IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received January 7, 2019, accepted January 19, 2019, date of publication February 1, 2019, date of current version February 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2894593

Detecting Interprocedural Infeasible Paths Based on Unsatisfiable Path Constraint Patterns

HONGLEI ZHU[1], DAHAI JIN[1], YUNZHAN GONG[1], YING XING[2], AND MINGNAN ZHOU[1]

[1]State Key Laboratory of Network and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China

[2]Automation School, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Dahai Jin (jindh@bupt.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant U1736110, Grant 61702044, and Grant 61502029, and in part by the Fundamental Research Funds for the Central Universities under Grant 2017RC27.

ABSTRACT The static analysis plays an important role in many software engineering activities. However, the existence of infeasible paths, which causes lower program test coverage and several false positives in the results of the static analysis, has become one of the biggest challenges for the static analysis. In this paper, based on unsatisfiable path constraint patterns, we present a new approach to detect interprocedural infeasible paths. In our approach, we first discover nine unsatisfiable path constraint patterns by mining the common path constraint features of a large number of infeasible paths. Then, we detect the interprocedural infeasible paths; a detected path is deemed to be an interprocedural infeasible path if its simplified constraint conditions match one of the nine unsatisfiable path constraint patterns. To illustrate and verify the approach, an experimental study is performed on five open source C projects. The results show that compared with the existing approach, our approach requires less time on average and detects more interprocedural infeasible paths among the given paths.

INDEX TERMS Static analysis, infeasible path, constraint pattern, interprocedural analysis.

I. INTRODUCTION

Software testing is an expensive, tedious, and labor-intensive task and accounts for up to 50% of the total cost of software development [1]. To improve the efficiency of software testing and find more defects, it is desirable to have matured static analysis tools that can automatically detect the defects in a program [2]–[4]. However, since static analysis is conservative and every path is considered to be executable, a large number of false positives come out as the results of static analysis. Therfore, one of the biggest challenges for static analysis is how to deal with the infeasible paths which are never executable for any inputs. Additionally, infeasible paths also have an effect on test case generation and the high test coverage of a program, i.e., the generation of more test cases to enhance the coverage is hindered, and the resources spent on improving the test coverage may be wasted [5]–[7]. Therefore, the detection of infeasible paths plays an important role in the static analysis of a program. If most of the infeasible paths can be detected and excluded automatically, the accuracy of static analysis and coverage analysis can be greatly improved.

The example in Fig.1 is used to illustrate the effect of infeasible path on the accuracy of static analysis. Fig.1(a) and Fig.1(b) show a C language code segment and the corresponding control flow graphs of functions **foo()** and **f1()** in the code segment, respectively. An invalid arithmetic operation (IAO) fault (at statement 7) is detected by the defect testing system (DTS)[4] which is a code static analysis tool for detecting the defects in program, but it is manually confirmed as a false positive. The path constraints[1] can be extracted after the symbolic execution, the constraint $X > 1 \wedge Y > 0$ is considered as the path constraint of subpath 1-2-3-4-5-6.[2] Moreover, since function **f1()** is called by function **foo()** at the statement 7, and the corresponding actual parameter of formal parameter *a* is *x*, the path constraint of the subpath 12-13-14-15 is $X < 0$. Accordingly, the constraint $X > 1 \wedge Y > 0 \wedge X < 0$ is considered as the path constraint of path 1-2-3-4-5-6-7-12-13-14-15. It is obvious that this path cannot be executed by any input values because of the conflict between constraint conditions $X > 1$ and $X < 0$. Therefore, the path 1-2-3-4-5-6-7-12-13-14-15 is an interprocedural infeasible path. That is,

[1]The constraint referred to in this paper is the explicit constraint.

[2]In this paper, the statement number is used to denote the corresponding node number of a statement in control flow graph.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

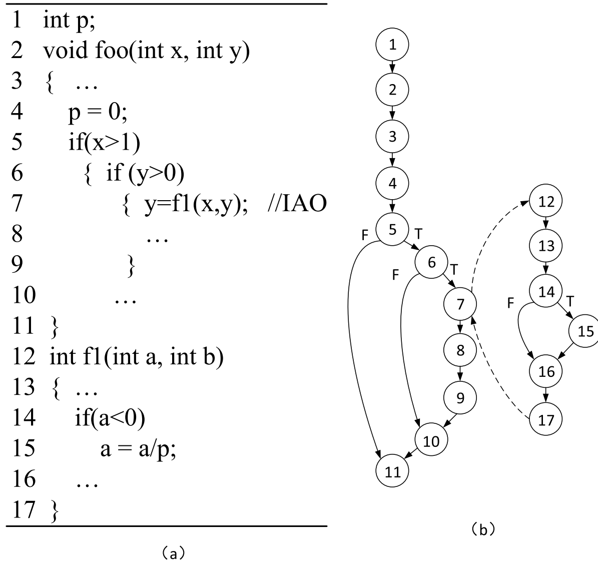

FIGURE 1. An example for illustrating the effect of an infeasible path. (a) A code segment. (b) The control flow graphs of foo() and f1().

the value of variable P is not equal to zero at statement 15 when function **f1()** is called by function **foo()** because of the effect of interprocedural infeasible path, and the IAO fault at statement 7 is a false positive.

Up to now, there exist a variety of approaches for detecting infeasible paths, among which the ones based on the satisfiability of the path constraint are most commonly used [8]–[15]. Although these approaches offer high precision, they depend on the ability of the constraint solver [16] and require expensive computations to determine the infeasibility of a path, especially for the large-scale systems. The approaches that based on branch correlation or code pattern have also proposed for detecting the infeasible paths [17]–[26]. However, these approaches mainly focus on the pairwise correlation analysis, the infeasible paths caused by the conflicts among more than two conditional branches can not be detected. In addition, dynamic test data generation algorithms [26]–[29] are also used to detect infeasible paths by monitoring the execution of a program, but the test data generation requires expensive computations.

In this paper, based on unsatisfiable path constraint patterns, a new approach for detecting interprocedural infeasible paths automatically is proposed. We evaluate the approach using five open source C projects, and the experimental results show that, on average, our approach is able to successfully detect 89.6% of the interprocedural infeasible paths among the given paths.

The contributions of this paper are as follows:

- We present a new approach to detect interprocedural infeasible paths that is based on unsatisfiable path constraint patterns.
- We discover nine unsatisfiable path constraint patterns by mining the common path constraint features of a large number of infeasible paths.
- We evaluate the effectiveness and efficiency of the approach on five open source C programs.
- We note that the interprocedural infeasible paths have an effect on not only the test case generation and high test coverage, but also the accuracy of program static analysis.

The rest of this paper is organized as follows: Section 2 surveys related work. Section 3 introduces some basic terms that will be used in this paper. Section 4 introduces our proposed approach for detecting interprocedural infeasible paths. Section 5 describes our experimental design and presents the experimental results. Section 6 concludes this paper.

II. RELATED WORK

In recent years, a large number of studies have been conducted to detect the infeasible paths in programs, and some techniques have been proposed. Generally, these techniques can be classified into two categories: static and dynamic. Static techniques are mainly based on the satisfiability of the path constraint and the characteristics of program code such as branch correlation, code pattern, etc. On the other hand, dynamic techniques are based on real execution of program and the execution flow is monitored. The existing static and dynamic techniques are briefly reviewed in next subsections, and the comparison between our approach and these approaches is also introduced.

A. STATIC TECHNIQUE

Coward [8] and Goldberg *et al.* [9] use equations to represent a path and determine the feasibility of a path by solving the equations. Similar to their approaches, Ruiz and Cassé [10] propose a method to detect infeasible paths based on static analysis of machine code and the feasibility test of conditions using Satisfiability Modulo Theory (SMT) solvers. Zhang and Wang [11] use the constraint-based tools to determine the feasibility of paths, the tool PAT extracts path conditions while BoNuS is used to solve them. The tools are highly automatic, and they allow the user to specify constraints and paths in a very natural way. Blackham *et al.* [12] apply SMT solvers to find classes of infeasible paths, and integrate the compute all minimal unsatisfiable subsets (CAMUS) algorithm for identifying unsatisfiable subsets within a system of constraints. Ansari [13] uses Unified Modeling Language (UML) for detecting the infeasible paths, the control flow graph is built from sequence diagram and then the independent paths are generated from it. Each path is converted into a set of a linear equation and solved. If there is an inconsistent solution, then the corresponding path is infeasible. Aissat *et al.* [14], [15] propose an approach that uses a transformed CFG to prune the infeasible paths. Furthermore, the symbolic execution technique and constraint solving technique are used to detect the infeasible paths.Although the method can prune lots of infeasible paths to facilitate Path-biased random testing, the construction of the transformed graph takes more time.It is obvious that, these approaches mainly depend on the ability of the constraint

solver and require expensive computations. Compared with these approaches, our approach is based on the unsatisfiable path constraint patterns to detect the infeasible paths and is independent of the constraint solvers. Furthermore, our approach has better scalability than these approaches, which can be applied to the detection of infeasible paths in large-scale programs.

Bodík *et al*. [17] use the infeasible path to improve the precision of traditional def-use pair analysis. Their approach detects the branch correlations by resolving predicate expressions backwards in the CFG during compile time. The infeasible paths are detected with a forward propagation algorithm. However, the branch correlations involving complex predicate expressions may not be detected by their work. Zhang and Chen [18] present an approach to detect infeasible paths by mining association rules of branch predicates. Program paths which break these rules are considered as suspicious infeasible paths. However, this approach heavily relies on execution data, it suffers an unsatisfiable precision when there are not adequate test cases. Barhoush and Alsmadi [19] develop a tool to automatically detect the infeasible paths caused by the logically inconsistent predicates related to dead codes and the correlated conditional statements with respect to a certain variable. However, only four examples are performed to evaluate the tool which can not handle the loop structures. Suhendra *et al*. [20] present an approach to detect infeasible paths by determining whether there exists a conflict between the assignment statement and branch statement, as well as among different branch statements. This approach tries to guarantee avoidance of the enumeration of a large number of execution paths. However, this approach detects only the pairwise conflicts and fails in the case of arbitrary infeasible paths and that it does not account for infeasibility in procedure calls. Compared with these approaches, our approach can convert the complex path constraint conditions into the simple path constraint condition, that is to say, our approach has the capacity of handling some complex predicate expressions. Furthermore, the unsatisfiable path constraint patterns proposed in our approach not only can represent the conflicts between two branches, but also can represent the conflicts among more than two branches, so the infeasible paths caused by the conflicts among more than two branches also can be detected by our approach. In addition, because of our approach adopts the interprocedural analysis, it can detect both intraprocedural and interprocedural infeasible paths without the execution data.

