SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2017, Vol. 60 012105:1–012105:15
doi: 10.1007/s11432-015-0450-5

An empirical study on constraint optimization techniques for test generation

Zhiyi ZHANG[1], Zhenyu CHEN[1]*, Ruizhi GAO[2], Eric WONG[2] & Baowen XU[1]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093*, China;*
[2]*Department of Computer Science, University of Texas at Dallas, Richardson* 75080-3021*, USA*

Received November 13, 2015; accepted February 3, 2016; published online October 13, 2016

Abstract Constraint solving is a frequent, but expensive operation with symbolic execution to generate tests for a program. To improve the efficiency of test generation using constraint solving, four optimization techniques are usually applied to existing constraint solvers, which are constraint independence, constraint set simplification, constraint caching, and expression rewriting. In this paper, we conducted an empirical study, using these four constraint optimization techniques in a well known test generation tool KLEE with 77 GNU Coreutils applications, to systematically investigate how these optimization techniques affect the efficiency of test generation. The experimental results show that these constraint optimization techniques as well as their combinations cannot improve the efficiency of test generation significantly for ALL-SIZED programs. Moreover, we studied the constraint optimization techniques with respect to two static metrics, lines of code (LOC) and cyclomatic complexity (CC), of programs. The experimental results show that the "constraint set simplification" technique can improve the efficiency of test generation significantly for the programs with high LOC and CC values. The "constraint caching" optimization technique can improve the efficiency of test generation significantly for the programs with low LOC and CC values. Finally, we propose four hybrid optimization strategies and practical guidelines based on different static metrics.

Keywords test generation, symbolic execution, constraint solving, constraint optimization, static metric

Citation Zhang Z Y, Chen Z Y, Gao R Z, et al. An empirical study on constraint optimization techniques for test generation. Sci China Inf Sci, 2017, 60(1): 012105, doi: 10.1007/s11432-015-0450-5

1 Introduction

Software testing is critical for software development, but it is labor-intensive and time-consuming. Testing cost can usually account for 30%–50% of total development cost [1]. Hence, it is imperative to implement automatic methods to reduce the cost of software testing. In the past decades, automatic test generation has been intensively studied in both academic and industry [2, 3]. Combination of symbolic execution and constraint solving is one of the popular solutions of test generation [4].

Symbolic execution is an old program analysis method, proposed by King in the late 1970s [5]. It uses symbolic values instead of actual data as input values and uses symbolic expressions to represent program variable values [6]. Then it generates a symbolic path condition which is a set of symbolic

* Corresponding author (email: zychen@nju.edu.cn)

expressions for conditions in a path. However, it lacked effective tools for several insurmountable technical challenges at that time, so the development was at a standstill. In recent years, due to the development of computer hardware and compute capacity, symbolic execution has attracted renewed attention of researchers, especially used for test generation [7].

Path conditions, i.e., constraints, constructed by symbolic execution will be solved to generate test data1) by constraint solvers [8–11]. This is a challenging task of test generation, because constraint solving is a classical mathematic problem and non-decidable in general [8]. These challenges have hindered extensive industrialization of test generation. Although some effective constraint solvers [12] have been developed in recent years, the cost of constraint solving is still extremely expensive and leads to a large percentage of total cost of test generation. Therefore, some constraint optimization techniques have been proposed to simplify constraints (i.e., path conditions) or reduce queries of constraint solvers, as well as reduce cost of test generation.

Many test generation tools have been developed based on symbolic execution and constraint solving [6]. These tools are employed with some constraint optimization techniques, including constraint independence, constraint set simplification, constraint caching, and expression rewriting, to improve their efficiency. Therefore, it is valuable to systematically study on these optimization techniques for test generation. In this paper, we empirically evaluate these four popular constraint optimization techniques in a well known test generation tool KLEE [8] with 77 GNU Coreutils applications. To the best of our knowledge, this is the first study on evaluating constraint optimization techniques for test generation.

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

(1) We conducted experiments on KLEE by running 77 GNU Coreutils applications. The experimental results show that the optimization techniques and their combinations cannot improve the efficiency of test generation for ALL-SIZED of programs.

(2) We investigated the constraint optimization techniques with respect to two static metrics of programs. The experimental results show that the "constraint set simplification" technique can improve the efficiency of test generation significantly for the programs with high LOC and CC values. The "constraint caching" technique can improve the efficiency of test generation significantly for the programs with low LOC and CC values.

(3) Some guidelines are provided to use appropriate constraint optimization techniques based on different static metrics. Four hybrid optimization strategies are proposed to be used in practice.

The rest of paper is as follows: Section 2 introduces test generation, constraints solving, and constraint optimization used in our study. Section 3 presents our experiment comparing optimization techniques and their combinations. Section 4 investigates the constraint optimization techniques with respect to static metrics. Section 5 studies hybrid optimization strategies. Section 6 discusses threats to validity and related work. Conclusion and future work are drawn in the last section.

2 Background

2.1 Test generation

Many automatic test generation tools have been developed in the past decades [7]. In this paper, we study test generation based on symbolic execution [5] and constraint solving [13]. Symbolic execution uses control flow information to identify paths to be covered and constraint solving generates test data for these paths. As shown in Figure 1, given a program, the path search strategy explores candidate paths in a program. We use depth-first-search (DFS) strategy in our study. DFS traverses the execution tree from the root and explores as far as possible. Symbolic execution [5] uses symbolic values instead of actual data as input values and simulates program execution. It uses symbolic expressions to represent intermediate variable values and records program execution status. Finally, symbolic path conditions ("PC" for short)

1) We use test and test data interchangeably in this paper.

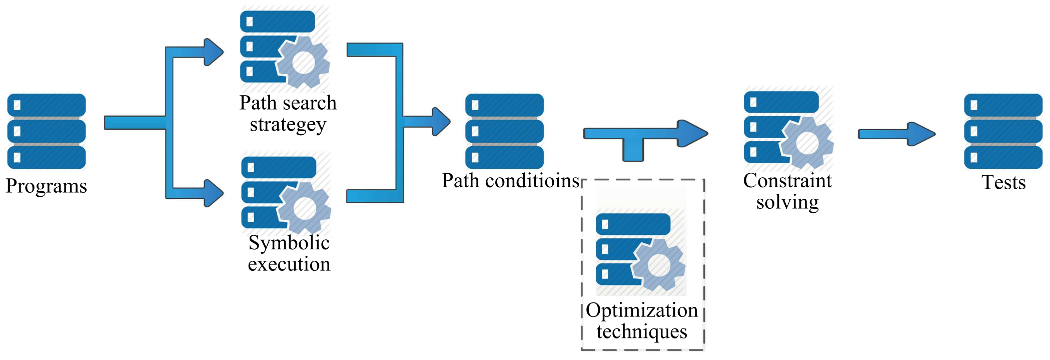

Figure 1 (Color online) Test generation based on symbolic execution and constraint solving.

```
function foo(int a, int b, int c, int d) {
1. if (d < 5)
2.         int e = b + c;
3.         if (a = e)
4.                 //do something
5.         else if (b > 10)
6.                         //do something
7.                 else
8.                         //do something
9.   else
10.     //do something
11. return
   }
```

Figure 2 An example program.

are generated. A PC is a set of constraints from program predicates with input parameters updated by symbolically executing program. A constraint solver is used to find solutions of PCs, i.e., constraints.

Symbolic execution has some practical challenges in handling large or complex programs. If execution paths are too long, then it is possible that the tools will not record all program execution status. If a program contains complex data types and external calls, the traditional symbolic execution may not be completed. In recent years, dynamic symbolic execution has been widely used in test generation [8–11, 14, 15], automated filter generation [16–20] and malware analysis [21–24]. Dynamic symbolic execution runs a program using concrete and symbolic inputs together. It uses some concrete values instead of symbolic values to significantly improve the efficiency of symbolic execution in practice.

