Hindawi
Mathematical Problems in Engineering
Volume 2020, Article ID 4258291, 12 pages
https://doi.org/10.1155/2020/4258291

Research Article

Infeasible Path Detection Based on Code Pattern and Backward Symbolic Execution

Yang Song, Xuzhou Zhang, and Yun-Zhan Gong

Beijing University of Posts and Telecommunications, Beijing, China

Correspondence should be addressed to Yang Song; isyang.song@gmail.com

Received 6 September 2019; Revised 11 April 2020; Accepted 30 April 2020; Published 25 May 2020

Academic Editor: Elena Zaitseva

This paper sets out to reveal the relationship between code pattern and infeasible paths and gives advices to the selection of infeasible path detection techniques. Lots of program paths are proved to be infeasible, which leads to imprecision and low efficiency of program analysis. Detection of infeasible paths is required in many areas of software engineering including test coverage analysis, test case generation, and security vulnerability analysis. The immediate cause of path infeasibility is the contradiction of path constraints, whose distribution will affect the performance of different program analysis techniques. But there is a lack of research on the distribution of contradict constraints currently. We propose a code pattern based on the empirical study of infeasible paths; the statistical result proves the correlation of the pattern with contradict constraints. We then develop a path feasibility detection method based on backward symbolic execution. Performance of the proposed technique is evaluated from two aspects: the efficiency of detecting infeasibility paths for specific program element and the improvement of applying the technique on code coverage testing.

1. Introduction

Static program analysis analyses the program without actually executing it, which plays an important role in software engineering such as verification [1] and validation [2]. A program path is infeasible if there is no input that can execute the program along the path. Many paths in a program are proved to be infeasible [3], which cause imprecision of static analysis, such as low coverage of automated software testing. So it is necessary to detect infeasible paths.

The primary cause of an infeasible path is the contradict logical formulas in path constraints. Due to the fact that the satisfiability of path constraints is undecidable, the detection of infeasible path is a significant scalability challenge. Existing techniques of infeasible path detection include matching contradict pattern [4] and checking the constraints' satisfiability by constraints solvers [5].

In recent years, intensive studies have been performed on infeasible path. Papadakis and Malevvris [6] formulated an automated symbolic execution tool that can be used for the detection of infeasible paths and generate test cases; the tool makes use of an efficient path heuristic, which is integrated with random testing, and efficiently handles the explosion of the path. Gong and Yao [5] formulated a tool for automatically identifying the branch correlations of various conditional statements by using the maximum likelihood estimation to detect infeasible paths. The achievement is important for improving the efficiency of software testing. Yano et al. [7] proposed the MOST approach, which is a search-based testing technique for the generation of a test case from Extended Finite State Machines (EFSM). MOST makes use of a multiobjective evolutionary algorithm so that the generation of test cases will be able to cover a given transition (test purpose). Wong et al. [8] suggested a method that utilizes the modified breadth first search with conflict checker, and a set of minimum Feasible Transition Paths (FTP) can be generated for each transition. They developed an EFSM executable model for algorithm modeling, algorithm verification, and performance assessment. Hermadi et al. [9] proposed and evaluated an approach for determining when further searches

for testing that covers exposed target paths are no longer worthwhile. It avoids the requirement for specifying limits to the number of search generations and can help in overcoming difficulties due to unfeasible pathing in search-based generation of data in tests for paths. Ruiz and Cassé [10] presented a new strategy for discovering unfeasible pathing in binary programs. Delahaye et al. [11] present a new approach to generalised unfeasible pathing from detections of single unfeasible paths and through this method demonstrated that this unfeasible path generalisation strategy compares with that of exhaustive unfeasible path detections in that the method can accelerate generating test inputs. Marashdih et al. [12] presented an approach for identifying cross site scripting (XSS) in PHP in accordance with genetic algorithms and static analyses, as well as another strategy to remove the detected XSS vulnerability from the source code [13, 14]. But as unfeasible pathing was eliminated manually, the strategy applies only to smaller programs.

In industry, the infeasible analysis is used for the programmable logic controllers (PLC); PLC are specially designed computers for automating electromechanical processes in industrial applications. There are works on formal verification of PLC [15–17]; the target PLC application such as single-task and multitask PLC programs [18] is first converted into formal model using code translation from source code to C code; then symbolic execution is performed on the C code, and after that verification is done.

The performances of infeasible path detection techniques are affected by different program structure, constraints type, distribution of contradict constraints, and so on. For example, the existence of input-dependent loop makes path explosion and unpredictable path length. Assuming that there are two contradict formulas at the end of a long infeasible path, analysing the path constraints reversely can immediately detect the contradiction, while forward technique has to traverse the whole path. This intuitively derives the idea of adopting appropriate analysis technique according to the various characters of infeasible paths. But there is lack of research on the distribution of contradict constraints to the best of our knowledge.

Symbolic execution is a static program analysis technique that explores program paths with symbolic values and collects execution information for further analysis. The applications of symbolic execution contain automated software testing [19], data-flow analysis [20], and worst-case execution time prediction [21]. In the application of infeasible paths detection, the path constraint collected by symbolic execution is solved by a satisfiability modulo theories (SMT) solvers. Symbolic execution can be classified into two kinds by the direction of exploration: forward symbolic execution and backward symbolic execution. Forward symbolic execution has been widely used in advanced automated testing tools [22, 23]. The application of backward symbolic execution is shown in Section 2.1.

This paper presents an empirical study on the common properties of the infeasible paths in real-world software program. Motivated by the result, we characterize the control-flow and data-dependency properties and conclude a code pattern. The validity of relationship between the pattern and infeasible paths is supported by statistical evidences. The path feasibility containing specific program elements of the code pattern is detected by backward symbolic execution. The experimental result shows the improvement of utilizing reverse analysis for the code pattern.

This paper contains the following contributions:

(i) We investigate the common properties of infeasible paths in real programs. These properties are presented for the first time.

(ii) Based on the investigation of infeasible paths, we propose a new code pattern and suppose its properties. The statistical results show the validity of the empirical properties.

(iii) We design an infeasible path detection technique for the proposed code pattern based on backward symbolic execution. The application of the pattern and detection technique is shown in code coverage testing. The experimental result confirms the promotion of the proposed method on the performance of covering specific program elements.

