ISSN 1330-3651 (Print), ISSN 1848-6339 (Online) https://doi.org/10.17559/TV-20190417093740
Original scientific paper

Inter-Procedural Diagnosis Path Generation for Automatic Confirmation of Program Suspected Faults

Honglei ZHU, Dahai JIN, Yunzhan GONG

Abstract: Static analysis plays an important role in the software testing field. However, the initial results of static analysis always have a large number of false positives, which need to be confirmed by manual or automatic tools. In this paper, a novel approach is proposed, which combines the demand-driven analysis and the inter-procedural dataflow analysis, and generates the inter-procedural diagnosis paths to help the testers confirm the suspected faults automatically. In our approach, first, the influencing nodes of suspected fault are calculated. Then, the CFG of each associated procedure is simplified according to the influencing nodes. Finally, the "section-whole" strategy is employed to generate the inter-procedural diagnosis path. In order to illustrate and verify our approach, an experimental study is performed on the five open source C language projects. The results show that compared with the traditional approach, our approach requires less time and can generate more inter-procedural diagnosis paths in the given suspected faults.

Keywords: automatic confirmation; path generation; static analysis; suspected fault

1 INTRODUCTION

Software testing is an inevitable step in software development, and it accounts for more than 50% of the cost of software development [1, 2]. In order to detect and repair the faults existing in the software as soon as possible, testers often analyze the program with the aid of defect detection tools [3,4], such as Astrée [5], DTS [6], Klocwork, etc. These tools employ the static analysis techniques and have some features such as the high defect detection rate, accurate fault location and high degree of automation. Due to the conservatism of static analysis, the initial results of static analysis tend to have a large number of false positives, testers need to confirm the initial results of static analysis by manual or automated confirmation tools [7, 8]. Therefore, to determine whether a suspected fault point is a real fault, all the execution status of suspected fault point or all the paths that go through the suspected fault point should be calculated [9]. However, in general, the actual application programs may have a lot of function calls, and the simple intra-procedural analysis cannot accurately determine whether the fault is a false positive [10]. To improve the accuracy of the suspected fault confirmation, it needs to analyze the whole paths from the entry point of the program or the external input points of the program to the suspected fault point [11].

Fig. 1 shows a C language code segment with an invalid arithmetic operation (IAO) suspected fault which was detected by DTS (defect testing system) at line 14. If the suspected fault was confirmed only by analyzing the procedure *func*2, it would be considered as a real fault. Because the parameter *c* can take the value from infinity to minus infinity during the intra-procedural data flow analysis, accordingly, the variable *c* at line 14 may take the value zero which gives rise to an IAO fault. However, the variable *c* cannot take the value zero by the inter-procedural analysis, and the inter-procedural diagnosis path 3-4-5-17-18-19-21-9-10-11-12-13-14 can be used for determining the IAO fault is a false positive. Therefore, in this paper, we target at the problem of inter-procedural diagnosis path generation for automatic confirmation of program suspected fault, which not only can make the confirmation of suspected fault accurately but also can help the developers to repair the fault.

```
1  void foo()                          12    b=func1(a)
2  { …                                 13  if(a>0)
3    scanf("%d%d", &x, &y);            14    t=1/c;          //IAO
4    if (y>0)                          15  ...
5      { z = func1(y);                 16 }
6        x = func2(x, y, z);           17  int func1(int m)
7    …                                 18 {if (m>0)
8  }                                   19      return 1;
9  int func2(int a, int b, int c)      20   return 0;
10 { float t;                          21 }
11   if(c<10)
```

Figure 1 An example of illustrating the importance of inter-procedural diagnosis path

The previous studies on suspected fault confirmation are mainly dependent on the execution traces or execution states which are generated by backward symbolic execution or backward inference [12-17]. Some researchers proposed the method called postcondition symbolic execution that eliminates redundant paths without reducing the search space during symbolic execution [18], while others use the fault correlation to identify suspected faults [19, 20]. Unfortunately, although these prior techniques can mitigate the risk of path explosion posed by forwarding symbolic execution [21], they also have some limitations, such as only analyzing the local program, using the function summary represents the concrete execution of callee during the inter-procedural analysis, which fail to guarantee complete accuracy of suspected fault confirmation and obtain the inter-procedural diagnosis paths to fix the fault [22-24]. Furthermore, due to only 43% of the nodes, and 52% of the function being useful during fault detection [25], it is important to prune the irrelevant nodes and functions during the generation of inter-procedural diagnosis paths, because it not only can accelerate the determination of diagnosis path feasibility but also can mitigate the risk of path explosion and indirectly improve the efficiency of suspect fault confirmation.

In this paper, we present a novel approach that combines the demand-driven analysis and the inter-

procedural data flow analysis to generate the inter-procedural diagnosis paths. Compared with the existing approaches, it has the following characteristics: Firstly, the backward analysis based on the demand-driven can omit the irrelevant predicates and procedures during the generation of inter-procedural diagnosis path. Secondly, unlike the existing approaches that use the intraprocedural analysis or simplified inter-procedural analysis, our approach employs the accurately inter-procedural analysis, which can derive benefit from improving the accuracy of suspected fault confirmation and can aid the developers to fix the faults. Thirdly, the "section-whole" strategy is adapted to generate the inter-procedural diagnosis paths, which can improve the scalability of the approach. We have evaluated the approach on five open source C language projects, and the experimental results show that our approach requires less time and can generate more inter-procedural diagnosis paths in the given suspected faults.