Ding *et al*. [21] present a code pattern based method for detecting infeasible branches and accordingly achieve infeasible paths detection. Although their method has high effectiveness and efficiency, they use a function summary to address the procedure call, which may lead to imprecise interprocedural analysis. Ngo and Tan [22] present an empirical approach to the problem of infeasible path detection. Their approach is based on the fact that many infeasible paths exhibit some common properties, which are caused by four code patterns. This approach can accurately detect 82.3% of all infeasible paths in the set of basis paths of seven systems by realizing these properties from the source code. However, there are also some cases that do not follow any of the four proposed code patterns, and the prototype tool only implements a simple constant substitution for predicates with arithmetic and bitwise operations. Kundu *et al*. [23] construct a graph model (called SIG), from which MM paths, execution sequences of model elements from the start to end of a method scope, are generated. Subsequently, they determine infeasibility of the MM paths by the Mutually Exclusive (MUX) and Null Reference Check (NLC) patterns. Delahaye *et al*. [24] present a method that takes opportunity of the detection of a single infeasible path to generalize to a family of infeasible paths, which will not have to be considered in further path conditions solving. Their method exploits non-intrusive constraint-based explanations to explain unsatisfiability and can save considerable time over an approach that does not make use of the generalization algorithm. However, only eight tiny C programs are performed to evaluate their method. Compared with these approaches, Although our approach is based on only nine unsatisfiable path constraint patterns, it is able to precisely detect 89.6% of the interprocedural infeasible paths among the given paths. Furthermore, we perform the experiments on the five open C programs to evaluate our approach, and the experimental results show that our approach requires less time on average and detects more interprocedural infeasible paths than the approach of Ngo and Tan [22].

B. DYNAMIC TECHNIQUE

Shujuan *et al*. [25] propose an approach based on data flow analysis and association analysis to detect infeasible paths. They built data sets that reflect the static dependencies and the dynamic execution information of conditional statements by combining static analysis and dynamic analysis and then determined the branch correlations based on the two types of branch correlations defined previously. Finally, the infeasible paths were detected. Although this approach can be used to detect infeasible paths effectively and accurately, it is difficult to build a data set that reflects the static dependencies and the dynamic execution information of conditional statements for a large-scale program. Gong and Yao [26] propose an approach that determines branch correlations based on the probabilities of the conditional distribution corresponding to the outcomes of different branches. Maximum likelihood estimation is employed to obtain the values of these probabilities. Then, infeasible paths are detected according to the branch correlations. However, this approach cannot detect the correlations between more than four conditional statements. Bueno and Jino [27] use the genetic algorithm to detect infeasible paths during test case generation; they also propose a fitness function that combines both data flow and control flow information to guide the search. However, this approach requires a higher computational cost to identify infeasible paths, and only a small-scale experiment was conducted

to validate the correctness of the approach. A heuristics approach is proposed by Ngo and Tan [28] to detect infeasible paths. This approach is based on the observation that many infeasible program paths exhibit some common properties. Through realizing these properties in execution traces collected during the test data generation process, the infeasible paths can be detected early with high accuracy. However, the proposed approach occasionally makes the wrong conclusion regarding path infeasibility. Delahaye *et al*. [29] propose a method that takes opportunity of the detection of a single infeasible path to generalize to a possibly infinite family of infeasible paths. The method first extracts an explanation of path condition, that is, the reason of the path infeasibility. Then, it determines conditions, using data dependency information, that paths must respect to exhibit the same infeasibility. Finally, it constructs an automaton matching the generalized infeasible paths. Although infeasible path generalization allows test generation to know of numerous infeasible paths ahead of time, and consequently to save the time needed to show their infeasibility, since most static analysis engines do not work incrementally, it is difficult to use this method in practice.

Compared with these approaches, since our approach belongs to the static technique and detects infeasible paths based on the unsatisfiable path constraint patterns, therefore, it does not need actual execute the program and build the data set. Accordingly, our approach requires a lower computational cost than the approaches mentioned above. In addition, the conflicts among more than two path constraint conditions can be represented by the proposed unsatisfiable path constraint patterns, so the infeasible paths caused by the conflicts among more than two branch statements can also be detected by our approach.

III. PRELIMINARIES

To help the reader to better understand this paper, in this section, we review some basic terms that will be used throughout the paper.

A control flow graph (CFG) of program P can be denoted as a four tuple $< N, E, s, e >$, where N is a set of nodes, E is a set of edges, s is the unique entry node and e is the unique exit node. A node n∈N represents a statement of P, and an edge $(n_i, n_j) \in E$ represents the control flow from statement n_i to statement n_j.

A program is represented by a directed graph $G* = (N*, E*)$ called a supergraph. $G*$ consists of a collection of control flow graphs CFG_1, CFG_2, ... (one for each procedure), one of which, CFG_{main}, represents the main procedure of the program. Each control flow graph CFG_i has a unique start node s_i and a unique exit node e_i. The other nodes of the control graph represent the statements and predicates of the procedure in the usual way, except that a procedure call is represented by two nodes: a call node and a return-site node.

A node in the CFG from which two edges may originate is called a predicate node, and the predicate nodes can be divided into two types: predicate nodes of the iteration construct (while statement) and predicate nodes of the selection construct (if statements). Each out-coming edge of a predicate node is called a branch, and each branch in the CFG is labeled using a predicate, referred to as a branch predicate, that describes the conditions under which the branch will be traversed.

A path constraint refers to the constraint of a path that is expressed as a conjunction of predicates, in which all the derived variables are substituted with their transitive definitions. If two predicates share the common variables, we say that they are mutually dependent, and a set of predicates in a path is called a set of dependent predicates if each predicate is dependent on another predicate in the same set. Let S be a set of dependent predicates of a path; an expression E is a basic expression if all the dependent predicates in S contain E. If there is a conflict among the dependent predicates of a path, then the dependent predicates are known as an unsatisfiable path constraint of the path.

For the nodes n_i and n_j in CFG, if all paths from the entry node s to node n_j go through node n_i, we call node n_i controls node n_j, and denote it as $n_i \xrightarrow{pre} n_j$. Let m is a branch statement, S is a statement set, and $s_i \in S$, $s_i \xrightarrow{pre} m$. If S makes m always take a true branch or a false branch, we call m and S have branch correlation. According to the type of statement in S, the branch correlation can be divided into A-B correlation (the correlation between assignment statement and branch statement) and B-B correlation (the correlation between branch statement and branch statement).

IV. AN APPROACH TO DETECTING INTERPROCEDURAL INFEASIBLE PATHS

A. OVERVIEW

In this section, an approach for detecting interprocedural infeasible paths based on the unsatisfiable path constraint patterns is presented. Fig. 2 shows the basic process of suspected faults confirmation; the dotted rectangle represents the basic process of detecting interprocedural infeasible paths based on unsatisfiable path constraint patterns. First, the constraint extraction for the selected path is executed. Then, the path constraints are clustered and simplified. Finally, a detected path is deemed to be an interprocedural infeasible path if its simplified constraint conditions match one of the nine unsatisfiable path constraint patterns, which are discovered by mining a large number of infeasible paths.

To better understand our basic idea, the code segment in Fig.1 is used to illustrate the basic process of path infeasibility analysis. We assume that the selected path is 2-3-4-5-6-7-12-13-14-15 with a backward path selection strategy and that the path constraint is $X > 1 \wedge Y > 0 \wedge X < 0$. Then, by executing constraint clustering, the constraint conditions $X > 1$ and $X < 0$, which are related to the variable X, are put into a dependent predicates set; the constraint condition $Y > 0$ is considered as another dependent predicates set related to the variable Y. Because each of the constraint conditions is already in its simplest form, the constraint

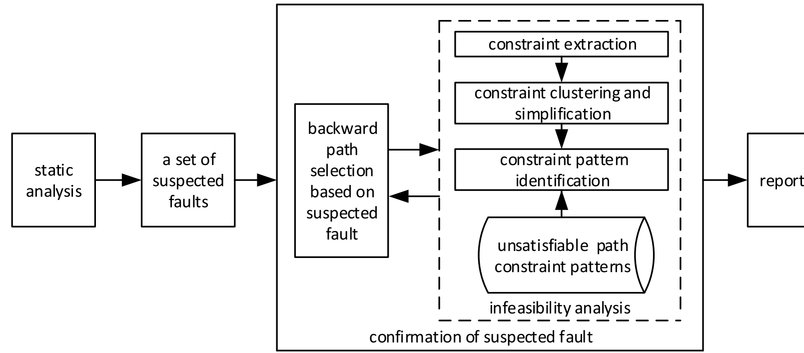

FIGURE 2. Basic process of suspected faults confirmation.

simplification step can be ignored. Finally, we compare each dependent predicates set of this path with the unsatisfiable path constraint patterns and find that the dependent predicates set $\{X > 1, X < 0\}$ is consistent with one of the unsatisfiable path constraint patterns. Therefore, the path 2-3-4-5-6-7-12-13-14-15 is determined as an interprocedural infeasible path without using a constraint solver to solve the path constraint.