The remainder of this paper is organized as follows: Section 2 reformulates the path feasibility program and presents the research backgrounds. Based on the investigation, a code pattern is defined and illustrated with an example in Section 3. The architecture of the backward reverse infeasible path detection technique for the code pattern is described in Section 4. Section 5 implements experiments and presents analysis to the results. The related work about infeasible path detection and backward program analysis are introduced in Section 6. Section 7 concludes this paper and provides the direction of future work.

2. Background

2.1. Application of Reverse Analysis. Reverse program analysis explores a program in reverse direction, which starts from a specific target until reaching the program entry point, thus considering only those paths that can reach the target. Another advantage of reverse analysis is that the target statement can be utilized to raise the efficiency. The work of Yu et al. [24] utilizes an exploited string pattern on the target statement to refine analysis result reversely.

These advantages can be illustrated with an example in Figure 1 from [25]. Statement 9 is specified as the target; all program paths containing the target are infeasible, because the logical formulas of statement, 7 if ($y > 0$) and 8 if ($y == 0$), are contradictory. Utilizing forward analysis may exceed the time limit to check feasibilities of 2^n paths. Meanwhile, reverse analysis detects contradiction immediately. The performance gap grows significantly with the increase of infeasible path length.

2.2. Path Feasibility Problem. A program path p consists of a sequence of nodes $n_1, n_2, \ldots, n_k$ and the edges between nodes, each node n is a statement in the program, and k is the length of p.

```
1 void unreachable(int x1, int x2, int x3 ..., int xn) {
2    int y = 0;
3    if (x1 > 0) { y = y + 1; } else { y = y + 2; }
4    ...
5    if (xn > 0) { y = y + 1; } else { y = y + 2; }
6
7    if ((yy > 0) {
8          if ((yy == 0) { // Error condition for, e.g., division−by−zero
9                error();
10         }
11     }
12 }
```

Figure 1: A motivating example of reverse analysis.

Definition 1 (feasible path). A path p is said to be feasible if there exists an input $\vec{x}$ in the input domain D of the program, for which executing the program with $\vec{x}$ derives a same execution trace as p. The feasibility of p is equivalent to the satisfiability of the path constraints C_p [26].

Definition 2 (Contradict Constraint). The path constraint C is the conjunction of logical formulas $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_k$, where φ_i is the formula of node n_i. If C is satisfiable, there exists at least one input $\vec{x}$ in D satisfying C, which is denoted as C $C|=_D \vec{x}$; otherwise C is unsatisfiable if no input in D satisfies C; we say C is contradict constraint, denoted as $C|=_D \varnothing$. The symbol D is omitted in the following discussion for simplicity.

Definition 3 (branch correlation). A conditional branch has correlation along a path if the output of the branch can be determined by other statements or branches along this path [27]. Take two branches br_i and br_j as an example; if br_i and br_j have branch correlation, their outputs res_i and res_j have two relationships, which are represented by a function symbol $\oplus$. If res_i and res_j are true or false simultaneously, represented by $\mathrm{res}_i \oplus \mathrm{res}_j| = 0$, then φ_i and φ_j have Accord-Correlation. If res_i and res_j are contrary whenever the formulas are assigned by any concrete values, represented by $\mathrm{res}_i \oplus \mathrm{res}_j| = 1$, then φ_i and φ_j have contra-correlation, which is denoted as $(\varphi_i, \varphi_j)\perp$.

A former work [5] reveals that branch correlation is the sufficient condition of unsatisfiable path constraints.

Definition 4 (contradict position). Let p be an infeasible path, whose constraint is $C = \varphi_1, \varphi_2, \ldots, \varphi_k$; if there exists $(\varphi_i, \ldots, \varphi_j)\perp$ $1 \le i < \cdots < j \le k$, we say that the contradict position is from i to j.

If $k\text{-}i < i - 1$, analysis path constraints reversely can detect path infeasibility faster than forward analysis. But there is insufficient study on the distribution of contradict path constraints. This motivates us to investigate the character of contradict constraints before selecting program analysis techniques.

2.3. Character Investigation for Infeasible Path. The infeasible paths are selected from academic papers about infeasible paths detection [1, 5, 28, 29] and six open-source practical projects (WCET-benchmark (http://www.mrtc.mdh.se/projects/wcet/benchmarks.html), CoreUtil (http://www.gnu.org/software/coreutils/coreutils.html), libPNG (http://libpng.org/pub/png/libpng.html), sqlite (http://sqlite.org/), nanoxm-l (http://nanoxml.sourceforge.net/orig/), and IPEG-benchmark (http://www.ipeg.com/tag/benchmark-data/)). If the path contains function call, whose return value cannot be judged directly, the function is handled as an uninterpreted function. The return value of an uninterpreted function is considered to be arbitrarily value that satisfies path constraints. This approximation of external function call preserves strict consistency for path constraints, which is used by many analysis tools [19].

The characters of infeasible paths are categorized from two aspects: control flow and data flow. Loop structure is not only a fundamental control structure but also the reason for path explosion. In constraint satisfaction problem, equation formulas are more difficult to be satisfied compared to in-equation formula [30]. So we investigate the proportion of loop structures and equality operations in infeasible paths.

The results of investigation are presented in Figure 2. The proportions of equality operation and loop are 76% (Figure 2(a)) and 54% (Figure 2(b)) separately. A further investigation on infeasible paths containing loop shows that 60% of their contradict constraints are related to equality operation (Figure 2(b)).

The investigation gives a preliminary presupposition that the existences of loop structures and equality equations are relevant to infeasible paths. An intuitive explanation is that the loop structure makes path length unpredictable, leading to a high probability of infeasible path according to a conclusion in previous study [31]; the equality equation formula makes path constraints hard to satisfy.

Therefore, we define a code pattern to characterize a program containing loops and equality equations in the next section.

3. Loop-Dependent Equation Pattern (LDEP)

This section firstly reviews some basic terms of control flow and data flow and then defines the code pattern and illustrates with an example.

3.1. Preliminary Definitions. A control flow graph (CFG) is used to represent the control structure of the program. It is a directed graph and is normally denoted as $G(P) = \{N, E, n_e, n_0\}$, where N is a set of nodes, E is a set of edges, and $\mathrm{n_e}$ and $\mathrm{n_0}$ are respective unique entry and exit nodes to the graph. Each node $n \in \mathrm{N}$ is a statement in the program, and each edge $e = (n_i, n_j)$ is the transfer of control between statements.

