Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Random or heuristic? An empirical study on path search strategies for test generation in KLEE[☆,☆☆]

Zhiyi Zhang[a,d,e], Ziyuan Wang[b,*], Fan Yang[b], Jiahao Wei[b], Yuqian Zhou[a,c], Zhiqiu Huang[a,c]

[a] Collage of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b] School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, Nanjing, China
[c] Ministry Key Laboratory for Safety-Critical Software Development and Verification, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[d] National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[e] Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University, Nanjing, China

A R T I C L E   I N F O

Article history:
Received 19 September 2021
Received in revised form 12 January 2022
Accepted 9 February 2022
Available online 17 February 2022

Keywords:
Path search strategies
Test generation
Symbolic execution
Constraint solving
Software testing

A B S T R A C T

Randomness may be the most straightforward strategy and has been widely used in software engineering. One criticism for random strategy is its aimless for tasks. Therefore, many heuristic strategies have been proposed to improve the random strategy. However, it is impossible to prove that heuristics are better than randomness in theory, especial for software test generation. This open question is always left to empirical study and researchers expect to conclude some guidelines in practice. This paper studies a key technique, path search strategies, of test generation. We conducted an empirical evaluation and comparison among ten concrete path search approaches provided by KLEE, where two approaches belong to random search strategy and eight belong to heuristic search strategy, with 53 GNU Coreutils applications. We also investigated both cases with and without constraint optimization techniques for KLEE-based test generation. The experimental results show that without optimization, one approach from random strategy – *random-path* – performs better than other techniques in terms of the number of completed paths, statement coverage and branch coverage. These results indicate that random strategy can be a better choice for test generation in most cases without optimization, and heuristic strategies need further investigation. However, when combined with optimization, the selection of search strategy depends on the specific optimization applied. We further analyze the reasons behind the statistical results and provide guidelines to select the appropriate path search strategy for test generation in practice.

1. Introduction

Random and heuristic are two representative ideas globally, and the investigation between them is one of the hottest spots. In software testing, when testers are concerned about improving testing efficiency, they inevitably investigate random and heuristic (Zhang et al., 2010; Chen and Yu, 1994; Bueno et al., 2014), such as in search strategies. During the execution of a search approach, the algorithm states continuously transform as the different inputs. However, the transition in an approach is not necessarily deterministic. The random strategy typically uses uniformly random bits as an auxiliary input for state transition, while the heuristic strategy transforms the states according to specific rules.

Random and heuristic strategies are often used to search paths in test generation based on symbolic execution, which King proposed in the late 1970s (King, 1976). It uses symbolic values instead of actual data as input values and uses symbolic expressions to represent program variable values during program execution (Chen et al., 2013). Different path search strategies are applied during symbolic execution to search the following path to continue symbolic execution.

At first, a random path search strategy is utilized to search paths since it is the most intuitive and most straightforward strategy. However, there are still several severe challenges that hinder symbolic execution from turning into reality. One of the

☆ This research is supported, in part, by National Key R&D Program of China (2020YFB1005500), National Natural Science Foundation of China (Grant No. 62002162, No. 61901218 and No. 61772259), Natural Science Foundation of Jiangsu Province, China (Grant No. BK20200442 and No. BK20190407), and Open Fund of the State Key Laboratory for Novel Software Technology (Grant No. KFKT2019B11).
☆☆ Editor: W. Eric Wong.
* Corresponding author.
E-mail addresses: zyzhang10@nuaa.edu.cn (Z. Zhang), wangziyuan@njupt.edu.cn (Z. Wang), 1319045411@njupt.edu.cn (F. Yang), 1217043030@njupt.edu.cn (J. Wei), zhouyuqian1989@nuaa.edu.cn (Y. Zhou), zqhuang@nuaa.edu.cn (Z. Huang).

challenges is path explosion. During dynamic symbolic execution (DSE), with the increase in the length of an execution path, the number of feasible execution paths of a program increases exponentially (Joshi et al., 2007); thus, exploring all feasible paths is typically prohibitively expensive.

To alleviate path explosion, in recent years, several heuristic search strategies, instead of random strategies, have been proposed in test generation tools, which are developed based on symbolic execution and constraint solving (Chen et al., 2013; Godefroid et al., 2008; Cadar et al., 2008b,a; Godefroid et al., 2005; Tillmann and De Halleux, 2008). These strategies transform the states according to specific rules. Graph search and coverage-optimized search are two popular and representative heuristic search strategies. However, for test generation based on DSE, which strategy performs better remains an open question. When a strategy performs better, it can help DSE generate more effective tests.

As a DSE engine, KLEE provides a total of ten path search approaches, where two belong to random search strategy and eight belong to heuristic search strategy. In this paper, to answer the question of which approach performs better, we conducted an empirical evaluation and investigation on two random and eight heuristic path search approaches using KLEE (Cadar et al., 2008a) with 53 GNU Coreutils applications. We evaluated tests generated with three popular metrics: the number of completed paths, statement coverage, and branch coverage. The experimental results show that without constraint optimization, although in some cases, the performance of the random approach and other approaches is not significantly different, however, in most cases, one approach from random strategy – *random-path* – performs better than other approaches in terms of completed paths, branch coverage, and branch coverage. The results indicate that random strategy can be a better choice for KLEE-based test generation and heuristic strategies need further investigation. Moreover, since KLEE always executes with constraint optimization techniques, we also studied the effectiveness of path search strategies when using single constraint optimization techniques (constraint caching, independent and simplification) during KLEE-based test generation together. Further, we also investigate the reasons behind the statistical results and provide guidelines to select the appropriate path search strategy for KLEE-based test generation. Last, we hope to inspire the investigation of randomness and heuristic in other research areas.

In summary, the main contributions of this paper are as follows.

- We conducted experiments on KLEE by running 53 GNU Coreutils applications. Our study empirically evaluated and investigated two random path search approaches against eight heuristic path search approaches by three test-sufficiency metrics: the number of completed paths, statement coverage, and branch coverage.
- Our results show that without constraint optimization, although in some cases, the performance of the random strategy and other strategies is not significantly different, however, in most cases, one approach from random strategy – *random-path* – performs better than other techniques in terms of completed paths, statement coverage and branch coverage. We also investigate the reasons behind the statistical results and provide guidelines to select the appropriate path search strategy for KLEE-based test generation.
- We also studied the effectiveness of path search strategies combined with single constraint optimization techniques (constraint caching, independent and simplification) in KLEE-based test generation. The results show that one approach from heuristic strategy – *nurs-qc* – performs better combined with constraint caching, while no approach performs best combined with constraint independence or simplification.
- Our study indicates that random strategy can be a better choice for KLEE-based test generation and heuristic strategies need further investigation. We further provide some guidelines to select appropriate strategies combined with constraint optimization techniques for KLEE-based test generation. This is also an inspiration for the investigation of randomness and heuristic in other research areas.

The rest of paper is as follows. Section 2 introduces the background of our study. Section 3 is our experiment design. Section 4 is the empirical study of path search strategies without and with constraints optimization techniques. Section 5 demonstrates the related work. Conclusions and future work are drawn in the last section.

2. Background

2.1. Test generation based on dynamic symbolic execution

Many automatic test generation tools have been developed in the past decades (Anand et al., 2013; Li and Wong, 2002). In this paper, we study test generation based on dynamic symbolic execution (Godefroid et al., 2005; Majumdar and Sen, 2007) and constraint solving (Schittkowski, 1986). DSE uses concrete execution drive traditional symbolic execution (King, 1976; Chien et al., 2021). It first runs a program using concrete values and keeps track of the symbolic state to get the path condition. Then, it negates the path condition corresponding to a branch not yet covered, and a constraint solver is used to generate the solution for the new alternative paths. This process is continued until all the feasible paths are covered, or a resource limit is reached. In recent years, DSE has been widely used in test generation (Cadar et al., 2008a; Cadar and Engler, 2005; Cadar et al., 2008b; Godefroid et al., 2008; Sen et al., 2005), automated filter generation (Brumley et al., 2008b; Newsome et al., 2006) and malware analysis (Brumley et al., 2008a; Moser et al., 2007; Song et al., 2008). Fig. 1 shows the process of test data generation based on DSE. Given a program under test and an initial input, DSE runs the program using concrete and symbolic inputs at the same time to get path conditions. The path search strategy explores candidate paths during symbolic execution. Then the path conditions will be optimized and sent to a constraint solver to get a solution. The test data of all feasible program paths can be generated by systematically repeating this process.

We use a simple program, shown in Fig. 2, to demonstrate how DSE generates test data. At first, there is a concrete input for the program, such as $(a = 13, b = 11, c = 1, d = 1)$, and its execution path is represented by a set of statement number, {1, 2, 3, 4, 5, 6, 8, 9, 15}. x_a, x_b, x_c, and x_d are symbolic inputs for a, b, c and d, respectively. In Statement 3, we get the constraint $x_d < 5$. In Statement 5 and 6, an intermediate variable e is replaced by input parameters to get the constraint $x_a \neq x_b + x_c$. This process continues until this path has been completely executed. Then, the path condition, i.e. a conjunction of all constraints in this path, is $(x_d < 5) \wedge (x_a \neq x_b + x_c) \wedge (x_b > 10)$. Next, the last constraint $x_b > 10$ is negated to obtain a new path condition, which is $(x_d < 5) \wedge (x_a \neq x_b + x_c) \wedge (x_b \leq 10)$, and the new path is {1, 2, 3, 4, 5, 6, 8, 10, 11, 15}. Finally, a constraint solver is used to get a solution of the new path condition, such as $(x_a = 13, x_b = 10, x_c = 1, x_d = 1)$, and this solution is a test which covers the new path {1, 2, 3, 4, 5, 6, 8, 10, 11, 15}.

DSE has some practical challenges in handling large or complex programs. One of the biggest challenge is path explosion (Chen et al., 2013). With the increase in the length of an execution path, the number of feasible execution paths of a program increases

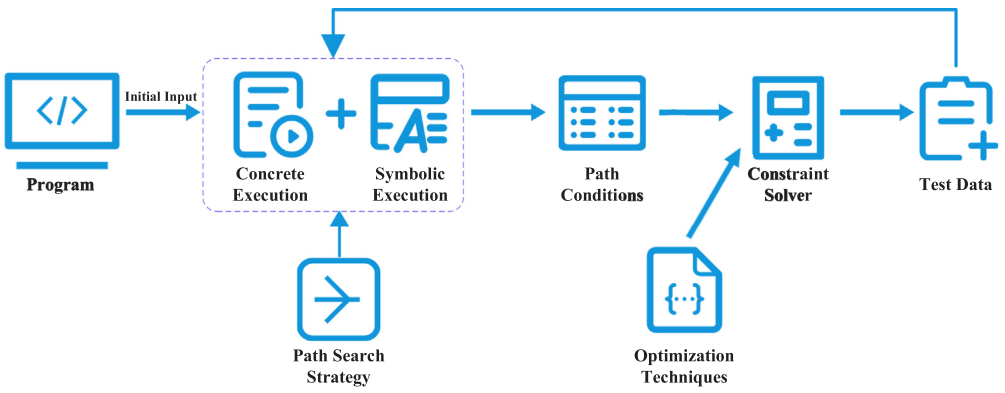


Fig. 1. Test generation based on dynamic symbolic execution and constraint solving.

```
foo(int a, int b, int c, int d)
{
    if (d < 5)
    {
        int e = b + c;
        if (a == e)
            //do something
        else if (b > 10)
            //do something
        else
            //do something
    }
    else
        //do something
}
```

Fig. 2. An example program.

exponentially (Joshi et al., 2007), thus exploring all feasible paths is typically prohibitively expensive. In recent years, heuristic path search strategies, such as generational search, depth-first search and coverage-optimized search, have been used in symbolic execution tools to alleviate this problem (Godefroid et al., 2008; Cadar et al., 2008b,a; Godefroid et al., 2005; Tillmann and De Halleux, 2008; Xie et al., 2009).

Moreover, during DSE, to improve the efficiency of test generation, several constraint optimization techniques have been proposed to simplify constraint expressions or reduce queries to constraint solvers. We introduce three representative constraint optimization techniques, constraint caching, independent and set simplification, which are used in our study.

Constraint caching (Cadar et al., 2008a) could quickly find out solution according to characteristics of subset and superset, which is, if a path condition is unsatisfiable then its superset is unsatisfiable; if a path condition is satisfiable then its subset is satisfiable. For example, the condition constraints $(x > 10) \wedge (x < 5)$ has no solution, neither does the original constraints $(x > 10) \wedge (x < 5) \wedge (y < 0)$. $x = 6$ is a solution for constraints $(x < 10) \wedge (x > 5)$, thus it also satisfies either $x < 10$ or $x > 5$ individually. Moreover, the new constraints often do not invalidate the solution to the existing subset. According to this heuristic, the existing solution of a subset can be used to verify the whole constraint set. It can reduce much time cost because verifying a constraint set is much faster than solving a constraint set.