The contribution of this paper is as follows:

- We present a novel approach that combines the demand-driven analysis and the inter-procedural data flow analysis to generate the inter-procedural diagnosis paths related to the confirmation of suspected faults.
- Our approach employs the "section-whole" strategy to construct the inter-procedural diagnosis path.
- Experimental studies, using five open source C language programs to illustrate the effectiveness and accuracy of our approach.

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 presents the approach and describes it in detail. Section 5 describes the experiment and evaluation. Section 6 concludes this paper.

2 RELATED WORK

Since a forward symbolic execution is non-demand-driven, it has to explore many paths that are not relevant to the suspected fault [19]. Recently, many post-failure-process approaches have been proposed for confirming the suspected faults, which use the backward symbolic execution or backward inference to generate the execution traces or execution states [12-17]. Manevich et al. [12] proposed a typical post-failure-process approach PSE which performs postmortem data-flow analysis to explain program failures with minimal information. However, PSE does not take into account the influencing nodes related to program failures, which may increase the time of computation. Cheng et al. [13] proposed an automatic verification method for suspected faults based on finite backtracking symbolic execution. Dillig et al. [14] proposed a new technique for assisting users in classifying error reports when automated static analyses fail to verify a program, and their technique allows verification tools to interact with users by computing small, relevant queries that capture exactly the facts that the analysis is missing to either verify the program or prove the existence of a real error. It is different from our approach which can generate an inter-procedural diagnosis path to help the developers confirm and fix the real fault. Chen et al. [15] proposed STAR, a novel approach which first computes the crash triggering precondition using a backward symbolic execution, and then identifies the complete crash path and constructs real test cases that can actually reproduce the crash. Although our approach is similar to this approach, our approach only takes into account the influencing nodes that are relevant to the suspected fault during the backward traversal CFG, which avoid generating a large number of redundant paths. In addition, the "section-whole" strategy is employed to alleviate the risk of path explosion. Yao et al. [16] proposed StatSym, a novel, automated Statistics-Guided Symbolic Execution framework that integrates the swiftness of statistical inference and the rigorousness of symbolic execution techniques to achieve precision, agility, and scalability in vulnerable program path discovery. Kasikci et al. [17] proposed failure sketching, an automated debugging technique that provides developers with an explanation of the root cause of a failure that occurred in production. Their approach combines static program analysis with a cooperative and adaptive form of dynamic program analysis, while our approach is a purely static approach.

3 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 $n \in N$ represents a statement of P, an edge $(n_i, n_j) \in E$ represents the control flow from statement n_i to statement n_j. A program can be represented as an inter-procedural control flow graph (ICFG), which intuitively is the union of control flow graphs for individual procedures comprising the program [26].

A Use-Definition Chain (UD Chain) is a data structure that consists of a use U of a variable, and all the definitions D of that variable that can reach that use without any other intervening definitions. A definition can have many forms but is generally taken to mean the assignment of some value to a variable. A counterpart of a UD Chain is a Definition-Use Chain (DU Chain), which consists of a definition D of a variable and all the uses U reachable from that definition without any other intervening definitions. A definition of variable n can be denoted as a four-tuple $<S, C, V, P>$, where S represents the definition expression of variable n, C represents a list of constants located in the definition expression, V represents a list of variables located in the definition expression, and P represents the location of variable n.

In a CFG, a node u dominates a node n if and only if every path from the entry node to n contains u. A node n post-dominates a node u if and only if every path from u to the exit node contains n. A node y is control dependent on a node x if and only if x has successors x' and x'' such that y post-dominates x' but y does not post-dominate x''. Furthermore, we say that node y is transitively controlled dependent on node x if there is a sequence of nodes, $x = x_0$, $x_1 \ldots x_n = y$, such that x_j is control dependent on x_{j-1}, $1 \leq j \leq n$. Let a node n (transitively) control be dependent on a

predicate p, then the predicate p is referred to as direct influencing predicate with respect to n. In addition, the predicate of which at least one of its branches contains the statement that the node n or its direct influencing predicate data is dependent on is referred to as indirect influencing predicate with respect to node n.

4 AN APPROACH FOR GENERATING THE INTER-PROCEDURAL DIAGNOSIS PATHS

In this section, first, we introduce the basic process of suspected faults confirmation. Then, we describe the basic idea of our approach by illustrating an example. Finally, we illustrate the approach in detail.

4.1 Approach Overview

Fig. 2 shows the basic process of suspected faults confirmation. First, the trigger condition of suspected fault is obtained by recognizing the type of suspected fault. Then, according to the associated variables and the trigger condition of a suspected fault, we backward analyze the inter-procedural control flow and inter-procedural data flow. After that, with the feasibility analysis of path, the inter-procedural paths are generated by employing the "section-whole" strategy. Finally, if there is a feasible path in the generated paths, along which the suspected fault can be triggered, then the suspected fault is confirmed as a real fault and the path is called the diagnosis path. Otherwise, if all the generated paths are either infeasible or fail to trigger the suspected fault, then the suspected fault is identified as a false positive case.

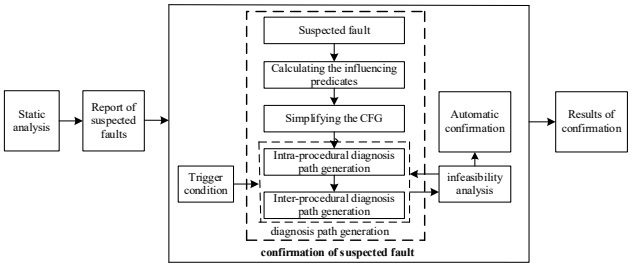

Figure 2 The basic process of suspected faults confirmation

The highlighted dotted rectangle in Fig. 2 represents the basic process of inter-procedural diagnosis paths generation. First, according to the associated variables of a suspected fault, we calculate the influencing predicates related to the suspected fault. Then, the CFG of associated procedures is simplified by pruning the non-influencing predicates nodes and the irrelevant statement nodes. Finally, with the "section-whole" strategy, the inter-procedural diagnosis path is generated, which depends on the generation of intra-procedural diagnosis path.