The CFG is a directed cyclic graph if it contains loop structure. A loop structure is denoted as $L = (\mathrm{in}_L, \mathrm{out}_L)$, where in_L and $\mathrm{out_L}$ are the entry and exit nodes to the loop.

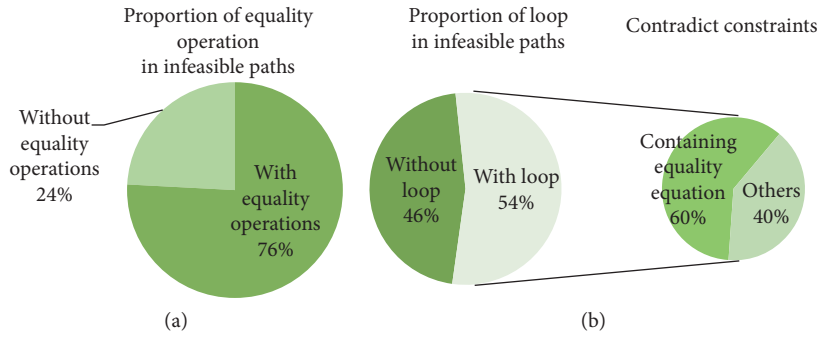

Figure 2: The distribution of loops and equations in infeasible paths.

In a CFG, a node n_j dominates a node n_i, if and only if every path from n_e to n_i contains n_j [31]. If there exist paths from n_i to n_j, $R(n)$ and $W(n)$ are the read and write operations to the memory location referenced by the variables of statement n, the node n_j has data dependency with n_i if formula (1) holds, and data dependency is denoted as $n_i \leftrightarrow n_j$.

$$\left(R(n_i) \cap W(n_j)\right) \cup \left[W(n_i) \cap R(n_j)\right] \cup \left[W(n_i) \cap W(n_j)\right] \neq \varnothing. \quad (1)$$

Let n be an *assignment statement* representing assigning an expression containing y to a variable v; we say the value of y propagates to v, which is denoted as $v \longrightarrow y$. Data propagation is a transitive relation in which $x \longrightarrow y$, $y \longrightarrow z \Rightarrow x \longrightarrow z$.

Based on these preliminary terms, we give the fundamental definitions for the code pattern.

Definition 5 (condition dependency). Let p be a path between two conditional statements n_i and n_j; if p does not contain assignment statement and $R(n_i) \cap R(n_i) \neq \varnothing$ or if p contains assignment statements $\{n_1, n_2, \ldots\}$and $x \in R(n_i) \wedge y \in R(n_j) \wedge x \mapsto y$, we say that n_i has condition dependency with n_j.

The condition dependency indicates that there exist constraints to same variables at different conditional statements, which is the necessary condition of condition correlation.

Definition 6 (dominate loop). Let $D(n) = dn_1, dn_2, \ldots, dn_m$ be the set of nodes dominating n; if dn_i and dn_i are the entry node and exit node of the same loop structure $L = \{dn_i, dn_j\}$, then L is the dominate j loop of n.

A node n and its dominate loop L manifest as a closed loop cycle before n in a CFG, where every path from the CFG's entry to n contains L.

Definition 7 (exclusive node). Let n be a node in a loop $L = \{in_L, in_L\}$; if every path from in_L to n contains pairs of entry node and exit node of another loop L' or does not contain any loop, then n is the exclusive node of L.

Definition 8 (concrete equation condition). Let $V_n = v_1, v_2, \ldots$ be the variable set of a conditional node n; if the logical formula of n contains an expression $v_i = N$, where N is a constant, then corresponding statement of n is a concrete equation condition.

Take example for C language; an expression containing equality operator and a constant operand is a concrete equation condition. For instance, the constraint of a conditional statement if $(v == 3)$ and its true branch requires the variable v to take a constant value of 3. Popular constraints solvers optimize the calculation by replacing variable with the constant 3 [32].

Besides the explicit form of concrete equation condition, some expressions also satisfy the definition implicitly. The implicit forms of concrete equation condition are listed in Table 1.

3.2. Loop-Dependent Equation Pattern (LDEP)

Definition 9. (loop-dependent equation pattern, *LDEP*). Let L be a loop structure and a concrete equation condition n_c; L is the dominate loop of n_c or n_c is the exclusive node of L; L and n_c constitute loop-dependent equation pattern if a node n_i in L has data dependency or condition dependency with n_c.

We introduce this definition to strictly describe a code pattern according to the result of the investigation in Section 2.3. Data dependency and condition dependency are the premises of branch correlation, which gains the possibility to infeasible paths. The fact that relevant loop cannot contain redefinition to the variables of n_c is worth the whistle, for the redefinition results in relocating memory for the variable, which violates the definition of data dependency [4]. Figure 3 gives three examples of *LDEP*. Assuming that these loops contain data-dependent or condition-dependent node of n_c, whether the program structures conform to *LDEP* or not is labelled with $\surd$ or $\times$. $L_4 + n_c$ and $L_5 + n_c$ are not LDEP because they are not dominate loops of n_c.

We assume an empirical property R of *LDEP*: the constraints of concrete-equation-condition node n can lead to contra-correlation, and the contradict position is close to n.

The validity of this property will be checked statistically in the next section.

3.3. Statistical Validation for Program Containing. In this section, we use binomial testing to check the validity of the proposed property. We first investigate the appearance of

Table 1: Occurrence of concrete equation condition in C language.

Grammar	Constraint
if($v == N$)	True branch: $v = N$
if($v! = N$)	False branch: $v = N$
if(v)	False branch: $-v \neq 0$
switch(v) case N:	Every case: $v = N$

LDEP in real software project. The statistic is drawn from a widespread open-source project Coreutils-8.24 (http://www.gnu.org/software/coreutils/coreutils.html). Coreutils provides some basic tools for Unix-like system. Table 2 gives the information of Coreutils and the appearance of *LDEP*.