Constraint independence optimization (Cadar et al., 2008b) divides independence constraints into different constraint subsets. Two constraints are considered as dependent if the solution of one can affect the solution of the other, which means they share one or more variables directly or indirectly. For example, the path condition $(x_a = x_b + x_c) \wedge (x_b > 10) \wedge (x_d < 5)$ could be divided into two groups $(x_a = x_b + x_c) \wedge (x_b > 10)$ and $(x_d < 5)$, since x_d is independent of x_a, x_b and x_c. Solving two small constraint subsets costs much less time than solving the whole constraint set. Moreover, it is easier to find a solution for independent constraint subsets than the whole constraint set in practice, although they have equivalent solvability in theory.

Constraint set simplification (Cadar et al., 2008a) could quickly find out whether existing constrains can be eliminated or if the new constraints conflict against the existing constraints. For example, there is a constraint $x_b > 10$ in the path condition $(x_a = x_b + x_c) \wedge (x_b > 10) \wedge (x_d < 5)$. If a new constraint $x_b > 100$ is added, then $x_b > 10$ will be eliminated, because $x_b > 100$ implies $x_b > 10$. On the contrary, if the new added constraint is $x_b < 5$, it is easier to detect that the path condition has no solution with two conflict constraints. Also, it can simplify equality expressions before querying the solver.

2.2. Path explosion and heuristic search strategies

During DSE, random path search strategies, such as random-path or random-state (Cadar et al., 2008a), are the simplest and most intuitive strategies for path searching. It has two significant advantages: One is that it favors paths with fewer constraints and so it has greater freedom to hit uncovered code (Chen et al., 2013); another is that random strategies could avoid starvation during symbolic execution, because execution paths with large numbers of branches are not able to predominate execution chances with the random selection (Chen et al., 2013).

Since the potential risk of path explosion, some heuristic path search strategies, such as graph and coverage-optimized search approach, have been proposed used in DSE in recent years. The aim of heuristic strategies is to maximizes the number of new tests generated and code coverage while avoiding redundant search in dynamic symbolic execution.

In this paper, we empirically studied two random and eight heuristic path search approaches, which are widely used for test generation (Cadar et al., 2008a). Table 1 describes these search approaches. The first column presents each strategy and its corresponding acronym. We use the acronym to represent corresponding approach in the remaining part of this paper. For example, $\mathcal{S}_{pa}$ represents the random approach — *random-path*. The second column represents whether this approach belongs to random or heuristic strategies. The last column are the simple descriptions of the approach. $\mathcal{BFS}$ and $\mathcal{DFS}$ are two representative search approaches. $\mathcal{S}_{cn}$ searches a new point which could cover new code. If there are many points which satisfy the requirement (covering new code), $\mathcal{S}_{cn}$ randomly selects a point and executes it symbolically. $\mathcal{S}_{md}$ means searching a uncovered point which

Table 1
Path search approaches.

Name	Classification	Introduction
BFS ($\mathcal{S}_{bf}$)	Heuristic	Breadth First Search
DFS ($\mathcal{S}_{df}$)	Heuristic	Depth First Search
nurs-covnew ($\mathcal{S}_{cn}$)	Heuristic	Non Uniform Random Search with Coverage-New
nurs-md2u ($\mathcal{S}_{md}$)	Heuristic	Non Uniform Random Search with Min-Dist-to-Uncovered
nurs-depth ($\mathcal{S}_{de}$)	Heuristic	Non Uniform Random Search with AAA
nurs-icnt ($\mathcal{S}_{ic}$)	Heuristic	Non Uniform Random Search with Instr-Count
nurs-cpicnt ($\mathcal{S}_{cp}$)	Heuristic	Non Uniform Random Search with CallPath-Instr-Count
nurs-qc ($\mathcal{S}_{qc}$)	Heuristic	Non Uniform Random Search with Query-Cost
random-path ($\mathcal{S}_{pa}$)	Random	Randomly select a path to explore
random-state ($\mathcal{S}_{st}$)	Random	Randomly select a state to explore

has the minimum distance from the current point. $\mathcal{S}_{de}$ selects the next point which has depth of "nth" power of two. $\mathcal{S}_{ic}$ means the selected point has the minimum instruction count, while $\mathcal{S}_{cp}$ means it has the minimum call-path instruction count. $\mathcal{S}_{qc}$ searches a new point which has the minimum query cost. $\mathcal{S}_{pa}$ and $\mathcal{S}_{st}$ are two random strategies which randomly select a path or a state to explore, respectively. In our study, we study "one path search approach" instead of "hybrid approaches" which means using different approaches alternately during symbolic execution. One reason is that the study of "one" is the basic before we study hybrid approaches. Only if we understand the effectiveness of a single approach, can we study hybrid approaches better. Another reason is that there are some tools that still use "one path search approach", such as EXE (Cadar et al., 2008b) and DART (Godefroid et al., 2005). Moreover, in future, we will do additional work to compare hybrid approaches.

3. Experimental design

3.1. Research question

This paper is to empirically study the effectiveness of random and heuristic path search strategies for test generation based on dynamic symbolic execution. In our study, we wish to evaluate and investigate on random and heuristic strategies under the following circumstances:

RQ1 When generating tests using dynamic symbolic execution, which strategy (random or heuristic) is better without constraint optimization techniques?

RQ2 When generating tests using dynamic symbolic execution, which strategy is better combined with single constraint optimization technique (caching, constraint independence and constraint set simplification)?

DSE is usually expensive, and the cost of constraint solving dominates the whole cost of symbolic execution (Chen et al., 2013). To reduce the cost and improve test generation efficiency, several constraint optimization techniques, such as constraint caching, constraint independence, and constraint set simplification, are usually applied during DSE (Cadar et al., 2008a). However, there are some side-effects when using the optimization (Zhang et al., 2017). During DSE, it takes a lot of time to optimize constrains, and sometimes optimizations are not suitable for some programs. For example, if one program has few inputs which are all dependent, so "constraint independence" is unusable and cannot improve the effectiveness, but DSE costs much time on "constraint independence". Hence, evaluating strategies effectiveness without constraint optimization techniques is necessary. Due to above the reason, we first evaluate strategies effectiveness without constraint optimization techniques, then we study the effectiveness of random and heuristic search strategies combined with constraint optimization techniques. Both research questions could guide testers to use optimization and path strategies better when generating tests based on DSE.

In our study, we use three metrics, the number of completed paths, statement coverage and branch coverage, to evaluate the performance of each path searching strategy. A completed path, which has been used as an evaluation metric in Zhang et al. (2017), means that a path whose entire execution process is completed, including path searching, symbolic execution, optimization (if there exists), and constraint solving. The more paths a search approach completes, the more test data it generates and the better the test suite is. Coverage analysis is usually used as a proxy method to assure the quality of tests. Statement coverage and branch coverage are two of the most popular coverage metrics. Our study demonstrates these two coverage for tests generated with heuristic and random path search strategies. The higher the code coverage is, the better the quality tests are and the better the search strategy is.

3.2. Tools and subjects for experiment

Many test generation tools have been developed based on symbolic execution and constraint solving (Chen et al., 2013). Specifically, we perform our empirical study on KLEE with STP for the following reasons. First and foremost, we believe that KLEE with STP is one of the best-known symbolic execution tools. It is mature and has been widely studied in academia. Secondly, KLEE contains two random and eight heuristic path search approaches, which could provide full support for our empirical study. These eight heuristic approaches could be divided into two categories, six optimized approaches and two graph approaches. Last but not least, we selected STP because STP was initially designed specifically for the symbolic execution tool EXE and could solve the queries generated by EXE well (Palikareva and Cadar, 2013; Cadar et al., 2008b). KLEE is redesigned based on EXE; thus, it generates the same types of queries as EXE, and STP is also suitable for KLEE (Palikareva and Cadar, 2013). In our experiments, we used the newest version of KLEE and STP with LLVM 3.4 (Lattner and Adve, 2004), a compilation infrastructure designed to compile program language such as C language.

We selected GNU CoreUtils programs as our experiment subjects. There are several reasons behind our selection. Firstly, GNU CoreUtils programs have been developed for decades and are widely used all over the world. Moreover, GNU CoreUtils programs have been studied on KLEE in many previous academic studies (Palikareva and Cadar, 2013; Wang et al., 2015). Secondly, KLEE is designed and configured specifically for GNU CoreUtils (Wang et al., 2015); thus, GNU CoreUtils programs are stable enough to run on KLEE and could reduce noises from software bugs. Lastly, library APIs may cause symbolic execution exploration to stop early and mask other potential limitations of symbolic execution (Wang et al., 2015). Since KLEE has built library APIs for GNU CoreUtils programs, running GNU programs could reduce the effect of lacking library APIs. In our experiments, we used the GNU CoreUtils 6.11 because this version is the most stable version for KLEE.

3.3. Experimental procedure

99 applications are included in GNU Coreutils version 6.11. Since some applications are not suitable for our study, we firstly discarded the applications for which either (a) KLEE ran into unsupported system calls of LLVM3.4 instructions, (b) KLEE finished in less than one hour (e.g. the application *false*), or (c) KLEE ran exhibited a multitude of nondeterminism (e.g. the application *date*, which depends on the current time). Finally, 46 GNU Coreutils applications were discarded, and 53 applications were used in our study.

For each program, we conducted test generation by KLEE with the single random or heuristic path search approach, respectively. Moreover, since we study the effectiveness of random and heuristic path search approaches combined with constraint optimization techniques, we also conducted test generation with single path search approach and single constraint optimization technique together. During symbolic execution, we set the execution time limit as one hour to constrain the time for test generation for a given program. The time limit is chosen as one hour because of two reasons. One is that if the time limit is too small, KLEE may complete only several paths with each search approach, so the evaluation results, such as numbers of completed paths, maybe similar, which threatens the validity of our conclusion. The other is that if the execution time is too large, KLEE may stop in the prescriptive time because all paths have been completed. For the same program with different search approaches, KLEE may stop at different times, so the comparisons are unfair. For example, if we set the time limit as two hours, for the program join, KLEE stops in 78 min with $\mathcal{S}_{cn}$ and 93 min with $\mathcal{S}_{de}$, respectively. When all paths have been completed, though tests generated with $\mathcal{S}_{de}$ completed more paths and have higher coverage, this may be due to the execution time of $\mathcal{S}_{de}$ is larger, not the effectiveness of $\mathcal{S}_{de}$ during symbolic execution. In addition, test resources are usually limited in practice. To make our experiment more convincing, the time limit should not be large. In our study, consider the size of experiment programs, "one hour" is suitable for KLEE execution. Moreover, "one hour" has been used in previous empirical studies about KLEE (Palikareva and Cadar, 2013). Please note that the execution time includes test generation, path searching, symbolic execution, optimization, and constraint solving, for fair comparisons.

To avoid nondeterminism freeze during symbolic execution, for each condition, we set the timeout to be 30 s for test generation due to the time limit and the programs, which means, if a single condition cannot be explored or solved in 30 s, we skip it. If the timeout value is too large, KLEE may be frozen for some reasons, such as floating numbers. If the value is too small, some complex paths may not be completed at this time. 30 s is an appropriate value in our experiments, which has been used in previous empirical studies about KLEE (Palikareva and Cadar, 2013). In practice, if the timeout value is set too large or too small, KLEE cannot run continually. To make our results could be used in practice, we set the timeout value as 30 s so that KLEE can run continually. Moreover, to reduce the effect of the randomness in our study, we repeat each approach ten times and average the results. We used 40 computers with Dual-CPU 3.70 GHz and memory 16 GB and cost about 9600 h to run our experiments. To reduce potential errors in our experiments, we used Gcov (License), a widely used and mature tool to measure coverage and collect the statement coverage and branch coverage information of tests.

4. Experiment results and analysis

In this section, we empirically studied the effectiveness of path strategies for test data generation based on DSE. We firstly present and analyze the results of our empirical study, then we explain the results and provide some guideline to select appropriate path strategies.

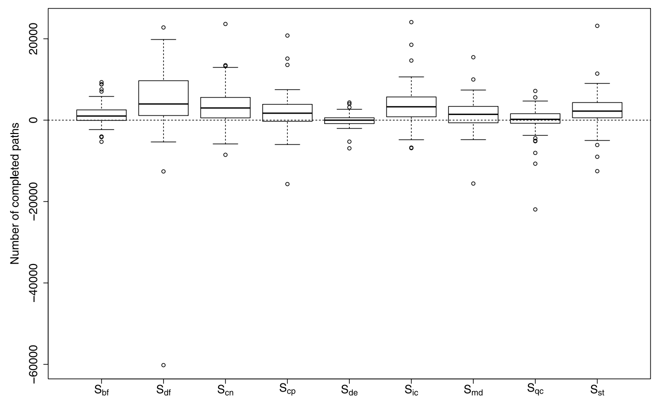
We compare the number of completed paths, statement coverage, and branch coverage of ten approaches under four conditions: no optimization, with constraint caching optimization, with constraint independence optimization, and with set simplification optimization. We first collected the number of completed paths and their corresponding test suites for each path search approach, then applied *Gcov* to get the statement and branch coverage information. However, during tests replay, some test cases, such as test cases of *chcon*, may change the system environment of Ubuntu, which resulted in KLEE stopping running. We discard these test cases that could not be replayed successfully. At last, test suites of 47 applications are used to analyze code coverage.