To better understand the basic idea of our approach, the code segment in Fig. 1 is used to illustrate the basic process of inter-procedural diagnosis of path generation. The associated variable and trigger condition of IAO suspected fault located at line 14 is variable c and constraint condition $c \neq 0$, respectively. The influencing predicates of this suspected fault are $a > 0$, $y > 0$, and $m > 0$ located at line 13, line 4, and line 18, respectively, which can be calculated by the inter-procedural data flow analysis. Accordingly, the associated procedures of this suspected fault are *func*2(), *foo*(), and *func*1(), the non-influencing predicate is $c< 10$ located at line 11. The statements located at line 11 and line 12 are considered as the irrelevant statements of the suspected fault, which should be pruned in the CFG of *func*2(). Therefore, the intra-procedural diagnosis path 9-10-13-14 can be generated by traversing the simplified CFG of *func*2(), which is a feasible path. After that, the inter-procedural paths 3-4-5-17-18-19-21-9-10-13-14 and 3-4-5-17-18-20-21-9-10-13-14 can be obtained by employing the "section-whole" path generation strategy, however, the path 3-4-5-17-18-20-21-9-10-13-14 is an infeasible path because of the existence of conflicting path constraint conditions $Y > 0$ and $Y < 0$. Therefore, the inter-procedural path 3-4-5-17-18-19-21-9-10-13-14, which can trigger the IAO suspected fault, is considered as the inter-procedural diagnosis path of this IAO fault. Obviously, through inter-procedural data flow analysis and the CFG simplification, one non-influencing predicate and one statement contained a callee are pruned, four inter-procedural paths are reduced during the backward diagnosis path generation, and the path constraint of diagnosis path has fewer constraint conditions.

4.2 Influencing Predicates Calculation

The influencing predicates calculation can be considered as the first phase of our approach. A report of suspected faults is obtained after the code static analysis, from which we can extract the essential information of suspected fault. And the essential information of a suspected fault can be denoted as a four-tuple <N, T, L, V>, where N, T, L, and V represent the fault ID, fault type, fault location, and the associated variables of fault, respectively. The fault location also can be denoted as a three-tuple <F, P, S>, where F, P, and S represent the file name, procedure name, and the line number, respectively.

To calculate the influencing predicates of the suspected fault, the essential information of the suspected fault is extracted firstly. Then, according to the essential information, we can obtain the direct influencing predicates of the suspected fault by the (transitively) control dependent analysis. After that, using the Use-Definition chain [27], we compute the definition statements of the associated variables of the suspected fault and the variables located in the direct influencing predicates. Through these definition statements, similar to the calculation of direct influencing predicates, the indirect influencing predicates of the suspected fault can be obtained by the (transitively) control dependent analysis. Since there are a large number of pointer aliases in the C program, and the backward analysis based on the demand-driven is adopted during the generation of inter-procedural diagnosis path. The method proposed by Zheng et al. is used to deal with pointer aliases in this paper [28]. Finally, we iteratively compute the definition statements of the variables located in the definition statements and the indirect influencing predicates, and the iterative operation will not stop until the value of each relevant variable depends on either the value of an external input source or a constant. The external input source contains the standard input stream functions such as *scanf*() and *getchar*(),

memory allocation functions such as *malloc*(), *calloc*() and *realloc*(), and the parameters of main procedure, etc. Accordingly, the rest of the indirect influencing predicates also can be obtained by the iterative calculation. Finally, the influencing predicates of a suspected fault can be obtained by combining its direct influencing predicates and indirect influencing predicates.

Algorithm 1: Influencing predicates calculation
Input: a suspected fault SF
Output: a set PS of influencing predicates
Begin
1 PS = Φ, VS = Φ;
2 PS←CD(SF);
3 VS←SF.V;
4 **if** (PS ≠ Φ) **then**
5 **for** (each p in PS) **do**
6 TVS= Φ;
7 TVS←ExtraVar(p);
8 VS←TVS ∪ VS;
9 **endfor**
10 **endif**
11 **for (**each v in VS**) do**
12 PreCalc(v);
13 **endfor**
14 return PS;
15 PreCalc(v){
16 VS'= Φ;
17 **for**(each ds in UD(v)) **do**
18 VS1'=Φ, PS'= Φ;
19 **if** (ds ∉ ES) **then**
20 PS'←CD(ds);
21 PS = PS ∪ PS';
22 VS'←ExtraVar(ds));
23 VS1'←ExtraVar(CD(ds));
24 VS' = VS' ∪ VS1';
25 **for**(each v' in VS') **do**
26 PreCalc(v');
27 **endfor**
28 **endif**
29 **endfor**
30 }
End