In order to perform the testing, the property R to be tested is separated into two hypotheses:

(i) R_1: in a program containing *LDEP*, the constraints of concrete-equation-condition node n can lead to contra-correlation

(ii) R_2: the contradict position caused by *LDEP* is close to n

We do statics of infeasible paths of Coreutils. As the statistic of infeasible path is conservative, we set that R_1 holds for equal or more than 70%. R_2 is an assumption on the basis of R_1; the alternate hypothesis for R_2 holds for equal or more than 93%.

$$\begin{aligned} H_0 \text{ (null hypothesis): } & p(\mathfrak{R}_1) < 0.7, \\ & p(\mathfrak{R}_2) < 0.93, \\ H_1 \text{ (alternative hypothesis): } & p(\mathfrak{R}_1) \geq 0.7, \\ & p(\mathfrak{R}_2) \geq 0.93. \end{aligned} \tag{2}$$

The statistics c is computed as , taking $a = 0.05$ as the level of significance; u_{1-a} is the median of normal distribution. The statistics of sample size for R_1 and R_2 is shown in Table 3. The statistical result is conservative for the strategy of handling external function in Section 2.3. The cases in testing $x_0 = 170 \geq c_0 = 169$ and $x_1 = 163 \geq c_1 = 163$ give affirmative result. We conclude that $R_1$and R_2 hold at 5% level of significance.

In summary, the proposed property of program containing *LDEP* in Section 2.3 holds: the constraints of concrete-equation-condition node n can lead to contra-correlation, and the contradict position is close to n.

4. Infeasible Path Detection Based on Backward Symbolic Execution

To make use of the properties of *LDEP*, an automated coverage testing tool Hybrid-SymExe based on backward symbolic execution is proposed in this section. Compared with the general symbolic execution based coverage testing tool, Hybrid-SymExe consists of a general symbolic execution engine, a module recognizing *LDEP,* and a backward symbolic execution engine to search feasible path for a specific target.

4.1. Architecture Overview. The overview of Hybrid-SymExe's architecture is depicted in Figure 4.

Firstly, a pretreatment is used to detect the existence of *LDEP* in the program under test (PUT). PUT that does not contain *LDEP* is handled to a coverage testing process based on general symbolic execution. If the PUT contains *LDEP*, the program element of concrete equation condition is selected as the target denoted as *LDEP–e* according to specified coverage criteria. Automated finding of feasible path covering *LDEP–e* is achieved by backward symbolic execution, while other program elements are managed by the general technique.

4.2. Backward Symbolic Execution. The advantages of backward symbolic execution are utilizing the concrete value to optimize constraints and traversing CFG reversely to searching only in the target space. Compared to general symbolic execution, backward symbolic execution is different in that it starts execution from the target and then searches the program backward, which leads to two changes in implementation:

(i) Variables are used before definition. The backward symbolic execution searches the closest definition to symbolize the variable when meeting the variable at first time. This is similar to lazy initialization [33].

(ii) The assignment operation affects the collected constraints rather than future constraints construction. If a variable v in the collected constraints C is assigned by an expression *exp* in the next assignment statement, then all the occurrences of v in C are replaced by *exp*.

The algorithm of backward symbolic execution is described by pseudocodes as shown in Algorithm 1. We use the usual description in symbolic execution [34]. C and A represent conditional statement and assignment statement. C is the collection of constraints. Given a map M representing the memory, we use $M' := m + [m \mapsto v]$ to denote the map M' that is the same as M except that $M'(m) = v$.

An initial procedure firstly initializes the memory map M, path $path_n$, and search state *state* and then starts symbolic execution from n. If current statement is C, the SymbolicExecution procedure extracts the symbolic expression of C; if current statement is A, let a variable v in the collected constraints C be assigned by an expression *exp* at A; then the all occurrences of v in C are replaced by *exp*. Each time C is updated, its satisfiability is checked by the ConstraintsConsistent procedure. The search achieves a feasible path when it reaches n_e or returns null if it backtracks to the initial state.

4.3. Case Study. This section gives an example to explain the reverse and forward infeasible path detection work on *LDEP*. The example code in Figure 5 is selected from a practical project. Statement 9 is selected as the target to be covered. For simplicity, only loop structures and the statements

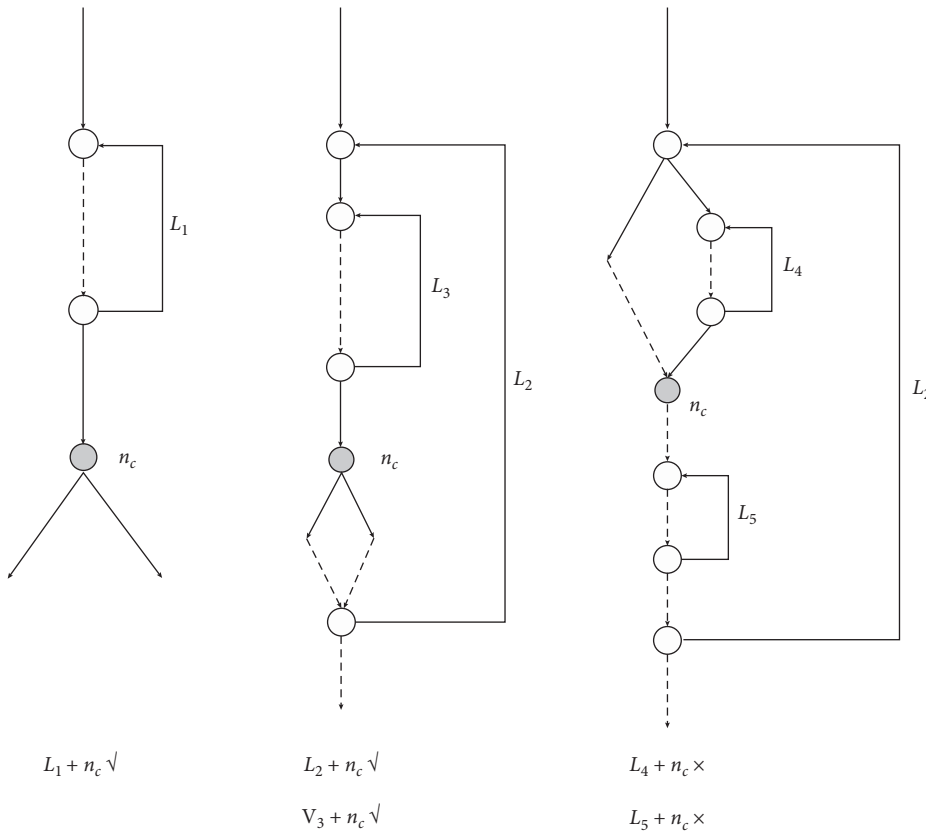

Figure 3: Example of LDEP.

Table 2: Appearance of LDEP in Coreutils.

Name	Files	LOC	LDEP	Proportion
Coreutils-8.24	174	60164	266	4.4/KLOC

Table 3: Samples for hypothesis validation.

