Demystifying Performance Regressions in String Solvers

Yao Zhang, Xiaofei Xie, Yi Li, Yun Lin, Sen Chen, Yang Liu, and Xiaohong Li

Abstract—Over the past few years, SMT string solvers have found their applications in an increasing number of domains, such as program analyses in mobile and Web applications, which require the ability to reason about string values. A series of research has been carried out to find quality issues of string solvers in terms of its correctness and performance. Yet, none of them has considered the performance regressions happening across multiple versions of a string solver. To fill this gap, in this paper, we focus on solver performance regressions (SPRs), *i.e.*, unintended slowdowns introduced during the evolution of string solvers. To this end, we develop *SPRFinder* to not only generate test cases demonstrating SPRs, but also localize the probable causes of them, in terms of commits. We evaluated the effectiveness of *SPRFinder* on three state-of-the-art string solvers, *i.e.*, Z3Seq, Z3Str3, and CVC4. The results demonstrate that *SPRFinder* is effective in generating SPR-inducing test cases and also able to accurately locate the responsible commits. Specifically, the average running time on the target versions is 13.2× slower than that of the reference versions. Besides, we also conducted the first empirical study to peek into the characteristics of SPRs, including the impact of random seed configuration for SPR detection, understanding the root causes of SPRs, and characterizing the regression test cases through case studies. Finally, we highlight that 149 unique SPR-inducing commits were discovered in total by *SPRFinder*, and 27 of them have been confirmed by the corresponding developers.

Index Terms—SMT string solver, performance regression, SPRFinder

1 Introduction

The Satisfiability Modulo Theories (SMT) problem is a class of decision problems for first-order formula extended with various background theories. A few well-established SMT solvers (*e.g.*, Z3 [1] and CVC4 [2]) have been developed for determining the satisfiability of SMT formula in the theories of Boolean, linear, non-linear arithmetics, as well as string operations, etc. These solvers have been widely adopted in supporting many practical applications, such as software/hardware verification [3]–[5], bug finding [6], [7], type inference [8], [9], synthetic biology [10]. String solvers are specialized SMT solvers with the ability to reason about string values, which find their applications in an increasing number of domains, such as the security analyses of mobile and Web applications [11], [12]. As a fundamental and universal reasoning engine, the quality of SMT solvers is crucial to the upper-layer applications.

Same as all other software systems, SMT solvers have quality issues (*e.g.*, correctness bugs [13] and performance slowdowns) which may greatly affect the soundness and efficiency of the applications. For example, bugs in SMT solvers may cause incorrect results in program verification or omission of test cases in test case generation [14]. The efficiency of SMT solvers also significantly impacts the performance of the applications. A well-known example is that the performance of solvers is one key factor that limits the scalability of symbolic execution. For example, when the path condition is too complex such as non-linear arithmetic, it may pose the scalability issues [15].

To this end, a few attempts have been made in finding bugs in SMT solvers, mostly focusing on *soundness bugs*. For example, recent studies (*e.g.*, YinYang [13], STORM [14], StringFuzz [16], and BanditFuzz [17]) identified multiple bugs in the arithmetic and string solvers of Z3 and CVC4, where either an "UNSAT" result is produced for satisfiable input formula or vice versa. The solvers can also be *incomplete*, in which case they are not able to determine the satisfiability of the inputs and report "UNKNOWN." Nevertheless, compared with these functional bugs, the performance of SMT solvers are less tested. Except for detecting functional bugs, StringFuzz [16] and BanditFuzz [17] were also evaluated on the performance issues of SMT solvers. Specifically, StringFuzz focuses on performance testing of solvers, while BanditFuzz uses RL-based algorithm to maximize the performance gap between different solvers. As fundamental reasoning engines, the solvers' runtime performance bottlenecks can influence the efficiency of applications (*e.g.*, symbolic execution, loop invariants inference). Researchers have proposed many methods (*e.g.*, some heuristics) to optimize their runtime performance [18]–[20].

There are a few hurdles to the detection and confirmation of performance issues for SMT solvers. **First**, unlike functional bugs, which can be manifested by deterministic test inputs, performance issues often also depend on the configurations, and sometimes non-deterministic factors such as

- *Yao Zhang, Sen Chen (corresponding author), and Xiaohong Li are with the College of Intelligence and Computing, Tianjin University. Emails: zzyy@tju.edu.cn, senchen@tju.edu.cn, xiaohongli@tju.edu.cn.*
- *Xiaofei Xie is with the Singapore Management University. Emails: xfxie@smu.edu.sg.*
- *Yi Li and Yang Liu are with the Nanyang Technological University. Emails: yi_li@ntu.edu.sg, yangliu@ntu.edu.sg.*
- *Yun Lin is with the Shanghai Jiao Tong University and National University of Singapore. Email: llmhyy@gmail.com*
- *This work was done while the first author was at Nanyang Technological University.*

random seeds. Therefore, it is harder for non-expert users to discover and report such issues. **Second**, debugging and localizing the causes for performance issues is more difficult, mainly because performance regressions are relative to a reference version of the solver. Looking at a single version alone, there may not be enough clue on where the actual problem is. **Finally**, performance changes—intended or not—can be introduced frequently during software evolution. For instance, a fix for a soundness bug may slow down the solver on certain inputs, and optimizations for some formulas may also cause a performance drop on other formulas. Many of such problems are unknown to the developers [21] and if remaining to be undetected, they may become technical debts and accumulate throughout the lifetime of the solvers. We refer to such unintended performance slowdowns of SMT solvers as *solver performance regressions* (SPRs). In this paper, we aim to shed light on the solver performance regressions. In particular, we focus on string solvers, *i.e.*, SMT solvers using the string theory, such as Z3seq [1], Z3Str3 [22], CVC4 [2], S3 [23], and S3N [12]. String solvers have been widely applied in security analysis and verification tasks [12] (*e.g.*, the ubiquitous string operations in Web applications), and have been shown to be susceptible to performance bugs [17]. To understand better the causes of performance slowdowns, we are particularly interested in how performance is affected by code changes (*i.e.*, commits) during the evolution of string solvers. Towards this goal, we attempt to address the following specific challenges.

1) How to effectively identify performance regressions, ideally with diverse underlying root causes. Joseph et al. [17] proposed a fuzzing technique, BanditFuzz, to discover inputs running slow on one solver, when compared with other reference solvers (*e.g.*, Z3, CVC4, and MathSAT), aiming to maximize the performance difference on distinct solvers. However, what they report is performance difference (between different solvers) while we are interested in finding performance regressions in this work. More specifically, we look at performance regression of an individual solver, *i.e.*, cases where a newer version runs much slower than an older version. Comparing to BanditFuzz [17], we aim to discover solver performance regressions (SPRs) that unveil more diverse root causes rather than a large number of regressions sharing the same root cause. **2)** How to debug and localize the root causes once a performance regression is detected. We use the commit(s) as an approximation to the root cause and aim to find minimal code changes responsible for the performance slowdown. **3)** How to better understand the different causes so that SPRs can be correctly mitigated. Performance debugging and analysis are a significant challenge [24]. Performance regressions on SMT solvers are rather convoluted and difficult to fix.

To address these challenges, we first implement a testing tool, named *SPRFinder*, to detect SPRs in a range of solver versions. *SPRFinder* is built on top of BanditFuzz from two perspectives, *i.e.*, an adaptive mutation and the diversity-aware feedback. Thus, *SPRFinder* can detect SPRs which can be traced back to a more diverse set of SPR-inducing code changes. To localize probable causes for an SPR, we then use a bisect-based algorithm to identify the responsible commit(s). Furthermore, to better understand the detected SPRs, we perform a case study on the intention of the localized commits to find out the reasons causing the SPRs.

We evaluated the usefulness of *SPRFinder* on the latest three versions of Z3 (*i.e.*, 4.8.7, 4.8.8, and 4.8.9) and CVC4 (*i.e.*, 1.7, 1.8, and 1.9[1]). To conduct a fair comparison on SPR detection, we first customized the functionality of BanditFuzz (named BanditFuzz_SPR) to use it on one type of solvers with different versions. We take random fuzzing and BanditFuzz_SPR as our experimental baselines. The experiments demonstrate that SPRFinder is effective in detecting SPRs when comparing with the baselines. Note that, the goal of BanditFuzz is to maximize the performance gap between different kinds of solvers, so that the results do not reflect the performance of BanditFuzz in its original domain.

By evaluating the detected SPR cases on different random seeds,[2] we found that 96.8% of the detected SPR cases (using the default seed) can still be reproduced with at least one fixed seed configuration. With the commit localization technique, we traced back to a total of 149 responsible commits, where 27 of them have been confirmed and 6 of them have been fixed. We further conducted an empirical study on the 149 commits and found that the objectives of these commits include: program fixes, algorithmic updates, background theory selection, and others. Through these results and feedback from the developers, we summarize common causes of SPRs. To the best of our knowledge, we are the first to detect and analyze performance regressions introduced during the evolution of SMT solvers. To summarize, we make the following contributions:

- We developed a regression testing tool, *SPRFinder*,[3] to detect the performance regressions between multiple versions of the solver. Besides, we empirically investigate the impact of random seed configuration when detecting SPRs.
- We proposed an automated localization technique to identify the commit that is responsible for the detected performance regression issues.
- We conducted the evaluation to demonstrate the effectiveness and the usefulness of our methods. We have identified a total of 146 commits that can cause SPRs in Z3Seq, Z3Str3, and CVC4, where 27 of them have been confirmed and 6 SRPs have been fixed by the developers.
- We carried out the first empirical study to explore the characteristics of SPRs, including understanding the root causes of SPRs and characterizing the regressing test cases based on developers' feedback, and provide useful findings for the research community.