B. UNSATISFIABLE PATH CONSTRAINT PATTERNS

In recent years, some researchers have used the pattern checking technology [30], [31] to solve the problems encountered in the study. In this paper, we use the unsatisfied constraint patterns to detect infeasible paths. An unsatisfiable path constraint pattern is the abstract description of the unsatisfiable constraint of a path. For the sake of obtaining the unsatisfiable constraint patterns, we manually confirm 12695 interprocedural infeasible paths distributed in 6 C language projects, which were written by programmers of various backgrounds, from undergraduate students to industrial software engineers, and these interprocedural infeasible paths are considered as the sample paths. Then we extract the corresponding path constraint of each infeasible path, and find out the unsatisfiable constraint (the path constraint conditions among which are conflict) of each path. After that, we mine the unsatisfiable path constraint features of these infeasible path. Finally, 9 unsatisfiable path constraint patterns are discovered, as shown in table 1. The first three columns represent the number of patterns, the abstract descriptions of the unsatisfiable constraint patterns and the constraint conditions of the corresponding pattern, respectively, while the last column contains the conflict types of dependent predicates.

In general, the infeasible paths are caused by the conflicts between assignment statements and branch statements or among two or more branch statements. Accordingly, we divide the unsatisfiable path constraint patterns into two categories, i.e., A-B conflict and B-B conflict, which represent the conflict between assignment statements and branch statements and that among the branch statements, respectively. As shown in table 1, patterns 1-3 are regarded as A-B conflict type because there is a branch predicate in the path constraint that always takes a false value. Moreover, patterns 4-9 are regarded as B-B conflict type because there are conflicts among the dependent predicates, i.e., there is no solution for the path constraint. To better understand these 9 unsatisfiable path constraint patterns described in table 1, in the following two subsections, we will illustrate each of them in detail.

TABLE 1. Unsatisfiable constraint patterns.

ID	Constraint pattern	Condition	Type
1	E	$E \in \{null, 0\}$	A-B
2	$\alpha\Theta\alpha$	$\alpha \in R, \Theta \in \{>, <, != \}$	conflict
3	$\alpha\Theta\beta$	$\alpha, \beta \in R, \alpha < \beta, \Theta \in \{>, >=, =\}$	
4	$E\Theta\alpha \wedge E\Psi\alpha$	$\alpha \in R, \Theta \in \{>, >=, =\}, \Psi \in \{<\}$ or $\Theta \in \{>\}, \Psi \in \{=, <, <=\}$ or $\Theta \in \{=\}, \Psi \in \{!=\}$	
5	$E\Theta\alpha \wedge E\Psi\beta$	$\alpha, \beta \in R, \alpha > \beta, \Theta \in \{>, >=, =\}, \Psi \in \{<, =, <=\}$	B-B
6	$E_1\Theta_1C_1 \wedge E_2\Theta_2C_2 \wedge \ldots \wedge \sum_{i=1}^{n-1} E_n\Psi C_n$	$C_i \in R, 0 < i < n+1, C_n < \sum_{m=1}^{n-1} C_m, \Theta_m \in \{>, >=, =\}, \Psi \in \{<, =, <=\}, 0 < m < n$ or $C_n > \sum_{m=1}^{n-1} C_m, \Theta_m \in \{<, =, <=\}, \Psi \in \{>, >=, =\}, 0 < m < n$	conflict
7	$E_1\Theta\alpha \wedge E_2\Theta\beta \wedge E_1\Omega E_2\Psi\gamma$	$\gamma = \alpha\Omega\beta, \alpha, \beta, \gamma \in R, \Theta \in \{=\}, \Psi \in \{<, !=, >\}, \Omega \in \{+, -, *, /\}$	
8	$E_i\Theta E_j \wedge E_j\Theta E_k \wedge E_k\Theta E_s \wedge \ldots \wedge E_t\Psi E_i$	i, j, k,...,t$\in N+, i \neq j \neq k \neq \ldots t$, $\Theta \in \{=\}, \Psi \in \{!=, >, <\}$	
9	$E\Theta\alpha \wedge E\Psi\alpha \wedge E\Omega\alpha$	$\alpha \in R, \Theta \in \{>=\}, \Psi \in \{<=\}, \Omega \in \{!=\}$	

1) ILLUSTRATIONS OF PATTERNS 1-3

As shown in Fig. 3, the C language code segments (a), (b), and (c), which are taken from the open source program **linux-3.4.113-master**, are given to introduce unsatisfiable constraint patterns 1-3, respectively. We use the symbol '...' to represent statements that are not important for the example. In code segment (a), the return value of function **hib_wait_on_bio_chain** is zero if a path takes the true branch of predicate node 'if (bio_chain == NULL)' at statement 10. Accordingly, the value of variable *ret* is zero at statement 3, and predicate ret of predicate node 'if (ret)' is zero at statement 4. Therefore, the path 1-2-3-8-9-10-11-4-5 is considered as an interprocedural infeasible path, as it cannot take the true branch of predicate node 'if (ret)' at statement 4. That is, the path is determined as an infeasible path while a constraint condition (branch predicate) of the path constraint is unsatisfiable. Therefore, the path constraint conditions

```
1  static int write_page(void *buf, sector_t offset, struct bio **bio_chain)
2  { …
3     ret = hib_wait_on_bio_chain(bio_chain);
4     if (ret)
5        return ret;
6      …
7  }
8  int hib_wait_on_bio_chain(struct bio **bio_chain)
9  { …
10    if (bio_chain == NULL)
11       return 0;
12    …
13 }
```

(a) an example for illustrating the pattern 1

```
1  static int swsusp_swap_check(void)
2  { …
3     res = set_blocksize(hib_resume_bdev, PAGE_SIZE);
4     if (res < 0)
5        blkdev_put(hib_resume_bdev, FMODE_WRITE);
6     return res;
7  }
8  int set_blocksize(struct block_device *bdev, int size)
9  { …
10    if (size < bdev_logical_block_size(bdev))
11       return -EINVAL;
12    …
13    return 0;
14  }
```

(b) an example for illustrating the pattern 2

```
1  static int pm_qos_power_open(struct inode *inode, struct file *filp)
2  { …
3     pm_qos_class = find_pm_qos_object_by_minor(iminor(inode));
4     if (pm_qos_class >= 0) {
5        …
6     }
7     …
8  }
9  static int find_pm_qos_object_by_minor(int minor)
10 { …
11    if (minor == pm_qos_array[pm_qos_class]->pm_qos_power
        _miscdev.minor)
12       return pm_qos_class;
13    return -1;
14  }
```

(c) an example for illustrating the pattern 3

FIGURE 3. Examples illustrating unsatisfiable path constraint patterns 1-3.

(dependent predicates) of path P follow unsatisfiable path constraint pattern 1 (shown in line 2 of table 1) if and only if the following two conditions are satisfied.

- One of the constraint conditions E of the path P contains only one variable V.
- The variable V, which was defined in the previous assignment statement, that can lead to the constraint condition E is unsatisfiable.

The example in Fig. 3(b) is used to illustrate unsatisfiable path constraint pattern 2. In the code segment, since the value of variable *ret* is related to the return value of function **set_blocksize**, the variable *ret* will take a value of zero if predicate node 'if (size < bdev_logical_block_size (bdev))' takes the false branch. Therefore, the predicate of predicate node 'if (res < 0)' is '0 < 0' at statement 4. Obviously, predicate node 'if (res < 0)' cannot take the true branch, as the constraint condition '0 < 0' is unsatisfiable and the path that must pass through sub-path 3-8-9-10-12-13-14-4-5 is an interprocedural infeasible path. Thus, if the path constraint of path P satisfies the following conditions, then we say that it is consistent with unsatisfiable path constraint pattern 2 (shown in line 3 of table 1).

- A constraint condition of path P has only a relational expression.
- One side of the relational expression is the constant α, $\alpha \in$ R (R represents a real number), while the other side is a variable V, which has been defined in the previous statement. Moreover, the value of V is equal to that of the constant α.
- The symbol Θ denotes the relational operator of the relational expression, with $\Theta \in \{>, <, !=\}$.

The code segment in Fig.3(c) is similar to the code segment in Fig.3(b). The value of variable *pm_qos_class* depends on the return value of function **find_pm_qos_object_by_minor**, and it takes a constant '-1' if the predicate node 'if(minor==pm_qos_array[pm_qos_class] →pm_qos_ power_miscdev.minor)' takes the false branch. Moreover, the constraint condition of the true branch of predicate node 'if (pm_qos_class >= 0)' is '−1 >= 0', which is an unsatisfiable constraint condition. As a result, the path that has to pass through the path segment 3-9-10-11-13-14-4-5 is an interprocedural infeasible path. We then say that the path constraint of path P is consistent with unsatisfiable constraint pattern 3 (shown in line 4 of table 1) if it satisfies the following conditions.

- A constraint condition of path P contains only a relational expression.
- The two sides of the relational expression are represented by α and β (α, $\beta \in$ R, R represents a real number, and $\alpha < \beta$); one of them is a variable V, which has been defined in the previous statement, and its value is a real number.
- Θ denotes the relational operator between α and β, with $\Theta \in \{>, >=, =\}$.

From the illustrations above, we can see that an assignment statement may have an effect on a predicate in a program, which can lead to the predicate always taking a false (true) value. Accordingly, this will result in any path that passes through the true (false) branch of this predicate being an infeasible path. Therefore, unsatisfiable path constraint patterns 1-3 can be considered as A-B conflict type.