4.1. Experiment results for RQ1

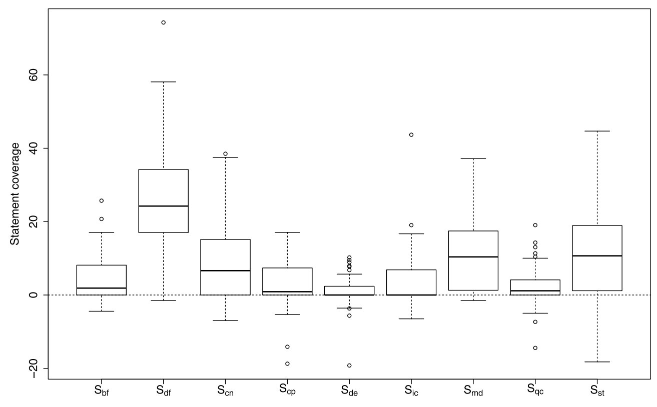
To answer RQ1, box plots are applied to display the results. Firstly, we choose random path approach as the baseline approach from ten approaches. Then, we use the number of completed paths, statement coverage, and branch coverage obtained by random path approach, to subtract corresponding values obtained by the other nine strategies. Fig. 3 shows the difference between random path approach and the other nine approaches under three test-sufficiency metrics. The horizontal axis represents nine approaches except for *random-path*, while the vertical axis represents the differences in completed paths numbers, statement coverage, and branch coverage between *random-path* and other approaches, respectively. A higher difference indicates a better performance of *random-path*. For example, the first boxplot in Fig. 3(a) presents the difference of numbers of completed paths between *random-path* and $\mathcal{S}_{bf}$. The median line (bold line) in the box plot represents the median value of the difference.

Fig. 3(a) shows that the number of paths completed by eight approaches, *BFS*$_{(\mathcal{S}_{bf})}$, *DFS*$_{(\mathcal{S}_{df})}$, *nurs-covnew*$_{(\mathcal{S}_{cn})}$, *nurs-icnt*$_{(\mathcal{S}_{ic})}$, *nurs-cpicnt*$_{(\mathcal{S}_{cp})}$, *nurs-md2u*$_{(\mathcal{S}_{md})}$, *nurs-qc*$_{(\mathcal{S}_{qc})}$, and *random-state*$_{(\mathcal{S}_{st})}$, is lower than *random-path* completed in most cases. And the number of paths completed by *nurs-depth*$_{(\mathcal{S}_{de})}$ greater and less than *random-path* completed accounts for half respectively, but the difference is small. From Fig. 3(b), we find that statement coverage of $\mathcal{S}_{de}$ and $\mathcal{S}_{ic}$ approaches is the same as that of *random-path* in about half of the cases, and is lower than that of *random-path* in most other cases. The statement coverage of the remaining seven approaches is lower than that of *random-path* in most cases. Fig. 3(c) shows that $\mathcal{S}_{df}$ and $\mathcal{S}_{st}$ have lower branch coverage than *random-path*. The branch coverage of five approaches, $\mathcal{S}_{bf}$, $\mathcal{S}_{cn}$, $\mathcal{S}_{ic}$, $\mathcal{S}_{md}$ and $\mathcal{S}_{qc}$, are as the same branch coverage as *random-path* in about half of the cases, and is mostly lower than *random-path* in the other half cases. The branch coverage of $\mathcal{S}_{cp}$ higher and lower than *random-path* accounts for half respectively, but the difference is tiny. $\mathcal{S}_{de}$ has the same branch coverage as *random-path* in half the cases. In the other half cases, though $\mathcal{S}_{de}$ has higher branch coverage than *random-path*, the difference is still negligible.

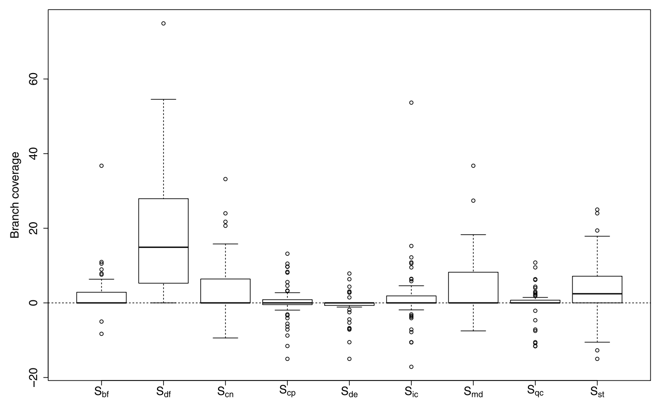
Although the box plot can intuitively show the difference value of each index between random path approach and other approaches, it is not enough to help us choose the best optimal approach. To confirm whether the differences between different approaches are significant, we do two hypothesis tests for any two approaches, $\mathcal{S}_i$ and $\mathcal{S}_{j, \{i,j \in \{st, pa, \ldots, bf\}\}}$, in terms of the number of completed paths, statement coverage, and branch coverage. The



(a) The difference of completed paths



(b) The difference of statement coverage



(c) The difference of branch coverage

Fig. 3. The difference between random-path and other approaches.

first hypothesis test is used to test whether approach $\mathcal{S}_i$ is better than approach $\mathcal{S}_j$, and the second hypothesis tests is used to test whether approach $\mathcal{S}_i$ is worse than approach $\mathcal{S}_j$. We believe that the more test paths or higher coverage a approach can get, the better the approach is. Therefore, we can set the two hypothesis tests as follows.

Hypothesis test 1:

- Null hypothesis: The performance of $\mathcal{S}_i$ is worse than or equal to the performance of $\mathcal{S}_j$
- Alternative hypothesis: The performance of $\mathcal{S}_i$ is better than the performance of $\mathcal{S}_j$

Hypothesis test 2:

- Null hypothesis: The performance of $\mathcal{S}_i$ is better than or equal to the performance of $\mathcal{S}_j$
- Alternative hypothesis: The performance of $\mathcal{S}_i$ is worse than the performance of $\mathcal{S}_j$

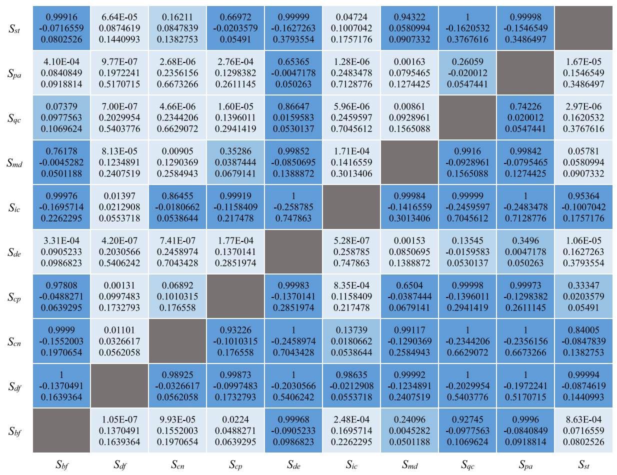
Without constraint optimization, we conducted three sets of hypothesis tests corresponding to three test-sufficiency metrics, the number of completed paths, statement coverage, and branch coverage. Specifically, each group of hypothesis tests compares ten approaches pairwise, and the results are shown in Fig. 4. For each approach, the data in its row indicates the result of hypothesis test 1, and in its column, it indicates the result of hypothesis test 2. There are three values in each cell in Fig. 4. The top value is the p-value of the hypothesis tests. The median value means effect size that measures the strength of the relationship between two approaches. The bottle value is the power, which is the probability that the test correctly rejects the null hypothesis. Moreover, we use different colors to represent different ranges of p-values. The brighter the color is, the higher the p-value is, and the higher the probability of accepting the null hypothesis is. Through the hypothesis test results in Fig. 4, we hope to find the overall optimal approach in different metrics without optimization conditions.

Fig. 4(a) shows that the number of paths completed by $\mathcal{S}_{pa}$, $\mathcal{S}_{de}$ is significantly higher than that completed by seven approaches, as well as $\mathcal{S}_{qc}$ is significantly higher than only six approaches. However, there is no significant difference in the number of paths completed by $\mathcal{S}_{pa}$, $\mathcal{S}_{de}$ and $\mathcal{S}_{qc}$. From Fig. 4(b), we found that the statement coverage of $\mathcal{S}_{pa}$ is significantly higher than that of the other nine approaches. Fig. 4(c) shows the branch coverage of $\mathcal{S}_{pa}$, $\mathcal{S}_{qc}$ and $\mathcal{S}_{de}$ is significantly higher than that of the five approaches, while $\mathcal{S}_{ic}$ and $\mathcal{S}_{cp}$ are significantly higher than only four. But $\mathcal{S}_{pa}$, $\mathcal{S}_{de}$, $\mathcal{S}_{qc}$, $\mathcal{S}_{ic}$ and $\mathcal{S}_{cp}$ have no significant difference without optimization. From the above results, we conclude that *random-path* performs better than other approaches in statement coverage. However, the advantage is not significant for *nurs-depth* and *nurs-qc* for the number of completed paths and branch coverage.

From the above results, when no constraint optimization technology is used, although in some cases, the performance of the random approach and other approaches is not significantly different, however, in most cases, the *random-path* search approach can significantly improve the efficiency of symbol execution to generate test cases. These results show that *random-path* is a better choice for DSE to generate test cases without constraint optimization, and heuristic search strategy needs further analysis and strengthening.

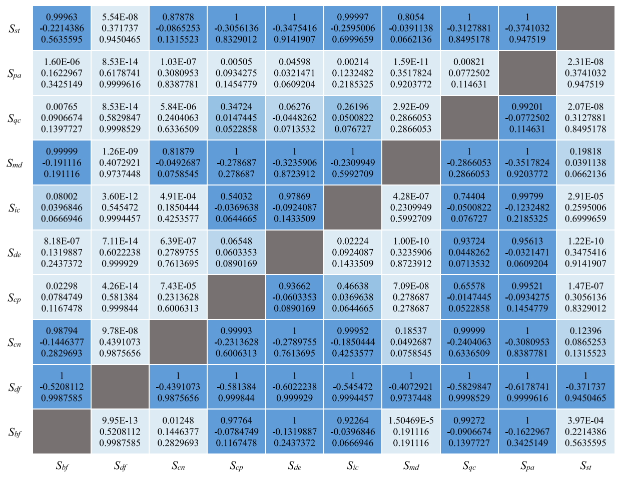
4.2. Experiment results for RQ2

In practice, some constraint optimization techniques, such as constraint caching, constraint independence, and constraint simplification (Cadar et al., 2008a), have been applied during DSE to improve test generation efficiency. RQ2 aims to investigate the underlying relations between path search strategies and constraint optimization techniques. To answer RQ2, we studied and investigated the test generated with random or heuristic path search strategies by three test-sufficiency metrics (the number of completed paths, statement coverage, and branch coverage) under three single constraint optimization conditions (constraint caching optimization, constraint independence optimization, and constraint set simplification optimization), respectively. box plots in Figs. 5, 9 and 4(a) show the difference between random path approach and the other nine approaches with single constraint optimization, respectively, while Figs. 6, 8 and 10 display the