2 Background

We begin by introducing the definitions of string solver and the solver performance regressions (SPRs), and presenting the strategy used in Banditfuzz [17].

1. Note that CVC4-1.9 is not released and we use its latest develop version as of January 2021.

2. Users can configure the parameter of "random seed" or use the default setting when running SMT solvers to control the propositional variable selection in the SMT core, which would probably affect the results of SPRs detection.

3. The source code of *SPRFinder* is public available on GitHub (https://github.com/ConfZ/SPRFinder).

2.1 String Solver

String constraint solving is the branch of the satisfiability modulo theories (SMT), whose typical constraint is on string length , concatenation, replace, regular expression, etc. For example, the following formula:

$$(str.contains\ (str.{+}{+}\ a\ b)\ c)$$

represents whether there exists an assignment for string variables a, b, c that can make the concatenation of a, b contains c. Such an assignment is also called a model of the formula. If the assignment exists, we say that the formula is "SAT". Otherwise, the formula is "UNSAT". String solver is developed to determine whether these string formulas are "SAT" or "UNSAT", and gives the corresponding models.

The growth of string manipulating programs in modern programming languages, including PHP, Python, JavaScript, et al., demands SMT solvers being capable of analyzing string constraints. Especially in fuzzing and software/hardware analysis domain, many new approaches [25]–[28] achieved better performance by adopting the string solvers. As a fundamental reasoning system, string solver plays an increasing significant role for the upper applications.

2.2 BanditFuzz

Banditfuzz banditfuzz is a fuzzer to conduct the performance testing and bug fuzzing for string and floating point solvers (e.g., CVC4 and Z3). The most related part to our work is the performance fuzzing for string solvers, so we mainly introduce the strategy of performance fuzzing in Banditfuzz as follows.

The motivation of BanditFuzz is to maximize the runtime gap of a case in the distinct solvers. They deploy their mutation strategy on a multi-armed bandit problem [29], which is a common reinforcement learning [30], [31] problem in MDP [32]. Specifically, Banditfuzz extends Stringfuzz [16] as the generator (G) to generate the input cases, and takes the SMT operator [33] (e.g., "str.++" and "str.replace") as the bandits (an action) in RL algorithm. In each testing loop, BanditFuzz takes a test case t from G and runs C on the distinct solvers S_1 and S_2. Meanwhile, it computes the score (SC_O), which is defined as:

$$Score := T_{S_1} - T_{S_2},$$

where T_{S_1} and T_{S_2} represent the running time of the solver S_1 and S_2.

In mutation section, BanditFuzz chooses an operator r as an action to generate a new mutant t' by replacing one of the operators in t with r, and then computes the score SC_r. If SC_r is higher than SC_O, Banditfuzz updates the Thompson sampling bandit [34]. That means the r can make the solver slower and more possible be selected in the next mutation. After that, BanditFuzz selects the top score ranks of cases and keep doing the next round. By this strategy, Banditfuzz can expand the runtime gap after each testing loop.

Note that, BanditFuzz aims to maximize the runtime gap of a case as much as possible on the distinct solvers. Even though Banditfuzz can better enlarge the performance gap on different solvers, it is limited to finding more diverse and unique cases. Specifically, SPR detection asks for finding more unique cases rather than only maximize the runtime gap, while Banditfuzz may easily fall into repeating mutation on the same case and lack diversity guidance for more unique cases so that it is not fit for SPR detection.

3 Overview

Figure 1 shows an overview of our approach including three major steps: 1) performance regression testing for detecting performance regression issues, 2) commit localization for identifying the commit(s) responsible for the detected SPRs, and 3) an empirical study to better understand the characteristics and causes of SPRs. All the results of SPR cases and SPR-inducing commits can be found on our website: https://sites.google.com/view/sprfinder

SPR Identification. Our test generation tool is extended based on a performance fuzzer for SMT solvers, *i.e.*, BanditFuzz. BanditFuzz is designed to generate inputs which maximize the performance difference between the target and reference solvers. In principle, it can also be applied to discover performance regressions, which are manifested by the performance gap between two versions of the same solver. Yet, the default test generator used by BanditFuzz has a fixed strategy to generate formulas, which may limit the diversity of tests. It can perform well when two solvers have different underlying algorithms and implementation strategies, but does not work well when the two solvers share great similarity (*e.g.*, in the regression testing setting). Moreover, BanditFuzz does not consider the code changes which are responsible for the performance regressions. Thus, the SPRs detected by BanditFuzz are often due to the same underlying causes, limiting its potential in detecting more bugs. Hence, we extended the design of BanditFuzz in two main directions: 1) We extended the test generator to support an adaptive configuration such that more diverse test cases can be generated. 2) We proposed a dynamic time warping (DTW)-based similarity [35], [36] for the guidance of test generation so that diverse test cases, exposing SPRs of different types, can be generated.

SPR Root Cause Localization. Each identified SPR by *SPRFinder* is triggered by a generated test input on a slow target version (newer) and a fast reference version (older). These regressions reflect performance slowdowns happened during the evolution of the solver. To help developers understand the root causes, we propose a method to localize the commit(s) responsible for the SPRs. Although the commit(s) may not correspond to the exact root cause of the issue, they serve as a good starting point for further investigation and debugging (the usefulness of these commits were confirmed by developers in our issue reports). Specifically, we employ an enhanced binary search over the commits between the target and the reference versions, to localize commit(s) introducing significant slowdowns.

Empirical Study. After localizing the responsible commit(s), it is sometimes still challenging for developers to understand and fix the issues. We conduct an empirical study on the detected SPRs to understand the intention of the localized commits. We also analyze the impact of incremental changes on the performance over a history range, and compare the characteristics of SPRs across different solvers.

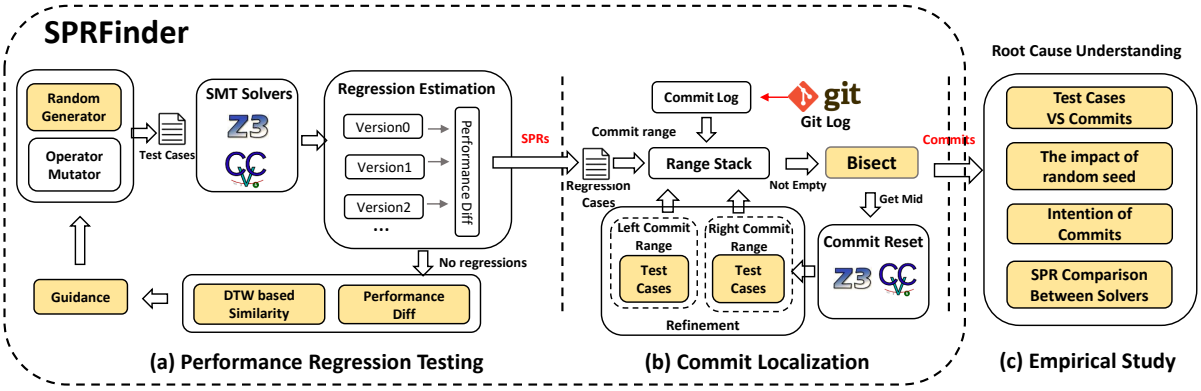


Fig. 1: Overview of our work

Additionally, random seed configuration to SMT solver can affect the detection performance of SPRs. Therefore, after the SPR detection, we run the test cases found by *SPRFinder* on 30 different random seeds, and see how much the random seed configuration can affect the SPR testing and also provide more information to the localization.

4 Methodology

4.1 Performance Regression Testing (*SPRFinder*)

Algorithm 1 shows how performance regression testing is performed. It takes two versions of the solver as inputs and returns a set of test cases that trigger SPRs. *SPRFinder* maintains a case queue T that contains a certain number of inputs (*i.e.*, β). At the beginning, it generates the initial cases (Line 2) with the function *Generator*, which produces some random SMT string formulas. Different from BanditFuzz, we propose an adaptive configuration strategy to diversify the generated test inputs. Then it starts the testing loop until the given time limit is reached (Lines 3 to 20). For each case t, if it causes a significant slowdown (Line 5), *i.e.*, the execution time on the new version v_1 is longer than the previous version v_0, a SPR is found and recorded (Line 6). We remove it from the case queue (Line 7).