We use a simple program, shown in Figure 2, to demonstrate how symbolic execution generates the condition of a path $\{1, 2, 3, 5, 6, 11\}$. x_1, x_2, x_3, and x_4 are symbolic inputs for a, b, c and d, respectively. In statement 1, we get the constraint $x_4 < 5$. In statements 2 and 3, an intermediate variable e is replaced by input parameters to get the constraint $x_1 = x_2 + x_3$. This process continues until this path has been completely executed. Finally, the path condition, i.e., a conjunction of all constraints in this path, is $(x_1 = x_2 + x_3) \wedge (x_2 > 10) \wedge (x_4 < 5)$.

2.2 Constraint solving

Constraint solving is a classical mathematic problem [13]. It is to find a solution of a set of constraints, e.g., a path condition for test generation [25–27]. A Boolean satisfiability (SAT) problem is a typical constraint solving problem and it is NP-complete [28]. A satisfiability modulo theory (SMT) [12] is an extension of SAT to solve more types of constraints, besides of Boolean constraints.

Some advanced constraint solvers, e.g., STP [29], Z3 [30], CVC3 [31], have been adopted in test generation tools. Using these constraint solvers, we can find a solution, i.e., test input, of path condition. For example, a solution $(x_1 = 17, x_2 = 11, x_3 = 6, x_4 = 1)$ can be generated by constraint solvers with the path condition $(x_1 = x_2 + x_3) \wedge (x_2 > 10) \wedge (x_4 < 5)$ for executing the path $\{1, 2, 3, 5, 6, 11\}$.

2.3 Constraint optimization

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 four representative constraint optimization techniques which are widely used for test generation [8].

2.3.1 *Constraint independence ($\mathcal{O}_{\rm i}$)*

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. Constraint independence optimization divides independence constraints into different constraint subsets. For example, the path condition $(x_1 = x_2 + x_3) \wedge (x_2 > 10) \wedge (x_4 < 5)$ could be divided into two groups $(x_1 = x_2 + x_3) \wedge (x_2 > 10)$ and $(x_4 < 5)$, since x_4 is independent of x_1, x_2 and x_3. 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.

2.3.2 *Constraint set simplification ($\mathcal{O}_{\rm s}$)*

New constraints are incrementally added into the path condition during symbolic execution. Constraint set simplification 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 $x_2 > 10$ in the path condition $(x_1 = x_2 + x_3) \wedge (x_2 > 10) \wedge (x_4 < 5)$. If a new constraint $x_2 > 100$ is added, then $x_2 > 10$ will be eliminated, because $x_2 > 100$ implies $x_2 > 100$. On the contrary, if the new added constraint is $x_2 < 5$, it is easier to detect that the path condition has no solution with two conflict constraints. Constraint set simplification can reduce queries of constraint solvers in this case.

2.3.3 *Constraint caching ($\mathcal{O}_{\rm c}$)*

Constraint caching 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, $(x > 10) \wedge (x < 5)$ is unsatisfiable. Then we can stop symbolic execution because any superset of $(x > 10) \wedge (x < 5)$ is unsatisfiable. The path condition $(x > 10) \wedge (y < 0)$ has a solution $(x = 11, y = -1)$, then two subsets, $x > 10$ and $y < 0$ have solutions. 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.

2.3.4 *Expression rewriting ($\mathcal{O}_{\rm r}$)*

Expression rewriting could transform the original path conditions into a much easier form before sending them to constraint solvers in order to speed up constraint solving. For example, the path condition $x + 0 > 5$ could be simplified to $x > 5$ and $2 \times x - x > 5$ could be rewritten as $x > 5$.

3 Comparing optimization techniques and their combinations

This paper is to empirically study constraint optimization techniques for test generation. The first research question is

(RQ1) Can the constraint optimization techniques or their combinations improve the efficiency of test generation?

3.1 Experiment design

KLEE [8], one of the most well known test generation tools, was used in our study. We adopted the depth-first-search (DFS) strategy in symbolic execution, with the constraint solver STP [29] for test generation. In our experiments, we used the computer with 2.66 GHz CPU and 4 GB memory.

99 GNU Coreutils applications were used as our experimental programs. 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 pervious academic studies. Secondly, KLEE is designed and configured specifically for GNU CoreUtils, thus GNU CoreUtils programs are stable enough running on KLEE and could reduce noises from software bugs [8, 32]. Lastly, library APIs may cause symbolic execution exploration to stop early and mask other potential limitations of symbolic execution. 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.

We discarded the applications for which either (a) our version of KLEE ran into unsupported LLVM instructionsor system calls (LLVM is a compilation infrastructure designed which could compile program language such as C language), or (b) KLEE finished in less than 360 s. Finally, 22 GNU Coreutils applications were discarded and 77 applications were used in our study.

For each program, we conducted test generation by KLEE without constraint optimization technique ($\mathcal{O}_{\rm non}$), with the single constraint optimization techniques ($\mathcal{O}_{\rm i}$, $\mathcal{O}_{\rm s}$, $\mathcal{O}_{\rm c}$, and $\mathcal{O}_{\rm r}$), and with the combined constraint optimization techniques ($\mathcal{O}_{\rm is}$, $\mathcal{O}_{\rm ic}$, $\mathcal{O}_{\rm ir}$, $\mathcal{O}_{\rm sc}$, $\mathcal{O}_{\rm sr}$, $\mathcal{O}_{\rm cr}$, and $\mathcal{O}_{\rm all}$), respectively. All together, there are 11 techniques. For example, $\mathcal{O}_{\rm is}$ is the combination of $\mathcal{O}_{\rm i}$ and $\mathcal{O}_{\rm s}$. $\mathcal{O}_{\rm all}$ is the combination of $\mathcal{O}_{\rm i}$, $\mathcal{O}_{\rm s}$, $\mathcal{O}_{\rm c}$, and $\mathcal{O}_{\rm r}$.

We give an example to show how to combine basic optimization techniques. Optimization combination means more than one basic optimization techniques are applied simultaneously during symbolic execution. For example, $(x_1 = x_2 + x_3) \wedge (x_2 > 10) \wedge (x_4 < 5)$ is a constraint set. When a new constraint $(x_2 > 5)$ is added. "Caching" is applied firstly to judge whether the new constraint set has a solution. Next, if it has a solution, "simplification" simplifies the constraint set. In our example, the constraint $(x_2 > 5)$ is eliminated because it is the sub-constraint of $(x_2 > 10)$. Lastly, "independence" divides the constraint set into two subsets — $(x_1 = x_2 + x_3) \wedge (x_2 > 10)$ and $(x_4 < 5)$.

We selected these seven combined methods for the following reasons:

Firstly, we consider that one optimization technique may affect the effectiveness of other optimization techniques during symbolic execution. That means, we consider that when using combined optimization techniques, symbolic execution may not perform better than when using single optimization techniques respectively. Similarly, though two single optimization techniques cannot improve the effectiveness of symbolic execution respectively, their combined optimization technique may improve the effectiveness. In order to make our experiment more convincing, we selected ($\mathcal{O}_{\rm is} - \mathcal{O}_{\rm cr}$) to find whether they perform better than non-optimization during symbolic execution.

Secondly, one hypothesis is that when using all optimization techniques, symbolic execution performs best. So in practical, researchers usually apply all optimization techniques during symbolic execution. To make our conclusion can be applied to practice and to prove whether this hypothesis is correct, in this paper, we selected $\mathcal{O}_{\rm all}$ in our study.

For each program, we set up the execution time to be 360 s for test generation. That is, we use 360 s to generate as many as paths and test data for a given program. During symbolic execution, KLEE may be frozen because of some reasons, such as complex paths, floating numbers and so on. To make programs can be executed continually, for each path, we set up the timeout to be 10 s for test generation. That is, if a path cannot be explored or solved in 10 s, we will skip it. Please note that the time cost includes test generation, path searching, symbolic execution, optimization, and constraint solving, for fair comparisons.