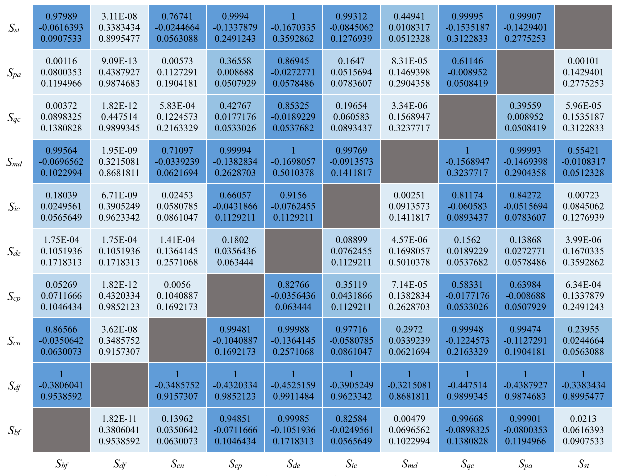
	S_{bf}	S_{df}	S_{cn}	S_{cp}	S_{de}	S_{ic}	S_{md}	S_{qc}	S_{pa}	S_{st}
S_{st}	0.99916 -0.0716559 0.0802526	6.64E-05 0.0874619 0.1440993	0.16211 0.0847839 0.1382753	0.66972 -0.0203579 0.05491	0.99999 -0.1627263 0.3793554	0.04724 0.1007042 0.1757176	0.94322 0.0580994 0.0907332	1 -0.1620532 0.3767616	0.99998 -0.1546549 0.3486497	
S_{pa}	4.10E-04 0.0840849 0.0918814	9.77E-07 0.1972241 0.5170715	2.68E-06 0.2356156 0.6673266	2.76E-04 0.1298382 0.2611145	0.65365 -0.0047178 0.050263	1.28E-06 0.2483478 0.7128776	0.00163 0.0795465 0.1274425	0.26059 -0.020012 0.0547441		1.67E-05 0.1546549 0.3486497
S_{qc}	0.07379 0.0977563 0.1069624	7.00E-07 0.2029954 0.5403776	4.66E-06 0.2344206 0.6629072	1.60E-05 0.1396011 0.2941419	0.86647 0.0159583 0.0530137	5.96E-06 0.2459597 0.7045612	0.00861 0.0928961 0.1565088		0.74226 0.020012 0.0547441	2.97E-06 0.1620532 0.3767616
S_{md}	0.76178 -0.0045282 0.0501188	8.13E-05 0.1234891 0.2407519	0.00905 0.1290369 0.2584943	0.35286 0.0387444 0.0679141	0.99852 -0.0850695 0.1388872	1.71E-04 0.1416559 0.3013406		0.9916 -0.0928961 0.1565088	0.99842 -0.0795465 0.1274425	0.05781 0.0580994 0.0907332
S_{ic}	0.99976 -0.1695714 0.2262295	0.01397 0.0212908 0.0553718	0.86455 -0.0180662 0.0538644	0.99919 -0.1158409 0.217478	1 -0.258785 0.747863		0.99984 -0.1416559 0.3013406	0.99999 -0.2459597 0.7045612	1 -0.2483478 0.7128776	0.95364 -0.1007042 0.1757176
S_{de}	3.31E-04 0.0905233 0.0986823	4.20E-07 0.2030566 0.5406242	7.41E-07 0.2458974 0.7043428	1.77E-04 0.1370141 0.2851974		5.28E-07 0.258785 0.747863	0.00153 0.0850695 0.1388872	0.13545 -0.0159583 0.0530137	0.3496 0.0047178 0.050263	1.06E-05 0.1627263 0.3793554
S_{cp}	0.97808 -0.0488271 0.0639295	0.00131 0.0997483 0.1732793	0.06892 0.1010315 0.176558		0.99983 -0.1370141 0.2851974	8.35E-04 0.1158409 0.217478	0.6504 -0.0387444 0.0679141	0.99998 -0.1396011 0.2941419	0.99973 -0.1298382 0.2611145	0.33347 0.0203579 0.05491
S_{cn}	0.9999 -0.1552003 0.1970654	0.01101 0.0326617 0.0562058		0.93226 -0.1010315 0.176558	1 -0.2458974 0.7043428	0.13739 0.0180662 0.0538644	0.99117 -0.1290369 0.2584943	1 -0.2344206 0.6629072	1 -0.2356156 0.6673266	0.84005 -0.0847839 0.1382753
S_{df}	1 -0.1370491 0.1639364		0.98925 -0.0326617 0.0562058	0.99873 -0.0997483 0.1732793	1 -0.2030566 0.5406242	0.98635 -0.0212908 0.0553718	0.99992 -0.1234891 0.2407519	1 -0.2029954 0.5403776	1 -0.1972241 0.5170715	0.99994 -0.0874619 0.1440993
S_{bf}		1.05E-07 0.1370491 0.1639364	9.93E-05 0.1552003 0.1970654	0.0224 0.0488271 0.0639295	0.99968 -0.0905233 0.0986823	2.48E-04 0.1695714 0.2262295	0.24096 0.0045282 0.0501188	0.92745 -0.0977563 0.1069624	0.9996 -0.0840849 0.0918814	8.63E-04 0.0716559 0.0802526

(a) Hypothesis test of path numbers



	S_{bf}	S_{df}	S_{cn}	S_{cp}	S_{de}	S_{ic}	S_{md}	S_{qc}	S_{pa}	S_{st}
S_{st}	0.99963 -0.2214386 0.5635595	5.54E-08 0.371737 0.9450465	0.87878 -0.0865253 0.1315523	1 -0.3056136 0.8329012	1 -0.3475416 0.9141907	0.99997 -0.2595006 0.6999659	0.8054 -0.0391138 0.0662136	1 -0.3127881 0.8495178	1 -0.3741032 0.947519	
S_{pa}	1.60E-06 0.1622967 0.3425149	8.53E-14 0.6178741 0.9999616	1.03E-07 0.3080953 0.8387781	0.00505 0.0934275 0.1454779	0.04598 0.0321471 0.0609204	0.00214 0.1232482 0.2185325	1.59E-11 0.3517824 0.9203772	0.00821 0.0772502 0.114631		2.31E-08 0.3741032 0.947519
S_{qc}	0.00765 0.0906674 0.1397727	8.53E-14 0.5829847 0.9998529	5.84E-06 0.2404063 0.6336509	0.34724 0.0147445 0.0522858	0.06276 -0.0448262 0.0713532	0.26196 0.0500822 0.076727	2.92E-09 0.2866053 0.2866053		0.99201 -0.0772502 0.114631	2.07E-08 0.3127881 0.8495178
S_{md}	0.99999 -0.191116 0.191116	1.26E-09 0.4072921 0.9737448	0.81879 -0.0492687 0.0758545	1 -0.278687 0.278687	1 -0.3235906 0.8723912	1 -0.2309949 0.5992709		1 -0.2866053 0.2866053	1 -0.3517824 0.9203772	0.19818 0.0391138 0.0662136
S_{ic}	0.08002 0.0396846 0.0666946	3.60E-12 0.545472 0.9994457	4.91E-04 0.1850444 0.4253577	0.54032 -0.0369638 0.0644665	0.97869 -0.0924087 0.1433509		4.28E-07 0.2309949 0.5992709	0.74404 -0.0500822 0.076727	0.99799 -0.1232482 0.2185325	2.91E-05 0.2595006 0.6999659
S_{de}	8.18E-07 0.1319887 0.2437372	7.11E-14 0.6022238 0.999929	6.39E-07 0.2789755 0.7613695	0.06548 0.0603353 0.0890169		0.02224 0.0924087 0.1433509	1.00E-10 0.3235906 0.8723912	0.93724 0.0448262 0.0713532	0.95613 -0.0321471 0.0609204	1.22E-10 0.3475416 0.9141907
S_{cp}	0.02298 0.0784749 0.1167478	4.26E-14 0.581384 0.999844	7.43E-05 0.2313628 0.6006313		0.93662 -0.0603353 0.0890169	0.46638 0.0369638 0.0644665	7.09E-08 0.278687 0.278687	0.65578 -0.0147445 0.0522858	0.99521 -0.0934275 0.1454779	1.47E-07 0.3056136 0.8329012
S_{cn}	0.98794 -0.1446377 0.2829693	9.78E-08 0.4391073 0.9875656		0.99993 -0.2313628 0.6006313	1 -0.2789755 0.7613695	0.99952 -0.1850444 0.4253577	0.18537 0.0492687 0.0758545	0.99999 -0.2404063 0.6336509	1 -0.3080953 0.8387781	0.12396 0.0865253 0.1315523
S_{df}	1 -0.5208112 0.9987585		1 -0.4391073 0.9875656	1 -0.581384 0.999844	1 -0.6022238 0.999929	1 -0.545472 0.9994457	1 -0.4072921 0.9737448	1 -0.5829847 0.9998529	1 -0.6178741 0.9999616	1 -0.371737 0.9450465
S_{bf}		9.95E-13 0.5208112 0.9987585	0.01248 0.1446377 0.2829693	0.97764 -0.0784749 0.1167478	1 -0.1319887 0.2437372	0.92264 -0.0396846 0.0666946	1.50469E-5 0.191116 0.191116	0.99272 -0.0906674 0.1397727	1 -0.1622967 0.3425149	3.97E-04 0.2214386 0.5635595

(b) Hypothesis test of statement coverage



	S_{bf}	S_{df}	S_{cn}	S_{cp}	S_{de}	S_{ic}	S_{md}	S_{qc}	S_{pa}	S_{st}
S_{st}	0.97989 -0.0616393 0.0907533	3.11E-08 0.3383434 0.8995477	0.76741 -0.0244664 0.0563088	0.9994 -0.1337879 0.2491243	1 -0.1670335 0.3592862	0.99312 -0.0845062 0.1276939	0.44941 0.0108317 0.0512328	0.99995 -0.1535187 0.3122833	0.99907 -0.1429401 0.2775253	
S_{pa}	0.00116 0.0800353 0.1194966	9.09E-13 0.4387927 0.9874683	0.00573 0.1127291 0.1904181	0.36558 0.008688 0.0507929	0.86945 -0.0272771 0.0578486	0.1647 0.0515694 0.0783607	8.31E-05 0.1469398 0.2904358	0.61146 -0.008952 0.0508419		0.00101 0.1429401 0.2775253
S_{qc}	0.00372 0.0898325 0.1380828	1.82E-12 0.447514 0.9899345	5.83E-04 0.1224573 0.2163329	0.42767 0.0177176 0.0533026	0.85325 -0.0189229 0.0537682	0.19654 0.060583 0.0893437	3.34E-06 0.1568947 0.3237717		0.39559 0.008952 0.0508419	5.96E-05 0.1535187 0.3122833
S_{md}	0.99564 -0.0696562 0.1022994	1.95E-09 0.3215081 0.8681811	0.71097 -0.0339239 0.0621694	0.99994 -0.1382834 0.2628703	1 -0.1698057 0.5010378	0.99769 -0.0913573 0.1411817		1 -0.1568947 0.3237717	0.99993 -0.1469398 0.2904358	0.55421 -0.0108317 0.0512328
S_{ic}	0.18039 0.0249561 0.0565649	6.71E-09 0.3905249 0.9623342	0.02453 0.0580785 0.0861047	0.66057 -0.0431866 0.1129211	0.9156 -0.0762455 0.1129211		0.00251 0.0913573 0.1411817	0.81174 -0.060583 0.0893437	0.84272 -0.0515694 0.0783607	0.00723 0.0845062 0.1276939
S_{de}	1.75E-04 0.1051936 0.1718313	1.75E-04 0.1051936 0.1718313	1.41E-04 0.1364145 0.2571068	0.1802 0.0356436 0.063444		0.08899 0.0762455 0.1129211	4.57E-06 0.1698057 0.5010378	0.1562 0.0189229 0.0537682	0.13868 0.0272771 0.0578486	3.99E-06 0.1670335 0.3592862
S_{cp}	0.05269 0.0711666 0.1046434	1.82E-12 0.4320334 0.9852123	0.0056 0.1040887 0.1692173		0.82766 -0.0356436 0.063444	0.35119 0.0431866 0.1129211	7.14E-05 0.1382834 0.2628703	0.58331 -0.0177176 0.0533026	0.63984 -0.008688 0.0507929	6.34E-04 0.1337879 0.2491243
S_{cn}	0.86566 -0.0350642 0.0630073	3.62E-08 0.3485752 0.9157307		0.99481 -0.1040887 0.1692173	0.99988 -0.1364145 0.2571068	0.97716 -0.0580785 0.0861047	0.2972 0.0339239 0.0621694	0.99948 -0.1224573 0.2163329	0.99474 -0.1127291 0.1904181	0.23955 0.0244664 0.0563088
S_{df}	1 -0.3806041 0.9538592		1 -0.3485752 0.9157307	1 -0.4320334 0.9852123	1 -0.4525159 0.9911484	1 -0.3905249 0.9623342	1 -0.3215081 0.8681811	1 -0.447514 0.9899345	1 -0.4387927 0.9874683	1 -0.3383434 0.8995477
S_{bf}		1.82E-11 0.3806041 0.9538592	0.13962 0.0350642 0.0630073	0.94851 -0.0711666 0.1046434	0.99985 -0.1051936 0.1718313	0.82584 -0.0249561 0.0565649	0.00479 0.0696562 0.1022994	0.99668 -0.0898325 0.1380828	0.99901 -0.0800353 0.1194966	0.0213 0.0616393 0.0907533

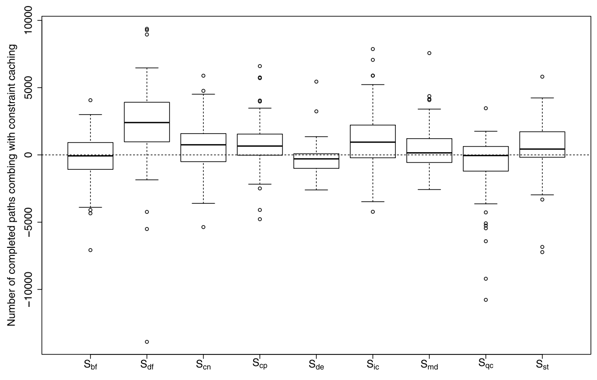
(c) Hypothesis test of branch coverage

p-value	0	0.05	0.2	0.5	1

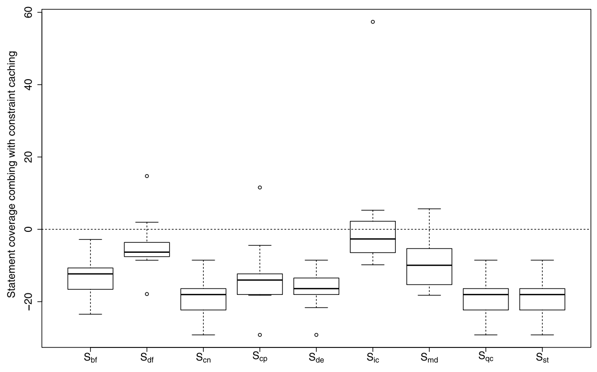
Fig. 4. Hypothesis test of ten approaches.

result of hypothesis tests with single constraint optimization, respectively.

