of skeletons mined from bug-triggering inputs. Besides, the effectiveness of association rules can be manifested by comparing $\texttt{HistFuzz}_r$ and `HistFuzz`.

Therefore, we utilize the two variants $\texttt{HistFuzz}_n$ and $\texttt{HistFuzz}_r$ to detect bugs in Z3 4.8.14 and cvc5 0.0.5, on which `HistFuzz` has unveiled the most bugs. Note that the relevant issues of all the 2,898 bug-triggering inputs we initially collect had been resolved before the versions were released. Next, we sample the same number of normal formulas from SMT-LIB benchmarks at random and use $\texttt{HistFuzz}_n$ to extract their skeletons. After that, we run the two variants and `HistFuzz` for 120 hours over the default mode of solvers. Finally, we examine the number of known unique bugs utilizing the Correcting Commit technique [36] as adopted in RQ2.

Ultimately, we find $\texttt{HistFuzz}_n$ identifies 7 bugs in the solvers. $\texttt{HistFuzz}_r$ outperforms it and reveals 9 bugs, which illustrates the effectiveness of historical skeletons, while `HistFuzz` can find the most bugs, whose number is 10. In addition, we find `HistFuzz` , which is under the direction of association rules, is more efficient compared to $\texttt{HistFuzz}_r$. Concretely, most of the bugs (6/10) detected by `HistFuzz` are uncovered within two hours, but $\texttt{HistFuzz}_r$ can only find 3 bugs in two hours. Thereby, we deduce that both of the major components of `HistFuzz` enhance its bug-finding ability, which is demonstrated in effectiveness and efficiency.

Furthermore, we also investigate the code coverage achieved by the variants in this experiment. The setup is the same as RQ2 except that the variant $\texttt{HistFuzz}_n$ extract skeletons from 100 normal formulas sampled from SMT-LIB benchmarks instead of the bug-triggering inputs. The results are shown in Table III. We can find that $\texttt{HistFuzz}_n$ can achieve more code coverage than the baselines. Nevertheless, via utilizing the skeletons from bug-triggering inputs, $\texttt{HistFuzz}_s$ can reach higher code coverage. Overall, we can conclude that both of `HistFuzz`'s components manifest the effectiveness, whether in terms of detecting bugs or improving code coverage.

V. Discussion

Generality. Bug-triggering inputs, including skeletons and atomic formulas, used by `HistFuzz` are derived from various SMT solvers and employed together. However, it is unclear whether the bug-triggering inputs of `HistFuzz` from one SMT solver are also effective in triggering bugs for other SMT solvers. To this end, we utilize `HistFuzz` to extract the skeletons, atomic formulas, and association rules from the bug-triggering inputs of Z3, denoted as $\texttt{HistFuzz}_{Z3}$, and cvc5, denoted as $\texttt{HistFuzz}_{cvc5}$, respectively. After that, we run them over Z3 4.8.14 and cvc5 0.0.5 respectively for 120 hours and check the known unique bugs found by them as RQ3. Ultimately, we find that $\texttt{HistFuzz}_{Z3}$ identifies 4 bugs in Z3 and 3 bugs in cvc5, while $\texttt{HistFuzz}_{cvc5}$ uncovers 1 bug in Z3 and 5 in cvc5. In other words, 3 Z3 bugs are exposed by $\texttt{HistFuzz}_{Z3}$ and 1 cvc5 bug is exposed by $\texttt{HistFuzz}_{cvc5}$. Thus, we deduce that `HistFuzz` is of generality to a certain extent but its performance is better on the originating solvers.

Threats to Validity. This study suffers from the following main threats. First, `HistFuzz` missed some bug-triggering formulas. As mentioned before, most of the reports are standard and easily accessible. However, some reports are hard to be obtained. For example, some issues may contain several zip files, so it is not easy to determine if these are the formulas we are interested in. Fortunately, there are only a very small number of reports in this case. Second, since our approach is based on historical data, many of the formulas generated by `HistFuzz` may be revealing duplicate bugs. Although we only collect the formulas in the resolved issues and have carefully de-duplicated the formulas before reporting, we still submitted some duplicate reports. Third, `HistFuzz` may generate invalid test formulas since historical bug-triggering formulas can be

also ill-formed. However, invalid inputs are still meaningful for fuzzing SMT solver as the developers state [39], which can be used to validate whether solvers can handle invalid inputs correctly. Furthermore, the ill-formed formulas in historical bug-triggering inputs can be easily filtered out by feeding them to solvers in advance, and thus this threat can be mitigated.