Algorithm 1 is an algorithm for calculating the influencing predicates of a suspected fault. The symbols *SF* and *SF.V* in algorithm 1 represent a suspected fault and the associated variables of a suspected fault, respectively. The functions *CD*(*s*), *UD*(*v*), and *ExtraVar*(*s*) are the functions that used to compute the direct influencing predicates of statement *s* with the relationship of control dependent, the definition statements of variable *v* with the Use-Definition chain, and the variables or parameters of called function in the statement *s* or predicate *s*, respectively. First, the algorithm calculates the associated variables of suspected fault which are considered as the initial value of the set VS and the direct influencing predicates of suspected fault which are considered as the initial value of the set PS (line1-3). If the suspected fault has the direct influencing predicates, then extracting the variables or parameters that are located in these influencing predicates and adding them into the set VS (line4-10). After that, the algorithm iteratively calculates the indirect influencing predicates of each variable in the set VS (line11-13). Finally, the influencing predicates of a suspected fault can be obtained (line 14). Obviously, the function *PreCalc*(*v*) is an iterative function that is used to calculate the indirect influencing predicates of variable *v*. It calculates the definition statements of variable *v* first. If a definition statement locates at an external input source or depends on the constant, then ends the analysis related to this definition statement. Otherwise, the direct influencing predicates related to this definition statement, and the variables or parameters that located in these influencing predicates and this definition statement are added into a set of the variable for iteratively calculating the influencing predicates (line15-30). It should be noted that if the value of a variable is dependent on the formal parameter of a function which is not the main function of a program, then the definition statements of the corresponding actual parameter related to the formal parameter of this function are considered as the definition statements of this variable.

To better understand algorithm 1, an example with IAO suspected fault in Fig. 1 is used to illustrate the algorithm. First, through identifying the type of suspected fault and analyzing the control dependent relationship of a suspected fault, the variable *c* and predicate $a > 0$ are considered as the associated variable and direct influencing predicate of a suspected fault, respectively. Then, the variable *c* and the variable *a*n extracted from the direct influencing predicate $a > 0$ are added into the set VS for iteratively calculating the indirect influencing predicates of the suspected fault. After that, we can obtain the definition statement of variable *c*(*z*) at line 5 (the variable *z* is the corresponding actual parameter of the formal parameter *c* in procedure *func*2()). With the further calculation, the direct influencing predicate and relevant variable of this definition statement also can be obtained, which are the predicate $y > 0$ and $m > 0$, and variable *y*, respectively. The parameter *y* of procedure *func*1() is regarded as the relevant variable of definition statement at line 5, and the procedure *func*1() should be analyzed in detail because of the value of variable *z* depends on the return value of procedure *func*1(), and the predicate $m > 0$ is regarded as the influencing predicate accordingly. Due to the value of the variable *y* depending on the external input, so the predicates $a > 0$ and $y > 0$ are regarded as the influencing predicates related to variable *c*. Then, similar to the calculation of influencing predicates related to variable *c*, the algorithm continues to calculate the influencing predicates related to variable *a*. Finally, the predicates $a > 0$ and $y > 0$ are regarded as the influencing predicates of the IAO suspected fault. It should be noted that because the procedure *func*1() at line 12 has no side effects on the variables *a* and *c* in set VS, it does not need to be analysed further.

4.3 CFG Simplification

The CFG simplification is the step followed by the influencing predicates calculation. As we can see from sub-section 3.2, not all of the statements or the called functions in a procedure have an effect on a statement or a variable in the procedure. Similarly, not all of the statements and the called functions in the fault procedure or other procedures have an effect on the suspected fault. If we do not consider these cases during the backward path generation, then not only the number of the generated path will increase sharply, but also the path constraint of each

generated path may contain some irrelevant constraint conditions for the suspected fault.

To alleviate the risk of path explosion during the backward path generation and eliminate the irrelevant constraint conditions to easily determine the path feasibility, we should simplify the CFG of procedures that are associated with the suspected fault before the backward path generation. First, we calculate the relevant procedures related to the suspected fault by the call graph and the inter-procedural data flow analysis. Then, we calculate the irrelevant statements and the called functions by the calculation of influencing predicates and definition statements in algorithm 1. Finally, we prune the corresponding nodes of these irrelevant statements and called functions in the CFG of the procedure.

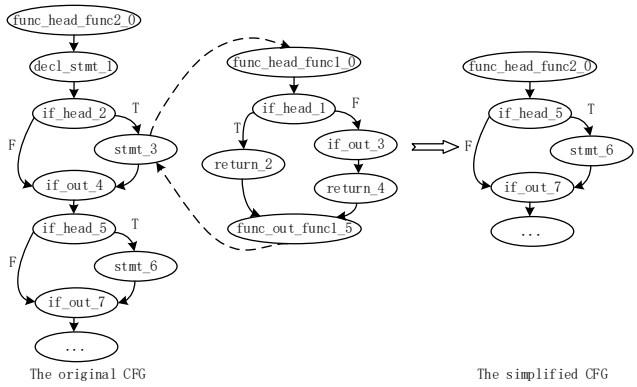

Figure 3 The original and simplified CFGs of *func*2()

Fig. 3 shows the original and simplified CFGs of *func*2() in Fig. 1. In the original CFG, the corresponding node of IAO suspected fault is node *stmt*_6. The node *stmt*_3 represents the definition statement of variable *b*, in which the variable *b* is determined as the irrelevant variable of the suspected fault and the called function *func*1() has no side effects on the *stmt*_6 during the influencing predicates calculation. Therefore, the node *stmt*_3 is determined as an irrelevant node, and the called function *func*1() does not need to be analyzed in detail. Similarly, due to the node *if_head*_2 being regarded as a non-influencing predicate in the step of influencing predicates calculation, it is also determined as an irrelevant node. Finally, the nodes *decl_stmt*_1, *if_head*_2, *stmt*_3, *and if_out*_4, and the edges originated these nodes can be pruned in the original CFG of *func*2(), and the simplified CFG of *func*2() is shown in the right of Fig. 3. By the backward traversing the simplified CFG from the node *stmt*_6, only one path can be generated from the entry of function *func*2() to the suspected fault point, which is less than three paths that are generated by the backward traversing the original CFG from the node *stmt*_6. Additionally, the path constraint of the path generated by the backward traversing the simplified CFG only has only one constraint condition, and the path constraint of the path generated by the backward traversing the original CFG has at least one constraint condition. Therefore, the CFG simplification not only can alleviate the risk of path explosion but also can improve the efficiency of path feasibility determination.