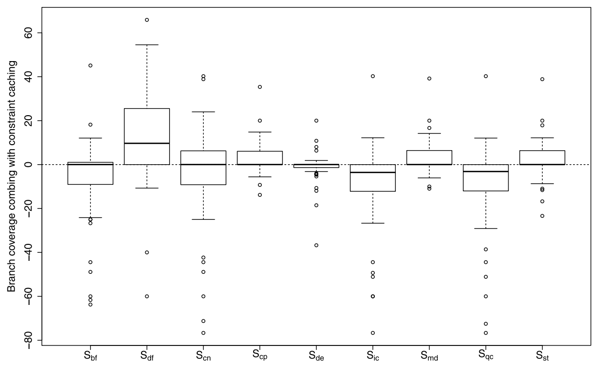
Fig. 5(a) shows that when using constraint caching, $\mathcal{S}_{de}$, $\mathcal{S}_{bf}$ and $\mathcal{S}_{qc}$ could complete more paths than *random-path* in most



(a) The difference of path numbers with constraint caching



(b) The difference of statement coverage with constraint caching

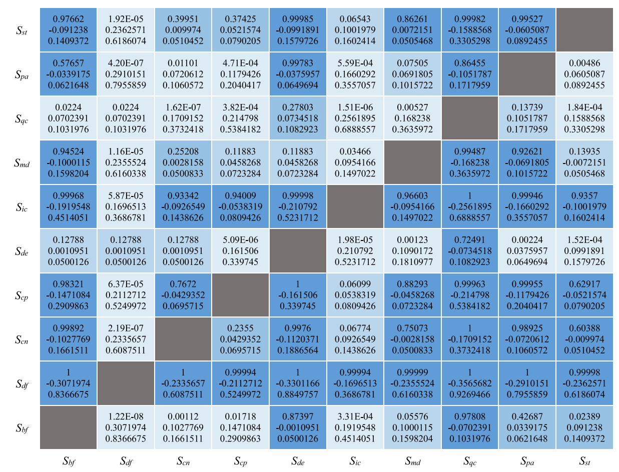


(c) The difference of branch coverage with constraint caching

Fig. 5. Hypothesis test of ten approaches with constraint caching.

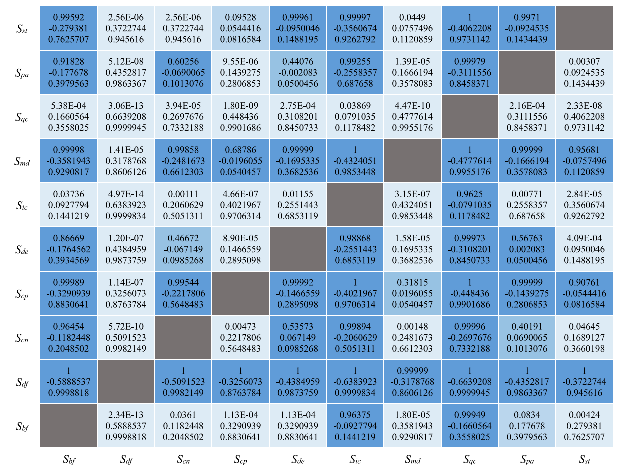
cases. The number of paths completed by the remaining six approaches, $\mathcal{S}_{df}$, $\mathcal{S}_{cn}$, $\mathcal{S}_{cp}$, $\mathcal{S}_{ic}$, $\mathcal{S}_{md}$, and $\mathcal{S}_{st}$, are lower than *random-path* completed in most cases. For statement coverage, the coverage of *random-path* is lower than of all nine other approaches. From Fig. 5(c), the branch coverage of $\mathcal{S}_{df}$ is lower than that of *random-path*, while the branch coverage of $\mathcal{S}_{ic}$ and $\mathcal{S}_{qc}$ are higher than that of *random-path* in most cases. $\mathcal{S}_{cp}$, $\mathcal{S}_{md}$ and $\mathcal{S}_{st}$ have the same coverage as *random-path* in half the cases, and have lower coverage than *random-path* in the other half. The coverage of $\mathcal{S}_{bf}$ and $\mathcal{S}_{de}$ are the same as that of *random-path* in about half of the cases, and are higher than that of *random-path* in most other cases. The coverage of $\mathcal{S}_{cn}$ higher and lower than that of *random-path* accounts for half, respectively.

Hypothesis test results in Figs. 6(a)–6(c) show that, under constraint caching optimization, the path completed by $\mathcal{S}_{qc}$ is overall optimal, which is significantly higher than seven approaches.



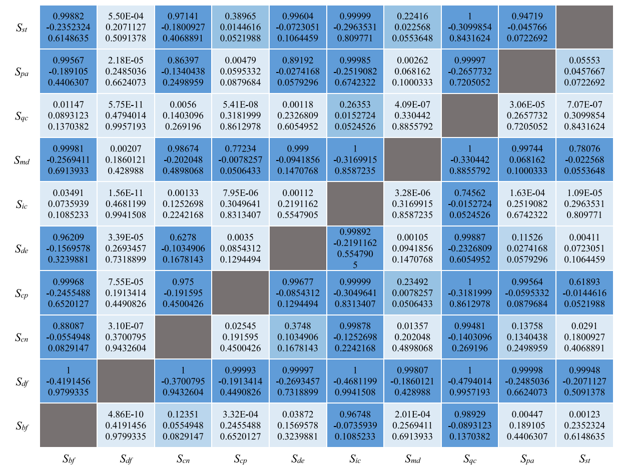
	S_{bf}	S_{df}	S_{cn}	S_{cp}	S_{de}	S_{ic}	S_{md}	S_{qc}	S_{pa}	S_{st}
S_{st}	0.97662 -0.091238 0.1409372	1.92E-05 0.2362571 0.6186074	0.39951 0.009974 0.0510452	0.37425 0.0521574 0.0790205	0.99985 -0.0991891 0.1579726	0.06543 0.1001979 0.1602414	0.86261 0.0072151 0.0505468	0.99982 -0.1588568 0.3305298	0.99527 -0.0605087 0.0892455	
S_{pa}	0.57657 -0.0339175 0.0621648	4.20E-07 0.2910151 0.7955859	0.01101 0.0720612 0.1060572	4.71E-04 0.1179426 0.2040417	0.99783 -0.0375957 0.0649694	5.59E-04 0.1660292 0.3557057	0.07505 0.0691805 0.1015722	0.86455 -0.1051787 0.1717959		0.00486 0.0605087 0.0892455
S_{qc}	0.0224 0.0702391 0.1031976	0.0224 0.0702391 0.1031976	1.62E-07 0.1709152 0.3732418	3.82E-04 0.214798 0.5384182	0.27803 0.0734518 0.1082923	1.51E-06 0.2561895 0.6888557	0.00527 0.168238 0.3635972		0.13739 0.1051787 0.1717959	1.84E-04 0.1588568 0.3305298
S_{md}	0.94524 -0.1000115 0.1598204	1.16E-05 0.2355524 0.6160338	0.25208 0.0028158 0.0500833	0.11883 0.0458268 0.0723284	0.11883 0.0458268 0.0723284	0.03466 0.0954166 0.1497022		0.99487 -0.168238 0.3635972	0.92621 -0.0691805 0.1015722	0.13935 -0.0072151 0.0505468
S_{ic}	0.99968 -0.1919548 0.4514051	5.87E-05 0.1696513 0.3686781	0.93342 -0.0926549 0.1438626	0.94009 -0.0538319 0.0809426	0.99998 -0.210792 0.5231712		0.96603 -0.0954166 0.1497022	1 -0.2561895 0.6888557	0.99946 -0.1660292 0.3557057	0.9357 -0.1001979 0.1602414
S_{de}	0.12788 0.0010951 0.0500126	0.12788 0.0010951 0.0500126	0.12788 0.0010951 0.0500126	5.09E-06 0.161506 0.339745		1.98E-05 0.210792 0.5231712	0.00123 0.1090172 0.1810977	0.72491 -0.0734518 0.1082923	0.00224 0.0375957 0.0649694	1.52E-04 0.0991891 0.1579726
S_{cp}	0.98321 -0.1471084 0.2909863	6.37E-05 0.2112712 0.5249972	0.7672 -0.0429352 0.0695715		1 -0.161506 0.339745	0.06099 0.0538319 0.0809426	0.88293 -0.0458268 0.0723284	0.99963 -0.214798 0.5384182	0.99955 -0.1179426 0.2040417	0.62917 -0.0521574 0.0790205
S_{cn}	0.99892 -0.1027769 0.1661511	2.19E-07 0.2335657 0.6087511		0.2355 0.0429352 0.0695715	0.9976 -0.1120371 0.1886564	0.06774 0.0926549 0.1438626	0.75073 -0.0028158 0.0500833	1 -0.1709152 0.3732418	0.98925 -0.0720612 0.1060572	0.60388 -0.009974 0.0510452
S_{df}	1 -0.3071974 0.8366675		1 -0.2335657 0.6087511	0.99994 -0.2112712 0.5249972	1 -0.3301166 0.8849757	0.99994 -0.1696513 0.3686781	0.99999 -0.2355524 0.6160338	1 -0.3565682 0.9269466	1 -0.2910151 0.7955859	0.99998 -0.2362571 0.6186074
S_{bf}		1.22E-08 0.3071974 0.8366675	0.00112 0.1027769 0.1661511	0.01718 0.1471084 0.2909863	0.87397 -0.0010951 0.0500126	3.31E-04 0.1919548 0.4514051	0.05576 0.1000115 0.1598204	0.97808 -0.0702391 0.1031976	0.42687 0.0339175 0.0621648	0.02389 0.091238 0.1409372