$a = 0.05$	p	n	c	x
$\mathfrak{R}_1$	0.7	216	169	170
$\mathfrak{R}_2$	0.93	170	163	163

relevant to the target condition true branch, $i == 20$, are kept.

The value of i is added as different constant by the parity of i at every iteration of loop. The jump out condition of loop is related to the input variable n. This code contains typical case of *LDEP* because $i == 0$ has data dependency and conditional dependency with statements in the loop. The loop must execute at least six iterations $(i = 9, n = 20)$ to cover the target statement.

The search tree of utilizing forward and reverse analysis to find feasible path containing the target is presented in Figure 6. Lines represent constraints and operations. Value domains in the box are the values satisfying the constraints of incoming edges.

The left figure is the process of reverse analysis. In reverse search, the target condition truebranch, $i == 20$, can be used as a precondition to optimize the constraints. Then the search forks two possible operations $i = i + 3$ and $i = i + 1$ and checks satisfiability of each constraint at statement 4. It is obvious that the available value 19 for i contradicts the predicate "i is an even number"; hence this branch of search tree is pruned. In every round of loop iteration, the reverse search detects the contradiction and discards these paths.

Meanwhile, the process of using forward analysis to find feasible path is shown in the right figure. The input domains for both variables are sets of concrete values, with which both true and false branches of statement 4 are satisfiable. The search tree cannot be pruned, and also the size of value domain cannot be reduced.

This case shows that utilizing reverse analysis is more efficient than forward analysis in finding feasible path for the target in the given code containing *LDEP*.

5. Experimental Evaluation

In this section, we firstly report the results of a binomial testing to statistically validate the property presented in Section 3.2. The second experiment compares the performance of using backward symbolic execution with using the general symbolic execution on finding feasible path for *LDEP–e*. The last experiment performs the improvement of introducing backward technique to automated coverage testing as an auxiliary for specific program element.

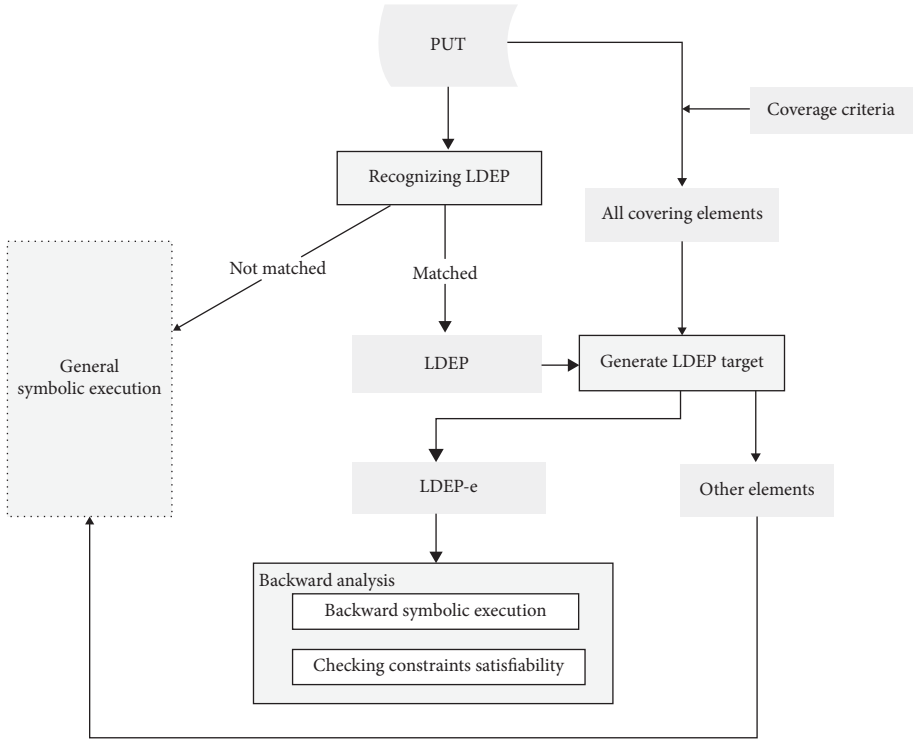

Figure 4: The architecture of Hybrid-SymExe.

Require: Target n, CFG $g = \{N, E, n_e, n_o\}$
Ensure: A feasible path from n_e to n path;
  if first then
    $\mathscr{M} \Leftarrow$ SYMBOLICINIT ($inputparams$);
    path $\Leftarrow$ INITPATH()
    state $\Leftarrow$ $initState$ ($\mathscr{C}, \mathscr{M}$);
  end if
  if n is in then
    Return $Path_{in}$;
  end if
  for each pre in precessors of n do
    STORESTATE();
    $b\_flag \Leftarrow$ EDGEOF(pre, n);
    if pre is C then
      $\mathscr{C} \Leftarrow \mathscr{C} \wedge$SymbolicExecution ($pre$, b_flag);
  end if
  if pre is A then
    [x: $symbolExpression$] $\Leftarrow$ SYMBOLICEXECUTION(pre);
    $\mathscr{M}$; = $\mathscr{M}$ + [$x \mapsto x'$];
    $\mathscr{C} \Leftarrow$ REPLACE X WITH EXPRESSION ((x: = $SymbolicExpression$]);
  end if
  if IsCONSTRAINTSCONSISTENT ($\mathscr{C}$) is $TRUE$ then
    BACKWARDSEARCH (pre, $state \Leftarrow (\mathscr{C}, \mathscr{M}, path \wedge n)$);
  else
    RESTORE STATE();
    continue;
  end if
end for

Algorithm 1: Search feasible path for n BACKWARDSEARCH.

```
1  input: i:[0,10], n:[−inf,+inf]
2  void foo(int i, int n){
3        while(i < n){-//1.loop structure
4              if(i%2 == 0)
5                    i = i + 1;//2data dependency with if condition
6              else
7                    i = i + 3;
       }
8        if(i == 20) // 3. concrete equation condition
9          target;
}
```

Figure 5: An example code containing LDEP.