We set 10 s for test generation of each path for two reasons. One is that we only execute 360 s for each program. If the timeout value is too large, KLEE may be frozen with some reasons, such as floating numbers. If the value is too small, some complex paths may not be completed in this time. "10 s" is an appropriate value in our experiments. Other is that we consider practical application of symbolic execution in our experiment. In practical, if the timeout value is set too large or too small, KLEE cannot run continually. To make our results can be used in practical, we set timeout value as 10 s, so KLEE can

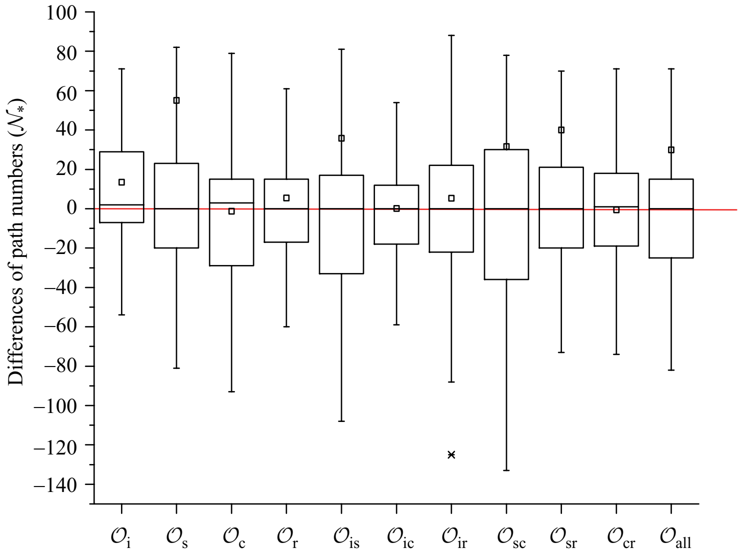

Figure 3 (Color online) $\mathcal{N}_*$ of each optimization technique.

Table 1 T-test of $\mathcal{N}_*$ for each optimization technique ($\alpha = 0.05$)

	$\mathcal{N}_\mathrm{i}$	$\mathcal{N}_\mathrm{s}$	$\mathcal{N}_\mathrm{c}$	$\mathcal{N}_\mathrm{r}$	$\mathcal{N}_\mathrm{is}$	$\mathcal{N}_\mathrm{ic}$	$\mathcal{N}_\mathrm{ir}$	$\mathcal{N}_\mathrm{sc}$	$\mathcal{N}_\mathrm{sr}$	$\mathcal{N}_\mathrm{cr}$	$\mathcal{N}_\mathrm{all}$
Average	13.5	55.0	−1.2	5.4	35.8	0.10	5.4	31.5	40.1	−0.61	29.9
P-value	0.09	0.17	0.89	0.53	0.27	0.99	0.44	0.34	0.23	0.93	0.35
Significant	×	×	×	×	×	×	×	×	×	×	×

run continually.

KLEE can provide the number of paths, denoted by $\mathcal{N}$, during test generation. A higher $\mathcal{N}$ indicates a better efficiency of test generation. In order to investigate the efficiency of constraint optimization techniques, we calculated the improvement of $\mathcal{N}$ for optimization techniques, i.e., $\mathcal{N}_{\mathcal{O}_*} - \mathcal{N}_{\mathcal{O}_\mathrm{non}}$, denoted by $\mathcal{N}_*$. For example, $\mathcal{N}_\mathrm{i} = \mathcal{N}_{\mathcal{O}_\mathrm{i}} - \mathcal{N}_{\mathcal{O}_\mathrm{non}}$ is the increased number of paths for the "constraint independence" optimization technique. A higher $\mathcal{N}_*$ indicates a better improvement of constraint optimization technique.

3.2 Experimental results and analysis

According to the number of completed paths in Appendix A, we collected $\mathcal{N}_*$ for each of the 11 optimization techniques and the 77 programs. The box-plots of optimization techniques are shown in Figure 3. The horizontal axis represents the optimization techniques and the vertical axis represents the increased number of paths ($\mathcal{N}_*$). It is surprising that all optimization techniques cannot improve the efficiency of test generation significantly for ALL-SIZED programs. The zero line crosses over all box-plots and is close to the median lines of all box-plots. Moreover, the averages of $\mathcal{N}_*$ (small boxes) of the optimization techniques $\mathcal{O}_\mathrm{c}$ and $\mathcal{O}_\mathrm{cr}$ are less than zero.

Furthermore, we collected the averages of $\mathcal{N}_*$ of each optimization technique, as shown in Table 1. The averages of $\mathcal{N}_*$ of $\mathcal{O}_\mathrm{c}$ and $\mathcal{O}_\mathrm{cr}$ are -1.2 and -0.61, respectively. It shows that the efficiency of test generation using these two optimization techniques are even worse than those without using optimization techniques. The average $\mathcal{N}_*$ of $\mathcal{O}_\mathrm{s}$ is the best one, 55.0. In order to investigate the experimental results, we did t-test for each optimization technique to $\mathcal{O}_\mathrm{non}$. All p-values are much larger than the standard significant threshold ($\alpha = 0.05$). That is, the differences of completed path numbers are not significant, even for $\mathcal{O}_\mathrm{s}$.

The experimental results from Figure 3 and Table 1 show that the constraint optimization techniques do not perform well. To insight into the results, we recorded the number of best times for each optimization technique. A "best" for one optimization technique means when using this optimization technique in test generation, it generates the most paths (and test data) among all optimization techniques (including

Table 2 The number of best times for each optimization technique

$\mathcal{O}_{\text{non}}$	$\mathcal{O}_{\text{i}}$	$\mathcal{O}_{\text{s}}$	$\mathcal{O}_{\text{c}}$	$\mathcal{O}_{\text{r}}$	$\mathcal{O}_{\text{is}}$	$\mathcal{O}_{\text{ic}}$	$\mathcal{O}_{\text{ir}}$	$\mathcal{O}_{\text{sc}}$	$\mathcal{O}_{\text{sr}}$	$\mathcal{O}_{\text{cr}}$	$\mathcal{O}_{\text{all}}$
4	8	7	8	8	3	8	10	6	8	12	7

$\mathcal{O}_{\text{non}}$). This number is denoted as $\mathcal{N}_{\mathcal{O}_*}$ which is the largest one for a given program. Table 2 shows the best times of each optimization technique. If we did not use any optimization technique ($\mathcal{O}_{\text{non}}$ in Table 2), we can get 4 times of best results from the 77 programs. The left 73 programs were covered by the 11 optimization techniques. For example, $\mathcal{O}_{\text{cr}}$ has 12 times of best results from the 77 programs. The experimental results show that constraint optimization techniques are useful for test date generation to some extend, but there is no one optimization technique which can win in most cases.

3.3 Explanation

From our experiment results, we found that all optimization techniques cannot improve the efficiency of test generation significantly for ALL-SIZED programs. We consider that there are some main reasons behind this phenomenon. For single optimization techniques, they may be not suitable for some programs. For example, if one program has few inputs and they are all dependent, so "independent" is not useable and cannot improve the effectiveness, but KLEE costs much time on "independent". For combined optimization techniques, two single optimization techniques may be conflict during symbolic execution. Take $\mathcal{O}_{\text{s}}$, $\mathcal{O}_{\text{c}}$ and $\mathcal{O}_{\text{sc}}$ for example, $\mathcal{O}_{\text{sc}}$ may cost much time to deal with constraint caching, so the performance of $\mathcal{O}_{\text{sc}}$ is worse than $\mathcal{O}_{\text{s}}$ but better than $\mathcal{O}_{\text{c}}$. Another reason is that some path conditions are not suitable to use combined optimization techniques. We introduce an example to show that the case is not suitable for the combination of $\mathcal{O}_{\text{s}}$ and $\mathcal{O}_{\text{r}}$. For a PC $2 \times x - x > 0 \wedge y > 0$, if $\mathcal{O}_{\text{r}}$ is used, then the PC is rewritten to $x > 0 \wedge y > 0$. When we add a new constraint $x > 10$ to the PC, if $\mathcal{O}_{\text{s}}$ is used, $2 \times x - x > 0$ is eliminated and the optimized PC is $x > 10 \wedge y > 0$. That means $\mathcal{O}_{\text{r}}$ can be skipped to avoid additional cost.