2) ILLUSTRATIONS OF PATTERNS 4-9

The six code segments shown in Fig. 4 and Fig. 5 are used to illustrate unsatisfiable path constraint patterns 4-9. The code segment in Fig. 4(a) is taken from the open source program **linux-3.4.113-master**. If predicate node 'if (handle→cur &&handle→cur_swap)' takes the true branch, then the constraint condition of this compound predicate is the conjunction of 'handle → cur ! = null' and 'handle → cur_swap ! = null'. Accordingly, the value of formal parameter *offset* is not null in function **write_page**, and there are no redefinition statements of parameter *offset* from the function entry to predicate node 'if (!offset)'.

```
1  static int flush_swap_writer(struct swap_map_handle *handle))
2  { if (handle->cur && handle->cur_swap)
3      return write_page(handle->cur, handle->cur_swap, NULL);
4    else
5      return -EINVAL;
6  }
7  static int write_page(void *buf, sector_t offset, struct bio **bio_chain)
8  { …
9    if (!offset)
10     return -ENOSPC;
11   …
12 }
```

(a) an example for illustrating the pattern 4

```
1  static int get_swap_writer(struct swap_map_handle *handle)
2  { int ret
3    ret = swsusp_swap_check();
4    if (ret){
5      …
6    }
7    …
8  }
9  static int swsusp_swap_check(void)
10 { int res;
11   res = swap_type_of(swsusp_resume_device, swsusp_resume_block,
           &hib_resume_bdev);
12   if (res < 0)
13     return res;
14   …
15 }
```

(b) an example for illustrating the pattern 5

```
1  static int32 buf_copy(cont_ad_t * r, int32 sf, int32 nf, int16 * buf)
2  { …
3    if (sf > CONT_AD_ADFRMSIZE) {
4      …
5    }
6    if (nf > 0) {
7      …
8    }
9    if ((sf + nf) >= CONT_AD_ADFRMSIZE){
10     …
11   }
12   …
13 }
```

(c) an example for illustrating the pattern 6

FIGURE 4. Examples illustrating unsatisfiable path constraint patterns 4-6.

Thus, the predicate node ‘if (!offset)’ will always take the false branch, and the paths traversing sub-path 1-2-3-7-8-9-11 are considered as interprocedural infeasible paths. Since the infeasible paths in Fig. 4(a) are caused by the unsatisfiable branch predicates, we say that the path constraint of path P is consistent with unsatisfiable path constraint pattern 4 (shown in line 5 of table 1) if and only if it satisfies the following conditions.

- The path constraint of path P contains at least two branch predicates p and q.
- p and q are defined as the relational expressions $E\Theta\alpha$ and $E\Psi\alpha$, respectively. The symbol E represents an expression, and α denotes a real number. E in p and E in q are identical, and α in p and α in q are identical.
- The relational operators Θ and Ψ satisfy one of the following three cases: (1) $\Theta \in \{>, >=, =\}$ and $\Psi \in \{<\}$. (2) $\Theta \in \{>\}$ and $\Psi \in \{=, <, <=\}$. (3) $\Theta \in \{=\}$ and $\Psi \in \{!=\}$.

The code segment in Fig. 4(b) is also taken from **linux-3.4.113-master**, from which we can see that if a path takes the true branch of predicate node ‘if (res < 0)’ at statement 12, the return value of function **swsusp_swap_check** is the variable *res*, and if the path also traverses the false branch of predicate node ‘if (ret)’ in function **get_swap_writer**, it is regarded as an interprocedural infeasible path. The constraint conditions $R < 0$ and $R = null$, which are extracted from branch nodes ‘if (res < 0)’ and ‘if (ret)’, respectively, cannot be satisfied simultaneously in one path (R represents the symbolic variable of variable *res*). It is obvious that the infeasible paths in Fig. 4(b) are caused by the conflict among the constraint conditions; thus, if the path

```
1  int op_add = 0;
2  static int f_cpx(int lpx, int lpy, char * choice)
3  { …
4    if(lpx == LTREE_P){
5      op_add+= lpx;
6      …
7      if(lpy == RTREE_P){
8        op_add+= lpy;
9        …
10       opt= f_dpx(lpx, lpy);
11       …
12     }
13   }
14   …
15 }
16 int f_dpx(int dm, int dn)
17 { …
18   dpx_s =dm+dn;
19   if(op_add<dpx_s)
20     …
21   …
22 }
```

(a) an example for illustrating the pattern 7

```
1  static int rout_sub(int major, int sline, int count, struct line_len *msg)
2  { …
3    if(major == sline){
4      …
5    }
6    if(major == count){
7      …
8      mdic=ident_line (sline, count, msg);
9      …
10   }
11   …
12 }
13 int ident_line(int argx, int argy, struct line_len * option)
14 { …
15   if(argx==argy)
16     …
17   …
18 }
```

(b) an example for illustrating the pattern 8

```
1  static int ffcopy (int from, int to)
2  { …
3    if (n < 0) {
4      error ("Cannot read file");
5      return (-1);
6    }
7    if (n == 0)
8      return (0);
9    p = buf;
10   while (n > 0) {
11     …
12   }
13   …
14 }
```

(c) an example for illustrating the pattern 9

FIGURE 5. Examples illustrating unsatisfiable path constraint patterns 7-9.

constraint of path P satisfies the following conditions, we say that it is consistent with unsatisfiable constraint pattern 5 (shown in line 6 of table 1).

- The path constraint of path P contains at least two branch predicates p and q.
- p and q are defined as the relational expressions $E\Theta\alpha$ and $E\Psi\beta$, respectively. The symbol E represents an expression, and the symbols α and β denote real numbers, with $\alpha > \beta$.
- The symbols Θ and Ψ denote the relational operators and satisfy the conditions $\Theta \in \{>, >=, =\}$ and $\Psi \in \{<, =, <=\}$, respectively.

To introduce unsatisfiable path constraint pattern 6, the code segment shown in Fig. 4(c), which is taken from an open source program called **sphinxbase-0.3**, is used. The function **buf_copy** has three predicate nodes which locate at statement 3, statement 6 and statement 9 of the code segment,respectively. If a path traverses the true branch of the first two predicate nodes and the false branch of the last predicate node, then this path is regarded as an infeasible path. The branch predicates S>256 and N>0 conflict with the branch predicate S+N < 256 (S and N denote the symbolic variables of variables *sf* and *nf* during the symbolic execution, respectively, and the value of constant CONT_AD_ADFRMSIZE is 256). Therefore, we say that the path constraint of path P is consistent with unsatisfiable path constraint pattern 6 (shown in line 7 of table 1) if and only if the following conditions are satisfied.

- The path constraint of path P contains branch predicates $p_1, p_2, p_3, \ldots, p_n$, with n > 2.
- The branch predicates $p_1, p_2, p_3, \ldots, p_n$ are defined as the relational expressions $E_1\Theta_1\ C_1$, $E_2\Theta_2\ C_2, \ldots,$ $E_n\Theta_n C_n$, respectively. The symbol E_i $(0 < i < n+1)$ represents an expression, the symbol C_i denotes a real number, and $E_n = \sum_{i=1}^{n-1} E_i$.
- The symbols Θ_m and Ψ denote the relational operator $(0 < m < n)$ and should satisfy one of the following two cases: (1) $\Theta_m \in \{>, >=, =\}$ and $\Psi \in \{<, =, <=\}$ if the constant $C_n < \sum_{m=1}^{n-1} C_m$; (2) $\Theta_m \in \{<, =, <=\}$ and $\Psi \in \{>, >=, =\}$ if the constant $C_n > \sum_{m=1}^{n-1} C_m$.

The code segment in Fig. 5(a), which is taken from a student's project, is used to illustrate unsatisfiable constraint pattern 7 (shown in line 8 of table 1). If a path starts from the entry node of function **f_cpx** and executes the true branches of predicate nodes 'if (lpx == LTREE_P)' and 'if (lpy == RTREE_P)', then the function **f_dpx** will be called in statement 10, and if the true branch of predicate node 'if (op_add<dpx_s)' is executed in function **f_dpx**, then the path is considered as an infeasible path, as there is a conflict among the constraint conditions that are extracted from the three predicate nodes above. Thus, we say that the path constraint is consistent with unsatisfiable path constraint pattern 7 if the following conditions are satisfied.

- The path constraint of path P contains at least three branch predicates p, q, and r.
- The branch predicates p, q, and r are defined as the relational expressions $E_p\Theta\alpha$, $E_q\Theta\beta$, and $E_r\Psi\gamma$, respectively. The symbol E represents an expression, the symbols α, β, and γ denote real numbers, and $E_r = E_p\Omega\, E_q$ and $\gamma = \alpha\Omega\beta$.
- The symbols Θ, Ψ, and Ω denote the relational operators and should satisfy the conditions $\Theta \in \{=\}$, $\Psi \in \{<, !=, >\}$ and $\Omega \in \{+, -, *, /\}$, respectively.

The code segment in Fig. 5(b) is used to illustrate unsatisfiable path constraint pattern 8 (shown in line 9 of table 1), it is also taken from a student's project. If a path starts from the entry of function **rout_sub** and takes the true branches of branch nodes 'if (major == sline)' and 'if (major == count)', then the function **ident_line** will be called at line 8, and if the false branch of predicate node 'if (argx == argy)' is traversed in function **ident_line**, then the path is regarded as an interprocedural infeasible path, as there is a conflict among the constraint conditions that are extracted from the three predicate nodes above. Therefore, if the path constraint of path P satisfies the following conditions, we say that it is consistent with unsatisfiable constraint pattern 8.

- The path constraint of path P contains branch predicates $p_1, p_2, p_3, \ldots, p_n$, with n > 2.
- The branch predicates $p_1, p_2, p_3, \ldots, p_n$ are defined as the relational expressions $E_1\Theta E_2$, $E_2\Theta E_3, \ldots, E_n\Psi E_1$, respectively, where E_i $(0 < i < n+1)$ represents an expression.
- The symbols Θ and Ψ denote the relational operators and should satisfy the conditions $\Theta \in \{=\}$ and $\Psi \in \{!=, >, <\}$, respectively.

To illustrate unsatisfiable path constraint pattern 9 (shown in the last line of table 1), the code segment in Fig. 5(c), which is taken from the open source project **deco,**[3] is used. The function **ffcopy** has two predicate nodes and one loop node which locate at statement 3, statement 7, and statement 10 of the program, respectively. If a path traverses the false branches of the three nodes above, then the path is considered to be an interprocedural infeasible path, as there is a conflict among the constraint conditions that are extracted from the three nodes above. Thus, we say that the path constraint of path P is consistent with unsatisfiable path constraint pattern 9 if and only if the following conditions are satisfied.