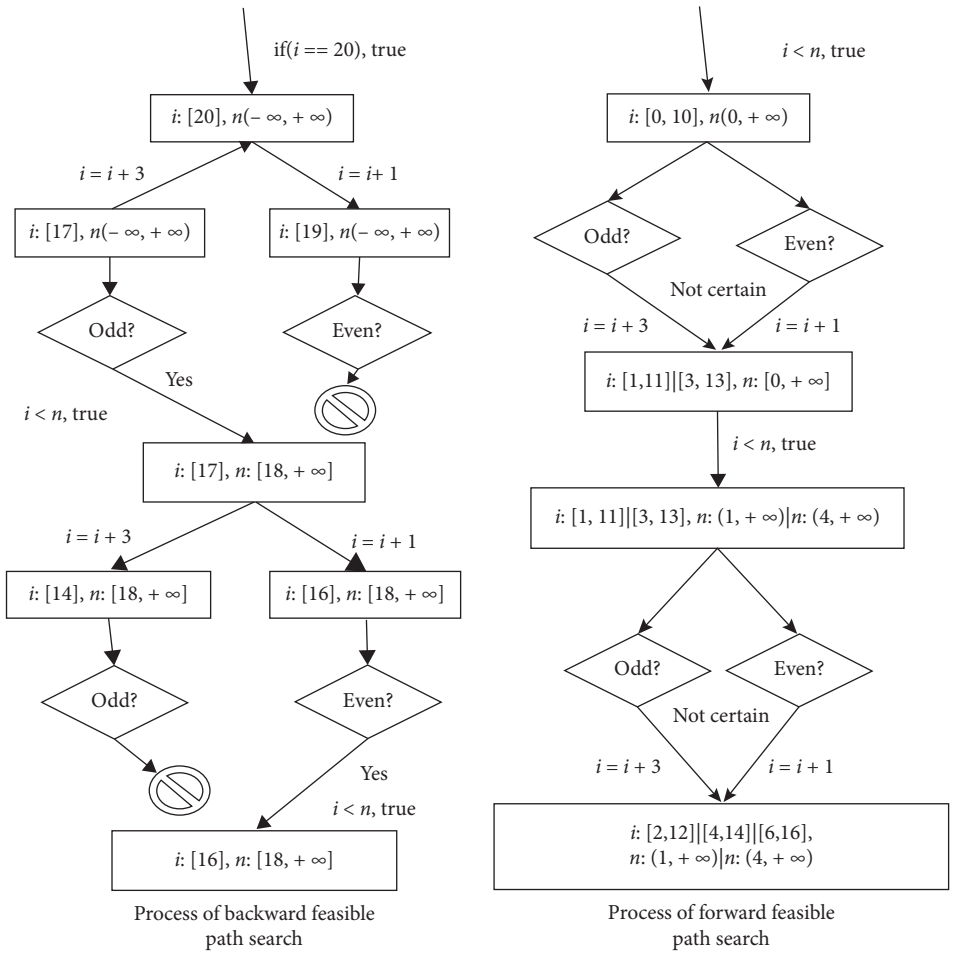

Figure 6: Applying backward analysis and forward analysis on the motivating example.

5.1. Comparison Experiment. The result of statistical validation verifies the property of *LDEP*, which is suitable for reverse analysis. This experiment compares the performance of utilizing backward symbolic execution and general symbolic execution to find feasible path for a specific target of *LDEP*.

The test programs are picked out from papers mentioned in Section 2.3 and Coreutils. The concrete equation conditions *LDEP–e* are selected as the targets. We run backward symbolic execution and general symbolic execution to search feasible paths containing *LDEP–e*. In addition, a popular automated testing tool Pex [35] is also used as a comparison. Pex cannot be appointed to generate paths for a single program element; only whether or not it succeeded is recorded. The results are shown in Table 4.

The data in the table shows that general symbolic execution (forward) fails to find feasible path for *LDEP–e* in *foo* and *fenyu* within time limits; meanwhile backward symbolic execution (backward) finds feasible paths for all *LDEP–e* in tested codes. The time consumption of backward symbolic execution is lower than that of forward technique except for *wanshu*. The reason is that the

LDEP–e in *wanshu* can be covered after executing the loop for 1 iteration, so the general symbolic execution can achieve feasible path faster than backward technique by saving the time of recognizing *LDEP*. Pex could not succeed for *foo* and *fenyu* as well.

In summary, using backward symbolic execution to find feasible path for *LDEP–e* is better than using forward symbolic execution in most cases. There exist exceptions because the correlation of *LDEP* and contra-correlation are not inevitable.

5.2. Performance Improvement Experiment. One of the direct applications of infeasible path detection is automated coverage testing. Most coverage testing tools are based on forward analysis techniques. This section performs experiment to evaluate the improvement of applying backward symbolic execution on automated coverage testing as an auxiliary for specific target to be covered.

We compare the performance of Hybrid-SymExe, Hybrid-SymExe without the backward analysis module (SymExe), and Pex on achieving full coverage of the test codes to show the improvement. The branch coverage and time assumption are shown in Table 5.

The data shows that Hybrid-SymExe receives better performance than SymExe except for *wanshu*. Pex is faster than Hybrid-SymExe and SymExe because Pex uses dynamic execution technique, but Pex fails to achieve full coverage for *foo* and *fenyu*.

In summary, it is hard to cover *LDEP–e* by general symbolic execution. Using backward symbolic execution as an auxiliary for *LDEP–e* can improve the coverage of automated coverage testing tool. The time consumption after integrating backward symbolic execution increases for spending time on recognizing *LDEP*.

6. Related Work

6.1. Backward Analysis. Backward execution is common in data flow analysis. Reps et al. [36] showed the research on improving the efficiency of interprocedure data flow analysis by utilizing the reachability of control flow based on the IFDS data flow framework. Building on this work, ESC/Java [37] and Chandra et al. [38] integrated backward analysis to compute the weakest precondition for a target statement. The key optimization of Snugglebug is pruning search space by constructing the call graph on-demand.

Similarly, Ma et al. [39] combined call graph with symbolic execution in constraint based testing. Besides, heuristic strategy is used to offer guidance to path generation. They concluded that the integration of different strategy is always better than using separate technique. This conclusion gives us inspirations of selecting different analysis techniques according to the different characters of program.

6.2. Infeasible Path Detection. The infeasible path detection has been applied in many software engineering fields [27, 40], in particular structural testing and coverage analysis. Detecting infeasible path can save considerable effort for automated coverage testing. The most two common approaches of infeasible path detection are dynamic analysis and static analysis. The dynamic technique uses the information of dynamic execution to predicate infeasible paths. The static technique analyses the existence of infeasible paths without executing the program.