To understand the phenomenon of Table 2, we analysed symbolic execution and constraint solving information of some programs. We observed that using $\mathcal{O}_{\text{s}}$ completed much more paths than using other optimization techniques for some programs. Take the program "expr" for example, $\mathcal{O}_{\text{s}}$ completed 2433 paths while other optimization techniques only completed around 30 paths. The main reason is that many concentrated intermediate variables is in one block of "expr", so that it is easy to simplify them. For other programs, there may have many input variables. In this way, $\mathcal{O}_{\text{s}}$ costs much time to find the same variables to simplify them.

4 Optimization techniques w.r.t. static metrics

Based on the experimental results in the previous section, no one optimization technique can win in most cases for test generation. This motivates us to study the constraint optimization techniques with regard to program characteristics. Static metrics have been widely used in software engineering [33, 34]. We introduce two simple static metrics: Line of Code (LOC) and Cyclomatic Complexity (CC) to investigate how the effect of constraint optimization techniques changes with respect to different programs. The second research question is

(RQ2) Do and how the constraint optimization techniques depend on static metrics of programs?

4.1 Experiment design

Two simple and widely-used metrics, LOC and CC, were adopted in our experiments. LOC is the number of lines only containing source code, without comments and blank. CC is the cyclomatic complexity to measure the complexity of control-flow graph. It is defined by $E - N + 2P$, in which E is the number of edges in a CFG, N is the number of nodes in a CFG, P is the number of connected components in a CFG.

Table 3 Averages of LOC and CC w.r.t. "best" programs

	$\mathcal{O}_\mathrm{i}$	$\mathcal{O}_\mathrm{s}$	$\mathcal{O}_\mathrm{c}$	$\mathcal{O}_\mathrm{r}$	$\mathcal{O}_\mathrm{is}$	$\mathcal{O}_\mathrm{ic}$	$\mathcal{O}_\mathrm{ir}$	$\mathcal{O}_\mathrm{sc}$	$\mathcal{O}_\mathrm{sr}$	$\mathcal{O}_\mathrm{cr}$	$\mathcal{O}_\mathrm{all}$
LOC	426.5	577	255.4	365.9	270.7	332.7	764.5	610.5	323.6	290.7	672.5
CC	68.4	109.4	49.9	73.8	65.6	50.0	155.1	103.6	59.6	47.9	126.3

Table 4 Averages and p-values of top-10 and bottom-10 programs w.r.t. LOC ($\alpha = 0.05$)

	$\mathcal{N}_\mathrm{i}$	$\mathcal{N}_\mathrm{s}$	$\mathcal{N}_\mathrm{c}$	$\mathcal{N}_\mathrm{r}$	$\mathcal{N}_\mathrm{is}$	$\mathcal{N}_\mathrm{ic}$	$\mathcal{N}_\mathrm{ir}$	$\mathcal{N}_\mathrm{sc}$	$\mathcal{N}_\mathrm{sr}$	$\mathcal{N}_\mathrm{cr}$	$\mathcal{N}_\mathrm{all}$
T10A	9.0	11.8	−25.9	6.2	5.7	−1.1	6.6	−10.5	−25.2	−3.7	11.9
T10p	0.23	**0.04**	0.25	0.69	0.62	0.88	0.43	0.58	0.24	0.74	0.33
B10A	−16.7	11.4	10.6	−7.3	−4.7	−12.5	13.8	−28.2	−0.1	−16.4	−12.5
B10p	0.47	0.31	**0.02**	0.69	0.81	0.56	0.40	0.14	1.00	0.45	0.45

Table 5 Averages and p-values of top-10 and bottom-10 programs w.r.t. CC ($\alpha = 0.05$)

	$\mathcal{N}_\mathrm{i}$	$\mathcal{N}_\mathrm{s}$	$\mathcal{N}_\mathrm{c}$	$\mathcal{N}_\mathrm{r}$	$\mathcal{N}_\mathrm{is}$	$\mathcal{N}_\mathrm{ic}$	$\mathcal{N}_\mathrm{ir}$	$\mathcal{N}_\mathrm{sc}$	$\mathcal{N}_\mathrm{sr}$	$\mathcal{N}_\mathrm{cr}$	$\mathcal{N}_\mathrm{all}$
T10A	11.1	16.8	−16.5	7.7	5.3	−4.2	−0.4	−15.2	−23.8	−10.3	11.9
T10p	0.15	**0.00**	0.50	0.62	0.65	0.58	0.97	0.43	0.27	0.43	0.33
B10A	11.0	8.3	11.5	11.9	−2.0	9.2	15.0	0.0	−1.6	−7.6	4.8
B10p	0.32	0.40	**0.01**	0.11	0.83	0.31	0.12	1.00	0.89	0.34	0.33

We used Understand [2)], a popular static analysis tool for measuring and analysing program code, to obtain the static metrics of the 77 programs respectively. In our study, we only calculate the code of the program without library, white line or comment. For example, "ls" is the largest program (LOC=2812, CC=701) among the 77 experimental programs. Understand completed static analysis and generated the LOC and CC in 3 s for this program. Please note that the static analysis only did once. Hence, we can ignore the cost of static analysis in our experimental result analysis.

4.2 Experimental results and analysis

To study the constraint optimization techniques with regard to static metrics, we firstly calculated the average LOC and CC of "best" programs for each optimization techniques, as shown in Table 3. The experimental results show that the average of LOC w.r.t. $\mathcal{O}_\mathrm{c}$ is the smallest (255.4) and the average of LOC w.r.t. $\mathcal{O}_\mathrm{ir}$ is the largest (764.5). The average of CC w.r.t. $\mathcal{O}_\mathrm{c}$ is the smallest (49.9) and the average of CC w.r.t. $\mathcal{O}_\mathrm{ir}$ is the largest (155.1).

Moreover, we selected the top-10 and bottom-10 largest programs among the 77 experimental programs w.r.t. LOC and CC, respectively. The averages and p-values of $\mathcal{N}_*$ of optimization techniques are calculated in Tables 4 and 5. T10A and B10A mean the average differences of top-10 and bottom-10 programs, respectively. T10p and B10p mean the p-values of top-10 and bottom-10 programs, respectively. The experimental results show that the averages of $\mathcal{O}_\mathrm{s}$ are 11.8 and 16.8 for top-10 programs w.r.t. LOC and CC, respectively. The p-values of $\mathcal{O}_\mathrm{s}$ are 0.04 and 0.00 for top-10 programs w.r.t. LOC and CC, respectively. That is, $\mathcal{O}_\mathrm{s}$ can improve the efficiency of test generation significantly for top-10 programs w.r.t. LOC and CC. The experimental results also show that the averages of $\mathcal{O}_\mathrm{c}$ are 10.6 and 11.5 for bottom-10 programs w.r.t. LOC and CC, respectively. The p-values of $\mathcal{O}_\mathrm{c}$ are 0.02 and 0.01 for bottom-10 programs w.r.t. LOC and CC, respectively. That is, $\mathcal{O}_\mathrm{c}$ can improve the efficiency of test generation significantly for bottom-10 programs w.r.t. LOC and CC. Therefore, we suggest to use $\mathcal{O}_\mathrm{s}$ for large and complex programs and $\mathcal{O}_\mathrm{c}$ for small and simple programs in test generation.