4.4 Diagnosis of Path Generation

The diagnosis of path generation can be regarded as the last phase of our approach. Because backward traversing the Inter-procedural control flow graph (ICFG) may generate a large number of paths with more constraint conditions, it is difficult to determine the feasibility of these paths. Therefore, in order to improve the efficiency of path feasibility determination and reduce the generation of inter-procedural paths, the "section-whole" strategy is employed in this phase, and it divides the diagnosis path generation into two steps: intra-procedural diagnosis path generation and inter-procedural diagnosis path generation.

It is necessary to detect the feasibility of a path as early as possible and prevent the delivery of path infeasibility during the inter-procedural path generation. The intra-procedural diagnosis path is generated first, then the inter-procedural diagnosis path is generated according to the intra-procedural diagnosis path and the function call relationship. We use the unsatisfiable path constraint patterns to detect the infeasible paths during the intra-procedural and inter-procedural diagnosis path generation [29]. With the simplified CFG of each associated procedure and the sequence of function calls related to the procedure that the suspected fault is located in it, we backward traverse the simplified CFG from the suspected fault point or a call site. If a node in the simplified CFG invokes a procedure which has a direct or indirect effect on the suspected fault, then the function summary of this procedure should be calculated. The function summary contains feasible paths information of the procedure, return value of the procedure, and the side effects of the procedure. The explored path which can reach this node should combine with each of feasible paths of the callee, and the feasibility of these combined paths should be determined so that the traversal can continue to be executed only along these feasible paths. Finally, we can generate the paths from the entry of procedure to a special point of procedure, in other words, the intra-procedural diagnosis paths are generated. It should be noted that the trigger condition of the suspected fault is regarded as a path constraint condition when determining the feasibility of the traversed paths because it can prune many of the traversed paths.

Generally, although a test case of intra-procedural path sometimes can trigger the suspected fault, due to the complex function call relationship may exist in a large program, the intra-procedural diagnosis path could not accurately confirm a suspected fault. By the call graph of the program, the function call sequences related to the procedure that the suspected fault locates in it can be calculated. Then, according to the simplified CFG of every procedure in these sequences, the inter-procedural diagnosis paths can be calculated by backward traversal along with these sequences. The traversal will not stop until all of the external input nodes that the suspected fault data are dependent on have been traversed.

To better understand the "section- whole" strategy, the example in Fig. 1 is used to illustrate these two steps. First, the intra-procedural path of *func*2() 9-13-14 is generated by traversing the simplified CFG of *func*2() from the suspected fault point. Then, according to the inter-procedural data-dependent analysis and the function call

relationship, the simplified CFG of procedure *foo*() should be calculated. And due to the procedure *func*1() located at line 5 being an associated procedure, the paths in procedure *func*1*()* should be combined with the traversed paths when the traversing arrives at this node. Two paths 3-4-5-17-18-20-21 and 3-4-5-17-18-19-21 can be generated by backward traversing the simplified CFG of *foo*() from the node *stmt*_6, but since the path 3-4-5-17-18-20-21 is an infeasible path, only the path 3-4-5-17-18-19-21 can combine with the path 9-13-14 to generate the inter-procedural path. Finally, the inter-procedural path 3-4-5-17-18-19-21-9-13-14 is generated, which is determined as a feasible path. Therefore, this path is regarded as an inter-procedural diagnosis path of suspected fault IAO. Compared with the strategy of directly backward traversing the entire ICFG, the "section-whole" strategy can detect the feasibility of a path as early as possible and prevent the transmission of path infeasibility.

5 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. Finally, we discuss the experimental results.

5.1 Experimental Design

To evaluate our approach, we conduct experiments on five open source C language programs. The basic information of these programs is given in Tab. 1, the columns *File*, *LOC*, *Function*, *SF*, *RF*, and *FF* represent the number of files, lines of code, the number of functions, the number of suspected faults detected by DTS, the number of real faults and the number of fake faults confirmed by manual, respectively. To ensure the effectiveness of the experiments, first, we confirm the suspected faults of each benchmark by manual, if there is a test case that can trigger a suspected fault, then the suspected fault is considered as a real fault (*RF*). In contrast, if none of the test cases can trigger a suspected fault, then the suspected fault is considered as a fake fault (*FF*). The number of real faults and fake faults in each benchmark is shown in Tab. 1. Then, we select 5 real faults and 5 fake faults from each benchmark by a random program. Finally, we treat the selected real faults and fake faults as the seeds.

Table 1 Basic information about the benchmark

Benchmark	*File*	*LOC*	*Function*	*SF*	*RF*	*FF*
spell-1.0	4	1820	26	38	14	24
a200c	37	6584	85	80	23	57
barcode-0.98	15	4166	56	73	26	47
antiword	78	20213	566	112	28	84
sphinxbase	68	22709	576	470	257	213

To better evaluate the efficiency of our approach, we compare the approach proposed in this paper with the one that generates the diagnosis paths regarding a suspected fault by directly backward traversing the whole ICFG. We count the number of paths traversed by each approach for the inter-procedural diagnosis path generation and also count the number of predicates that the traversed paths go through. In addition, the number of predicates and paths generated by these two approaches are compared, respectively. Moreover, we also simply evaluate the accuracy of our approach. If a suspected fault is a real fault, and the inter-procedural diagnosis path is not generated by an approach, then a false negative case is counted for that approach. In contrast, if a suspected fault is not a real fault, the inter-procedural diagnosis path is generated by an approach, then a false positive case is counted for that approach. Finally, to further illustrate the effectiveness of our approach, we select a real fault whose inter-procedural diagnosis path can be generated by each of these two approaches, and the number of predicates in the inter-procedural diagnosis path is respectively counted.