6.2.1. Dynamic Technique. Dynamic technique can be used to find solution for path-oriented test case generation problem. Bueno and Jino [41] detected infeasible path by monitoring the fitness value of execution trace. The proposed fitness function combined control flow and data flow information to improve analysis precision. The definition of LDEP also involves control and data flow information. Ngo and Tan [42] refined previous work [41] by taking empirical correlation of statement into consideration. The target path is reported as infeasible once an undesired branch is taken.

A recent dynamic technique [11] utilized data dependency information to construct a generalised infeasibility automaton for matching infeasible paths in the future search.

The major problem of dynamic technique is the huge consumption of computation.

6.2.2. Static Technique. The immediate cause of infeasible path is the contradictory path constraints [34, 40, 43]. The constraint based testing uses symbolic execution to generate the constraints representing the path. If the constraints are found to be unsatisfiable by a constraint solver, then the path is pronounced to be infeasible. The performance of symbolic execution is subject to the ability of its underneath constraint solver and the scalability to large real-world projects.

Instead of detecting one infeasible path at a time by examining path constraint, recognizing infeasible code pattern can detect a group of infeasible paths. The infeasible code pattern [4, 44] can be extracted from empirical study on the common properties of infeasible paths. Paths containing infeasible code pattern can be reported as infeasible immediately. The code pattern *LDEP* we proposed is different from the infeasible code pattern as *LDEP* reveals a potential correlation between program character and the distribution of contradict constraints. Therefore, *LDEP* and infeasible code pattern can work together to handle infeasible path.

Another detection approach based on code pattern is branch correlation. Bodiket al. concluded in [27] that 9% to 40% of conditional statements exhibit correlation which can lead to infeasible path. Gong and Yao [5] proposed a new approach to detect branch correlation by dynamic technique, which employed maximum likelihood estimation for the probability of branch correlation based on the output values of branches. However, only part of branch correlation can be detected.

Table 4: Comparison between forward analysis and reverse analysis.

Test code	LOC	Success		Time (ms)		Total paths explored		Pex
		Forward	Backward	Forward	Backward	Forward	Backward	Success
Foo	11	No	Yes	Timeout	729.62	—	273	No
pel	21	Yes	Yes	892.23	96.41	123	29	Yes
do_decode	43	Yes	Yes	263.12	173.07	89	71	Yes
wanshu	22	Yes	Yes	14.5	16.1	21	40	Yes
fenyu	17	No	Yes	Timeout	54.3	—	39	No

Table 5: Improvement of applying reverse analysis to SymExe.

Test code	Branch coverage			Time consumption (ms)		
	SymExe (%)	Hybrid-SymExe (%)	Pex (%)	SymExe	Hybrid-SymExe	Pex
Foo	67.7	100	90.48	2000.0	1321.2	1634
pel	100	100	100	1352.3	877.2	252
do_decode	100	100	100	420.8	460.7	288
wanshu	92	100	100	223.4	289.3	264
fenyu	83.3	100	91.67	2000.0	639.3	287

7. Conclusion

In this paper, we propose a code pattern based on empirical study of infeasible paths and exhibit its correlation with the distribution of contradict path constraints. This achievement can promote the effectiveness of infeasible path detection through utilizing backward symbolic execution for the specific program element. With the current prototype tool, a significant improvement can be seen by applying the proposed approach on coverage testing as an auxiliary.

The novelty of the research results lies in the statistical validation for the empirical property of the proposed code pattern and providing the basis for employing suitable techniques according to the character of program to be analysed.

Some researchers define the infeasible path as any path that is not executable under any test cases [45] and remove infeasible paths from the entire path of the control flow graph [46]. Balakrishnan et al. [47, 48] stated that developers need to distinguish these infeasible paths from the other paths of the entire control flow graph. The majority of the approaches on eliminating infeasible paths focus on C, C++, and Java programming languages. It is noteworthy that all current approaches cannot determine most of the infeasible paths effectively.

In this paper, based on the investigation of infeasible paths, we explore the common properties of infeasible paths in real programs. These properties are presented for the first time. We propose a new code pattern and suppose its properties. The statistical results show the validity of the empirical properties.

However, it is in a first stage of research on the distribution of contradict path constraints. We believe that this is a promising future direction of integrating different techniques to improve the efficiency of software engineering activities from different aspects [20, 49–51].

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] M. Delahaye, B. Botella, and A. Gotlieb, "Explanation-based generalization of infeasible path," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pp. 215–224, Paris, France, April 2010.

[2] A. Goldberg, T. C. Wang, and D. Zimmerman, *Applications of Feasible Path Analysis to Program Testing*, ISSTA, Seattle, WA, USA, 1994.

[3] D. Hedley and M. A. Hennell, "The causes and effects of infeasible paths in computer programs," in *Proceedings of the 8th International Conference on Software Engineering*, pp. 28–30, London, UK, August 1985.

[4] M. N. Ngo and H. B. K. Tan, "Detecting large number of infeasible paths through recognizing their patterns," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 215–224, Dubrovnik, Croatia, September 2007.

[5] D. Gong and X. Yao, "Automatic detection of infeasible paths in software testing," *IET Software*, vol. 4, no. 5, pp. 361–370, 2010.

[6] M. Papadakis and N. Malevris, "A symbolic execution tool based on the elimination of infeasible paths," in *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances*, pp. 435–440, Nice, France, August 2010.

[7] T. Yano, E. Martins, and F. L. de Sousa, "MOST: a multi-objective search-based testing from EFSM," in 2011 IEEE fourth international conference on software testing, verification and validation workshops (ICSTW)," pp. 164–173, 2011.

[8] S. Wong, C. Y. Ooi, Y. W. Hau, M. N. Marsono, and N. Shaikh- Husin, "Feasible transition path generation for EFSM-based system testing," in *Proceedings of the 2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1724–1727, Beijing, China, 2013.

[9] I. Hermadi, C. Lokan, and R. Sarker, "Dynamic stopping criteria for search-based test data generation for path testing," *Information and Software Technology*, vol. 56, no. 4, pp. 395–407, 2014.

[10] J. Ruiz and H. Cassé, "Using smt solving for the lookup of infeasible paths in binary programs," *OASIcs-OpenAccess Series in In- formatics*, vol. 47, 2015.

[11] M. Delahaye, B. Botella, A. Gotlieb et al., "Infeasible path generalization in dynamic symbolic execution," *Information & Software Technology*, vol. 58, pp. 403–418, 2015.