In order to enable this suggestion to be used in practice, we classify programs into three types based on LOC [33]. The LOC of tenth largest and smallest program are 853 and 84, respectively. So the three program types based on LOC are small programs with LOC$\leqslant$100, medium programs with 101$\leqslant$LOC$\leqslant$800, and large programs with LOC$\geqslant$801. Similarly, We also classify programs into three types based on CC [34]: simple programs with CC$\leqslant$20, medium programs with 21$\leqslant$CC$\leqslant$150, and complex programs

2) Understand. http://scitools.com/.

Table 6 Averages and p-values of large, medium and small programs w.r.t. LOC ($\alpha = 0.05$)

	$\mathcal{N}_{\rm i}$	$\mathcal{N}_{\rm s}$	$\mathcal{N}_{\rm c}$	$\mathcal{N}_{\rm r}$	$\mathcal{N}_{\rm is}$	$\mathcal{N}_{\rm ic}$	$\mathcal{N}_{\rm ir}$	$\mathcal{N}_{\rm sc}$	$\mathcal{N}_{\rm sr}$	$\mathcal{N}_{\rm cr}$	$\mathcal{N}_{\rm all}$
L-A	10.8	12.8	−28.5	7.6	8.8	−0.8	6.0	−9.2	−25.7	−1.7	16.0
L-p	0.19	**0.04**	0.26	0.66	0.48	0.92	0.52	0.66	0.29	0.89	0.22
M-A	17.7	70.4	−0.9	5.2	47.39	0.4	1.2	48.4	58.4	0.0	40.1
M-p	0.08	0.21	0.94	0.65	0.29	0.97	0.90	0.29	0.21	1.00	0.37
S-A	−2.1	18.8	16.4	4.7	3.7	−0.7	22.7	−11.8	8.0	−2.5	−3.5
S-p	0.92	0.12	**0.02**	0.76	0.82	0.97	0.11	0.47	0.46	0.89	0.80

Table 7 Averages and p-values of complex, medium and simple programs w.r.t. CC ($\alpha = 0.05$)

	$\mathcal{N}_{\rm i}$	$\mathcal{N}_{\rm s}$	$\mathcal{N}_{\rm c}$	$\mathcal{N}_{\rm r}$	$\mathcal{N}_{\rm is}$	$\mathcal{N}_{\rm ic}$	$\mathcal{N}_{\rm ir}$	$\mathcal{N}_{\rm sc}$	$\mathcal{N}_{\rm sr}$	$\mathcal{N}_{\rm cr}$	$\mathcal{N}_{\rm all}$
C-A	10.0	14.7	−27.1	8.6	8.0	−1.2	7.3	−11.7	−26.4	−4.1	13.2
C-p	0.23	**0.01**	0.28	0.62	0.53	0.88	0.43	0.58	0.28	0.74	0.34
M-A	19.1	78.6	−1.2	6.9	53.4	1.6	1.6	54.3	63.3	−0.3	44.8
M-p	0.08	0.19	0.92	0.58	0.28	0.88	0.87	0.27	0.20	0.97	0.36
S-A	−1.3	5.4	12.5	−0.6	−2.2	−3.8	15.6	−13.9	5.6	0.4	−5.8
S-p	0.93	0.65	**0.04**	0.96	0.87	0.78	0.16	0.28	0.49	0.98	0.58

with CC$\geqslant$151.

Tables 6 and 7 shows the experimental results of three types of programs based on LOC and CC, respectively. L-A, M-A and S-A mean the average differences of completed path numbers of large, medium and small programs, respectively. L-p, M-p and S-p mean the p-values of large, medium and small programs, respectively. The averages of $\mathcal{O}_{\rm s}$ are 12.8 and 14.7 and the p-values of $\mathcal{O}_{\rm s}$ are 0.04 and 0.01 for large and complex programs. That is, $\mathcal{O}_{\rm s}$ can improve the efficiency of test generation significantly for large and complex programs. It can also be observed that the averages of $\mathcal{O}_{\rm c}$ are 16.4 and 12.5 and the p-values of $\mathcal{O}_{\rm c}$ are 0.02 and 0.04 for small and simple programs, respectively. In this way, $\mathcal{O}_{\rm c}$ can improve the efficiency of test generation significantly for small and simple programs. However, there is no significant conclusion can be derived for medium programs. The best one is $\mathcal{O}_{\rm i}$ with the p-value 0.08, which is close to $\alpha = 0.05$.

We suggest testers using an optimization technique that satisfies two conditions. One condition is that the average $\mathcal{N}_*$ for medium programs is larger than zero, which means symbolic execution could complete more paths when using this optimization technique. Thus, we do not suggest an optimization technique with the average $\mathcal{N}_*$ less than zero. Other condition is that the p-value is less than 0.05, which means the improvement is significant. If all p-values of optimization techniques, which the average $\mathcal{N}_*$ larger than zero, are less than 0.05, we prefer the optimization techniques that p-value is closed to 0.05, because we believe that it means the increase is nearly significant. In our experiment, though $\mathcal{O}_{\rm i}$ does not have the largest average $\mathcal{N}_*$ for medium programs, it has the smallest p-value for medium programs. On the other hand, though $\mathcal{O}_{\rm s}$ has the largest average $\mathcal{N}_*$ for medium programs, the p-value is much larger than 0.05, which means the increase is not significant. So in summary, since there is no one optimization technique that its p-value is less than 0.05, we suggest that testers can use $\mathcal{O}_{\rm i}$ for medium programs.

$\mathcal{O}_{\rm c}$ uses supersets and subsets for optimization based on the existing solutions of path conditions. A small and simple program usually consists of path conditions which are small and simple as well, such that it can easily generates some solutions for these path conditions. Hence, $\mathcal{O}_{\rm c}$ performs well on small and simple programs. $\mathcal{O}_{\rm s}$ simplifies the redundant constraints in path conditions. Because a large and complex program usually contains large and complex path conditions which are highly possible to have some redundant constraints. As a result, $\mathcal{O}_{\rm s}$ performs well on large and complex programs.

5 Hybrid optimization strategies

The guidelines with program classifying based on LOC and CC suggest testers to use $\mathcal{O}_{\rm c}$ for small (LOC$\leqslant$100) and simple (CC$\leqslant$20) programs and use $\mathcal{O}_{\rm s}$ for large (LOC$\geqslant$801) and complex (CC$\geqslant$150)

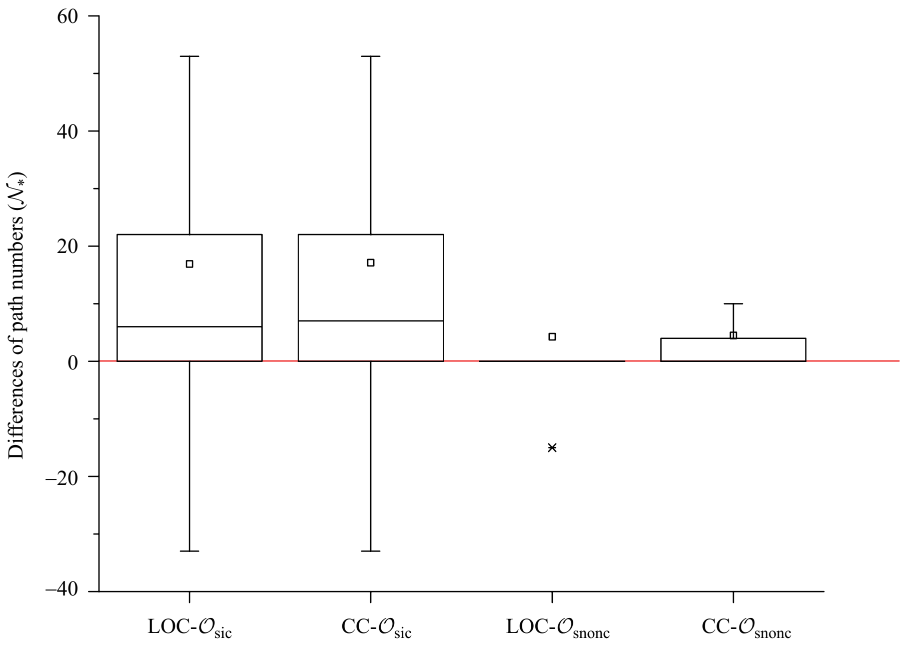

Figure 4 (Color online) Box-plots of $\mathcal{N}_*$ of hybrid optimization strategies.