5.2 Evaluation Metrics

We use standard *Precision* and *Recall* metrics to evaluate the accuracy of an approach. *Precision* measures the actual inter-procedural diagnosis path that is correctly generated in terms of a percentage of the total number of inter-procedural diagnosis paths, while *Recall* measures the ability of an approach to find the actual inter-procedural diagnosis path. By using *TP*, *FP*, and *FN* to denote true positive, false positive and false negative detection results, respectively, the *Precision* and *Recall* can be computed using Eqs. (1) and (2).

$$Precision = \frac{TP}{(TP + FP)} \tag{1}$$

$$Recall = \frac{TP}{(TP + FN)} \tag{2}$$

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

5.3 Experimental Results

Tab. 2 lists the experimental results of the two approaches mentioned above. The first column gives the name of each benchmark. Columns *RF*, *FF*, *TravPath*, and *TravPre* represent the number of real faults taken into consideration, the number of fake faults taken into consideration, the number of traversed paths for generating the inter-procedural diagnosis paths and the number of traversed predicates for generating the inter-procedural diagnosis paths, respectively. Columns *TP*, *FP*, *FN*, *TN* and *Time* give the number of the real faults that are confirmed as the real faults by the diagnosis paths (true positive cases), the number of the fake faults that are wrongly confirmed as the real faults by the diagnosis path (false positive cases), the number of the real faults that cannot be confirmed as the real faults by the diagnosis paths (false negative cases), the number of the fake faults that are confirmed as fake faults by the diagnosis paths (true negative cases), and the cost time of an approach, respectively. Furthermore, the columns Approach I and Approach II represent the approach proposed in this paper and the approach that generates the inter-procedural

diagnosis path by directly backward traversing the whole ICFG, respectively

Tab. 2 shows that, for the 25 real faults in the seeds, 16 inter-procedural diagnosis paths and 12 inter-procedural diagnosis paths were generated by Approach I and Approach II, respectively. That is to say, Approach I and Approach II only can confirm 16 real faults and 12 real faults respectively, which have 9 false negative cases and 13 false negative cases respectively. For each fake fault in each benchmark, no diagnosis path is generated during the experiment (the generated paths are either infeasibility or fail to trigger the fake fault), that is to say, there is no test case that can trigger the fake fault. Therefore, these fake faults are identified as fake faults, and the number of true negative cases is the same as the number of fake faults. According to equation (2), Recall values can be calculated. The former approach achieves higher *Recall* value than the latter one and the *Recall* improvement of the former over the latter 16% for all the subject programs on average. Moreover, because neither of these two approaches causes any false positives for the real faults, each of inter-procedural diagnosis paths can trigger a real fault. Thus, the *Precision* also can be calculated by the equation (1), and the *Precision* of each approach is 100%.

Table 2 Experimental results

Benchmark	Seed		Approach I							Approach II						
	RF	*FF*	*TraPath*	*TraPre*	*TP*	*FP*	*FN*	*TN*	*Time*	*TraPath*	*TravPre*	*TP*	*FP*	*FN*	*TN*	*Time*
spell-1.0	5	5	236	1 013	5	0	0	5	264	647	15942	5	0	0	5	591
a200c	5	5	3938	26174	4	0	1	5	759	17421	374692	3	0	2	5	976
barcode-0.98	5	5	3747	30219	3	0	2	5	872	13487	353746	2	0	3	5	1187
antiword	5	5	6475	76986	2	0	3	5	1 184	10109	428751	1	0	4	5	1409
sphinxbase	5	5	8089	139712	2	0	3	5	1351	13524	681536	1	0	4	5	1593
Total	25	25	22485	274 104	16	0	9	25	4430	55188	1854667	12	0	13	25	5756

To generate the inter-procedural diagnosis paths, we set the maximum expansion of loop to 1, and the maximum computation time for one suspected fault to 180 seconds during our experiments (further increasing the maximum computation time cannot bring noticeable improvement to the current experimental results). Accordingly, 22485 paths and 55188 inter-procedural paths were traversed by Approach I and Approach II respectively, which totally contain 274104 predicates and 1854667 predicates respectively. Furthermore, comparing the total time costs of Approach I and Approach II, Approach I requires 4430 seconds less than Approach II that needs 5756 seconds. That is to say, in terms of efficiency, Approach I increased by 29.9% compared with Approach II.

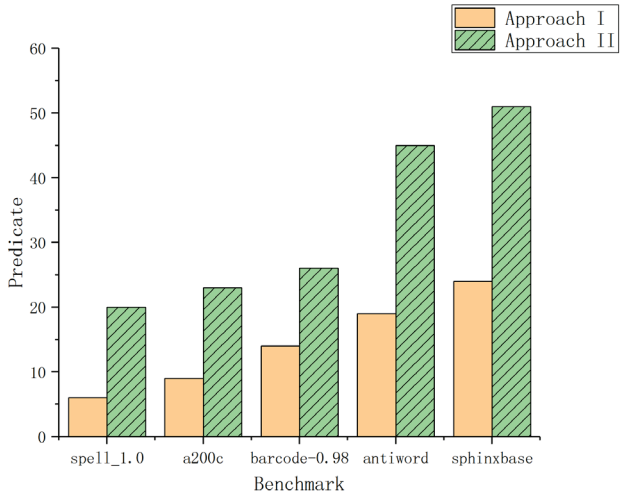

Figure 4 A comparison of the number of predicates in the diagnosis paths generated by two approaches