[12] A. W. Marashdih, Z. F. Zaaba, and H. K. Omer, "Web security: detection of cross site scripting in PHP web application using genetic algorithm," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 8, no. 5, 2017.

[13] J. Ruiz, H. Cassé, and M. de Michiel, "Working around loops for infeasible path detection in binary programs," in *Proceedings of the 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 1–10, Shanghai, China, September 2017.

[14] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: past, present and future," *In OASIcs-OpenAccess Series in Informatics*, vol. 15, 2010.

[15] L. Baresi, C. Ghezzi, and L. Mottola, "On accurate automatic verification of publish-subscribe architectures," in *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pp. 199–208, Washington, DC, USA, 2007.

[16] A. Aiken, M. Fähndrich, and Z. Su, "Detecting races in relay ladder logic programs," in *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pp. 184–200, Berlin, Heidelberg, 1998.

[17] J. Nellen, K. Driessen, M. Neuhäußer, E. Ábrahám, and B. Wolters, "Two cegar-based approaches for the safety verification of plc-controlled plants," *Information Systems Frontiers*, vol. 18, no. 5, pp. 927–952, 2016.

[18] S. C. Park, C. M. Park, G.-N. Wang, J. Kwak, and S. Y.. Plcstudio, "Simulation based plc code verification," in *Proceedings of the 40th Conference on Winter Simulation*, pp. 222–228, Miami, Florida, 2008.

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

[20] T. Su, Z. Fu, G. Pu et al., "Combining symbolic execution and model checking for data flow testing," in *Proceedings of the International Conference on Software Engineering*, pp. 654–665, Florence, Italy, May 2015.

[21] D. Kebbal, "Automatic flow analysis using symbolic execution and path enumeration," in *Proceedings of the International Conference on Parallel Processing*, pp. 397–404, Columbus, OH, USA, August 2006.

[22] S. Anand, C. S. Psreanu, and W. Visser, "JPFõSE: a symbolic execution extension to java pathfinder," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 134–138, Warsaw, Poland, 2007.

[23] C. Cadar, D. Dunbar, D. R. Engler et al., "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," *Operating Systems Design and Implementation*, pp. 209–224, Prentice Hall, Upper Saddle River, NJ, USA, 2008.

[24] F. Yu, M. Alkhalaf, and T. Bultan, "Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 605–609, Auckland, New Zealand, November 2009.

[25] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pp. 31–36, Vasteras, Sweden, April 2014.

[26] J. Yan and J. Zhang, "An efficient method to generate feasible paths for basis path testingfficient method to generate feasible paths for basis path testing," *Information Processing Letters*, vol. 107, no. 3-4, pp. 87–92, 2008.

[27] R. Bodik, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 6, pp. 361–377, 1997.

[28] C. Ting, M. Tulika, R. Abhik et al., "Exploiting branch constraints without exhaustive path enumeration," in *Proceedings of the Fifth International Workshop on Worst-Case Execution Time Analysis*, Palma, Spain, July 2005.

[29] B. Blackham, M. H. Liffiton, G. Heiser et al., "Trickle: automated infeasible path detection using all minimal unsatisfiable subsets," in *Proceedings of the 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 169–178, Berlin, Germany, April 2014.

[30] U. Feige and D. Reichman, "On the hardness of approximating Max-Satisfy," *Information Processing Letters*, vol. 97, no. 1, pp. 31–35, 2006.

[31] N. Malevris, "A path generation method for testing LCSAJs that restrains infeasible paths," *Information and Software Technology*, vol. 37, no. 8, pp. 435–441, 1995.

[32] L. De Moura and N. Bjrner, "Z3: an efficient SMT solver," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Budapest, Hungarya, April 2008.

[33] P. Godefroid, N. Klarlund, K. Sen et al., "Dart," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 213–223, 2005.

[34] K. Sen, D. Marinov, and G. Agha, "Cute," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[35] N. Tillmann and J. De Halleux, "PEX: white box test generation for .NET[C]," in *Proceedings of the Tests and Proofs: Second International Conference*, pp. 134–153, Prato, Italy, April 2008.

[36] T. Reps, S. Horwitz, M. Sagiv et al., "Precise interprocedural data flow analysis via graph reachability," in *Proceedings of the Symposium on Principles of Programming Languages*, pp. 49–61, San Francisco, CA, USA, 1995.

[37] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 234–245, 2002.

[38] S. Chandra, S. J. Fink, M. Sridharan et al., "Snugglebug," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 363–374, 2009.

[39] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," *Static Analysis*, Springer, Berlin, Germany, pp. 95–111, 2011.

[40] I. Forgács and A. Bertolino, "Feasible test path selection by principal slicing," *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 378–394, 1997.

[41] P. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pp. 209–218, Grenoble, France, September 2000.

[42] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Information and Software Technology*, vol. 50, no. 7-8, pp. 641–655, 2008.

[43] J. Zhang and X. Wang, "A constraint solver and its application to path feasibility analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 2, pp. 139–156, 2001.

[44] D. Kundu, M. Sarma, D. Samanta et al., "A UML model-based approach to detect infeasible paths," *Journal of Systems and Software*, vol. 107, pp. 71–92, 2015.

[45] A. W. Marashdih and Z. F. Zaaba, "Cross site scripting: detection approaches in web application," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 10, 2016.

[46] B. Barhoush and I. Alsmadi, "Infeasible paths detection using static analysis," *The Research Bulletin of Jordan ACM*, vol. 2, no. 3, pp. 120–126, 2013.

[47] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, O. Wei, and A. Gupta, "SLR: path-sensitive analysis through infeasible-path detection and syntactic language refinement," in *Proceedings of the 2008 International Static Analysis Symposium*, pp. 238–254, Valencia, Spain, July 2008.

[48] T. Ball, "Paths between imperative and functional programming," *ACM SIGPLAN Notices*, vol. 34, no. 2, pp. 21–25, 1999.

[49] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Transactions on Electronic Computers*, vol. EC-15, no. 5, pp. 757–763, 1966.

[50] V. Chipounov, V. Kuznetsov, G. Candea et al., "The S2E platform: design, implementation, and applications[J]," *ACM Transactions on Computer Systems*, vol. 30, no. 1, 2012.

[51] V. Garousi and J. Zhi, "A survey of software testing practices in Canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.