The variable t_v is used to represent the execution time[4] of test case t on the version v. If the execution time of the new version v_1 is longer than that of the previous version v_0 by a pre-defined threshold α (Line 5), an SPR is identified. Note that, because the performance of string solvers may not be stable, we choose a sufficiently large threshold α to reduce noises introduced by the performance fluctuations. Besides, the threshold is adjustable to, for example, drive the algorithm towards identifying SPRs with significant impact. We adopt the Thompson sampling method [17] to select an action (*i.e.*, a mutation operator) that is more likely to trigger SPRs (Line 9). With the selected operator, we generate a new mutant t' by replacing one of the operators in t with $action$ (Line 10). Note that we use the same selection and mutation strategy as BanditFuzz did, with details described in [17]. Then we compute the scores (*c.f.* Definition 1) of the mutant and the original case respectively (Line 11). If the score of the mutant is better than the original case, it indicates that the selected action works well and the reward of the action is updated with the Thompson sampling bandit [17]

4. We set a time limit for constraint solving to avoid non-termination.

Algorithm 1: SPR Testing based on BanditFuzz [17]

Input : v_0, v_1: two versions of a SMT solver
Output: R: a set of performance regression test cases
Const : α: the threshold for performance regression
β: the number of test cases in the queue
c: the configuration of the generator

1 $R := \emptyset$;
2 $T := Generator(\beta, c)$;
3 **while** *time limit is not reached* **do**
4 **for** $t \in T$ **do**
5 **if** $t_{v_1} - t_{v_0} > \alpha$ **then**
6 $R \leftarrow R \cup \{t\}$;
7 $T \leftarrow T \setminus \{t\}$;
8 **continue**;
9 $action \leftarrow SelectAct()$;
10 $t' \leftarrow Mutate(t, action)$;
11 **if** $Score(t') > Score(t)$ **then**
12 $Dis := Distance(t)$;
$UpdateReward(action, Dis)$;
13 **if** $t'_{v_1} - t'_{v_0} > \alpha$ **then**
14 $R \leftarrow R \cup \{t'\}$;
15 **else**
16 $T \leftarrow T \cup \{t'\}$;
17 $T \leftarrow KeepBestScore()$;
18 $NT \leftarrow Generator(\beta - len(T), c)$;
19 $T \leftarrow T \cup NT$;
20 $c \leftarrow AdaptiveConfiguration(c)$;
21 **return** R;

(Line 12). If a SPR is triggered by t' (Line 13), we add it into R. Otherwise, it is added into the case queue (Line 16). After mutation, *SPRFinder* only keeps the test case that achieve the best score and removes the rest (Line 17). We then generate a number of fresh cases using the generator (Line 18) and add them into the queue (Line 19) so that the total number is still β. Finally, the configuration c is updated if no SPR is discovered after a time window (see details below).

We highlight the novelties in Algorithm 1 of *SPRFinder* compared with BanditFuzz as follows.

- In the $Score$ calculation (Line 11), we propose a more relaxed score calculation method to better guide the following RL-based algorithm (*c.f.* Section 4.1.1).
- In the $UpdateReward$ method (Line 12), we adopt a novel DTW-based method (*c.f.* Definition 2 in Section 4.1.2) to calculate the time distance between SPR-inducing cases to adjust the reward increment. Thus,

the action selection can be better guided for more diverse SPR-inducing cases.
- In the $AdaptiveConfiguration$ method (Line 20), we propose a self-adaptive configuration strategy to dynamically adjust the seed generator so that it can effectively generate diverse cases (*c.f.* Section 4.1.3).

4.1.1 Performance Score for SPR Detection

When deciding if an input triggers an SPR, we compare the solving time taken by the target version (the latest one) with the minimum solving time over all other versions. If the gap is bigger than the threshold, we consider it as an SPR. The performance score used in Algorithm 1 is defined as follows.

Definition 1 (Performance Score)**.** *Given an input t and its execution time $(t_{v_0}, t_{v_1}, \ldots, t_{v_n})$ on multiple versions (v_0, v_1, $\ldots$, v_n) of a solver, its performance score is calculated as*

$$Score(t) = t_{v_n} - min(t_{v_0}, \ldots, t_{v_{n-1}}),$$

where t_{v_n} refers to the latest version.

4.1.2 Distance-based Reward Calculation

To guide towards more diverse SPRs, we design the reward function used in Algorithm 1 to take into account the similarity between test cases. Ideally, we would like to know if two given test cases may trigger SPRs due to the same underlying root cause. However, this cannot be determined before the actual causes are localized. Instead, we approximate the similarity in root causes with the similarity in the duration of the caused slowdowns. This also reflects our observation in practice that similar slowdown patterns are often caused by the same commit(s).

Thus, we modify the reward function used by BanditFuzz with discount factors to consider the similarity of slowdown patterns, so that reward is discounted when a test input share similar pattern with an existing case.

To this end, we use $TS(t) = (t_{v_0}, t_{v_1}, \ldots, t_{v_n})$ to represent the execution time sequence of the test case t at each version.

Definition 2 (Distance of Time Sequence)**.** *Suppose T is the test cases generated before, the distance between t and T is:*

$$Dis(t) = min(\{DTW(TS(t), TS(t'))|\forall t' \in T\}),$$

where DTW (Dynamic time warping) [37] is a classic algorithm to compute the distance between two temporal sequences.

Intuitively, when the distance is large, the new input t is considered to be more different from the test cases in T. Then the reward is updated as:

$$Reward(A) := Reward_{pre}(A) + \gamma \cdot Dis(t),$$

where γ is the discount factors. A represents the selected action and $Reward(A)$ is the corresponding reward of the action A. $Reward_{pre}(A)$ refers to the reward of A in the previous detection round.

4.1.3 Adaptive Configuration Update

We empirically observed that the generator chosen has a direct impact on *SPRFinder*'s capability of discovering SPRs. Our study also revealed that the following parameters may affect the complexity of the generated constraints: 1) the length of the string constants, 2) the number of variables, 3) the number of sub-formulas (*i.e.*, asserted statements), and 4) the depth of the nested operations.

Further investigation confirmed that the generated tests by BanditFuzz do not cover more complex constraints. Besides, increasing the complexity parameters (*e.g.*, the length and the number of formulae) can result in more complex constraints, and thereby longer unit solving time for both the target and reference versions.

To balance between diverse constraints and short unit solving time, and also to avoid local optimum, we propose to adjust the complexity parameters based on the number of SPRs identified during a time window, to allow the constraint complexity to be adjusted gradually (*c.f.* Section 5.2).

These parameters will be updated if no SPR was found during a time interval t. We add or subtract a random value in each update. Specifically, the parameters are updated based on the time interval during which no SPR is detected. Suppose the starting time is t_0, then

- If no SPR was found within one interval (i.e., $(t_0, t_0 + t)$), the length of string constant L would be increased or decreased a random value in $[20, 50]$, where $L \in [10, 500]$.
- If no SPR was found within two intervals (i.e., $(t_0, t_0 + 2t)$), the number of variables V would be increased or decreased a random value in $[1, 5]$, where $V \in [3, 20]$.
- If no SPR was found within three intervals (i.e., $(t_0, t_0 + 3t)$), the number of sub-formulas S would be increased or decreased a random value in $[1, 3]$, where $S \in [4, 15]$.
- If no SPR was found within four intervals (i.e., $(t_0, t_0 + 4t)$), the depth of the nested operations D would be increased or decreased by 1, where $D \in [2, 6]$.
- If no SPR was found within five intervals (i.e., $(t_0, t_0 + 5t)$), we update the starting time as $t_0 := t_0 + 5t$ and continue to repeat the process from the first step.

We empirically set the range of each parameter (e.g., $L \in [10, 500]$) and then increased/decreased values. If the new value is out of range, we will ignore the update. Each parameter is increased if $t_{ave} < \theta \cdot T_{timeout}$, or decreased if $t_{ave} \geq \theta \cdot T_{timeout}$, where t_{ave} is the average unit solving time of all generated cases and $T_{timeout}$ is the timeout threshold. θ is a configurable threshold. Intuitively, if the average solving time exceeds $\theta \cdot T_{timeout}$, the complexity of the generated case is too high, which may affect the testing performance. Thus the parameters should be decreased to derive a reasonable unit solving time. Otherwise, the parameters are increased to grow the constraint complexity. Owing to this strategy, we can generate more diverse test cases (*c.f.* Section 5.2).

4.2 Commit Localization

After SPRs are successfully identified, we localize commit(s) responsible for these regressions as an approximation to the root cause, aiming to find the code changes which led to the performance slowdown. By narrowing the causes down to specific commits, we can then analyze and debug them more easily. A naïve method is to compile the solver at each commit and compare the performance of each input with/without the commit. But, it is impossible to enumerate

Algorithm 2: Commit Localization

```
Input : T: A set of test cases
        v0, v1: two versions of a SMT solver
Output: R: the localized commits
1  R ← ∅;
2  Let σ be an empty stack;
3  σ.push((C_v0, C_v1), T);
4  while σ is not empty do
5      (C_vs, C_ve), T' ← σ.pop();
6      if C_vs and C_ve are adjacent then
7          for t ∈ T' do
8              R ← R ∪ {(t, C_ve)};
9          continue;
10     C_vm ← BiSect(C_vs, C_ve);
11     v_m ← Reset(C_m);
12     left ← ∅, right ← ∅;
13     for t ∈ T' do
14         t_vm ← execute(v_m, t);
15         if |t_vm − t_vs| < |t_vs − t_ve| / 3 then
16             right ← right ∪ {t};
17         else if |t_vm − t_ve| < |t_vs − t_ve| / 3 then
18             left ← left ∪ {t};
19         else
20             right ← right ∪ {t};
21             left ← left ∪ {t};
22     if left is not empty then
23         σ.push((C_vs, C_vm), left);
24     if right is not empty then
25         σ.push((C_vm, C_ve), right);
26 return R;
```

all combinations considering the compilation and solving time on each commit and input.

The traditional bisect-based tool such as git-bisect [38] is not suitable for this specific task due to the following three reasons. 1) The traditional bisect-based approaches can only deal with one case in each localization round, while testing hundreds of cases found by *SPRFinder* requires repeated compilation and test runs, taking a huge amount of time and effort. 2) Many abnormal solver feedback such as unknown results, crashes, and timeouts require special treatment in the performance testing scenario. 3) The running time of string solvers is unstable, and small fluctuations may mislead the localization. Therefore, we propose an enhanced binary search to localize commits for a batch of test cases together with customized support for solver performance testing. The goal is to locate the relevant commits while minimizing the time taken. The basic idea is to map test cases into consecutive commits (*i.e.*, *commit range*), which are gradually narrowed down by binary search, until the target commit is located.