Ethical Considerations. To avoid flooding issues with low priorities to issue trackers, we mainly tested the solvers in default mode (i.e., with no extra option) or limited significant options and reported the relevant issues, which is also mentioned in Section IV. Besides, we perform bug reduction and deduplication to our best before reporting, which facilitates developers to debug. Furthermore, we try to contribute more to the community, e.g., submitting some bug-triggering instances generated by `HistFuzz` to the regression test suite of Z3, which are also appreciated by the developer.[3]

Future Work. In this work, we focus on the historical data in several SMT solvers' bug tracking systems, especially the bug-triggering formulas, and utilize them to better fuzz SMT solvers. In fact, there have many other SMT solvers and their bug-triggering formulas can be also used. Besides, there are many bug-triggering formulas reported every day and this information can be also used in the future. Therefore, one possible direction is to further exploit more historical data to test the solvers more effectively.

VI. Related Work

SMT Solver Fuzzing. In recent years, SAT/SMT solvers have been widely used in research to analyze programs and solve constraints [40], [41], [42], [43]. Therefore, the correctness and robustness of SMT solvers become more significant and attract the interest of researchers. Initially, a grammar-based blackbox fuzzing tool FuzzSMT [44] was designed for validating SMT solvers. Winterer et al. [13] use the method of semantic fusion to synthesize two formulas and obtain satisfiability-preserving formulas. However, the logics applicable to these tools is not comprehensive. STORM [14] obtains satisfiable formulas with the Boolean structure different from the original seed through construction. OpFuzz [15] tests the SMT solvers via simply mutating operators with the same properties and achieves satisfactory results. Sparrow [16] leverages the idea of approximations to mutate the operators and generate formulas with test oracles. More recently, Typefuzz [17] improve OpFuzz to combine mutation and generation to yield more diverse mutants. However, different from all of the above techniques, we concentrate on historical data and utilize them to validate SMT solvers effectively.

Mutation-based Testing and Fuzzing. Mutation testing is a widely-used technique for software testing and helpful for improving quality of software [45]. Researchers have proposed many mutation-based testing techniques for different software and languages [46], [47], [48], [49], [50], [51], [52], [53]. Besides, fuzzing is one of the most popular testing techniques, which is of good extensibility and easy to deploy. Mutation-based fuzzing technique generates test cases via mutating valid initial seeds,e.g., American fuzzy lop (AFL) [54]. More relevant, Zhang et al. [55] proposed Skeletal Program Enumeration (SPE) to test compilers [56]. Then, both Sparrow [16] and TypeFuzz [17] refer to the concept of skeleton for SMT solver testing. However, they focus on different points. Sparrow mainly mutates the operators and constants in formulas based on approximation relations to generate mutants with test oracles. For TypeFuzz, it generates new expressions to replace expressions of the same type in seeds while keeping the overall structure of the formula unchanged. Unlike both, we find that the skeletons play a critically significant role in the historical bug-triggering formulas. Consequently, we directly extract the skeletons from them and perform mutation, which achieves inspiring effects as well.

[3] https://github.com/Z3Prover/z3test/pull/47

VII. Conclusion

In the study, we introduce a novel method that leverages historical bug-triggering inputs to validate SMT solvers. Specifically, we first extract valuable skeletons from the collected bug-triggering formulas and enumerate each skeleton with the associated atomic formulas to generate effective new formulas to test SMT solvers. We have implemented our approach as a practical fuzzing tool `HistFuzz`. To evaluate the effectiveness of `HistFuzz`, we conduct a bug hunting campaign for exposing real bugs in Z3 and cvc5, two well-known SMT solvers. `HistFuzz` has detected 111 confirmed bugs in Z3 and cvc5, of which 108 have been fixed by developers. Note that a number of bugs uncovered by `HistFuzz` are critical or long-latent bugs in the SMT solvers. In addition, `HistFuzz` outperforms the state-of-the-art fuzzers in terms of code coverage and effectiveness. Our experimental results also demonstrate that it is promising for SMT solver testing to mine historical information.

Acknowledgement

We would like to extend our heartfelt gratitude to the developers of Z3 and cvc5, especially Nikolaj Bjørner, Christoph M. Wintersteiger, Andrew Reynolds, Andres Noetzli, Aina Niemetz, Gereon Kremer, and Martin Brain, for their valuable information and efforts to address the bug reports we submitted. We are also grateful to all the anonymous reviewers for their insightful feedback. Furthermore, we would like to express our appreciation to the authors of YinYang and STORM for generously sharing their artifacts. This work is supported by the Hubei Province Key R&D Technology Special Innovation Project under Grant No. 2021BAA032, the National Natural Science Foundation of China (Grant No. 62002125, No. 62072194, No. 62172205), the CCF-Huawei Populus euphratica Innovation Research Funding and the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001).