(a) Hypothesis test of path numbers with constraint caching



	S_{bf}	S_{df}	S_{cn}	S_{cp}	S_{de}	S_{ic}	S_{md}	S_{qc}	S_{pa}	S_{st}
S_{st}	0.99592 -0.279381 0.7625707	2.56E-06 0.3722744 0.945616	2.56E-06 0.3722744 0.945616	0.09528 0.0544416 0.0816584	0.99961 -0.0950046 0.1488195	0.99997 -0.3560674 0.9262792	0.0449 0.0757496 0.1120859	1 -0.4062208 0.9731142	0.9971 -0.0924535 0.1434439	
S_{pa}	0.91828 -0.177678 0.3979563	5.12E-08 0.4352817 0.9863367	0.60256 -0.0690065 0.1013076	9.55E-06 0.1439275 0.2806853	0.44076 -0.002083 0.0500456	0.99255 -0.2558357 0.687658	1.39E-05 0.1666194 0.3578083	0.99979 -0.3111556 0.8458371		0.00307 0.0924535 0.1434439
S_{qc}	5.38E-04 0.1660564 0.3558025	3.06E-13 0.6639208 0.9999945	3.94E-05 0.2697676 0.7332188	1.80E-09 0.448436 0.9901686	2.75E-04 0.3108201 0.8450733	0.03869 0.0791035 0.1178482	4.47E-10 0.4777614 0.9955176		2.16E-04 0.3111556 0.8458371	2.33E-08 0.4062208 0.9731142
S_{md}	0.99998 -0.3581943 0.9290817	1.41E-05 0.3178768 0.8606126	0.99858 -0.2481673 0.6612303	0.68786 -0.0196055 0.0540457	0.99999 -0.1695335 0.3682536	1 -0.4324051 0.9853448		1 -0.4777614 0.9955176	0.99999 -0.1666194 0.3578083	0.95681 -0.0757496 0.1120859
S_{ic}	0.03736 0.0927794 0.1441219	4.97E-14 0.6383923 0.9999834	0.00111 0.2060629 0.5051311	4.66E-07 0.4021967 0.9706314	0.01155 0.2551443 0.6853119		3.15E-07 0.4324051 0.9853448	0.9625 -0.0791035 0.1178482	0.00771 0.2558357 0.687658	2.84E-05 0.3560674 0.9262792
S_{de}	0.86669 -0.1764562 0.3934569	1.20E-07 0.4384959 0.9873759	0.46672 -0.067149 0.0985268	8.90E-05 0.1466559 0.2895098		0.98868 -0.2551443 0.6853119	1.58E-05 0.1695335 0.3682536	0.99973 -0.3108201 0.8450733	0.56763 0.002083 0.0500456	4.09E-04 0.0950046 0.1488195
S_{cp}	0.99989 -0.3290939 0.8830641	1.14E-07 0.3256073 0.8763784	0.99544 -0.2217806 0.5648483		0.99992 -0.1466559 0.2895098	1 -0.4021967 0.9706314	0.31815 0.0196055 0.0540457	1 -0.448436 0.9901686	0.99999 -0.1439275 0.2806853	0.90761 -0.0544416 0.0816584
S_{cn}	0.96454 -0.1182448 0.2048502	5.72E-10 0.5091523 0.9982149		0.00473 0.2217806 0.5648483	0.53573 0.067149 0.0985268	0.99894 -0.2060629 0.5051311	0.00148 0.2481673 0.6612303	0.99996 -0.2697676 0.7332188	0.40191 0.0690065 0.1013076	0.04645 0.1689127 0.3660198
S_{df}	1 -0.5888537 0.9998818		1 -0.5091523 0.9982149	1 -0.3256073 0.8763784	1 -0.4384959 0.9873759	1 -0.6383923 0.9999834	0.99999 -0.3178768 0.8606126	1 -0.6639208 0.9999945	1 -0.4352817 0.9863367	1 -0.3722744 0.945616
S_{bf}		2.34E-13 0.5888537 0.9998818	0.0361 0.1182448 0.2048502	1.13E-04 0.3290939 0.8830641	1.13E-04 0.3290939 0.8830641	0.96375 -0.0927794 0.1441219	1.80E-05 0.3581943 0.9290817	0.99949 -0.1660564 0.3558025	0.0834 0.177678 0.3979563	0.00424 0.279381 0.7625707

(b) Hypothesis test of statement coverage with constraint caching



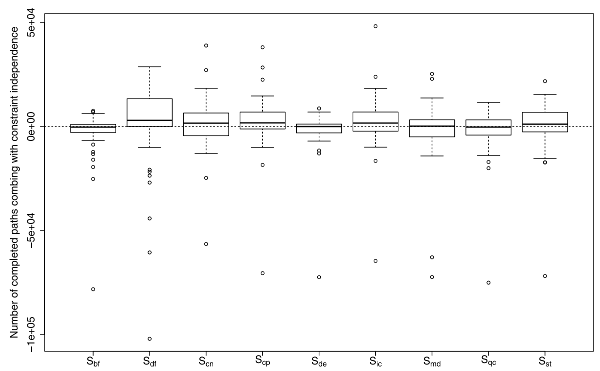
	S_{bf}	S_{df}	S_{cn}	S_{cp}	S_{de}	S_{ic}	S_{md}	S_{qc}	S_{pa}	S_{st}
S_{st}	0.99882 -0.2352324 0.6148635	5.50E-04 0.2071127 0.5091378	0.97141 -0.1800927 0.4068891	0.38965 0.0144616 0.0521988	0.99604 -0.0723051 0.1064459	0.99999 -0.2963531 0.809771	0.22416 0.022568 0.0553648	1 -0.3099854 0.8431624	0.94719 -0.045766 0.0722692	
S_{pa}	0.99567 -0.189105 0.4406307	2.18E-05 0.2485036 0.6624073	0.86397 -0.1340438 0.2498959	0.00479 0.0595332 0.0879684	0.89192 -0.0274168 0.0579296	0.99985 -0.2519082 0.6742322	0.00262 0.068162 0.1000333	0.99997 -0.2657732 0.7205052		0.05553 0.0457667 0.0722692
S_{qc}	0.01147 0.0893123 0.1370382	5.75E-11 0.4794014 0.9957193	0.0056 0.1403096 0.269196	5.41E-08 0.3181999 0.8612978	0.00118 0.2326809 0.6054952	0.26353 0.0152724 0.0524526	4.09E-07 0.330442 0.8855792		3.06E-05 0.2657732 0.7205052	7.07E-07 0.3099854 0.8431624
S_{md}	0.99981 -0.2569411 0.6913933	0.00207 0.1860121 0.428988	0.98674 -0.202048 0.4898068	0.77234 -0.0078257 0.0506433	0.999 -0.0941856 0.1470768	1 -0.3169915 0.8587235		1 -0.330442 0.8855792	0.99744 0.068162 0.1000333	0.78076 -0.022568 0.0553648
S_{ic}	0.03491 0.0735939 0.1085233	1.56E-11 0.4681199 0.9941508	0.00133 0.1252698 0.2242168	7.95E-06 0.3049641 0.8313407	0.00112 0.2191162 0.5547905		3.28E-06 0.3169915 0.8587235	0.74562 -0.0152724 0.0524526	1.63E-04 0.2519082 0.6742322	1.09E-05 0.2963531 0.809771
S_{de}	0.96209 -0.1569578 0.3239881	3.39E-05 0.2693457 0.7318899	0.6278 -0.1034906 0.1678143	0.0035 0.0854312 0.1294494		0.99892 -0.2191162 0.554790 5	0.00105 0.0941856 0.1470768	0.99887 -0.2326809 0.6054952	0.11526 0.0274168 0.0579296	0.00411 0.0723051 0.1064459
S_{cp}	0.99968 -0.2455488 0.6520127	7.55E-05 0.1913414 0.4490826	0.975 -0.191595 0.4500426		0.99677 -0.0854312 0.1294494	0.99999 -0.3049641 0.8313407	0.23492 0.0078257 0.0506433	1 -0.3181999 0.8612978	0.99564 -0.0595332 0.0879684	0.61893 -0.0144616 0.0521988
S_{cn}	0.88087 -0.0554948 0.0829147	3.10E-07 0.3700795 0.9432604		0.02545 0.191595 0.4500426	0.3748 0.1034906 0.1678143	0.99878 -0.1252698 0.2242168	0.01357 0.202048 0.4898068	0.99481 -0.1403096 0.269196	0.13758 0.1340438 0.2498959	0.0291 0.1800927 0.4068891
S_{df}	1 -0.4191456 0.9799335		1 -0.3700795 0.9432604	0.99993 -0.1913414 0.4490826	0.99997 -0.2693457 0.7318899	1 -0.4681199 0.9941508	0.99807 -0.1860121 0.428988	1 -0.4794014 0.9957193	0.99998 -0.2485036 0.6624073	0.99948 -0.2071127 0.5091378
S_{bf}		4.86E-10 0.4191456 0.9799335	0.12351 0.0554948 0.0829147	3.32E-04 0.2455488 0.6520127	0.03872 0.1569578 0.3239881	0.96748 -0.0735939 0.1085233	2.01E-04 0.2569411 0.6913933	0.98929 -0.0893123 0.1370382	0.00447 0.189105 0.4406307	0.00123 0.2352324 0.6148635

(c) Hypothesis test of branch coverage with constraint caching

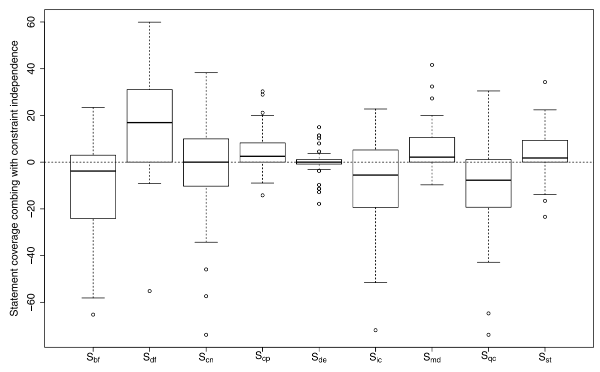
p-value	0	0.05	0.2	0.5	1

Fig. 6. Hypothesis test of ten approaches with constraint caching.

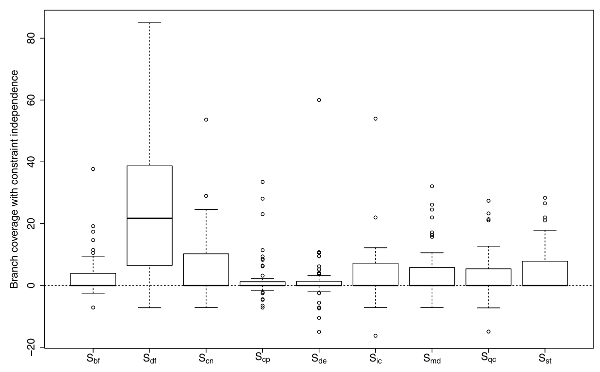
It also performs best for statement coverage. For branch coverage, $\mathcal{S}_{qc}$ and $\mathcal{S}_{ic}$ have no significant difference. Both of them are significantly higher than eight approaches. Thus, $\mathcal{S}_{qc}$ performs better than other approaches when using caching during symbolic execution.



(a) The difference of paths numbers with constraint independence



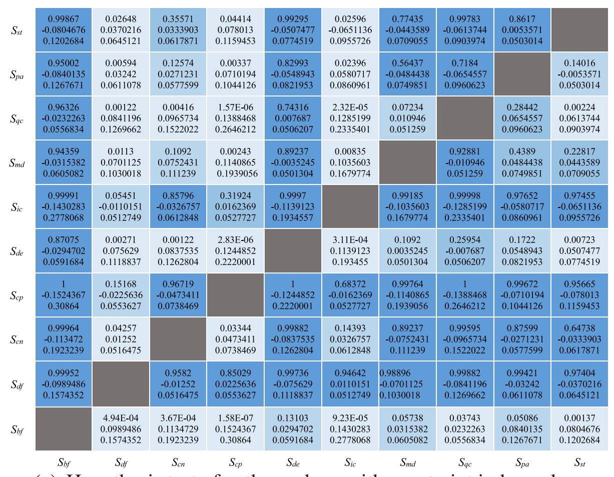
(b) The difference of statement coverage with constraint independence



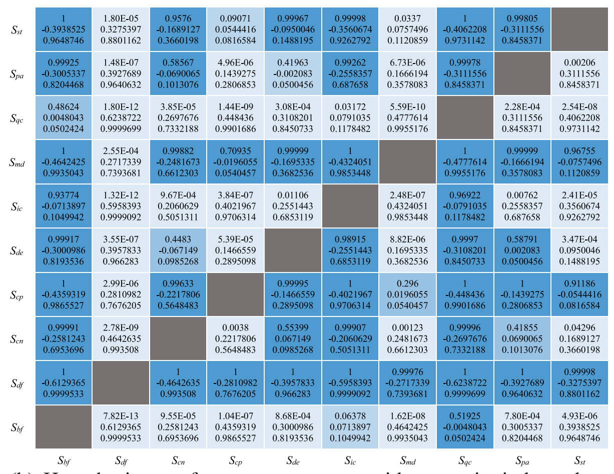
(c) The difference of branch coverage with constraint independence

Fig. 7. The difference between random-path and other approaches with constraint independence.