Algorithm 2 shows the main procedure that identifies the commits for a set of test cases. The inputs to the algorithm include two versions of the target solver (*e.g.*, CVC4-1.7 and CVC4-1.8) and a set of test cases that trigger the SPRs. Note that, the algorithm can easily be extended to work on multiple versions. A stack σ is used to maintain the updated commit ranges during the search (Line 2). Each item in σ includes a range and the test cases that fall within this range (*i.e.*, the range of commits containing the responsible commit). At the beginning, all test cases T belong to the range (C_{v_0}, C_{v_1}) (Line 3), where (C_{v_0}, C_{v_1}) represent all the commits in between the two versions v_0 and v_1.

SPRFinder then refines the ranges with binary search, until the commit can be localized for each test case (Lines 4 to 24). *SPRFinder* updates the commit range (C_{v_s}, C_{v_e}) of T' (Lines 23 to 25) in each bisection round, where C_{v_s} and C_{v_e} represent the starting commit and the ending commit of the commit range, respectively. Specifically, if two commits are adjacent (Line 6), we can already localize the commit, *i.e.*, C_{v_e}, for the test cases T' (Lines 7-8). Otherwise, we pick a middle commit C_{v_m} (Line 10) and build the new version v_m (Line 11). The commit C_{v_m} splits the original commit range into two parts and we perform binary search over them. We use $left$ and $right$ to represent the test cases that belong to the first and the second halves of the range, respectively. For each test case t, we run it at v_m and obtain the running time t_{v_m} (Line 14). We use t_{v_s} and t_{v_e} to respectively represent the running time of a test case on the two versions v_s and v_e, *i.e.*, after the starting commit C_{v_s} and the ending commit C_{v_e}. In the binary search, we need to determine whether the responsible commit should be closer to the start version (*i.e.*, in left part) or the end version (*i.e.*, in the right part). Thus, we compare the closeness between the running time of the two versions based on a threshold (*i.e.*, $\frac{|t_{v_s}-t_{v_e}|}{3}$). If the time difference is less than this threshold, two versions are assumed to have similar running time. Specifically, if $|t_{v_m} - t_{v_s}| < \frac{|t_{v_s}-t_{v_e}|}{3}$, it means that t_{v_m} is closer to t_{v_s} (Line 15), so that the responsible commit for t should fall into the $right$ half (C_{v_m}, C_{v_e}) (Line 16). Similarly, if $|t_{v_m} - t_{v_e}| < \frac{|t_{v_s}-t_{v_e}|}{3}$, it means that t_{v_m} is closer to t_{v_e} (Line 17), and then it falls into the $left$ half (C_{v_s}, C_{v_m}) (Line 18). Otherwise, if $|t_{v_m} - t_{v_s}| \geq \frac{|t_{v_s}-t_{v_e}|}{3}$ and $|t_{v_m} - t_{v_e}| \geq \frac{|t_{v_s}-t_{v_e}|}{3}$, it demonstrates that t_{v_m} is neither closer to t_{v_s} nor closer to t_{v_e}, the responsible commit falls into both halves. Finally, we put the refined ranges into the range stack (Line 23 and Line 25) and *SPRFinder* continues the refinement process in the next iteration. This way, we perform the search for a batch of tests T in one go. The complexity of the algorithm is $\mathcal{O}(\log n \times |T|)$, where n stands for the number of commits between v_0 and v_1.

Even if the seed generator (*c.f.* Algorithm 1, Line 2) and the operator based mutator (*c.f.* Algorithm 1, Line 10) are designed only for string solvers in this work, they can be further customized for other SMT solvers to solve a similar problem. Besides, the commit localization technique for batch of cases is also generally applicable in root cause identification for performance issues of other solvers.

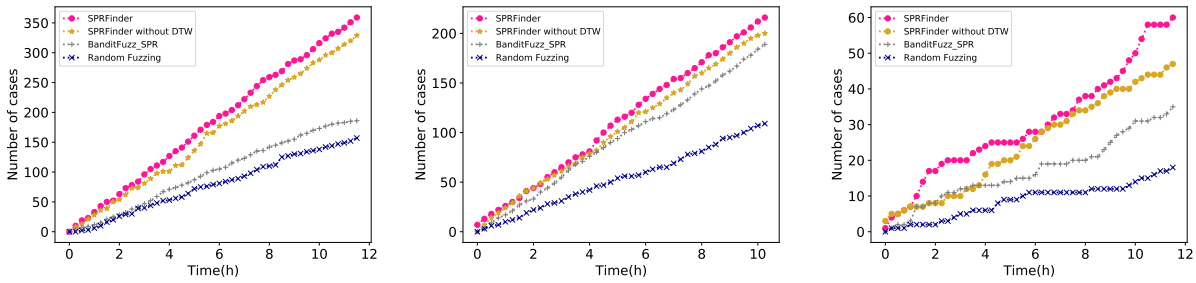
5 Evaluation of *SPRFinder*

In this section, we evaluate our approach in order to answer the following research questions.

- **RQ1:** How effective is *SPRFinder* in detecting performance regressions?
- **RQ2:** How much does the random seed configuration affect the SPR detection?
- **RQ3:** How accurately and efficiently can *SPRFinder* localize responsible commit(s) for the SPR-inducing test cases?

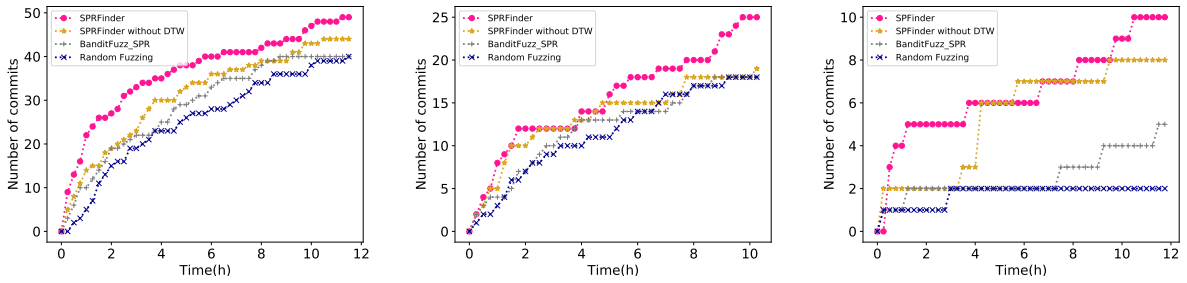
5.1 Setups

Benchmarks. We selected two widely used string solvers (*i.e.*, Z3 [1] and CVC4 [2]) for the evaluation of our method.



(a) Z3Seq (b) Z3Str3 (c) CVC4

Fig. 2: The total number of test cases generated by BanditFuzz_SPR, Random Fuzzing, and *SPRFinder*



(a) Z3Seq (b) Z3Str3 (c) CVC4

Fig. 3: The total number of localized unique commits detected by BanditFuzz_SPR, Random Fuzzing, and *SPRFinder*

Specifically, we selected the latest three versions for each solver, *i.e.*, 4.8.7, 4.8.8 and 4.8.9 for Z3, as well as 1.7, 1.8, and 1.9 for CVC4. Note that for Z3, there are two alternatives for string solving: the theory of strings (via Z3Str3 solvers) and the theory of sequences (via Z3Seq solver). We used both of them in our experiments. There is a "random seed" parameter (*e.g.*, controlled by "`--random-seed`" for CVC4 and "`smt.random_seed`" for Z3) in all the target solvers which controls heuristic selection in the SMT core. The random seeds chosen may have an impact on the solver's runtime performance even on the same input formulas. For each solver, we feed the three selected versions to *SPRFinder* and generate test cases that trigger SPRs between any two of the three versions by the default random seed configuration, since non-expert users are more likely to use the solver with the default settings.

Approaches under Comparison. To be consistent with the threshold defined in Algorithm 1, we also define the performance regression threshold of the approaches under comparison as α. We compared *SPRFinder* with two approaches (*i.e.*, random fuzzing and BanditFuzz_SPR) in the following experiments to demonstrate the effectiveness of SPR detection. Specifically, we implemented a random fuzzer which adopts a random seed generator. In addition, we further customized BanditFuzz (named BanditFuzz_SPR) because original BanditFuzz is limited in detecting performance regression issues. The main reasons include: 1) If BanditFuzz finds an SPR-inducing case, it spends much time mutating it, instead of the other seeds. This is because BanditFuzz aims to maximize the time difference, rather than finding a set of diverse SPR-inducing cases. 2) The seed generator of BanditFuzz is configured with fixed parameters, which limits the diversity of the generated test cases (more detailed analysis can be found in Section 5.2)

Thus, we built BanditFuzz_SPR, which customized BanditFuzz in the following two aspects. 1) After finding a SPR, BanditFuzz_SPR restarts and initializes all the settings, while BandiFuzz keeps running until time is up. 2) BanditFuzz_SPR randomly adjusts parameters for the generator in each test case generation, while BanditFuzz adopts the default fixed parameters.

5.2 RQ1: Effectiveness of Performance Regression Testing

Setup (RQ1). We ran SPRFinder, SPRFinder without DTW, BanditFuzz_SPR, and Random fuzzing on a target solver (*i.e.*, Z3Str3, Z3Seq, and CVC4) for 12 hours to generate test cases triggering performance regressions. To reduce randomness, we repeated the process for 5 times. For each test case, we set a 20-second timeout for the solver and ran them with the default random seed configuration. We set the number of test cases in the queue (β in Algorithm 1) as 5. We set the discount factor γ in Section 4.1.2 as 0.05, and set θ in Section 4.1.3 as 3/4. We compared the results using two metrics: the number of unique SPR-inducing test cases generated and the number of unique commits localized based on the generated test cases.