References

[1] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking*. Springer, 2018, pp. 305–343.

[2] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, "Symbolic optimization with SMT solvers," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 607–618. [Online]. Available: https://doi.org/10.1145/2535838.2535857

[3] M. Y. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Smt-based refutation of spurious bug reports in the clang static analyzer," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 11–14. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00026

[4] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Practical constraint solving for generating system test data," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 2, pp. 11:1–11:48, 2020. [Online]. Available: https://doi.org/10.1145/3381032

[5] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. S. Pasareanu, "Nnrepair: Constraint-based repair of neural network classifiers," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12759. Springer, 2021, pp. 3–25. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_1

[6] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 14:1–14:27, 2021. [Online]. Available: https://doi.org/10.1145/3418461

[7] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 419–429. [Online]. Available: https://doi.org/10.1145/3238147.3238225

[8] Y. Luo, A. Filieri, and Y. Zhou, "Symbolic parallel adaptive importance sampling for probabilistic program analysis," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 1166–1177. [Online]. Available: https://doi.org/10.1145/3468264.3468593

[9] E. Sherman and A. Harris, "Accurate string constraints solution counting with weighted automata," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 440–452. [Online]. Available: https://doi.org/10.1109/ASE.2019.00049

[10] W. Grieskamp, X. Qu, X. Wei, N. Kicillof, and M. B. Cohen, "Interaction coverage meets path coverage by SMT constraint solving," in *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings*, 2009, pp. 97–112. [Online]. Available: https://doi.org/10.1007/978-3-642-05031-2_7

[11] E. Kang, S. Lafortune, and S. Tripakis, "Automated synthesis of secure platform mappings," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11561. Springer, 2019, pp. 219–237. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_12

[12] J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand, "Search-driven string constraint solving for vulnerability detection," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 198–208. [Online]. Available: https://doi.org/10.1109/ICSE.2017.26

[13] D. Winterer, C. Zhang, and Z. Su, "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*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 718–730. [Online]. Available: https://doi.org/10.1145/3385412.3385985

[14] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in smt solvers using blackbox mutational fuzzing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 701–712.

[15] D. Winterer, C. Zhang, and Z. Su, "On the unusual effectiveness of type-aware operator mutations for testing SMT solvers," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 193:1–193:25, 2020. [Online]. Available: https://doi.org/10.1145/3428261

[16] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "Skeletal approximation enumeration for SMT solver testing," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 1141–1153. [Online]. Available: https://doi.org/10.1145/3468264.3468540

[17] J. Park, D. Winterer, C. Zhang, and Z. Su, "Generative type-aware mutation for testing SMT solvers," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–19, 2021. [Online]. Available: https://doi.org/10.1145/3485529

[18] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

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

[20] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

[21] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for jvm testing," in *44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 21-29, 2022*. IEEE, 2022. [Online]. Available: https://doi.org/10.1145/3510003.3510059

[22] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "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, 2019, pp. 305–316. [Online]. Available: https://doi.org/10.1109/ASE.2019.00037

[23] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2613–2630. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung

[24] H. Zhong, "Enriching compiler testing with real program from bug report," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 40:1–40:12. [Online]. Available: https://doi.org/10.1145/3551349.3556894

[25] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, 2015, pp. 39–48. [Online]. Available: https://doi.org/10.1109/ICSE.2015.133

[26] L. Moonen, S. Di Alesio, D. W. Binkley, and T. Rolfsnes, "Practical guidelines for change recommendation using association rule mining," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 732–743. [Online]. Available: https://doi.org/10.1145/2970276.2970327

[27] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 64:1–64:20, 2017. [Online]. Available: https://doi.org/10.1145/3133888

[28] C. Barrett, P. Fontaine, and C. Tinelli. (2022) The satisfiability modulo theories library (SMT-LIB). [Online]. Available: http://smtlib.cs.uiowa.edu/

[29] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in *VLDB’94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 1994, pp. 487–499. [Online]. Available: http://www.vldb.org/conf/1994/P487.PDF

[30] SMT-LIB. (2021) SMT-LIB Benchmarks. [Online]. Available: http://smtlib.cs.uiowa.edu/benchmarks.shtml

[31] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. 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, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 322–335. [Online]. Available: https://doi.org/10.1145/3460319.3464803

[32] cvc5. (2022) cvc5 fuzzing guidelines. [Online]. Available: https://github.com/cvc5/cvc5/wiki/Fuzzing-cvc5

[33] G. Kremer, A. Niemetz, and M. Preiner, “ddsmt 2.0: Better delta debugging for the smt-libv2 language and friends,” in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12760. Springer, 2021, pp. 231–242. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_11

[34] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker,” in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[35] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, “BEACON: directed grey-box fuzzing with provable path pruning,” in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 36–50. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833751

[36] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, “An empirical comparison of compiler testing techniques,” in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 180–190. [Online]. Available: https://doi.org/10.1145/2884781.2884878

[37] “Using the gnu compiler collection:gcov,” 2021.