- The path constraint of path P contains at least three branch predicates p, q, and r.
- The branch predicates p, q, and r are defined as the relational expressions $E\Theta\alpha$, $E\Psi\alpha$, and $E\Omega\alpha$, respectively. E represents an expression, and α represents a real number.
- The symbols Θ, Ψ, and Ω denote the relational operators and should satisfy the conditions $\Theta \in \{>=\}$, $\Psi \in \{<=\}$, and $\Omega \in \{!=\}$, respectively.

From the illustrations above, we find that a path will be regarded as an infeasible path if there is a conflict among the branch predicates of the path constraint. In general, there are several reasons for the conflicts that between assignment

[3] http://deco.sourcearchive.com/documentation/3.9/main.html

statement and branch statement or among branch predicates, one of which is that an assignment statement affects the value of a predicate node statement, and the assignment statement is associated with the branch statement. As shown by the examples in Fig. 3, a variable is usually referenced in a predicate node statement, and its value is assigned by a return value of a function. If the return value cannot satisfy one of the branch predicates of the predicate node, then the path that passes through these nodes is determined as an infeasible path. Another major reason is that branches are not totally independent of each other and instead may be correlated. Considering the examples in Fig. 4 and Fig. 5, because the interprocedural branches in the program are correlated, a conflict will exist among the interprocedural branch predicates. Accordingly, the path that passes through these correlated branch predicates will be regarded as an interprocedural infeasible path.

C. EXTRACTION, CLUSTERING AND SIMPLIFICATION OF PATH CONSTRAINTS

Although the extraction, clustering and simplification of path constraints are the basic processes of detecting infeasible paths, they are not the focus of this paper; thus, we will simply introduce these techniques. As shown in Fig. 2, to detect whether a path is an interprocedural infeasible path, we first need to extract its path constraint. It should be noted that the path used for extraction is an interprocedural path; therefore, the supergraph of the program under test should be constructed during symbolic execution. Then, we can extract the constraint of the interprocedural path as the constraint of the intraprocedural path. For example, the path 1-2-3-7-9-10-12 in Fig. 4(f) is an intraprocedural path, and the path 1-2-3-9-10-11-12-13-4-6 in Fig. 4(b) is an interprocedural path. Then, when extracting their path constraints, path constraint $N >= 0 \wedge N! = 0 \wedge N <= 0$ is extracted from the former, while path constraint $R < 0 \wedge R = 0$ is extracted from the latter. The symbols N and R represent the corresponding variables *n* and *res*, respectively, used during the symbolic execution.

To better recognize the path constraint using the unsatisfiable path constraint patterns, we need to perform several conversions for the path constraint. Therefore, the operation of constraint clustering is executed after the extraction of the path constraint. Because the path constraint of a path contains a certain number of predicates, and because some of them may have common symbolic variables, a predicate may be dependent on another predicate. Therefore, according to the dependent relationships among the predicates in the path constraint, a path constraint can be divided into several parts in the process of constraint clustering, with every part being considered as a set of dependent predicates. It should be noted that a predicate may be put into different sets at the same time. For example, path constraint $X > 1 \wedge X < 0 \wedge Y > 0 \wedge X + Y > 0 \wedge Z > 0$ can be divided into three sets of dependent predicates: $\{X > 1 \wedge X < 0\}$, $\{X > 1 \wedge X < 0 \wedge Y > 0 \wedge X + Y > 0\}$, and $\{Z > 0\}$. Moreover, each of the sets can be considered as a sub-constraint of the path constraint. If one of the sub-constraints is unsatisfiable, then the path will be considered as an infeasible path. Considering the example in Fig. 1, the path constraint of path 2-3-4-5-6-7-14-15-16-17 is $X > 1 \wedge Y > 0 \wedge X < 0$, while $\{X > 1, X < 0\}$ and $\{Y > 0\}$, which can be considered as the sub-constraints of this path, are the sets of dependent predicates. Finally, the path is determined to be an infeasible path because the sub-constraint in the former is unsatisfiable.

The simplification of the path constraints is a step following the clustering of the path constraints. In this step, the dependent predicates in the set of dependent predicates are simplified so that we can improve the accuracy and the speed of the matching between the set of dependent predicates and the unsatisfiable path constraint patterns. First, the basic expression is obtained by analyzing the expression of each dependent predicate in the set. Then, a symbolic variable is used to replace the basic expression so that the complicated predicates in the set can be simplified to produce simple predicates. Finally, the redundant predicates in the set are removed so that the set of dependent predicates can be recognized based on the unsatisfiable path constraint patterns. For example, the set $\{M + N > 1, M + N > 0, M + N = 0\}$ is a set of dependent predicates, and the expression $M + N$ is a basic expression of the set. When executing the operation of constraint simplification, the basic expression is replaced by a symbolic variable MN, and the redundant dependent predicate $MN > 1$ is removed because the dependent predicate $MN > 0$ contains it. After that, the set of dependent predicates is updated to the set $\{MN > 0, MN = 0\}$, which can easily be recognized based on the unsatisfiable path constraint pattern.

D. ALGORITHMS

Based on the unsatisfiable path constraint patterns that have been introduced above, in this subsection, we present a set of algorithms for detecting whether a path is an interprocedural infeasible path. Algorithm 1 is the main algorithm, which is used to detect whether a path p is an interprocedural infeasible path in program M. Each function **PatternMatch_i(s)** **(PM_i(s))** that is called in the algorithm represents one of the matching algorithms between the sub-constraint (dependent predicates in set s) and the unsatisfiable path constraint patterns. Additionally, the functions **SG()**, **ConstraintExtract()**, and **ConstraintCluster()** are used to construct the supergraph of the program, extract the path constraints and cluster the path constraints, respectively. For simplicity, we describe only two representative matching algorithms: **PatternMatch_5(s)** for pattern 5 and **PatternMatch_8(s)** for pattern 8.

In algorithm 1, the supergraph G* of program M should be constructed in the initialization step, which is executed only once to detect the feasibility of any path of the program M (line 1). By extracting and clustering the path constraints of path P, accordingly, the set λ, which consists of several sets of dependent predicates, can be obtained (lines 2-3). Each set of dependent predicates in the set λ represents a

Algorithm 1 An Algorithm for Detecting Infeasible Paths

```
   Input: a path P of program M, program M
   Output: true or false
1  G* ← SG(M);
2  θ ← ConstraintExtract(p, G*);
3  λ ← ConstraintCluster(θ);
4  for each s in λ; do
5      if (s.length == 1) then
6          if (PM_1(s)||PM_2(s)||PM_3(s)) then
7              return true;
8          end
9          else if (s.length == 2) then
10             if (PM_4(s)||PM_5(s)) then
11                 return true;
12             end
13         end
14         else if (s.length == 3) then
15             if ((PM_9(s)||PM_6(s)||PM_7(s)||PM_8(s))
               then
16                 return true;
17             end
18         end
19         else if (s.length >= 4) then
20             if ((PM_6(s)||PM_7(s)||PM_8(s)) then
21                 return true;
22             end
23         end
24     end
25 end
26 return false;
```

sub-constraint of path P, and the appropriate matching function, i.e., the algorithm used to match an unsatisfiable path constraint pattern with the sub-constraint, is chosen according to the number of dependent predicates that the set contains. If one of the sub-constraints of path P can be successfully matched with one of the nine patterns, the path p can be regarded as an interprocedural infeasible path; otherwise, the path P may be a feasible path (lines 4-25). Because the number of dependent predicates in a sub-constraint may be greater than the number of elements that the unsatisfiable path constraint pattern contains after path constraint clustering is performed, at some time, the sub-constraint can be matched with one of the patterns, such as pattern 6, pattern 7, and pattern 8, that can be matched with the sub-constraint, which contains more than four dependent predicates based on an analysis of the results of path constraint clustering. Therefore, the functions **PM_6(s)**, **PM_7(s)**, and **PM_8(s)** are called twice in algorithm 1. The overall complexity of algorithm 1 is O(M*P), where M denotes the number of dependent predicate sets after simplification of the path constraint has been carried out, P denotes the maximum complexity of the algorithms corresponding to nine patterns.

Algorithm 2 PatternMatch_5(s)

```
   Input: a set s of dependent predicates
   Output: true or false
1  s1 = {>, >=, =}, s2 = {<, =<, =};
2  for each predicate pair (p,q), p,q∈ s do
3      if (p_le = q_le) then
4          if (p_re > q_re) then
5              if (the operator of p ∈ s1) then
6                  if (the operator of q ∈ s2) then
7                      return true;
8                  end
9              end
10         end
11         else
12             if (the operator of p ∈ s2) then
13                 if (the operator of q ∈ s1) then
14                     return true;
15                 end
16             end
17         end
18     end
19 end
20 return false;
```

Algorithm 2 is an algorithm for determining whether the sub-constraint can be matched with pattern 5. The symbols *p_le* and *p_re* represent the left expression and the right expression of predicate *p*, respectively. For each dependent predicate pairs in the set *s*, the algorithm identifies whether the left expressions of the two dependent predicates are equal firstly. If they are not equal, then it will choose the next predicate pair to determine whether the dependent predicates are consistent with pattern 5. Otherwise, it will identify whether the right expressions of the two predicates are equal (lines 1-3). If the right expressions are not equal, then the matching operation of the operators will continue, and if the operators of the dependent predicates belong to the sets of operators that have been defined previously, then the sub-constraint can match with pattern 5. Otherwise, it will choose the next predicates pair to determine whether the dependent predicates are consistent with pattern 5. If none of the predicate pairs in set *s* are consistent with pattern 5, it means that the sub-constraint cannot match with pattern 5 and should match with the other patterns to determine the feasibility of the path (lines 4-20). In addition, the complexity of algorithm 2 is O(Q), where Q represents the number of predicate pairs in set *s*. The number of predicate pairs can be calculated by counting the number of dependent predicates in set *s*, and Q is equal to N*(N-1)/2 where N denotes the number of dependent predicates in set *s*, therefore, the complexity of algorithm 2 is O(N^2).