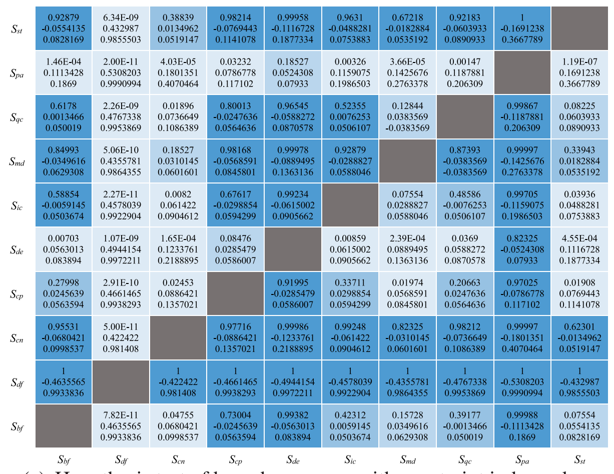
When using constraint independence, Fig. 7(a) shows that the numbers of paths completed by five approaches, $\mathcal{S}_{df}$, $\mathcal{S}_{cn}$, $\mathcal{S}_{cp}$, $\mathcal{S}_{ic}$, and $\mathcal{S}_{st}$, are lower than *random-path* completed in most cases. The number of paths completed by $\mathcal{S}_{de}$ and $\mathcal{S}_{md}$ greater and less than random path approach completed accounts for half, respectively. $\mathcal{S}_{bf}$ and $\mathcal{S}_{qc}$ could complete more paths than *random-path* in most cases. From Fig. 7(b), we found that the branch coverage of *random-path* is higher than that of four approaches, $\mathcal{S}_{df}$, $\mathcal{S}_{cp}$, $\mathcal{S}_{md}$, and $\mathcal{S}_{st}$, and lower than that of three approaches, $\mathcal{S}_{bf}$, $\mathcal{S}_{ic}$, and $\mathcal{S}_{qc}$. The branch coverage of $\mathcal{S}_{cn}$ and $\mathcal{S}_{de}$ higher and lower than that of *random-path* accounts for half, respectively, but the difference between $\mathcal{S}_{de}$ and *random-path* is negligible. Fig. 7(c) shows that the branch coverage of $\mathcal{S}_{df}$ is lower than that of *random-path* in most cases. And the branch coverage of the other eight approaches are the same as that of the *random-path*



(a) Hypothesis test of path numbers with constraint independence



(b) Hypothesis test of statement coverage with constraint independence



(c) Hypothesis test of branch coverage with constraint independence

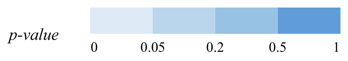
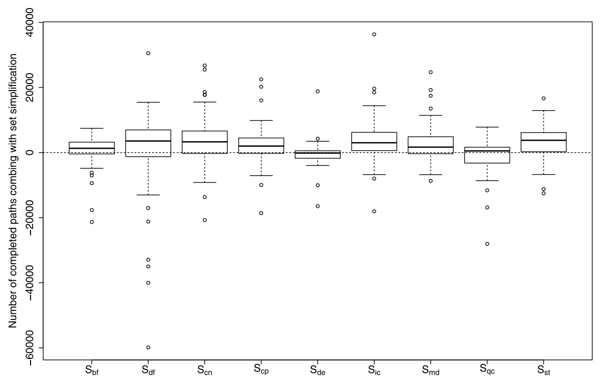
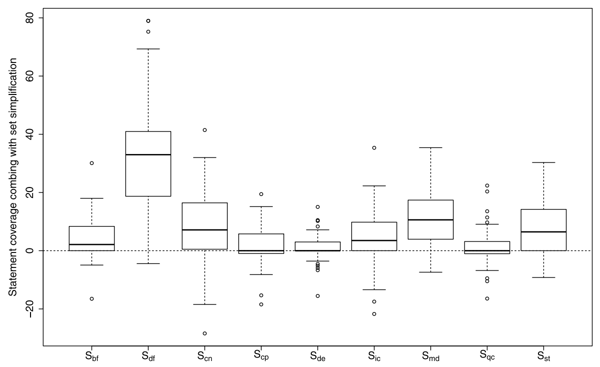


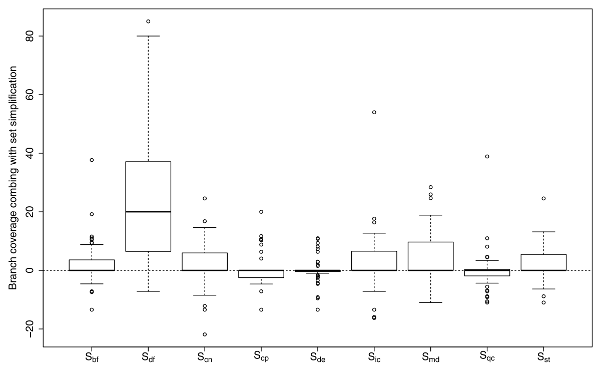
Fig. 8. Hypothesis test of ten approaches with constraint independence.



(a) The difference of path numbers with set simplification



(b) The difference of statement coverage with set simplification



(c) The difference of branch coverage with set simplification

Fig. 9. The difference between random-path and other approaches with set simplification.

approach in about half of the cases, and are lower than that of *random-path* in the other half cases.

Figs. 8(a)–8(c) shows that $\mathcal{S}_{bf}$ could complete more paths than other approaches, but it is only significantly more than five approaches. The statement coverage achieved by $\mathcal{S}_{qc}$ is significantly higher than eight approaches, and $\mathcal{S}_{pa}$ has significantly higher coverage than eight approaches. From the hypothesis test results, no one search path can win in most cases for all evaluation metrics combined with constraint independence optimization.

With constraint set simplification optimization, Fig. 9(a) shows only $\mathcal{S}_{de}$ completes more paths than *random-path* in most cases. The numbers of paths completed by the other eight approaches are lower than *random-path* completed in most cases. From Fig. 9(b), we found that the statement coverage of six approaches, $\mathcal{S}_{bf}$, $\mathcal{S}_{df}$, $\mathcal{S}_{cn}$, $\mathcal{S}_{ic}$, $\mathcal{S}_{md}$, and $\mathcal{S}_{st}$, are lower than that of *random-path*. The coverage of $\mathcal{S}_{de}$ is the same as that of *random-path* in about half of the cases, and lower than that of *random-path* in the other half cases. The statement coverage of $\mathcal{S}_{cp}$ and $\mathcal{S}_{qc}$ greater and less than that of *random-path* accounts for half, respectively. Fig. 9(c) shows that only the branch coverage of $\mathcal{S}_{df}$ is lower than *random-path* in most cases. The branch coverage of five approaches, $\mathcal{S}_{bf}$, $\mathcal{S}_{cn}$, $\mathcal{S}_{ic}$, $\mathcal{S}_{md}$ and $\mathcal{S}_{st}$, are the same as that of *random-path* in about half of the cases, and lower than that of *random-path* in the other half cases. $\mathcal{S}_{cp}$ and $\mathcal{S}_{qc}$ also have the same branch coverage as *random-path* in half the cases. In other half cases, their branch coverage are higher than *random-path*. The coverage of $\mathcal{S}_{de}$ is the same as that of *random-path* in most cases.

Fig. 10(a) shows that $\mathcal{S}_{de}$ is effective in completing more paths, significantly more than eight approaches. Figure shows that the statement coverage of $\mathcal{S}_{pa}$, $\mathcal{S}_{qc}$, $\mathcal{S}_{de}$ and $\mathcal{S}_{cp}$ are significantly higher than five approaches. $\mathcal{S}_{qc}$ performs best for branch coverage, since its branch coverage is significantly higher than six approaches. Like using constraint independence optimization, we calculate that no one search approach performs best under all evaluation metrics with constraint simplification optimization.

Moreover, we found that *DFS* has poor performance when using any of the three constraint optimization techniques. Fig. 6 shows that no matter which approach is compared with *DFS*, the *p*-value of Hypothesis test 1 is larger than 0.05, which means the number of completed paths, statement coverage and branch coverage of *DFS* are lower than other approaches.

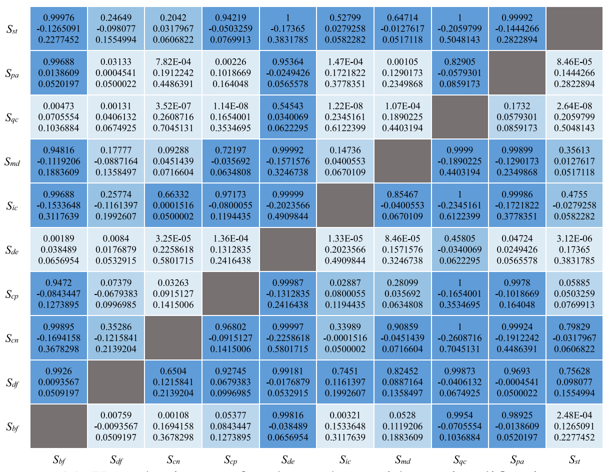
In summary, we believed that there is not a best path search approach in all situations when constraint optimization is used. The choice of path search strategy depends on the specific application of optimization technology. The *nurs-qc* search approach is suitable under constraint caching. In contrast, no one search approach performs best combined with independence or simplification optimization. Moreover, *DFS* is not a good choice when using any of the three constraint optimization techniques.

4.3. Analysis

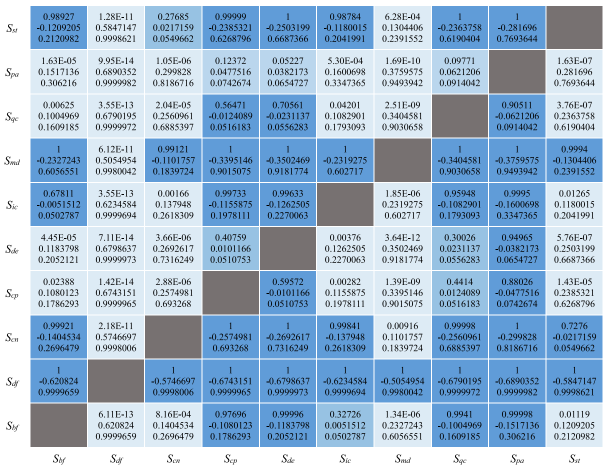
The above results show that the optimal search strategy is also different under different evaluation methods and optimization techniques. However, we found that *random-path* and *nurs-qc* perform better than other approaches when using no and caching optimization, respectively. To understand such a phenomenon, we analyzed the GNU CoreUtils programs structure programs and observed several reasons to explain the statistical results.

Firstly, GUN Coreutils programs contain plenty of conditions and decision statements. Thus, it costs many times to calculate the current state of symbolic execution and find the best condition to execute. Otherwise, *random-path* randomly selects a condition to execute, so it reduces the time cost of calculation. Moreover, *random-path* favors paths with fewer constraints (Chen et al., 2013), so it has greater freedom to hit uncovered code. Moreover, it also avoids starvation since execution paths with large numbers of branches cannot predominate execution chances with random selection. So coverage of tests generated with *random-path* is high.

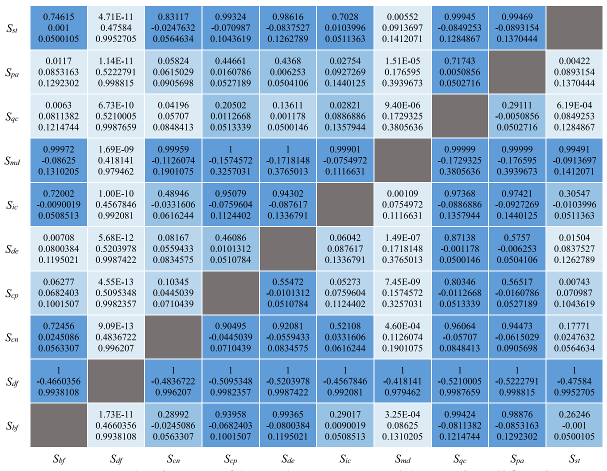
Secondly, by analyzing the path conditions we got in the study, we find heuristic strategies usually generate long and complex path conditions at first. Generating and solving these conditions is exceedingly expensive, so the number of completed paths with heuristic strategies is less than with *random-path*. Take program *join* as an example. When using *DFS*, the constraint conditions of paths that executed by KLEE in the beginning phase have high depth, which costs much time to generate this set of path conditions and solve it. There has an average of 51 constraints at the beginning of 22 sets of path conditions. However, path conditions of *random-path* are relatively short and straightforward, so the generation and solution of path conditions with *random-path* are much quicker than with *DFS*. For program *join* with *random-path*, the average number of constraints at the beginning of 22 sets of



(a) Hypothesis test of path numbers with set simplification



(b) Hypothesis test of statement coverage with set simplification



(c) Hypothesis test of branch coverage with set simplification

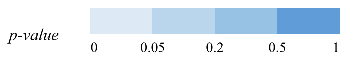


Fig. 10. Hypothesis test of ten approaches with set simplification.

path conditions is 16, much less than average with *DFS*. Solving short path conditions costs much less time than solving long path conditions.

```
set_prefix (char *p)
{
    char *s;
    prefix_lead_space = 0;
    while (*p == ' ')
    {
        prefix_lead_space++;
        p++;
    }
    quad prefix = p;
    prefix_full_length = strlen (p);
    s = p + prefix_full_length;
    while (s > p && s[-1] == ' ')
        s--;
    *s = '\0';
    prefix_length = s - p;
}
```

Fig. 11. A fragment of program *fmt*.

Thirdly, heuristic strategies are not practical for some program structures to find the following steps in a path. When there exist nonterminating loops with symbolic conditions in a program, heuristic strategies may get stuck. Fig. 11 is a fragment of program *fmt* containing many loops. We can find from the fragment that the loops could run plenty of times with high depth, so heuristic strategies may always select the loops with different depths. Our results show that the number of completed paths generated with heuristic strategies is much less than with *random-path*, and coverage with heuristic strategies is lower than with *random-path*.

Fourthly, verifying a small constraint set costs less time for constraint caching than verifying a large constraint set. *nurs-qc* selects the following path with minimum query cost, so the constraint selected by *nurs-qc* is more straightforward than selected by other approaches. Caching could quickly find out whether the constraint set has the solution. Hence, *nurs-qc* performs better than other approaches when using constraint caching. *DFS* usually generates long and complex path conditions at first and may get stuck when it encounters nonterminating loops, so it still performs worse than other approaches with optimization techniques.