To determine the threshold, we evaluated the performance stability of running solvers. Specifically, we generated 500 string formulas and ran each case 10 times on the same version to test the stability of these solvers. Table 1 shows the average time, average time differences, and the maximal time difference of each string formula. We can see that the maximal time difference and the maximal average time are 2.02s and 9.91s on Z3Str3, respectively. Hence, in our paper, we chose 10s as the threshold, which is a safe setting considering the performance instability (compared to 2.02s). Moreover, the slowdowns caused by the identified

TABLE 1: Running time on the same versions of solvers

	Avg time	Avg diff	Max diff
# Z3Seq	2.05	0.11	1.69
# Z3Str3	**9.91**	0.24	**2.02**
# CVC4	1.43	0.03	0.35

SPRs would be significant enough, i.e., the regression can affect the normal usage of the solvers (compared to 9.91s).

Results. Figures 2 and 3 show the averaged results of the total number of generated test cases and the total number of commits localized based on the test cases. The x-axis represents the running time, ranging from 0 to 12 hours. Overall, we can see that *SPRFinder* is more effective than random fuzzing, BanditFuzz_SPR, and *SPRFinder* without DTW. We also noticed that *SPRFinder* and BanditFuzz_SPR outperform random testing (*i.e.*, the Non-RL fuzzer), demonstrating that reinforcement learning can be helpful in this task. Furthermore, *SPRFinder* is more effective than BanditFuzz_SPR, which indicates the usefulness of our strategies, *i.e.*, the guidance from the performance score (*c.f.* Definition 1), the adaptive configuration and the distance-based reward. Note that, even if Fig. 2 shows that the improvement by using DTW seems to be limited in terms of the total number of generated cases, we still see that *SPRFinder* generates more unique SPR cases with a higher speed when adopting DTW, based on Fig. 3. That fits our goal well of finding unique SPRs which have different underlying causes, rather than only optimizing the total number of possibly repeated cases.

Table 2 shows the average results of both tools after the 12-hour experiment. We can see that, on average, *SPRFinder* detects 362.4, 232.6, and 59.2 SPR test cases in total, on the three versions of the target solvers, respectively. With the commit localization technique applied on the detected test cases, *SPRFinder* localizes 49.8, 25.2, and 10.0 SRP-triggering commits, respectively. Considering all repeated runs combined, *SPRFinder* found a total of 66, 61, and 22 unique commits for Z3Seq, Z3Str3, and CVC4, respectively. We reported 10 and 15 issues to the developers of Z3 and CVC4, respectively, where 4 and 12 of them have been confirmed. Note that, 3 SPRs in CVC4 have been fixed up to today.

We found less SPR-inducing commits in CVC4 compared with Z3Seq and Z3Str3, which is a potential indication of the tools' performance stability.

Answer to RQ1: *SPRFinder* is more effective than the baseline approach (*i.e.*, BanditFuzz_SPR) in detecting performance regressions, and DTW helps to detect more cases with unique causes. With *SPRFinder*, we found 66, 61, and 22 unique SPR-inducing commits in total. This results also show that there are many performance regression issues in well-established string solvers, which should be considered especially when applied in performance-critical applications.

5.3 RQ2: Impact of the Random Seed Configuration

Setup. We empirically evaluated the test cases under different choices of random seeds and further investigated the impact of random seeds for SPR detection. Specifically, we

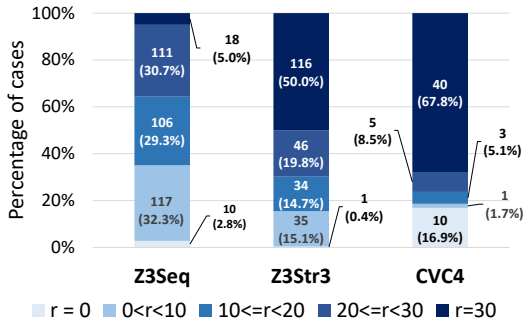


Fig. 4: Percentage of the cases that triggered SPRs by varying random seeds, and n represents the number of SPRs triggered by fixing random seeds to 1-30

took all 653 cases (362 for Z3Seq, 323 for Z3Str3, and 59 for CVC4, respectively) detected by *SPRFinder* in RQ1 using the *default* random seed as the subjects. The problem with the default random seed is that it is based on heuristics and may incur different solver behaviors across multiple versions. This brings a threat to the validity of our detected SPRs. To eliminate the noise from random seeds, we ran *each detected case* on the corresponding solver by fixing the random seeds across versions, with different choices ranging from 1 to 30.[5] Similar to RQ1, we set a 20-second timeout for each solving and the performance regression threshold (α) as 10 seconds. Finally, we calculated the number of SPR cases triggered under different random seed configurations.

Results. Figure 4 shows the percentage of cases that successfully triggered SPRs with various choices of random seeds. The value of n represents the number of random seeds caused SPRs on a *test case*. We classified n values into the following five ranges, namely, $n = 0$, $0 < n < 10$, $10 \leq n < 20$, $20 \leq n < 30$, and $n = 30$, representing the different levels of generality of the detected SPRs. For example, $n = 0$ represents that the test case was triggered by the default seed, but cannot be triggered by any of the 30 fixed random seeds. This indicates that the SPR case is likely due to the noise incurred by the default random seed when applied across multiple versions. In contrast, $n = 30$ represents that the test case can be reproduced by all of the 30 random seeds. This serves as a strong evidence for the generality of the test, which also indicates a real reproducible performance regress issue.

Based on the experiment results, we found that 67.8% of the SPR-triggering cases can be reproduced on all the 30 random seeds for CVC4, while only 5% of the cases in Z3Seq and 50% of the cases in Z3Str3 can be reproduced. This shows that the random seed settings have a larger impact on Z3seq and Z3Str3, when compared with CVC4. In addition, there are 16.9% of the cases in CVC4 which cannot be reproduced with the fixed random seeds. The same values for Z3Seq and Z3Str3 are 2.8% and 0.4%, respectively. In general, 27.8% of the detected cases with the default random seed configuration can be reproduced on all the tested random seed configurations.

Answer to RQ2: The random seed configuration has an impact on SPR detection. Still, most (96.8%) of the

5. The seeds 1 to 30 were suggested by the Z3 developer [39].

TABLE 2: The average results of BanditFuzz_SPR, Random Fuzzing, and *SPRFinder* after 12 hours

		Z3Seq			Z3Str3			CVC4		
		min	max	avg	min	max	avg	min	max	avg
#Test	SPRFinder	359	371	362.4	221	238	232.6	52	68	59.2
	SPRFinder without DTW	330	346	337	213	224	218.4	46	49	47.4
	BanditFuzz_SPR	170	192	186	183	191	187.5	31	39	35.6
	Random	146	161	157.2	133	142	137.6	22	28	24
#Commit	SPRFinder	49	51	49.8	24	26	25.2	9	11	10
	SPRFinder without DTW	42	44	43.6	19	20	19.2	8	9	8.2
	BanditFuzz_SPR	38	42	40.2	17	20	18.6	4	7	5.2
	Random	36	43	39.8	17	19	18.2	1	4	2

TABLE 3: The average time overload before and after a commit

	#Commit	Before	After	Difference
Z3Seq	66	0.7	19.5	18.8
Z3Str3	61	0.7	20.0	19.3
CVC4	22	3.1	19.9	16.8
Average (second)		1.5	19.8	18.3

SPR cases found in RQ1 can be triggered with at least one fixed seed configuration. SPR detection on Z3Seq is more easily affected by the choices of random seeds, compared with other solvers.

5.4 RQ3: Effectiveness and Efficiency of Commit Localization

Setup. We evaluated effectiveness of the commit localization technique by comparing the performance differences between the version before the localized commit (*a.k.a.* before-commit version) and the version after this commit (*a.k.a.* after-commit version). Note that, in the localization, for each case we randomly selected one specific seed from the SPR-inducing seeds (identified in RQ2). For example, if a SPR is successfully reproduced on seeds 1 to 20 (in RQ2), we randomly pick a seed from the range. For each commit, we built two versions of the solver, *i.e.*, the before-commit version and the after-commit version and measured the respective running time. A larger time difference indicates a better accuracy in the commit localization.

Besides, we aim to investigate if *SPRFinder* can handle all the conditions along the whole commit histories. To do this, we built the commit histories of each solver (i.e., Z3 and CVC4) and run these SPR-inducing cases (in RQ1) on the whole commit histories of these solvers. Then, we selected the representative cases of the results and classified them into different categories. Finally, we checked if *SPRFinder* can accurately locate theses cases by analyzing the time trend of version updates. Finally, we evaluate the efficiency of our commit localization approach by comparing *SPRFinder* and git Git. We used all test cases (generated by *SPRFinder*) in RQ1, and We ran both *SPRFinder* and Git-bisect for 12 hours on the target solvers (*i.e.*, Z3Seq, Z3Str3, and CVC4) to test which approach can locate more cases during the time range.

Results. Table 3 shows the averaged results. Column #*Commit* shows the total number of localized commits for each solver. Column *Before* shows the average time overhead of all test cases on the corresponding before-commit versions and Column *Right* shows the average time overhead of all test cases on the corresponding after-commit versions. Column *Difference* shows the average time difference between the before-commit version and the after-commit versions. The results show that with the localized commits, the test cases have a considerable time difference (*i.e.*, 18.8, 19.3, and 16.8 seconds), which indicates that *SPRFinder* is able to localize SPR-inducing commits accurately.