Algorithm 3 is an algorithm for determining whether the sub-constraint can be matched with pattern 8. First, according to the operators of the dependent predicates, the dependent

Algorithm 3 PatternMatch_8(s)

```
   Input: a set s of dependent predicates
   Output: true or false
1  s' = φ, s'' = φ, TS1 = φ, TS2 = φ;
2  for (each predicate p in s) do
3      if (the operator of p∈ {! =, >, <}) then
4          add p to s' and continue;
5      end
6      TS1 = φ, TS2 = φ;
7      if (the operator of p equals '=') then
8          for (each cs in s'') do
9              if (p_le∈cs) then
10                 TS1 = cs;
11             end
12             if (p_re∈cs) then
13                 TS2 = cs;
14             end
15         end
16         if (TS1 == φ&&TS2 == φ) then
17             create a new set cs_new;
18             add p_le and p_re to cs_new;
19             add cs_new to s'';
20         end
21         else if (TS1! = φ&&TS2 == φ) then
22             add p_re to TS1, and update the set s'';
23         end
24         else if (TS1 == φ&&TS2! = φ) then
25             add p_le to TS2, and update the set s'';
26         end
27         else if (TS1! = φ&&TS2! = φ) then
28             merge the sets TS1, TS2;
29             update the set s'';
30         end
31     end
32 end
33 for (each p in s') do
34     for (each cs in the s'') do
35         if (p_le and p_re both in set cs) then
36             return true;
37         end
38     end
39 end
40 return false;
```

predicates in the set s are divided into two types and stored in sets s' and s''. The former set stores the dependent predicates whose operators belong to set $\{>, <, !=\}$, while the latter one stores the dependent predicates whose operators are '=' (lines 1-7). Furthermore, when adding a dependent predicate to set s'', only the left and right expressions of the predicate are added, and they are considered as a set. At the same time, if a dependent predicate is added to set s'', it needs to be determined whether a set in set s'' contains the left and right expressions of the dependent predicate. If the set in set s'' does not contain any expression of the dependent predicate, then a new set must be created and should be added to set s''. If the set in set s'' contains only one side of the expression of the dependent predicate, then the other side of the expression of the dependent predicate should be added to set s''. If one set in set s'' contains one side of the expression of the dependent predicate and another set in set s'' contains the other side of the expression of the dependent predicate, then it must merge the two sets of set s'' and update the set s'' (lines 8-32). Finally, if a set in set s'' contains both the left expression and right expression of a dependent predicate in set s', then the output of the algorithm is 'true', which denotes that the sub-constraint can be matched with pattern 8; otherwise, it needs to be determined whether the sub-constraint can be matched with one of the other patterns (lines 33-40). The complexity of algorithm 3 is O(N*W), where N denotes the number of dependent predicates in set s and W denotes the number of set cs. Since the set cs is composed of left or right expressions of dependent predicates with the same value, that is, in the worst case, W is equal to N/2 where N denotes the number of dependent predicates in set s, the complexity of algorithm 3 is $O(N^2)$.

V. EVALUATION AND DISCUSSION

In this section, we first introduce the experimental design and evaluation metrics. Then, we introduce the experimental results in detail to verify the accuracy and efficiency of our approach. After that, we discuss the experimental results. Finally, the threats to validity are introduced.

A. EXPERIMENTAL DESIGN

We implement our approach on DTS (defect testing system) which is a code static analysis tool for detecting the defects in program. To evaluate our approach, we conduct experiments on five open source C language programs. The information regarding these programs is given in table 2; the columns File, LOC, and Function present the number of files, lines of code and number of functions of the program, respectively. As it is difficult to determine the actual number of interprocedural infeasible paths of a complex program, to ensure the effectiveness of the experiments, we select 10% of the total number of interprocedural paths which are generated in the previous experiments by a random program for detection and manually confirm the interprocedural infeasible paths, treating these infeasible paths as seeds. Furthermore, although our approach can detect both the intraprocedural and interprocedural infeasible paths, we mainly illustrate the detection of

TABLE 2. Basic information regarding the benchmark.

Benchmark	File	LOC	Function
Spell-1.0	4	1991	26
Barcode-0.98	15	4858	56
deco	22	9566	319
a200c	37	6584	85
qlib	105	24494	367

TABLE 3. Experimental results of interprocedural infeasible paths detection.

Benchmark	Path	Seed	C_{max}	C_{min}	Approach I								Approach II							
					TP	FP	FN	Time(s)	AT(s)	Prec	Rec	F	TP	FP	FN	Time(s)	AT(s)	Prec	Rec	F
Spell-1.0	2136	469	9	3	451	0	18	2007	0.94	100%	96.2%	98.0%	406	0	63	1892	0.89	100%	86.6%	92.8%
Barcode-0.98	2981	413	7	4	387	0	26	4163	1.40	100%	93.7%	96.8%	331	0	82	4481	1.50	100%	80.1%	89.0%
deco	4392	672	7	3	630	0	42	7165	1.63	100%	93.8%	96.8%	519	0	153	8709	1.98	100%	77.2%	87.2%
a200c	3960	506	8	3	417	0	89	6302	1.59	100%	82.4%	90.4%	396	0	110	6025	1.52	100%	78.3%	87.8%
qlib	4157	728	6	4	613	0	115	8396	2.02	100%	84.2%	91.4%	440	0	288	9712	2.34	100%	60.4%	75.3%
Total/Avg	17626	2788	7.4	3.4	2498	0	290	28033	1.59	100%	90.6%	94.7%	2092	0	692	30819	1.75	100%	76.5%	86.4%

interprocedural infeasible paths in this paper; thus, we only conduct experiments on the interprocedural paths.

To better evaluate the effectiveness and efficiency of our approach, we compare the approach proposed in this paper with one that detecting infeasible paths by recognizing the identical/complement-decision, mutually-exclusive-decision, check-then-do, and looping-by-flag patterns (the overall complexity of algorithm is O(N^2E), where N denotes the number of predicate nodes of selection construct in the CFG and E denotes the number of edges of the CFG) [22]. We then confirm and count the number of false positives and false negatives of each approach. If an interprocedural non-infeasible path is detected as an interprocedural infeasible path by an approach, a false positive is counted for that approach. In contrast, if an infeasible path is wrongly detected as a feasible path by an approach, a false negative is counted for that approach. Moreover, we also count the number of infeasible paths detected based on each pattern presented in this paper. To evaluate the efficiency, the time costs of these two approaches are compared.

B. EVALUATION METRICS

We use the standard Precision, Recall, and F-Score metrics to evaluate the effectiveness of an approach. Precision measures the actual interprocedural infeasible paths that are correctly detected in terms of a percentage of the total number of interprocedural infeasible paths, while Recall measures the ability of an approach to find actual infeasible paths. By using TP, FP, and FN to denote true positive, false positive and false negative detection results, respectively, the Precision, Recall, and F-Score can be computed using Equations (1), (2), and (3), respectively.

$$Precision = TP/(TP + FP). \quad (1)$$

$$Recall = TP/(TP + FN). \quad (2)$$

$$F = 2 * Precision * Recall/(Precision+Recall) \quad (3)$$

Additionally, we evaluate the efficiency of an approach by computing its time cost, the total time required for each benchmark and the average processing time for each path.

C. EXPERIMENTAL RESULTS

Table 3 shows the experimental results of the two approaches mentioned above. Columns Path, Seed, C_{max} and C_{min} present the number of interprocedural paths taken into consideration, the number of truly interprocedural infeasible paths, and the maximum and minimum call depth of the functions traversed by the interprocedural infeasible paths, respectively. Columns Time, AT, Pre, Rec, and F present the total time cost, the average time cost for processing one path, the Precision value, the Recall value, and the F-Score value, respectively. Furthermore, columns Approach I and Approach II represent the approach proposed in this paper and the approach proposed by Ngo and Tan [22], respectively.

According to table 3, we randomly select 17626 interprocedural paths from the five open source programs, 2788 paths of which are manually confirmed to be infeasible paths. It is important to note that the paths selected for detection are different from the paths selected for mining the unsatisfiable path constraint patterns. The value of C_{max} ranges from 6 to 9, and the value of C_{min} ranges from 3 to 4. Approach I and Approach II are able to detect 2498 interprocedural infeasible paths and 2092 interprocedural infeasible paths, respectively, with each of these paths being a true positive. Moreover, the former approach causes 290 false negatives, while the latter causes 692 false negatives. Fortunately, because neither of these two approaches causes any false positives for the seeds, all the detected interprocedural infeasible paths are the actual infeasible paths. Thus, the Precision of each approach is 100%. Accordingly, the Recall values and F-Score values also can be calculated. The former approach achieves higher Recall values and F-Score values than those of the latter one, the Recall improvement of the former over the latter ranges from 4.1% to 23.8% for all the subject programs,and the F-Sore improvement of the former over the latter ranges form 2.6% to 16.1% for all the subject programs. On average, the former achieves an average Recall of 90.6% and an average F-Score of 94.7%, while the latter achieves an average Recall of 76.5% and an average F-Score of 86.4%.

With respect to the efficiency of infeasible paths detection, Approach I requires 1.59 seconds to process a path, while Approach II requires 1.75 seconds on average. Furthermore, comparing the total time costs of Approach I and Approach II, Approach I requires approximately 46 minutes less than Approach II to detect the infeasible paths among the given paths. That is, the average time cost of Approach I decreases by 9.1% compared to that of Approach II.

Table 4 shows the results detected based on each pattern proposed in this paper. Columns P1, P2, P3, P4, P5, P6, P7, P8, and P9 give the number of interprocedural infeasible paths detected based on pattern 1, pattern 2, pattern 3, pattern 4, pattern 5, pattern 6, pattern 7, pattern 8, and pattern 9, respectively.