Although it is different than the number of inter-procedural diagnosis paths that were generated by Approach I and Approach II respectively, for the given real faults, any of the inter-procedural diagnosis paths that were generated by Approach II also can be generated by Approach I. Therefore, to illustrate the advantages of our approach, we choose a real fault of which the inter-procedural diagnosis path can be generated by each of these two approaches from each benchmark program, and compare the number of predicates included in these two inter-procedural diagnosis paths. Fig. 4 shows the number of predicates in two inter-procedural diagnosis paths that were generated for the same real fault by Approach I and Approach II, respectively. The orange cylinder represents the experimental results of Approach I, and the green cylinder with oblique line represents the experimental results of Approach II. As we can see from Fig. 4, five inter-procedural diagnosis paths are generated by Approach I in which only 72 predicates are traversed totally, while 171 predicates are traversed for Approach II. On average, Approach I needs to traverse 12.2 predicates to generate an inter-procedural diagnosis path while Approach II needs to traverse 33.6 predicates. Therefore, the above results show that our approach achieves higher efficiency than Approach II.

5.4 Discussion

Although the experimental results show that 64% of the inter-procedural diagnosis paths can be generated for the real faults by using our approach, there are 9 false negative cases, which account for 36% of all real faults. We carefully analyzed all the false negative cases and found that the following issues still have an effect on the generation of inter-procedural diagnosis path, such as complex cyclic structures and arrays, recursive function.

As our approach employs the "0-1" cycle strategy during the traversal of CFG, if a definition statement of an associated variable related to the suspected fault locates in the complex cycle structure, then the accurate data dependencies of this associated variable cannot be obtained. Accordingly, if the actual diagnosis path of a real fault must go through this cycle body, while any traversed paths through this cycle body may be identified as the diagnosis path of this real fault, then the exploration of diagnosis path will continue until the other inter-procedural diagnosis path is generated or the traversal exceeds the limitation of maximum computation time. Therefore, the

complex cycle structure is an influencing condition that may give rise to a false negative.

Although our approach can prune a lot of redundant paths and predicates during the diagnosis path generation by using the analysis of inter-procedural data dependency and the "section-whole" strategy, if the value of an associated variable depends on the return value of a complex function, and the function summary of this function that has a number of callees with complex invocation relationship, then the computation of function summary needs cost much time and the function summary will have a large number of paths, which would lead to the diagnosis path being generated in the limited time. Similarly, the recursive function is also an influencing condition of diagnosis path generation because of the complex computation of function summary.

Additionally, from Tab. 1 and Tab. 2, we can see that the more functions or lines in the program, the more difficult to generate an inter-procedural diagnosis path, especially the traditional approach that uses the directly backward traversing ICFG strategy to generate the inter-procedural diagnosis path. Therefore, compared with the traditional approach, although our approach may give rise to some false negative cases and need to improve in some aspects, such as the processing of complex cycle structure and recursive function, the generation of function summary for the complex function, the Recall value of our approach increases by 16%, and both the number of traversed paths and the number of traversed predicates for generating the inter-procedural diagnosis path are greatly reduced.

6 CONCLUSION

Because initial results of code static analysis always have a large number of false positives, they need to be confirmed by manual or automatic confirmation tools. To aid the testers to confirm the suspect faults as soon as possible, we present a novel approach that combines the demand-driven analysis and the inter-procedural data flow analysis. In our approach, we first compute the influencing nodes of the suspected fault. Then, according to the influencing nodes, we simplify the CFG of each associated procedure. Finally, we use the "section-whole" strategy to generate the inter-procedural diagnosis path which can confirm the suspect faults automatically. To evaluate the accuracy and efficiency of our approach, we also conduct experiments on five open source C projects. And a comparison study between our approach and the traditional approaches also demonstrated that our approach effectively outperforms the traditional approaches.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (U1736110, 61702044, 61502029).

7 REFERENCES

[1] Hailpern, B. & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM System Journal, 41*(1), 4-12. https://doi.org/10.1147/sj.411.0004

[2] Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, *RTI Project*, 7007(11), 1-309.

[3] Bessey, A., Block, K., & Chelf, B. (2010). A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM, 53*(2), 66-75. https://doi.org/10.1145/1646353.1646374

[4] Li, Z., Zhang, J., Liao, X., & Ma, J. (2015). Survey of Software Vulnerability Detection Techniques. *Chinese Journal of Computer, 38*(4), 717-732. https://doi.org/10.3724/SP.J.1016.2015.00717

[5] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., & Rival, X. (2005). The Astrée analyzer. In *Proceedings of 14th European Symposium on Programming, ESOP 2005*, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, 21-30. https://doi.org/10.1007/978-3-540-31987-0_3

[6] Xiao, Q., Gong, Y., Yang, Z., Jin, D., & Wang, Y. (2010). Path-Sensitive Static Defect Detecting Method. *Journal of Software, 21*(2), 209-217. https://doi.org/10.3724/SP.J.1001.2010.03782

[7] Zhu, H., Jin, D., & Gong, Y. (2018). False positive elimination in suspected code fault automatic confirmation. *International Journal of Computers and Application, 40*(3), 1-9. https://doi.org/10.1080/1206212X.2017.1397342

[8] Mei, H. Wang, Q. Zhang, L., & Wang, J. (2009). Software Analysis: A Road Map. *Chinese Journal of Computer, 32*(9), 1697-1710. https://doi.org/10.3724/SP.J.1016.2009.01697