Figure 5 shows the representative results of single cases running on the commit histories of the target solvers. The x-axis represents the commit history of the solver versions, *i.e.*, Z3-4.8.7 to Z3-4.8.9 or CVC4-1.7 to CVC4-1.9, and the y-axis represents the running time of the solver. C and C' represent responsible commits and C_m represents the middle commit of the commit histories. We classify the results by two dimensions, *i.e.*, Monotonic/Non-monotonic, Single-Dominant/Multi-Dominant. Specifically, Monotonic/Non-monotonic represents whether the time varies monotonously, and Single-dominant/Multi-dominant represents if the SPR is dominated by one commit or multiple commits. We analyzed the localization ability of *SPRFinder* from the following four conditions.

1) *Monotonic & Single-Dominant:* As shown in Fig. 5 (a), the running time increases monotonically during the commit histories, and obviously there is a steep soar that makes the main contribution to the SPR (*a.k.a.* SPR is dominated by single commit in the commit histories). In this condition, *SPRFinder* can accurately locate the commit C accordingly.
2) *Monotonic & Multi-Dominant:* Compared with Single-Dominant scenario, Multi-Dominant is more complicated. As shown in Fig. 5 (b), the time grows monotonously but more than one commit induced a soar. Note that our algorithm (Line 15 and Line 17 in Algorithm 2) disregards the little fluctuations, so we mainly focus on the significant variations. Obviously, both C and C' are dominating commits for the SPR. In this condition, when the first round bisection locates C_m as the middle commit, *SPRFinder* can move to both left and right, so that both the responsible commits C and C' can be accurately located.
3) *Non-Monotonic & Single-Dominant:* In this condition, the solver performance is non-monotonic but SPR is dominated by a single commit. For example, in Fig. 5 (c), both C and C' can make the running time grow up, while C' is not the domination because the running time here is less than our threshold in Algorithm 2 (Line 15). Therefore, *SPRFinder* can accurately locate the dominating commits.
4) *Non-Monotonic & Multi-Dominant:* This condition is the most complicated one because *SPRFinder* may not be correctly guided to find all the dominating responsible

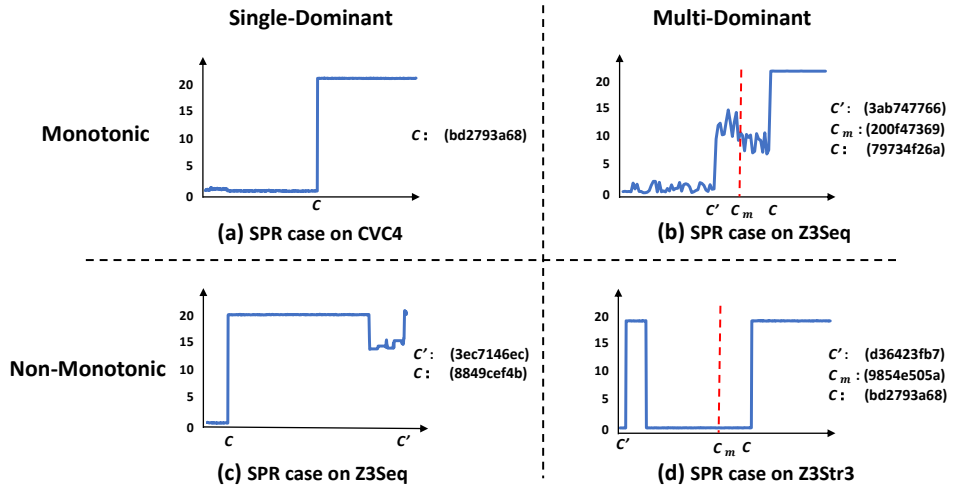


Fig. 5: Running time of single cases on commit histories of string solvers

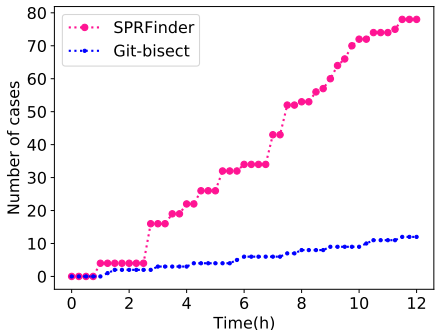


Fig. 6: The average cases localized by Git-bisect and *SPRFinder*

commits. For example, as shown in Fig. 5 (d), both C and C' are dominating commits. Once *SPRFinder* bisects to C_m, it will determine that the target commit falls into right. Therefore, *SPRFinder* can only find C but miss and C'. Even though *SPRFinder* would miss some results, it can achieve a better performance and locate to at lest one accurate commit, since localization for a batch of test cases is quite a time-consuming task in practice. In fact, it is a trade-off solution between performance and accuracy in terms of the localization algorithm.

Figure 6 shows the average cases localized by Git-bisect and *SPRFinder*. The x-axis represents the time ranging from 0 to 12 hours. We can clearly see that *SPRFinder* is much more efficient, which is about 8× faster than Git-bisect in our commit localization setting.

Answer to RQ3: In the most conditions of the commit histories, *SPRFinder* is effective and efficient in localizing the responsible commits for the given performance regression cases. The average time taken at the before-commit version is 1.5 seconds, while that at the after-commit version is 19.8 seconds. This is a strong indication that the localized commits are indeed responsible for the SPRs. *SPRFinder* is more efficient (8x) than Git-bisection in our batch scenario.

6 Empirical Study

Owing to the ability of *SPRFinder* and our own experiences in performing the experiments, we conducted an empirical study to have a deeper understanding of the characteristics of SPRs and attempt to answer the following research questions.

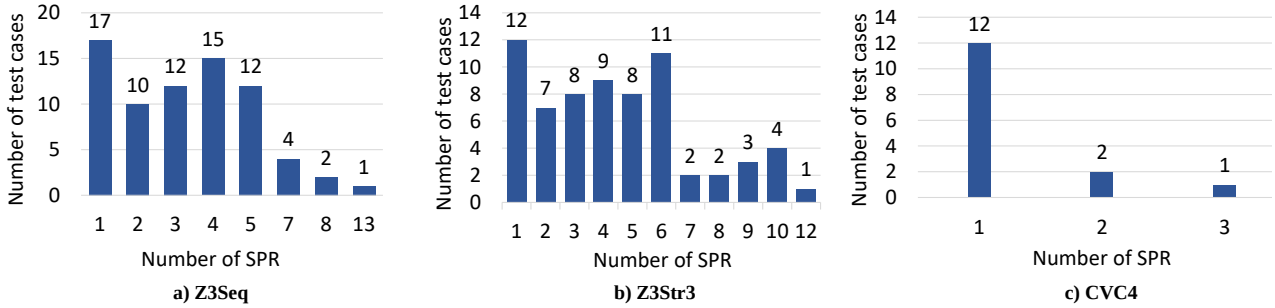
- **RQ4:** How do incremental code changes impact the solver performance?
- **RQ5:** Why do the localized commits cause performance regression issues?

6.1 RQ4: Impact of Incremental Code Changes

Setup. First, we aim to study the performance impact along the whole commit histories. Specifically, we would like to observe whether a single test case may trigger multiple SPRs along the evolving versions of the solvers. We used all the localized commits in RQ1 (*i.e.*, 66, 61, and 22 in Z3Seq, Z3Str3, and CVC4) as reference points and built two reference versions, before and after each commit. We then randomly selected one test case for each such commit, which runs significantly slower at the newer version. Thus, 66, 61, and 22 test cases were selected for Z3Seq, Z3Str3, and CVC4, respectively. Finally, we ran each test case on all the reference versions and checked whether a SPR is triggered at this version. We calculated how many SPRs can be discovered along the commit histories using each test.

Results. Figure 7 shows the detailed results on each solver. The x-axis represents the number of SPRs discovered by a single test case along the version histories. The y-axis represents the number of test cases that fall into the corresponding category. For example, for Z3Seq, there are 15 test cases which trigger SPRs on 4 different commits and 1 case which triggers SPRs on 13 commits. In general, we find that a single test case may trigger multiple SPRs through the commit histories, which indicates the instability of the solver performance during incremental updates. For example, a test may run faster after some commits, but become slower again after the subsequent commits. In particular, some cases in Z3Seq and Z3 Str3 have gone through more than 10 fast-to-slow changes during the evolution of the solvers.

Figure 8 shows the detailed performance vibration of an example case, by fixing the random seed to 2. In the corresponding commit histories, it triggered three performance regression issues. Specifically, after the commit *78feac446*, the SPR was triggered for the first time. Then the issue was



a) Z3Seq b) Z3Str3 c) CVC4

Fig. 7: Distribution of test cases based on the number of SPR-inducing commits exposed by the test case

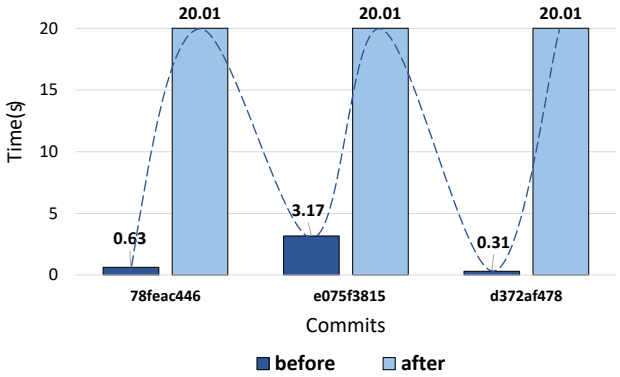


Fig. 8: The performance vibration of a test case On Z3Seq by fixing random seed = 2

fixed by a later commit between *78feac446* and *e075f3815*. Next, it triggered the second SPR (right after *e075f3815*), which is then fixed before the commit *d372af478*. In the end, it triggered the third regression right after *d372af478*.

In general, CVC4 tends to be more stable while Z3Str3 seems the most unstable. For example, for CVC4, most test cases (12) only trigger SPRs once. The maximum number of SPRs triggered by one test case was 3. For Z3Str3, the number of SPRs triggered by one test case may vary from 1 to 12.