Table 8 Averages and p-values of hybrid optimization strategies

	LOC-$\mathcal{O}_{\rm sic}$	CC-$\mathcal{O}_{\rm sic}$	LOC-$\mathcal{O}_{\rm snonc}$	CC-$\mathcal{O}_{\rm snonc}$
Average	16.9	17.2	4.3	4.9
P-value	0.02	0.02	0.00	0.00

programs. For medium programs, testers can use $\mathcal{O}_{\rm i}$ or skip constraint optimization techniques directly.

Based on the guidelines from the previous section, we can design some hybrid optimization strategies. First, given a program under test, we can calculate LOC and CC values by the Understand tool. Test generation tools can then use appropriate optimization techniques based on the LOC and CC values to generate test cases.

- LOC-$\mathcal{O}_{\rm sic}$: $\mathcal{O}_{\rm s}$ for large programs (LOC$\geqslant$801), $\mathcal{O}_{\rm i}$ for medium programs (101$\leqslant$LOC$\leqslant$800), and $\mathcal{O}_{\rm c}$ for small programs (LOC$\leqslant$100).
- CC-$\mathcal{O}_{\rm sic}$: $\mathcal{O}_{\rm s}$ for complex programs (CC$\geqslant$151), $\mathcal{O}_{\rm i}$ for medium programs (21$\leqslant$LOC$\leqslant$150), and $\mathcal{O}_{\rm c}$ for simple programs (LOC$\leqslant$20).
- LOC-$\mathcal{O}_{\rm snonc}$: $\mathcal{O}_{\rm s}$ for large programs (LOC$\geqslant$801), $\mathcal{O}_{\rm non}$ for medium programs (101$\leqslant$LOC$\leqslant$800), and $\mathcal{O}_{\rm c}$ for small programs (LOC$\leqslant$100).
- CC-$\mathcal{O}_{\rm snonc}$: $\mathcal{O}_{\rm s}$ for complex programs (CC$\geqslant$151), $\mathcal{O}_{\rm non}$ for medium programs (21$\leqslant$LOC$\leqslant$150), and $\mathcal{O}_{\rm c}$ for simple programs (LOC$\leqslant$20).

Figure 4 describes the box-plots of hybrid optimization strategies. The horizontal axis represents the hybrid optimization techniques and the vertical axis represents the increased number of paths. The whole boxes of LOC-$\mathcal{O}_{\rm sic}$, CC-$\mathcal{O}_{\rm sic}$, and CC-$\mathcal{O}_{\rm snonc}$ are above the zero line. The box of LOC-$\mathcal{O}_{\rm snonc}$ is degraded as a straight line and almost hit zero. Specially, for LOC-$\mathcal{O}_{\rm snonc}$, there are some outliers with positive value, and the values is larger than 60, so the outliers are not shown in the figure. Consider these outliers, the mean value of LOC-$\mathcal{O}_{\rm snonc}$ is higher than zero and the maximum value is beyond the range. Figure 4 suggests that LOC-$\mathcal{O}_{\rm sic}$ and CC-$\mathcal{O}_{\rm sic}$ have a better performance than LOC-$\mathcal{O}_{\rm snonc}$ and CC-$\mathcal{O}_{\rm snonc}$. Furthermore, the averages and p-values of hybrid optimization technique are shown in Table 8. The experimental results show that all hybrid optimization strategies can improve the efficiency of test generation significantly. The average improvement of LOC-$\mathcal{O}_{\rm sic}$ (16.9) and CC-$\mathcal{O}_{\rm sic}$ (17.2) are better than those of LOC-$\mathcal{O}_{\rm snonc}$ (4.3) and CC-$\mathcal{O}_{\rm snonc}$ (4.9).

6 Discussion

6.1 Threats to validity

Threats to internal validity are concerned with the uncontrolled factors that may be also responsible for the results. There are many constraint optimization techniques and symbolic execution tools. The selection of tools and optimization techniques may influence the results. In this paper, we include KLEE and four optimization techniques in our empirical study. KLEE is a well-known test generation tool which has been used in many research projects or real programs. Constraint independence $\mathcal{O}_{\rm i}$, constraint set simplification $\mathcal{O}_{\rm s}$, constraint caching $\mathcal{O}_{\rm c}$ and expression rewriting $\mathcal{O}_{\rm r}$ are also four popular optimization techniques. Moreover, Understand is a professional static analysis tool for measuring and analyzing program code bases. 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. The selection of experiment objects is a key point for our study. GNU Coreutils applications are real programs which form the core user-level environment of Unix systems. They have been used for previous test generation research. The 77 GNU Coreutils programs used in our study are also diversified in program size. We preferred to use these programs to reduce the external threats to our empirical study. In the future, we definitely will include more real-life programs to enhance our empirical study.

Threats to construct validity are concerned with the uncontrolled factors that may be also responsible for the results. The evaluation metric is a major concern that may affect the experimental results. Since the execution paths play a key role in software testing, we used the number of completed paths as the criterion to evaluate optimization techniques. Furthermore, we calculate the improvement of optimization techniques $\mathcal{N}_* = \mathcal{N}_{\mathcal{O}_*} - \mathcal{N}_{\mathcal{O}_{\rm non}}$ which is a reasonable evaluation metric to study the utilization of constraint optimization techniques in test generation.

Threats to conclusion validity are concerned with that can lead researchers to reach an incorrect conclusion about a relationship in the observations. To reduce the threats to the conclusion of our empirical study, we firstly count several characters of programs, including MaxCyclomatic, AvgEssential and so on. Then, for each program character, we analyze whether there exists a relationship between the character and the optimization technique, by evaluating the effectiveness of all, top-ten and bottom-ten programs, respectively. These ensure that we do not miss relationships or conclude a relationship when in fact there is not.

6.2 Related work

Ferguson and Korel [35] divided automatic test generation approaches to three classes: random, path-oriented and goal-oriented. Symbolic execution with constraint solving is a typical path-oriented test generation approach [7]. Random test data generation is considered to be the simplest method for test data generation. The study [36] indicated that random test generation always had low code coverage because of its narrow ranges of inputs. This paper studies the impact of constraint optimization techniques on test generation and does not study how test generation is performed in general.

Palikareva et al. [32] conducted an empirical study about multi-solver support to symbolic execution in test generation. In their study, they ran KLEE on GNU Coreutils applications with different constraint solvers, such as STP [29], Z3 [30] and Boolector [37]. They compared the execution time with the same number of instructions. The path search strategy DFS is adopted in test generation with or without constraint caching respectively. Their experimental results show that STP perform best. STP is initially designed specifically for EXE, which has the same types queries as KLEE. Their study only focused on the efficiency of different constraint solvers and did not compare the efficiency of constraint optimization techniques.