From table 4, the number of interprocedural infeasible paths detected based on each pattern can be determined. Pattern 4 is able to detect 853 infeasible paths in total, while pattern 7 detects only 33 infeasible paths. Moreover, according to table 2, the percentage of infeasible paths detected based on each pattern over all the infeasible paths detected is as follows: 24.3% are detected based on pattern 1, 7.6% are detected based on pattern 2, 5% are detected based on pattern 3, 34.1% are detected based on pattern 4, 17.7% are detected based on pattern 5, 3.4% are detected based on pattern 6, only 1.3% are detected based on pattern 7, 1.4% are detected based on pattern 8 and 5.2% are detected based on pattern 9.

TABLE 4. Results detected based on each pattern.

Benchmark	P1	P2	P3	P4	P5	P6	P7	P8	P9
Spell-1.0	183	80	21	136	15	0	0	9	7
Barcode-0.98	135	14	30	122	58	6	14	0	8
deco	151	18	43	233	90	9	4	16	66
a200c	78	31	14	198	52	27	0	5	12
qlib	59	48	17	164	226	42	15	6	36
Total	606	191	125	853	441	84	33	36	129

D. DISCUSSION

The above experimental results show that more infeasible paths can be detected by our approach than by approach II, because approach II detects the infeasible paths by recognizing four patterns which are just the common properties of some infeasible paths in the program, compared with the nine unsatisfiable path constraint patterns proposed in this paper, the types of infeasible paths detected by our patterns include not only the types of infeasible paths identified by the four patterns, but also include some types of infeasible paths that cannot be detected by approach II. Such as the pattern 4 of our approach, it not only can be used to detect the infeasible paths caused by the completely different path constraint conditions (the identical/complement-decision pattern of approach II), but also can be used to detect the infeasible paths caused by the other type of conflicts among the constraint conditions. That is, no matter how many experiments are performed and the sample paths are selected in an experiment, the number of infeasible paths detected by our approach would not less than the number of infeasible paths detected by approach II.

Furthermore, in terms of the overall complexity of the algorithm, algorithm 1 is the main algorithm of our approach, and its overall complexity is O(M*P), where M denotes the number of dependent predicate sets, P denotes the maximum complexity of the our matching algorithms. P is equal to O(N^2), where N denotes the number of predicates in the dependent predicate set, i.e., the overall complexity of our main algorithm is O(N^2*M). Compare with the overall algorithm complexity O(N^2*E) of approach II, where N denotes the number of predicate nodes of selection construct in the CFG (is equal to N*M in our algorithm complexity) and E denotes the number of edges of the CFG, it obviously that the algorithm complexity of our approach is less than approach II. In other words, our approach would take less time than approach II to detect the infeasible paths, which is consistent with the experimental results. Therefore, although we have not perform more experiments on more paths, it has only a little effect on the reliability of experimental results.

Fig.6 shows a code segment which is taken from project **a200c**. In the code segment, the subpath 3-4-5-6-7-10-11-12-13-14-15 is considered as an infeasible path because of the conflict between the constraint conditions 'Y>TABEND' and 'Y<TABSTART'. However, the infeasible path not caused by one of the four patterns proposed in approach II, as a result, it cannot be detected by approach II. Fortunately, the path constraint of this infeasible path can match with the pattern 5 proposed in this paper, so it can be detected by our approach.

```
1   #define TABSTART 1620
2   #define TABEND 2017
3   int update()
4    { double T;
5      T = 2000.0 + (JD - J2000)/365.25;
6       …
7       UT = TDT - deltat(T)/86400.0;
8       …
9     }
10  double deltat(double Y)
11   { …
12      if( Y > TABEND )
13      …
14      if( Y < TABSTART ){
15          if( Y >= 948.0 ){
16              B = 0.01*(Y - 2000.0);
17              ans = (23.58 * B + 100.3)*B + 101.6;
18             }
19          else
20              …
21            return(ans);
22          }
23   }
```

FIGURE 6. A case not detected by approach II.

Although the experimental results show that 89.6% of the interprocedural infeasible paths can be detected and an average Recall of 90.6% is achieved using our approach, there are 290 false negative cases, which account for 10.4% of all truly infeasible paths. We manually analyze all the false negative cases and find that 12 cases do not follow any of the patterns proposed in this paper. For example, Fig.7 shows a false negative case which is taken from the project **qlib**. Because the path constraint of path 3-4-5-6-7-8-11-12-13-15-17-19 is $aa <= 1.0 \wedge bb <= 1.0 \wedge aa > 0.0 \wedge bb > 0.0 \wedge (bb * (aa/(aa+bb))) > 1.0$, which has no solution, the path is considered as an interprocedural infeasible path. Unfortunately, the path constraint of this path cannot be matched with the unsatisfiable path constraint patterns proposed in this paper; therefore, this path cannot be regarded as an infeasible path. Accordingly, the path that contains this path also cannot be regarded as an interprocedural infeasible path, thus causing a true negative result.

The remaining 278 false negative cases are not detected because of the limitations of our approach. According to the basic process of our approach, to detect the feasibility of a path, the path constraint must be extracted first. Then, the path constraint can be analyzed. However, since the extraction of a path constraint depends on the symbolic execution in our

```
1   double incbi(double aa, double bb, double yy0)
2   { …
3     if(aa <= 1.0 && bb <= 1.0)
4     { …
5        a = aa;
6        b = bb;
7        x = a/(a+b);
8        y = incbet( a, b, x);
9        …
10   }
11  double incbet(double aa, double bb, double xx)
12  { …
13    if (aa <= 0.0)
14       …
15    if (bb <= 0.0)
16       …
17    if ((bb * xx) <= 1.0)
18       …
19    …
20  }
```

FIGURE 7. A case not detected by the proposed approach.

approach, and because the predicates in programs may have a number of operations of the complicated data type, such as the operation of struct, the symbolic execution technique is unable to better address them. Therefore, although the path constraint is processed according to our approach after performing symbolic execution, it cannot be matched with the unsatisfiable path constraint patterns. As such, the infeasible paths in code segments of Fig. 8 and Fig. 9 cannot be detected.

```
1   void read_file(pipe_t * the pipe, FILE * stream, char * file)
2   { str_t *str = str_make(0);
3     …
4     if (str->str[str->len – 1]!= '\n')
5         str_add_char(str, '\n');
6     …
7   }
8   int str_add_char(str_t * str, char c)
9   { if (!str)
10       str =str_make(str);
11    …
12  }
```

FIGURE 8. A case not detected by the proposed approach.

```
1   void reread ( int k, int sk)
2   { …
3     if (! (p = getwstring (50, d->cwd, " Change directory ",
         "Change directory to")))
4     …
5   }
6   char *getwstring (int w, char *str, char *head, char *mesg)
7   { int len;
8     …
9     len = strlen (mesg);
10    if (len > w)
11       w = len;
12    …
13  }
```

FIGURE 9. A case not detected by the proposed approach.

In the code segment of Fig.8, variable *str* is a pointer with respect to a *str_t* struct. 'if (str → str[str → len − 1] ! = '\n')' and 'if (!str)' are predicate nodes located in function **read_file** and function **str_add_char**, respectively. If a path chooses the true branch of the former and the true branch of the latter, then the branch predicates '(str → str[str → len − 1] ! = '\n')' and '!str' are the constraint conditions of path 1-2-3-4-5-8-9-10. Moreover, these two constraint conditions actually follow pattern 4, and the path should be considered as an interprocedural infeasible path. However, because the symbolic execution cannot address the predicate '(str→ str[str→len - 1] ! = '\n')', which has a complicated struct, the path constraint cannot be matched with pattern 4, and our approach fails to detect the path.

In the code segment of Fig.9, a path which passes the subpath 3-6-7-8-9-10-11 should be considered as an infeasible path, because the predicate node 'if (len > w)' in function **getwstring()** will choose the false branch to execute when the function **getwstring()** is called by function **reread()**, and this situation actually follows pattern 3. However, since the library function **strlen()** cannot be correctly addressed and the value of variable *len* cannot be determined during the symbolic execution, the constraint condition 'len > w' also cannot be matched with pattern 3, and our approach fails to detect the paths which pass subpath 3-6-7-8-9-10-11.

Additionally, as shown in table 5, we also summarized the causes for these false negative cases and divided them into three types. Columns ID, Number, Situation and Example present the type of false negative cases, the number of false negative cases, the cause for generation of false negative cases and an example to illustrate a false negative case with specified type, respectively.

TABLE 5. A summary for the false negative cases.

ID	Number	Situation	Example
1	12	outside constraint patterns 1-9	example in Fig.6
2	247	the effects of complex structs and arrays	example in Fig.7
3	31	the effects of library functions	example in Fig.8

E. THREATS TO VALIDITY

In this subsection, we will discuss the validity threats applicable to our approach.

1) CONSTRUCT VALIDITY

Although we perform an experimental study on five open source C projects and the experimental results show that 89.6% of the interprocedural infeasible paths can be accurately detected using our approach, there are still some false negative cases. One cause of false positives is that, the path constraint of the detected paths should be obtained firstly during the detection of interprocedural infeasible paths. Unfortunately, the extraction of the path constraint depends on the symbolic execution which can not handle some complex data types. As a result, some infeasible paths contained complex data types can not match with the nine unsatisfiable path constraint patterns we found, and these paths can not be determined as interprocedural infeasible paths. In the future work, we will pay more attention to study how to

tackle complex type data during symbol execution, so as the interprocedural infeasible path can match with one of the unsatisfiable path constraint patterns proposed in this paper, and more interprocedural infeasible paths can be detected. In addition,the DTS uses the function summary to handle recursion functions in C program during the code static analysis, which may lead to imprecise interprocedural data flow analysis, but it has no impact on the effectiveness of our approach, because our approach is based on the unsatisfiable path constraint patterns to detect the infeasible paths.

2) INTERNAL VALIDITY

The nine unsatisfiable path constraint patterns are discovered by mining the unsatisfiable path constraint features of 12695 infeasible paths in six C projects, which may be incomplete because of the limited sample paths. Fortunately, the experimental results show that the approach proposed in this paper can detect the interprocedural infeasible paths accurately, and only a small part of the interprocedural infeasible paths can not be detected due to the fact that it can not match with one of the nine unsatisfiable path constraint patterns. Furthermore, we can extract the constraint features of the infeasible paths which can not detected by our approach, and establish one or more new unsatisfiable path constraint patterns for detecting the interprocedural infeasible paths in future.