Answer to RQ4: The performance of the string solver is unstable. During the evolution of the solver, some test cases may trigger SPRs multiple times, *i.e.*, a test case can reveal SPRs caused by multiple commits.

6.2 RQ5: Empirical Study on SPR-inducing Commits

After the SPR-triggering commits are localized, the natural next step is to understand and fix the issues. Fixing performance regressions is a challenging task. Therefore, we conducted an empirical study to understand the developers' intentions behind these commits. We manually analyzed the commits in combination with the developers' feedback and summarized the following reasons.

1) *Program Fixes*. Some commits aim to fix known bugs in the program. As a result, it may cause performance degradation on some input examples.
2) *Algorithmic Update/Optimization*. There are a major part of commits that update algorithms used by the string solvers. It may improve the performance on some inputs, but we found it worsens the performance on other inputs by a large margin.

TABLE 4: Commit triage based on its intention

		Z3Seq	Z3Str3	CVC4	Total
Intention	Program Fixes	29	24	4	57
	Algorithmic Update	24	26	9	59
	Theroy Selection	4	4	0	8
	Unknown	9	7	9	25
	Total	66	61	22	149
Class.	Compromise	11	9	4	24
	Performance Bug	0	0	3	3

3) *Theory Selection*. String solvers rely on other background theory solvers to handle arithmetic, array, and other functions. Yet, some commits update these theory solvers, which results in performance regressions.
4) *Unknown*. There are also some other commits for which we fail to identify the intentions.

Table 4 shows the number of commits based on the intentions behind the update. We found that many SPRs were caused by program fixes (38.3%) and algorithmic updates/optimizations (39.6%). In total, *SPRFinder* discovered 66, 61, and 22 SPR-inducing commits in Z3Seq, Z3Str3, and CVC4, respectively. Based on the feedback from the developers, we further classified the commits into two types: *performance compromises* and *performance bugs*. For the performance compromise commits, developers aim to fix bugs or improve performance on some test examples while sacrificing that of others. We observed that programmers made the compromise intentionally, as a more comprehensive solution was yet ready. For the performance bugs, developers were unaware that such commits may have side effects, which could have been mitigated if known in prior.

Note that it is sometimes difficult to judge objectively whether a commit is performance compromise or performance bug, as it requires confirmations from the developers. We could not always obtain such confirmation, and thus classified the commits conservatively with best efforts. If the commit messages clearly indicate that there were known issues and compromises made, we regarded them as the performance compromise commits. For performance bugs, we only calculated the commits that have been confirmed or fixed by developers. Row *Class.* of Table 4 shows the classified results and other un-classified commits were regarded as unknown.

We also include some feedback from the developers of CVC4 on two types of the cases as follows. Note that, for the second case, the developer marked them with a *bug* tag and fixed them subsequently.

Performance compromise: "*The commit bd2793a fixed a refutation soundness bug. That commit was a straightforward fix of a lemma that was unsound. Thus if CVC4 solved this quickly before that commit and does not solve now, it may have been due to unsound reasoning.*"

Performance bug: "*...This lead to an infinite loop of inferences because we effectively were just splitting...a component into two skolems and the only restriction was that the first one was non-empty.*"

We also investigated SPR-triggering test cases common across different solvers based on the results from RQ1. Specifically, for Z3Str3 and Z3Seq, there are 7 test cases, traced back to 17 commits, that can trigger SPRs on both. There are 9 test cases that trigger SPRs on both Z3Str3 and CVC4, and one test case that triggers SPRs on both Z3Seq and CVC4.

Answer to RQ5: The SPRs are usually caused when developers aim to fix known bugs, update the algorithms, and make changes to the other theory solvers. There are some comprises that developers made intentionally, and some are because of the developers' unawareness of potential side effects of their changes on solver performance. In addition, some common test cases seem to trigger SPRs on multiple solvers.

6.3 Threats to Validity

The selection of versions and the string solvers could be a threat to validity, which may affect the generalizability of the RQ1 results. To mitigate it, we selected 2 state-of-the-art string solvers and the latest 3 versions for the evaluation. The randomness of solver performance may be another threat to the results of RQ1. We repeated the process for 5 times and calculated the average results. Another threat comes from the hyper parameters used in the algorithm, *e.g.*, the threshold of performance regressions and the number of seeds. The random seed chosen is another threat. In answering RQ1, we selected the default random seed, which may introduce noises in determining some SPR-inducing test cases. We systematically experimented on the choices of random seeds in RQ2. Finally, the classification of performance compromises and bugs may not be accurate, because the commits, which are classified as performance compromises, may also contain unknown bugs.

7 Related Work

SMT Solver Testing. In recent years, there have been many studies focusing on functional testing of SMT solvers. Brummayer et al. [40] developed FuzzSMT to randomly generate SMT formulas, essentially performing fuzz testing on SMT solvers. Yet, it does not handle formulas in the String theories. Bugariu et al. [41] proposed a formula synthesis approach that is able to generate "SAT" or "UNSAT" formulas, which are then used as test oracle to detect soundness bugs. Similarly, our work also relies on a seed generator, which performs adaptive adjustments to the predefined parameters, thus is able to balance between diversity and performance. Moreover, we not only generate SMT formulas, but also use the distance-based reward to guide the subsequent mutations. Most of the recent studies construct new test cases by transforming existing seed tests according to certain relations [13], [14], [42]–[45]. The key idea is to ensure that the satisfiability of the newly generated tests can be predicted via the transformation rules. For example, Mansur et al. [14] proposed to mutate SMT tests by breaking up and rebuilding the formulas based on their models. A limitation is that their mutator is not able to construct "UNSAT" formulas. Winterer et al. proposed a series of transformations [13], [42], [43] to either fusing two equisatisfiable formulas together, replacing operators, or changing variables semantically. Yao et al. [45] introduced a simple but effective SMT fuzzing technique that combines two different input spaces, i.e., configuration space and semantic space, to detect soundness bugs. They also proposed a mutation approach [44], which produces mutants by over-approximating or under-approximating a SMT formula. In their work, the test oracle of the transformed formula can be inferred from the original seed. All these works above mainly focus on detecting soundness bugs, while our work targets performance regression issues across different versions of a solver.

As for the performance testing of SMT solvers, Dmitry et al. proposed StringFuzz [16], which is an effective generator and transformer for SMT formula. They used the generation-based approach to produce and mutate valid test cases to detect bugs and the performance limitations in string solvers. To further study the performance gap between different solvers, Scott et al. [17] proposed to adopt a reinforcement learning algorithm to detect performance issues and soundness bugs. They also used the Thompson sampling method to select the best operator for the following mutation. Both works studied the performance testing of SMT solvers, but none of them targets the performance regression issue concerning multiple versions of a solver. Our work is the first to analyze the performance regression during the evolution of a solver. Moreover, we also localize the root causes of these cases to assist debugging.

BanditFuzz [17] is the most related one to our work. *SPRFinder* is built based on BanditFuzz but can be distinguished from it in the following aspects: 1) *SPRFinder* is the first work to study the SPR detection problem on different versions of a solver, while BanditFuzz aims to find cases causing a large performance gap between different types of solvers. 2) *SPRFinder* cares more about the diversity of the SPR-inducing cases, while BanditFuzz mainly focuses on enlarging the performance gap between solvers. 3) *SPRFinder* not only detects performance regressions, but also automatically localizes the responsible commits of SPRs, and we conducted empirical studies on these commits to have a deeper understanding of the root causes. The evaluation results (*c.f.* Figures 2 & 3 and Table 1) demonstrate that BanditFuzz is limited on SPRs detection. We adopted a dynamically adaptive mutation strategy to generate more suitable cases to trigger more performance regressions and adopted a DTW-based similarity approach to improve diversity of the mutants. Owing to these strategies, *SPRFinder* works better on SPR detection.

Regression Testing. To ensure that software evolution does not affect the existing functionalities of software, regression testing [46] has been adopted. It is a time-consuming task to run the entire test cases during regression testing [47], therefore, regression test selection [48]–[52], test suite minimization [53]–[58], test case prioritization [59]–[62] have been widely studied in this research area. Different from these research directions, to identify the root causes of issues (e.g., performance issues, bugs) during regression testing, many researchers focused on identifying issue-change commits, which is also a prevalent challenge in regression testing. Because developers have to spend extra time and efforts narrowing down which commit caused the issues.

To figure out the real cause of the program failures, Zeller [63] proposed delta-debugging to find the minimal failure-inducing set by simplifying the input. But, this approach mainly focused on the failure-inducing inputs rather than commits localization, while our work aims to locate SPRs to the corresponding commits. Coulder et al. proposed a git bisect strategy [38] that can help debugger to locate the regression by bisecting the commit history. However, both of the approaches above can only work on a single case, so that not efficient enough on a batch of cases. Besides, they are also not appropriate for SMT solver, while our work mainly focuses on localizing commits for a batch of test cases, which can minimize the compilation time and customize to better handle the localization on solvers.

Cito et al. [64] used exploratory visual analysis and change point analysis to identify the root causes of web performance degradation issues. Similarly, Daly et al. [65] also adopted change point detection to represent the significant changes from a given history of performance results. Mühlbauer et al. [66] proposed an approach to identify configuration-dependent performance changes. Huang et al. [67] proposed performance risk analysis (PRA) to estimate the risk of performance changes based on static analysis, so that performance regression testing can further leverage the analysis result to test commits with high risks first. Our work distinguishes from these existing studies in two aspects: 1) *SPRFinder* may generate many test cases and each test case needs much time to solve (*e.g.*, 20s), thus it is expensive to localize the commit one by one. To improve the efficiency, *SPRFinder* adopts the bisection-based approach to localize commits for a batch of test cases. 2) Our work mainly focuses on the discovering performance regressions of SMT solver and analyze these regressions, where commit localization is only one of the process.