[9] Barik, T. (2016). How Should Static Analysis Tools Explain Anomalies to Developers? In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1118-1120. https://doi.org/10.1145/2950290.2983968

[10] Zhang, D., Sui, J., & Gong, Y. (2017). Large scale software test data generation based on collective constraint and weighted combination method. *Tehnicki vjesnik, 24*(4), 1041-1049. https://doi.org/10.17559/TV-20170319045945

[11] Bush, W. R. Pincus, J. D., & Sielaff, D. J. (2000). A static analyzer for finding dynamic programming errors. *Software Practice and Experience, 30*(7), 775-802. https://doi.org/10.1002/(SICI)1097-024X(200006)30:7%3C775::AID-SPE309%3E3.0.CO;2-H

[12] Manevich, R., Sridharan, M., Adams, S., Das, M., & Yang, Z. (2004). PSE: explaining program failures via postmortem static analysis. *ACM SIGSOFT Software Engineering Notes, 29*(6), 63-72. https://doi.org/10.1145/1029894.1029907

[13] Cheng, S. Jiang, F. Lin, J., & Tang, Y. (2009). Automatic verification of possible software doubtful defects. *Journal of Tsinghua University (Science and Technology)*, 49, 2222-2227. https://doi.org/10.16511/j.cnki.qhdxxb.2009.s2.011

[14] Dillig, I., Dillig, T., & Aiken, A. (2012). Automated Error Diagnosis Using Abductive Inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 181-192. https://doi.org/10.1145/2254064.2254087

[15] Chen, N. & Kim, S. (2015). Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering, 41*(2), 198-220. https://doi.org/10.1109/TSE.2014.2363469

[16] Yao, F., Li, Y., Chen, Y., Xue, H., & Venkataramani, G. (2017). StatSym: Vulnerable Path Discovery through Statistics-Guided Symbolic Execution. *IEEE/IFIP International Conference on Dependable Systems and Networks*, 109-120. https://doi.org/10.1109/DSN.2017.57

[17] Kasikci, B., Schubert, B., Pereira, C., Pokam, G., & Candea, G. (2015). Failure sketching: a technique for automated root cause diagnosis of in-production failures. *Proceedings of the 25th Symposium on Operating Systems Principles*, 344-360. https://doi.org/10.1145/2815400.2815412

[18] Yi, Q., Yang, Z., Guo, S., Wang, C., Liu, J., & Zhao, C. (2018). Eliminating path redundancy via post-conditioned symbolic execution. *IEEE Transactions on Software Engineering, 44*(1), 25-43. https://doi.org/10.1109/TSE.2017.2659751

[19] Zhang, D. (2017). High-speed Train Control System Big Data Analysis Based on Fuzzy RDF Model and Uncertain Reasoning. *International Journal of Computers, Communications & Control, 12*(4), 577-591. https://doi.org/10.15837/ijccc.2017.4.2914

[20] Zhang, D., Jin, D., Gong, Y., Chen, S., & Wang, C. (2015). Research of alarm correlations based on static defect detection. *Tehnicki vjesnik, 22*(2), 311-318. https://doi.org/10.17559/TV-20150317102804

[21] Das, M., Lerner, S., & Seigle, M. (2002). ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, 57-68. https://doi.org/10.1145/512529.512538

[22] Xie, Y., Aiken, A. (2007). Saturn: A scalable framework for error detection using Boolean satisfiability. *ACM Transactions on Programming Languages & Systems (TOPLAS), 29*(3), 16-58. https://doi.org/10.1145/1232420.1232423

[23] Hallem, S., Chelf, B., Xie, Y., & Engler, D. (2002). A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, 69-82. https://doi.org/10.1145/512529.512539

[24] Dillig, I. Dillig, T. Aiken, A., & Sagiv, M. (2011). Precise and compact modular procedure summaries for heap-manipulating programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 567–577. https://doi.org/10.1145/1993498.1993565

[25] Le, W. & Soffa, M. L. (2008). Marple: A demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, 272-282. https://doi.org/10.1145/1453101.1453137

[26] Pande, H. & Landi, D. W. (1991). Interprocedural Def-Use Associations in C Programs. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, 139-153. https://doi.org/10.1145/120807.120820

[27] Harrold, M. J. & Soffa, M. L. (2004). Efficient Computation of Interprocedural Definition-Use Chains. *ACM Transactions on Programming Languages and Systems (TOPLAS), 16*(2), 175-204. https://doi.org/10.1145/174662.174663

[28] Zheng, X. & Rugina, R. (2008). Demand-driven alias analysis for C. *ACM SIGPLAN Notices, 43*(1), 197-208. https://doi.org/10.1145/1328438.1328464

[29] Zhu, H., Jin, D., Gong, Y., Xing, Y., & Zhou, M. (2019). Detecting interprocedural infeasible paths based on unsatisfiable path constraint patterns. *IEEE Access*, 7, 15040-15055. https://doi.org/10.1109/ACCESS.2019.2894593

Contact information:

Honglei ZHU, Corresponding author
State Key Laboratory of Network and Switching Technology,
Beijing University of Posts and Telecommunications,
No. 10 Xitucheng Road, 100876 Beijing, China
E-mail: zhuhonglei@bupt.edu.cn

Dahai JIN
State Key Laboratory of Network and Switching Technology,
Beijing University of Posts and Telecommunications,
No. 10 Xitucheng Road, 100876 Beijing, China
E-mail: jindh@bupt.edu.cn

Yunzhan GONG
State Key Laboratory of Network and Switching Technology,
Beijing University of Posts and Telecommunications,
No. 10 Xitucheng Road, 100876 Beijing, China
E-mail: gongyz@bupt.edu.cn