4.4. Implications

In summary, *random-path* is the best choice to be used for KLEE-based test generation with no optimization, especially for programs which contains plenty of condition and decision statements such as nonterminating loops. It has greater freedom to hit uncovered path avoids starvation, so it could complete more path, and achieve higher coverage than heuristic path search strategies. Moreover, testers could select *nurs-qc* combined with constraint caching. But when using constraint independence or simplification, there is not a superior strategy, each approach have its own strengths, and testers should select suitable strengths according the evaluation metrics.

4.5. Threats to validity

Threats to internal validity are concerned with the uncontrolled factors that may also be responsible for the results. The major internal validity in our study is the parameter values in the experimental procedure. To reduce this threat, we applied the same parameter values of the time limit and timeout, which have been used in previous empirical studies about KLEE (Palikareva and Cadar, 2013). Another internal validity is the potential faults in our implementations and the tools we used to carry out our experiments. Gcov is a professional coverage analysis tool for code coverage. These well-known tools and techniques help reduce the internal threats to our empirical study to certain extends.

Threats to external validity are concerned with whether the findings in our experiments are generalizable for other settings. GNU Coreutils applications are real programs that form the core user-level environment of Unix systems. They have been used for previous test generation research. Moreover, KLEE is designed and configured specifically for GNU CoreUtils (Wang et al., 2015). We preferred to use these programs to reduce the external threats to our empirical study.

The evaluation metric is a major concern that may affect the experimental results. To reduce construct validity for our study, we used three popular metrics for assessing search strategies. Furthermore, we calculate the difference between random and heuristic strategies to study the utilization of search strategies in test generation. In the future, we will do additional studies on subject systems with mutation testing (Andrews et al., 2005; Do and Rothermel, 2006; Andrews et al., 2006) as the metric.

5. Related work

The are some research work about path search strategies for test generation. T. Xie et al. proposed an heuristic strategy, which named Fitness-Guided search approach, and compared with random strategy on 30 subjects (Xie et al., 2009). The experiment results shows that Fitness-Guided search approach is more effective in achieving high code coverage faster. Y. Li et al. introduced a approach to guide symbolic execution to less explored paths and compared with one random approach and six heuristic approaches (Li et al., 2013). T. Su et al. combined symbolic execution and model checking approach to automate data flow testing (Su et al., 2015). Their experiment results showed that it could reduce 40% testing time while improve data-flow coverage by 20% compared with state-of-the-art search strategies. W. Visser used state matching based techniques during test input generation and compared with random selection in terms of testing coverage (Visser et al., 2006). However, these work focused on new heuristic strategy and mainly compared with some heuristic strategy. In our paper, we focus on comparing randomness and intelligence for test generation. We compared eight heuristic approaches and two random approaches. Moreover, we used 53 subjects and three popular metrics to evaluate these approaches.

Many symbolic execution tools use heuristic path search strategies. For example, depth-first search is applied in early symbolic execution tools, such as EXE (Cadar et al., 2008b) and DART (Godefroid et al., 2005). P. Godefroid (Godefroid et al., 2008) et al. proposed generational search to select paths during symbolic execution. Generational search attempts to expand every constraint in the path constraint, so it could partially explore the state spaces of large applications executed with large inputs and very deep paths. Since the large number of constraints, the generated tests and code coverage are also maximum. Pex (Tillmann and De Halleux, 2008) devises a meta-strategy during symbolic execution. It makes a fair choice between all unexplored branches, which means that it partitions all branches into equivalence classes and picks a representative of the least often chosen class. Thus, it can avoid getting stuck in a particular area of the program. In recent years, some research has combined fuzzing testing to guide symbolic execution to improve the efficiency of vulnerability verification. Stephens (Stephens et al., 2016) and Zhang (Zhang and Thing, 2017) alternately explore program execution paths, hoping to guide fuzzing to explore more profound levels of the program through symbolic execution and solve slower growth of fuzzing code coverage. Ognawala et al. Ognawala et al. (2018) performed fuzzing and symbolic execution to improve the software's functional coverage. Sabbaghi et al. Sabbaghi et al. (2019) took into account factors such as path dependence and path length. They used a fuzzy search strategy to help symbolic execution find the following execution path, which improved the test efficiency. However, most of the papers only proposed the heuristic path search strategies, they do not study the effectiveness of these strategies.

6. Conclusion and future work

In this paper, we conducted an empirical evaluation and investigation between random and heuristic path search strategies using KLEE with 53 GNU Coreutils applications. We investigated both cases with and without constraint optimization techniques for KLEE-based test generation. The experimental results show that without optimization, although in some cases, the performance of the random approach and other approaches is not significantly different, however, in most cases, one approach from random strategy – *random-path* – universally performs better than other techniques in terms of completed paths and coverage metrics. These results indicate that *random-path* can be a better choice for KLEE-based test generation and heuristic path search strategies need further investigation. However, when combined with optimization, the selection of strategy depends on the specific optimization applied. We further analyze the reasons behind the statistical results and provide guidelines to select the appropriate path search strategy for KLEE-based test generation.

In the future, we will extend our work in two directions. Firstly, we plan to use mutation testing as the metric to evaluate path search strategies. Furthermore, we plan to empirically study the effectiveness of interleaving path search strategies which means using more than one strategy by turns during symbolic execution.

CRediT authorship contribution statement

Zhiyi Zhang: Conceptualization, Methodology, Software, Investigation, Writing – original draft, Supervision. **Ziyuan Wang:** Validation, Data curation, Writing – review & editing. **Fan Yang:** Formal analysis, Investigation. **Jiahao Wei:** Formal analysis, Investigation. **Yuqian Zhou:** Writing – review & editing. **Zhiqiu Huang:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86 (8), 1978–2001.

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering. ACM, pp. 402–411.

Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S., 2006. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Softw. Eng. 32 (8), 608–624.

Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H., 2008a. Automatically identifying trigger-based behavior in malware. In: Botnet Detection. Springer, pp. 65–88.

Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S., 2008b. Theory and techniques for automatic generation of vulnerability-based signatures. IEEE Trans. Dependable Secur. Comput. 5 (4), 224–241.

Bueno, P.M., Jino, M., Wong, W.E., 2014. Diversity oriented test data generation using metaheuristic search techniques. Inform. Sci. 259, 490–509.

Cadar, C., Dunbar, D., Engler, D.R., 2008a. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. Vol. 8. pp. 209–224.

Cadar, C., Engler, D., 2005. Execution generated test cases: How to make systems code crash itself. In: Model Checking Software. Springer, pp. 2–23.

Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R., 2008b. EXE: automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. 12 (2), 10.

Chen, T.Y., Yu, Y.-T., 1994. On the relationship between partition and random testing. IEEE Trans. Softw. Eng. 20 (12), 977–980.

Chen, T., Zhang, X.-s., Guo, S.-z., Li, H.-y., Wu, Y., 2013. State of the art: Dynamic symbolic execution for automated test generation. Future Gener. Comput. Syst. 29 (7), 1758–1773.

Chien, H.-Y., Huang, C.-Y., Fang, C.-C., 2021. Applying slicing-based testability transformation to improve test data generation with symbolic execution. Int. J. Perform. Eng. 17 (7).

Do, H., Rothermel, G., 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans. Softw. Eng. 32 (9), 733–752.

Godefroid, P., Klarlund, N., Sen, K., 2005. DART: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223.

Godefroid, P., Levin, M.Y., Molnar, D.A., et al., 2008. Automated whitebox fuzz testing. In: NDSS. Vol. 8. pp. 151–166.

Joshi, P., Sen, K., Shlimovich, M., 2007. Predictive testing: amplifying the effectiveness of software testing. In: The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. ACM, pp. 561–564.

King, J.C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, pp. 75–86.

Li, Y., Su, Z., Wang, L., Li, X., 2013. Steering symbolic execution to less traveled paths. In: ACM SigPlan Notices. Vol. 48. No. 10. ACM, pp. 19–32.

Li, J.J., Wong, W.E., 2002. Automatic test generation from communicating extended finite state machine (CEFSM)-based models. In: Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002, IEEE, pp. 181–185.

License, G., Gcov: Gnu coverage tool.

Majumdar, R., Sen, K., 2007. Hybrid concolic testing. In: 29th International Conference on Software Engineering. ICSE'07, IEEE, pp. 416–426.

Moser, A., Kruegel, C., Kirda, E., 2007. Exploring multiple execution paths for malware analysis. In: Security and Privacy, 2007. SP'07. IEEE Symposium on. IEEE, pp. 231–245.

Newsome, J., Brumley, D., Song, D., 2006. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. Internet Society.

Ognawala, S., Hutzelmann, T., Psallida, E., Pretschner, A., 2018. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1475–1482.

Palikareva, H., Cadar, C., 2013. Multi-solver support in symbolic execution. In: Computer Aided Verification. Springer, pp. 53–68.

Sabbaghi, A., Kanan, H.R., Keyvanpour, M.R., 2019. FSCT: A new fuzzy search strategy in concolic testing. Inf. Softw. Technol. 107, 137–158.

Schittkowski, K., 1986. NLPQL: A FORTRAN subroutine solving constrained nonlinear programming problems. Ann. Oper. Res. 5 (2), 485–500.

Sen, K., Marinov, D., Agha, G., 2005. CUTE: A Concolic Unit Testing Engine for C. Vol. 30. No. 5. ACM.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P., 2008. BitBlaze: A new approach to computer security via binary analysis. In: Information Systems Security. Springer, pp. 1–25.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In: NDSS. Vol. 16. No. 2016. pp. 1–16.

Su, T., Fu, Z., Pu, G., He, J., Su, Z., 2015. Combining symbolic execution and model checking for data flow testing. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 654–665.

Tillmann, N., De Halleux, J., 2008. Pex–white box test generation for. net. In: Tests and Proofs. Springer, pp. 134–153.

Visser, W., Păsăreanu, C.S., Pelánek, R., 2006. Test input generation for java containers using state matching. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. ACM, pp. 37–48.

Wang, X., Zhang, L., Tanofsky, P., 2015. Experience report: how is dynamic symbolic execution different from manual testing? a study on KLEE. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM, pp. 199–210.
Xie, T., Tillmann, N., De Halleux, J., Schulte, W., 2009. Fitness-guided path exploration in dynamic symbolic execution. In: Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on. IEEE, pp. 359–368.
Zhang, Z., Chen, Z., Gao, R., Wong, E., Xu, B., 2017. An empirical study on constraint optimization techniques for test generation. Sci. China Inf. Sci. 60 (1), 012105.
Zhang, L., Hou, S.-S., Hu, J.-J., Xie, T., Mei, H., 2010. Is operator-based mutant selection superior to random mutant selection? In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM, pp. 435–444.
Zhang, L., Thing, V.L., 2017. A hybrid symbolic execution assisted fuzzing method. In: TENCON 2017-2017 IEEE Region 10 Conference. IEEE, pp. 822–825.

Zhiyi Zhang received the M.S. degree and the Ph.D. degree both in Software Engineering from Nanjing University, China, in 2011 and 2016, respectively. He is currently an associate professor at the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. His research interests include software engineering and software testing.

Ziyuan Wang is an associate professor at School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, China. He received the B.S. degree in Mathematics and the Ph.D. degree in Computer Science from Southeast University, China, in 2004 and 2009, respectively. He worked as a Postdoctoral Researcher at Department of Computer Science and Technology, Nanjing University, China. His research interest mainly focuses on automated software testing techniques.

Fan Yang received the B.S. degree in Computer Science at Xi'an University of Finance and Economics, China in 2019. She is currently a postgraduate student in Software Engineering at Nanjing University of Posts and Telecommunications, China. Her research interest is software testing.

Jiahao Wei received the M.S. degree in Computer Science at Nanjing University of Posts and Telecommunications, China in 2020. He is currently engaged in cloud computing and big data work at China Mobile Communication Group. His research interest is software testing and cloud computing.

Yuqian Zhou received the M.S. degree in Probability and Mathematical Statistics from Zhengzhou University, China, in 2013 and the Ph.D. degree in Cryptography from Beijing University of Posts and Telecommunications, China, in 2018. From 2018 to now, she is a lecturer at Nanjing University of Aeronautics and Astronautics, China. Her research interest includes trust-worthy quantum random number generations and fundamental study of quantum nonlocality.

Zhiqiu Huang is a professor at the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China, and the director of Key Laboratory of Safety-Critical Software, Ministry of Industry and Information Technology, China. He received the M.S. degree and the B.S. degree from National University of Defense Technology, China, and the Ph.D. degree from Nanjing University of Aeronautics and Astronautics, China. His research interest is system software, software engineering, and formal methods, etc.