8 Conclusion

In this paper, we studied performance regressions in string solvers introduced during evolution. We developed an automated tool, named *SPRFinder*, which detects and localizes solver performance regressions. *SPRFinder* is designed to generate tests which maximize performance gaps between different versions of the same solver. Based on the SPR-inducing tests, we then localize the commits responsible, which can be used as a starting point for further investigation. Based on the 149 commits identified, we also conducted an empirical study to better understand the performance issues in string solvers.

The future work mainly includes: 1) exploring more advanced reward function for improving the performance of SPR detection and 2) applying our framework to more applications such as other types of solvers.

Acknowledgments

This work was partially supported by the National Science Foundation of China (No. 62102284, 61872262), the Ministry of Education, Singapore under its Academic Research Fund Tier 1 (21-SIS-SMU-033, T1-251RES1901), Tier 2 (MOE2019-T2-1-040, T2EP20120-0019), and Tier 3 (MOET32020-0004).

References

[1] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[2] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6806. Springer, 2011, pp. 171–177. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_14

[3] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[4] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[5] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010, pp. 348–370.

[6] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[7] M. Vujošević-Janičić and V. Kuncak, "Development and evaluation of lav: an smt-based error finding platform," in *International Conference on Verified Software: Tools, Theories, Experiments*. Springer, 2012, pp. 98–113.

[8] M. Hassan, C. Urban, M. Eilers, and P. Müller, "MaxSMT-based type inference for Python 3," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 12–19.

[9] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 470–487.

[10] B. Yordanov, C. M. Wintersteiger, Y. Hamadi, and H. Kugler, "SMT-based analysis of biological computation," in *NASA formal methods symposium*. Springer, 2013, pp. 78–92.

[11] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, and M. Deters, "An efficient SMT solver for string constraints," *Formal Methods in System Design*, vol. 48, no. 3, pp. 206–234, 2016.

[12] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "Inter-theory dependency analysis for SMT string solvers," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[13] D. Winterer, C. Zhang, and Z. Su, "Validating SMT solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 718–730.

[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] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[16] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, "Stringfuzz: A fuzzer for string solvers," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 45–51.

[17] J. Scott, T. Sudula, H. Rehman, F. Mora, and V. Ganesh, "Banditfuzz: Fuzzing smt solvers with multi-agent reinforcement learning," in *International Symposium on Formal Methods*. Springer, 2021, pp. 103–121.

[18] J. Chen and F. He, "Control flow-guided SMT solving for program verification," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 351–361.

[19] M. Balunovic, P. Bielik, and M. T. Vechev, "Learning to solve SMT formulas," in *NeurIPS*, 2018, pp. 10 338–10 349.

[20] S. Shen, S. Shinde, S. Ramesh, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: Augmenting symbolic execution with neural constraints," in *NDSS*, 2019.

[21] (2020) performance regression issue on github. [Online]. Available: https://github.com/CVC4/cvc4-projects/issues/217

[22] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," in *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2017, pp. 55–59.

[23] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in Web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1232–1243.

[24] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 145–155.

[25] G. Fraser and A. Arcuri, "Evosuite at the sbst 2017 tool competition," in *10th International Workshop on Search-Based Software Testing (SBST'17) at ICSE'17*, 2017, pp. 39–42.

[26] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190.

[27] G. Redelinghuys, W. Visser, and J. Geldenhuys, "Symbolic execution of programs with strings," in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, 2012, pp. 139–148.

[28] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 179–180.

[29] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *European conference on machine learning*. Springer, 2005, pp. 437–448.

[30] M. A. Wiering and M. Van Otterlo, "Reinforcement learning," *Adaptation, learning, and optimization*, vol. 12, no. 3, 2012.

[31] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[32] M. L. Puterman, "Markov decision processes," *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990.

[33] "Smt-lib," https://smtlib.cs.uiowa.edu/.

[34] O. Chapelle and L. Li, "An empirical evaluation of thompson sampling," *Advances in neural information processing systems*, vol. 24, pp. 2249–2257, 2011.

[35] M. Müller, "Dynamic time warping," *Information retrieval for music and motion*, pp. 69–84, 2007.

[36] T. Giorgino *et al.*, "Computing and visualizing dynamic time warping alignments in r: the DTW package," *Journal of statistical Software*, vol. 31, no. 7, pp. 1–24, 2009.

[37] T. Giorgino, "Computing and visualizing dynamic time warping alignments in r: The DTW package," *Journal of Statistical Software*, vol. 031, 2009.

[38] C. Couder, "Fighting regressions with git bisect," *Online: https://www. kernel. org/pub/software/scm/git/doc s/git-bisect-lk2009. html, The Linux Kernel Archives, GIT Repository, Version*, vol. 4, no. 5, 2008.

[39] https://github.com/Z3Prover/z3/issues/4962.

[40] R. Brummayer, "Fuzzsmt," 2009.

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

[42] D. Winterer, C. Zhang, and Z. Su, "On the unusual effectiveness of type-aware operator mutations for testing smt solvers," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.

[43] J. Park, D. Winterer, C. Zhang, and Z. Su, "Generative type-aware mutation for testing smt solvers," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–19, 2021.

[44] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "Skeletal approximation enumeration for smt solver testing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1141–1153.

[45] ——, "Fuzzing smt solvers via two-dimensional input space exploration," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 322–335.

[46] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[47] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999, pp. 179–188.

[48] L. Zhang, "Hybrid regression test selection," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 199–209.

[49] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic le dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[50] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 23–32.

[51] H. Hemmati and L. Briand, "An industrial investigation of similarity measures for model-based test case selection," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 141–150.

[52] G. Xu and A. Rountev, "Regression test selection for aspectj software," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 65–74.

[53] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie, "How do assertions impact coverage-based test-suite reduction?" in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 418–423.

[54] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396, 2015.

[55] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 738–748.

[56] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[57] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 246–256.

[58] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 237–247.

[59] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–31, 2014.

[60] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 268–279.

[61] K. Zhai, B. Jiang, and W. Chan, "Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study," *IEEE Transactions on Services Computing*, vol. 7, no. 1, pp. 54–67, 2012.

[62] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 192–201.

[63] A. Zeller, "Yesterday, my program worked. today, it does not.

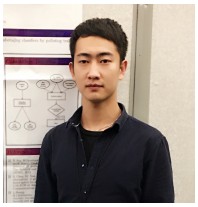
why?" in *ACM SIGSOFT Software engineering notes(ESEC/FSE)*. ACM, 1999, pp. 253–267.
[64] J. Cito, D. Suljoti, P. Leitner, and S. Dustdar, "Identifying root causes of web performance degradation using changepoint analysis," in *International Conference on Web Engineering*. Springer, 2014, pp. 181–199.
[65] D. Daly, W. Brown, H. Ingo, J. O'Leary, and D. Bradford, "The use of change point detection to identify software performance regressions in a continuous integration system," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 67–75.
[66] S. Mühlbauer, S. Apel, and N. Siegmund, "Identifying software performance changes across variants and versions," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 611–622.
[67] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 60–71.

Yao Zhang is currently a Ph.D student in College of intelligence and Computing of Tianjin University, China. His research focuses on software security, SMT solver analysis, software analysis and testing.

Xiaofei Xie received his Ph.D, M.E. and B.E. from Tianjin University. He is currently an assistant professor in Singapore Management University, Singapore. His research mainly focuses on program analysis, traditional software testing and quality assurance analysis of artificial intelligence. In particular, he won two ACM SIGSOFT Distinguished Paper Awards in FSE'16 and ASE'19.

Yi Li (Member, IEEE) is an Assistant Professor from the School of Computer Science and Engineering at the Nanyang Technological University, Singapore. He received his M.Sc. and Ph.D. in Computer Science from the University of Toronto in 2013 and 2018. Yi's research interests are in software engineering, software security, and software sustainability. His recent work on software change history analysis won an IEEE Best Artifact Award at ICSME'20 and ACM Distinguished Paper Award at ASE'15.

Yun Lin (Member, IEEE) He is now a Senior Research Fellow in SoC, National University of Singapore. He received his Ph.D degree in Computer Science from Fudan University and Bachelor degree in Software Engineering from East China Normal University. His research work focus on explainable and interactive root cause inference techniques for both traditional and AI programs.



Sen Chen (Member, IEEE) is an Associate Professor in the College of Intelligence and Computing, Tianjin University, China. Before that, he was a research Assistant Professor in the School of Computer Science and Engineering, NTU, Singapore. He received his Ph.D. degree in Computer Science from East China Normal University, China, in June 2019. His research focuses on software security and analysis.

Liu Yang (Senior Member, IEEE). graduated in 2005 with a Bachelor of Computing (Honours) in the National University of Singapore (NUS). In 2010, he obtained his PhD and started his post doctoral work in NUS, MIT and SUTD. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang Assistant Professor. He is currently a full professor and the director of the cybersecurity lab in NTU. He specializes in software verification, security and software engineering. By now, he has over 300 publications and 6 best paper awards in top tier conferences and journals.

Xiaohong Li received the Ph.D. degree from Tianjin University, Tianjin, China. She is a Full Tenured Professor with the School of Computer Science and Technology, College of Intelligence and Computing, Tianjin University. Her current research interests include knowledge engineering, trusted computing, and security software engineering. More information is available on http://cic.tju.edu.cn/faculty/lxh/index-english.html.