Erete et al. [38] proposed a constraint solving optimization method DomainReduce. DomainReduce which refers to restricting the domain for path conditions could help the constraint solver to find a solution faster than when considering the complete input domain. DomainReduce negates the last constraint in the path condition set that corresponds to a branch not yet covered after a path condition set has been solved. It finds the constraints which are independent from the negated constraint and reuses the existing values of these constraints. Finally, it solves the remaining symbolic variables and gets a solution. The result indicated that using DomainReduce optimization could help the constraint solver to find a solution with higher probability and faster. CUTE [11] employed incremental solving in test generation. That is, path conditions were solved incrementally during the path exploring process. If two path conditions from similar paths differ in a small number constraints, solutions could be reused to speed up test generation. Sen et al. used branch coverage as a criterion to evaluate CUTE. Their experimental results show that incremental solving was capable of drastically speeding up constraints solving since it reduced the size of normal path conditions between 49% with 88%. However, both DomainReduce and incremental solving are solving optimization techniques, not constraint optimization techniques. We will study the impact of these two types of optimization techniques on test generation in the future.

Cadar et al. [8] conducted a simple experiment to verify the efficiency of $\mathcal{O}_\mathrm{i}$ and $\mathcal{O}_\mathrm{c}$ in test generation. In their experiment, they firstly ran KLEE on GNU Coreutils applications for 5 min without optimization, then they rerun the same workload with constraint independence and constraint caching enabled separately and together for the same number of instructions. After that, they compared the running time. The result showed that for the same workload, using optimization will reduce much of time during symbolic execution. To the best of our knowledge, this paper is the first systematical study on constraint optimization techniques for test generation.

7 Conclusion and future work

In this paper, four single optimization techniques and seven combined optimization techniques were investigated. The experimental results show that these constraint optimization techniques cannot improve the efficiency of test generation significantly for ALL-SIZED programs. Furthermore, we investigated the constraint optimization techniques with regard to static metrics LOC and CC. It can be observed that $\mathcal{O}_\mathrm{s}$ works well for large and complex programs and $\mathcal{O}_\mathrm{c}$ works well for small and simple programs. Therefore, we proposed four hybrid optimization strategies in practice which are able to improve the efficiency of test generation significantly.

In the future, we will extend our work in two directions. Firstly, we plan to use code coverage and mutation testing as the metrics to evaluate optimization techniques. Furthermore, we plan to use more real-world programs as experiment objects.

Acknowledgements This work was supported in part by National Basic Research Program of China (973 Program) (Grant No. 2014CB340702) and National Natural Science Foundation of China (Grant Nos. 91418202, 61472178, 61373013).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Beizer B. Software Testing Techniques. 2nd ed. New York: International Thomson Computer Press, 1990
2 Fang C R, Chen Z Y, Xu B W. Comparing logic coverage criteria on test case prioritization. Sci China Inf Sci, 2012, 55: 2826–2840
3 Yang R, Chen Z Y, Zhang Z Y, et al. Efsm-based test case generation: sequence, data, and oracle. Int J Softw Eng Knowl Eng, 2015, 25: 633–667
4 Orso A, Rothermel G. Software testing: a research travelogue (2000–2014). In: Proceedings of the IEEE International Conference on Future of Software Engineering (ICSE). New York: ACM, 2014. 117–132
5 King J C. Symbolic execution and program testing. Commun ACM, 1976, 19: 385–394
6 Chen T, Zhang X-S, Guo S-Z, et al. State of the art: dynamic symbolic execution for automated test generation. Future Gener Comput Syst, 2013, 29: 1758–1773
7 Anand S, Burke E K, Chen T Y, et al. An orchestrated survey of methodologies for automated software test case generation. J Syst Softw, 2013, 86: 1978–2001

8 Cadar C, Dunbar D, Engler D R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2008. 209–224

9 Cadar C, Ganesh V, Pawlowski P M, et al. Exe: automatically generating inputs of death. ACM Trans Inf Syst Secur, 2008, 12: 10

10 Godefroid P, Klarlund N, Sen K. Dart: directed automated random testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2005. 40: 213–223

11 Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly With 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2005. 263–272

12 Barrett C, de Moura L, Stump A. Design and results of the first satisfiability modulo theories competition (smt-comp 2005). J Autom Reasoning, 2005, 35: 373–390

13 Schittkowski K. NLPQL: a fortran subroutine solving constrained nonlinear programming problems. Ann Oper Res, 1986, 5: 485–500

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

15 Godefroid P, Levin M Y, Molnar D A, et al. Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium, San Diego, 2008. 8: 151–166

16 Brumley D, Newsome J, Song D, et al. Towards automatic generation of vulnerability-based signatures. In: Proceedings of IEEE Symposium on Security and Privacy, Berkeley/Oakland, 2006. 15–16

17 Brumley D, Newsome J, Song D, et al. Theory and techniques for automatic generation of vulnerability-based signatures. IEEE Trans Depend Secure Comput, 2008, 5: 224–241

18 Brumley D, Wang H, Jha S, et al. Creating vulnerability signatures using weakest preconditions. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium, Venice, 2007. 311–325

19 Liang Z K, Sekar R. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. New York: ACM, 2005. 213–222

20 Newsome J, Brumley D, Song D. Vulnerability-specific execution filtering for exploit prevention on commodity software. In: Proceedings of the Network and Distributed System Security Symposium, San Diego, 2006. 58–71

21 Brumley D, Hartwig C, Kang M G, et al. Bitscope: Automatically Dissecting Malicious Binaries. School of Computer Science, Carnegie Mellon University, Technology Report CMU-CS-07-133. 2007

22 Brumley D, Hartwig C, Liang Z K, et al. Automatically identifying trigger-based behavior in malware. In: Botnet Detection. Berlin: Springer, 2008. 65–88

23 Moser A, Kruegel C, Kirda E. Exploring multiple execution paths for malware analysis. In: Proceedings of IEEE Symposium on Security and Privacy, Berkeley, 2007. 231–245

24 Song D, Brumley D, Yin H, et al. Bitblaze: a new approach to computer security via binary analysis. In: Information Systems Security. Berlin: Springer, 2008. 1–25

25 Chandra A K, Iyengar V S. Constraint solving for test case generation: a technique for high-level design verification. In: Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors, Cambridge, 1992. 245–248

26 DeMilli R A, Offutt A J. Constraint-based automatic test data generation. IEEE Trans Softw Eng, 1991, 17: 900–910

27 Gotlieb A, Botella B, Rueher M. Automatic test data generation using constraint solving techniques. ACM SIGSOFT Softw Eng Notes, 1998, 23: 53–62

28 Tovey C A. A simplified np-complete satisfiability problem. Discrete Appl Math, 1984, 8: 85–89

29 Ganesh V, Dill D L. A decision procedure for bit-vectors and arrays. In: Computer Aided Verification. Berlin: Springer, 2007. 519–531

30 de Moura L, Bjørner N. Z3: an efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2008. 337–340

31 Barrett C, Tinelli C. Cvc3. In: Computer Aided Verification. Berlin: Springer, 2007. 298–302

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

33 Jones C. Software metrics: good, bad and missing. Computer, 1994, 27: 98–100

34 Shepperd M. A critique of cyclomatic complexity as a software metric. Softw Eng J, 1988, 3: 30–36

35 Ferguson R, Korel B. The chaining approach for software test data generation. ACM Trans Softw Eng Meth, 1996, 5: 63–86

36 Offutt A J, Hayes J H. A semantic model of program faults. ACM SIGSOFT Soft Eng Notes, 1996, 21: 195–200

37 Brummayer R, Biere A. Boolector: an efficient smt solver for bit-vectors and arrays. In: Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2009. 174–177

38 Erete I, Orso A. Optimizing constraint solving to better support symbolic execution. In: Proceedings of IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Berlin, 2011. 310–315