3) EXTERNAL VALIDITY

Although we only perform an experimental study on five open source C projects to evaluate our approach, the interprocedural infeasible paths in the large C projects also can be precisely detected by our approach. Compared with middle-scale or small-scale C projects, the large-scale C projects may have more functions and the length of the path may be longer. Accordingly, more path constraints are included in one path, and the constraint solving time of this path will be increased in large-scale C projects. However, our approach does not adopt the constraint solving technique, but uses the path constraint of the detected path match with one of the unsatisfiable path constraint patterns for detecting the infeasibility of the path. Furthermore, the constraint clustering and simplified technology are used to improve the detection efficiency. Therefore, the increase of path length has little impact on our approach, and our approach can be applied to the large-scale C projects to detect the interprocedural infeasible paths. Furthermore, our approach is used for detecting the infeasible paths in C language programs at present, and cannot be used for detecting the infeasible paths in object-oriented languages programs, such as Java and python, since there are many differences between C language and object-oriented programming languages.

VI. CONCLUSIONS

An infeasible path, which is a path in a control flow graph, cannot be executed for any input values. A large number of infeasible paths make static analysis overly conservative in software testing. In this paper, we put forward a new approach based on unsatisfiable constraint patterns to detect interprocedural infeasible paths. First, we discover nine unsatisfiable path constraint patterns by mining the common path constraint features of a large number of infeasible paths. Then, by comparing the simplified constraint conditions of detected paths with the nine unsatisfiable constraint patterns, we can unambiguously detect interprocedural infeasible paths. To evaluate the effectiveness and efficiency of our approach, we also conduct experiments on five open source C projects. The results show that, on average, our approach is able to successfully detect the interprocedural infeasible paths with an average Recall of 90.6%. Moreover, it is also shown that our approach outperforms the code pattern approach in the sense that our approach requires less time and detects more interprocedural infeasible paths.

However, our approach is unable to adequately address more complex data types. In the future, we will focus on improving our approach to overcome this limitation,such as modeling complex data types during symbolic execution and improving the constraint patterns, so that more infeasible paths in complex programs can be precisely detected.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, Aug. 1990.

[2] P. Cousot *et al.*, "The ASTREÉ analyzer," *Programming Languages and Systems*. Berlin, Germany: Springer, 2005, pp. 21–30.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Softw., Pract. Exper.*, vol. 30, no. 7, pp. 775–802, Jun. 2000.

[4] Q. Xiao *et al.*, "Path sensitive static defect detecting method," (in Chinese), *J. Softw.*, vol. 21, no. 2, pp. 209–217, Feb. 2010.

[5] Y. Zhang, S. Jiang, and H. Han, "Research progress on infeasible path detecting problem," *J. Comput. Theor. Nanosci.*, vol. 12, no. 8, pp. 1931–1935, Aug. 2015.

[6] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

[7] S. Ding and H. B. K. Tan, "Detection of infeasible paths: Approaches and challenges," *Eval. Novel Approaches Softw. Eng.*, vol. 410, pp. 64–78, 2012.

[8] P. D. Coward, "Symbolic execution and testing," *Inf. Softw. Technol.*, vol. 33, no. 1, pp. 53–64, 1991.

[9] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Seattle, WA, USA, 1994, pp. 80–94.

[10] J. Ruiz and H. Cassé, "Using SMT solving for the lookup of infeasible paths in binary programs," in *Proc. WCET*, Lund, Sweden, 2015, pp. 95–104.

[11] J. Zhang and X. Wang, "A constraint solver and its application to path feasibility analysis," *Int. J. Softw. Eng. Knowl. Eng.* , vol. 11, no. 2, pp. 139–156, 2001.

[12] B. Blackham, M. Liffiton, and G. Heiser, "Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets," in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Berlin, Germany, Apr. 2014, pp. 169–178.

[13] G. A. Ansari, "Detection of infeasible paths in software testing using UML application to gold vending machine," *Int. J. Edu. Manage. Eng.*, vol. 7, no. 4, pp. 21–28, 2017.

[14] R. Aissat *et al.*, "A method for pruning infeasible paths via graph transformations and symbolic execution," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Vienna, Austria, Aug. 2016, pp. 144–151.
[15] R. Aissat, F. Voisin, and B. Wolff, "Infeasible paths elimination by symbolic execution techniques: Proof of correctness and preservation of paths," in *Proc. 7th Int. Conf. Interact. Theorem Proving (ITP)*, Nancy, France, 2016, pp. 36–51.
[16] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. TACAS*, Budapest, Hungary, 2008, pp. 337–340.
[17] R. Bodík, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," in *Proc. 6th Eur. Softw. Eng. Conf., 5th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Zurich, Switzerland, 1997, pp. 361–377.
[18] C. Zhang and Y. Chen, "Detecting infeasible paths via mining branch correlations," *J. Softw. Eng.*, vol. 6, no. 4, pp. 65–78, Apr. 2012.
[19] B. Barhoush and I. Alsmadi, "Infeasible paths detection using static analysis," *Res. Bull. Jordan ACM*, vol. 2, no. 3, pp. 1120–1126, 2013.
[20] T. V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Efficient detection and exploitation of infeasible paths for software timing analysis," in *Proc. 43rd Annu. Design Automat. Conf.*, San Francisco, CA, USA, 2006, pp. 358–363.
[21] S. Ding, H. Zhang, and H. B. K. Tan, "Detecting infeasible branches based on code patterns," in *Proc. CSMR-WCRE*, Antwerp, Belgium, Feb. 2014, pp. 74–83.
[22] M. N. Ngo and H. B. K. Tan, "Detecting large number of infeasible paths through recognizing their patterns," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Dubrovnik, Croatia, Sep. 2007, pp. 215–224.
[23] D. Kundu, M. Sarma, and D. Samanta, "A UML model-based approach to detect infeasible paths," *J. Syst. Softw.*, vol. 107, pp. 71–92, Sep. 2015.
[24] M. Delahaye, B. Botella, and A. Gotlieb, "Explanation-based generalization of infeasible path," in *Proc. ICST*, Paris, France, Apr. 2010, pp. 215–224.
[25] J. Shujuan, H. Han, S. Jiaojiao, Z. Yanmei, J. Xiaolin, and Q. Junyan, "Detecting infeasible paths based on branch correlations analysis," (in Chinese), *J. Comput. Res. Develop.*, vol. 53, no. 5, pp. 1072–1085, May 2016.
[26] D. Gong and X. Yao, "Automatic detection of infeasible paths in software testing," *IET Softw.*, vol. 4, no. 5, pp. 361–370, Oct. 2010.
[27] P. M. S. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data," in *Proc. 15th IEEE Int. Conf. Automated Softw. Eng.*, Grenoble, France, Sep. 2000, pp. 209–218.
[28] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Inf. Softw. Technol.*, vol. 50, nos. 7–8, pp. 641–655, 2008.
[29] M. Delahaye, B. Botella, and A. Gotlieb, "Infeasible path generalization in dynamic symbolic execution," *Inf. Softw. Technol.*, vol. 58, pp. 403–418, Feb. 2015.
[30] H. Gao, S. Mao, W. Huang, and X. Yang, "Applying probabilistic model checking to financial production risk evaluation and control: A case study of Alibaba's Yu'e Bao," *IEEE Trans. Computat. Social Syst.*, vol. 5, no. 3, pp. 785–795, Sep. 2018.
[31] H. Gao, D. Chu, Y. Duan, and Y. Yin, "Probabilistic model checking-based service selection method for business process modeling," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 27, no. 6, pp. 897–923, 2017.

HONGLEI ZHU was born in Henan, China, in 1983. He received the master's degree in computer science and technology from Henan Polytechnic University, Henan, China, in 2010. He is currently pursuing the Ph.D. degree in computer science and technology with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications.

From 2011 to 2014, he was an Assistant with the Henan University of Chinese Medicine. His research interests include software testing and software engineering.

DAHAI JIN was born in Liaoning, China, in 1974. He received the master's and Ph.D. degrees in information security from the Armored Engineering Institute of the PLA, in 2002 and 2006, respectively. He was a Postdoctoral Fellow with the Beijing University of Posts and Telecommunications, in 2008.

He has been an Assistant Professor with the Beijing University of Posts and Telecommunications, since 2010. His current research interests include software testing and static analysis.

YUNZHAN GONG was born in Shandong, China, in 1962. He received the master's degree in computer from the National University of Defense Technology, in 1986, and the Ph.D. degree in computer from the Institute of Computing Technology, Chinese Academy of Sciences, in 1991.

From 1995 to 2006, he was a Professor with the Armored Engineering Institute of the PLA. He has been a Professor with the Research Institute of Networking Technology, Beijing University of Posts and Telecommunications, since 2006. His current research interests include software testing and software reliability.

He received China Engineering Science Realistic Award, the Chinese Academy of Natural Sciences Award, and the Military Progress Prize in Science and Technology.

YING XING received the Ph.D. degree in computer science from the Beijing University of Posts and Telecommunications, in 2014, where she was a Postdoctoral Fellow, in 2014, and is currently an Assistant Professor with the Automation School. Her research interests include artificial intelligence and automatic test data generation. She is a member of CCF.

MINGNAN ZHOU was born in Jilin, China, in 1993. He received the B.Admin. degree in E-commerce with law from the Beijing University of Posts and Telecommunications and the B.E. degree from the Queen Mary University of London, in 2016. He is currently pursuing the Ph.D. degree in computer science and technology with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications.

He has been an R & D Personnel with the State Key Laboratory of Network and Switch Technology, since 2015. His research interests include software testing and distributed systems.

He was a member of the Institution of Engineering and Technology, from 2012 to 2016.