[38] H. B. Mann and D. R. Whitney, “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other,” *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50 – 60, 1947. [Online]. Available: https://doi.org/10.1214/aoms/1177730491

[39] A. Niemetz, M. Preiner, and C. W. Barrett, “Murxla: A modular and highly extensible API fuzzer for SMT solvers,” in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Shoham and Y. Vizel, Eds., vol. 13372. Springer, 2022, pp. 92–106. [Online]. Available: https://doi.org/10.1007/978-3-031-13188-2_5

[40] P. Tolmach, Y. Li, S. Lin, Y. Liu, and Z. Li, “A survey of smart contract formal specification and verification,” *ACM Comput. Surv.*, vol. 54, no. 7, pp. 148:1–148:38, 2022. [Online]. Available: https://doi.org/10.1145/3464421

[41] R. Tzoref and O. Grumberg, “Automatic refinement and vacuity detection for symbolic trajectory evaluation,” in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 190–204. [Online]. Available: https://doi.org/10.1007/11817963_20

[42] R. Degiovanni, D. Alrajeh, N. Aguirre, and S. Uchitel, “Automated goal operationalisation based on interpolation and SAT solving,” in *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 129–139. [Online]. Available: https://doi.org/10.1145/2568225.2568323

[43] M. Mendonça, A. Wasowski, and K. Czarnecki, “Sat-based analysis of feature models is easy,” in *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, ser. ACM International Conference Proceeding Series, D. Muthig and J. D. McGregor, Eds., vol. 446. ACM, 2009, pp. 231–240. [Online]. Available: https://dl.acm.org/citation.cfm?id=1753267

[44] R. Brummayer and A. Biere, “Fuzzing and delta-debugging smt solvers,” in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, 2009, pp. 1–5.

[45] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just, “Does mutation testing improve testing practices?” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 910–921. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00087

[46] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon, “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*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 329–340. [Online]. Available: https://doi.org/10.1145/3293882.3330576

[47] R. Jabbarvand and S. Malek, “μdroid: an energy-aware mutation testing framework for android,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 208–219. [Online]. Available: https://doi.org/10.1145/3106237.3106244

[48] Q. Zhu, A. Panichella, and A. Zaidman, “An investigation of compression techniques to speed up mutation testing,” in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 274–284. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICST.2018.00035

[49] A. S. Ami, K. Kafle, A. Nadkarni, D. Poshyvanyk, and K. Moran, “μse: Mutation-based evaluation of security-focused static analysis tools for android,” in *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 53–56. [Online]. Available: https://doi.org/10.1109/ICSE-Companion52605.2021.00034

[50] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, “Predictive mutation testing,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 342–353. [Online]. Available: https://doi.org/10.1145/2931037.2931038

[51] D. Fortunato, J. Campos, and R. Abreu, “Qmutpy: a mutation testing tool for quantum algorithms and applications in qiskit,” in *ISSTA ’22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 797–800. [Online]. Available: https://doi.org/10.1145/3533767.3543296

[52] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney, “Applying mutation analysis on kernel test suites: An experience report,” in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, 2017, pp. 110–115. [Online]. Available: https://doi.org/10.1109/ICSTW.2017.26

[53] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti, “XACMUT: XACML 2.0 mutants generator,” in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 28–33. [Online]. Available: https://doi.org/10.1109/ICSTW.2013.11

[54] M. Zalewski. (2022) American fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl

[55] Q. Zhang, C. Sun, and Z. Su, “Skeletal program enumeration for rigorous compiler testing,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, A. Cohen and M. T. Vechev, Eds. ACM, 2017, pp. 347–361. [Online]. Available: https://doi.org/10.1145/3062341.3062379

[56] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, “A survey of compiler testing,” *ACM Comput. Surv.*, vol. 53, no. 1, pp. 4:1–4:36, 2020. [Online]. Available: https://doi.org/10.1145/3363562