Appendix A The number of completed paths

Program	$\mathcal{O}_{\rm non}$	$\mathcal{O}_{\rm i}$	$\mathcal{O}_{\rm s}$	$\mathcal{O}_{\rm c}$	$\mathcal{O}_{\rm r}$	$\mathcal{O}_{\rm is}$	$\mathcal{O}_{\rm ic}$	$\mathcal{O}_{\rm ir}$	$\mathcal{O}_{\rm sc}$	$\mathcal{O}_{\rm sr}$	$\mathcal{O}_{\rm cr}$	$\mathcal{O}_{\rm all}$
base64	899	908	893	894	920	904	896	920	886	893	917	893
cat	2143	2292	2225	2210	2294	2070	2021	2063	2017	2128	2006	2252
chcon	1168	1171	1087	1171	1186	1087	1150	1168	1102	1158	1155	1153
chgrp	1175	1190	1141	1229	1141	1116	1116	1170	1121	1155	1173	1134
chown	1136	1170	1155	1190	1134	1173	1137	1134	1170	1119	1159	1139
comm	1152	1150	1132	1134	1121	1134	1139	1135	1116	1099	1153	1118
cp	1099	1075	1084	1054	1039	1072	1057	1022	1056	1039	1057	1058
csplit	1075	1134	1098	1078	1172	1072	1072	1057	1057	1057	1078	1058
cut	912	916	1161	919	916	1137	888	888	1121	1134	890	1141
date	1042	1075	918	914	975	995	1075	1029	1028	1041	1057	961
dd	959	972	976	755	1025	1040	1005	992	1023	969	1030	1058
df	1241	1245	1241	1245	1247	1241	1243	1243	1241	1249	1245	1241
dir	545	616	755	696	758	697	545	626	763	717	758	545
dircolors	1906	2006	2062	2019	2019	2036	2036	1994	1984	2036	2067	1829
echo	202	206	198	208	208	191	208	207	183	198	206	192
env	1132	1222	1229	1209	1193	1209	1176	1185	1188	1202	1173	1193
expand	1878	1878	2145	1887	1899	2154	1899	1884	2076	2145	1896	2091
expr	38	15	2432	38	38	2430	38	15	2432	2432	44	2432
factor	450	901	377	882	898	629	855	782	761	945	423	375
fmt	234	224	235	226	225	235	234	224	235	234	236	236
fold	1697	1619	1621	1620	1619	1595	1603	1607	1603	1571	1603	1611
ginstall	1096	1096	1081	1081	1081	1081	1096	1096	1096	1081	1096	1096
groups	1859	1894	1897	1875	1908	1869	1901	1876	1881	1848	1879	1881
head	2866	2887	2901	2945	2866	2847	2835	2796	2819	2866	2800	2866
hostid	1126	1164	1162	1163	1156	1176	1160	1197	1176	1141	1161	1141
id	2808	2857	2817	2698	2718	2686	2792	2818	2825	2710	2857	2809
join	944	937	947	941	937	922	940	956	922	922	922	919
kill	1549	1549	1549	1549	1549	1549	1549	1549	1549	1549	1549	1549
link	1105	1051	1051	1110	1109	1051	1087	1106	1051	1051	1120	1088
ln	675	743	658	611	675	692	674	743	743	675	760	773
logname	1145	1194	1159	1157	1142	1141	1163	1195	1124	1192	1148	1145
ls	678	713	708	705	597	687	678	722	540	484	618	669
md5sum	3	3	1772	3	3	3	3	3	11	11	3	11
mkdir	1120	1105	1123	1123	1123	1105	1131	1123	1105	1147	1131	1123
mkfifo	1060	1060	1042	1075	1096	1060	1078	1131	1096	1043	1150	1060
mknod	1082	1049	1049	1049	1031	1085	1030	1028	1104	1013	1031	1013
mktemp	1077	994	996	995	1028	1012	1078	1046	1012	1012	1012	995
mv	1069	1093	1054	1074	1071	1054	1051	1054	1069	1036	1071	1069
nice	690	710	681	707	682	690	684	690	692	693	693	683
nl	2601	2530	2503	2508	2569	2608	2650	2651	2602	2602	2527	2561
od	1292	1292	1328	1292	1292	1340	1281	1322	1357	1340	1292	1344
paste	2936	2943	2927	2878	2846	2846	2858	2892	2873	2871	2903	2895
pathchk	366	378	346	368	367	333	365	376	326	383	366	366
pinky	1002	1002	981	981	1002	1133	1143	1153	1143	1133	1153	1143
pr	646	653	655	655	655	646	657	661	651	655	656	655
printenv	2045	2030	2071	2060	2060	1937	2071	1957	1945	2066	1931	2024
printf	1454	1507	1515	1425	1508	1557	1455	1538	1519	1557	1465	1469
ptx	1812	1796	1813	1755	1804	1771	1771	1780	1771	1771	1772	1820
readlink	93	93	95	98	95	95	95	93	96	95	93	95
rm	1038	1007	974	1006	1021	1024	1023	1007	1040	1042	1043	1007
rmdir	842	804	827	786	842	838	860	837	858	871	856	802
seq	946	946	946	946	946	946	946	946	946	946	946	946
setuidgic	903	903	870	924	889	870	870	907	888	906	921	888
sha1sum	1752	1871	1809	1882	1908	1979	1822	1762	1982	2004	1867	1794
shred	628	675	570	526	488	589	572	564	658	450	450	450
sleep	583	576	573	575	573	575	572	576	573	575	573	573
stat	1851	1851	1844	1882	1889	1807	1837	1844	1837	1837	1851	1837
stty	661	661	664	640	661	661	661	639	637	637	661	661
su	2133	2069	2097	2061	2045	2133	2025	2105	1787	2141	2081	2113

Continued on next page

Program	$\mathcal{O}_{\text{non}}$	$\mathcal{O}_{\text{i}}$	$\mathcal{O}_{\text{s}}$	$\mathcal{O}_{\text{c}}$	$\mathcal{O}_{\text{r}}$	$\mathcal{O}_{\text{is}}$	$\mathcal{O}_{\text{ic}}$	$\mathcal{O}_{\text{ir}}$	$\mathcal{O}_{\text{sc}}$	$\mathcal{O}_{\text{sr}}$	$\mathcal{O}_{\text{cr}}$	$\mathcal{O}_{\text{all}}$
sum	3490	3474	3287	3499	3477	3356	3380	3365	3404	3417	3391	3420
tac	2287	2308	2172	2306	2300	2224	2260	2253	2196	2243	2268	2259
tee	1832	1835	1849	1736	1811	1821	1818	1776	1832	1835	1727	1818
touch	494	494	494	494	494	538	539	538	537	537	538	537
tr	1085	1084	1096	1087	1086	1085	1076	1089	1089	1079	1086	1087
tsort	717	746	720	750	722	718	739	722	748	737	746	743
tty	1106	910	1174	1110	947	1226	922	1199	973	1131	956	959
uname	1075	1077	1078	1077	1078	1090	1078	1060	1090	1059	1060	1060
unexpand	920	926	919	923	918	918	911	912	908	907	901	912
uniq	457	479	479	472	469	469	469	496	472	479	472	479
unlink	1003	985	1002	1018	985	967	1002	1000	985	949	986	1021
uptime	1322	1307	1209	1279	1271	1268	1304	1289	1250	1306	1340	1289
users	1289	1340	1340	1304	1326	1307	1343	1322	1325	1341	1289	1307
vdirr	2374	2469	2434	2428	2418	2408	2474	2459	2345	2331	2413	2433
wc	1208	1219	1077	1034	1169	1019	1161	1103	1084	1085	1227	1126
who	1494	1490	1459	1465	1441	1520	1478	1516	1513	1525	1537	1565
whoami	1159	1159	1156	1169	1178	1142	1128	1163	1141	1177	1213	1174
yes	3	3	3	3	3	3	3	3	3	3	3	3