Dependence Guided Symbolic Execution

Haijun Wang, Ting Liu, Member, IEEE, Xiaohong Guan, Fellow, IEEE, Chao Shen, Member, IEEE, Qinghua Zheng, Member, IEEE, and Zijiang Yang, Senior Member, IEEE

Abstract—Symbolic execution is a powerful technique for systematically exploring the paths of a program and generating the corresponding test inputs. However, its practical usage is often limited by the path explosion problem, that is, the number of explored paths usually grows exponentially with the increase of program size. In this paper, we argue that exploring more paths does not necessarily reveal more program behavior. If the program paths is also prohibitively expensive. For a medium-size program, systematically exploring the paths is also prohibitively expensive.

In this paper, we argue that exploring more paths does not necessarily reveal more program behavior. If the program behavior exhibited in a path has been collectively manifested by other previously explored paths, such path is redundant and thus needs not be explored. Based on this observation, we propose a path reduction that is enabled by the concept of symbolic value. Through pruning away those paths without unique symbolic values at its statement instances, our approach can achieve the equivalent fault detection capability as traditional symbolic execution without exhaustive path exploration.

We utilize a simple program shown in Figure 1 to illustrate our idea. The program is used to compute the absolute values of two inputs \( x \) and \( y \). The right-hand side of Figure 1 shows that two paths \( \pi_1 = [T, 2, 5]^T, 6 \) and \( \pi_2 = [1^F, 4, 5^F, 8] \) have been explored, and a path \( \pi_3 = [1^F, 2, 5^F, 8] \) is being explored. Note that the superscripts \( T \) and \( F \) are used to indicate whether the true or false branch is taken at a conditional statement. We exploit symbolic value to explain whether \( \pi_3 \) needs to be explored. The symbolic value at a statement instance is the symbolic expression connecting the instance with the symbolic input variables [9]. In fact, the symbolic value at an instance represents a way in which the instance is computed from the symbolic input variables and thus can be considered as a “unit” of program behavior. For example, the symbolic values at four instances of \( \pi_3 \) are:

\[
\pi_3^1(x_0 > 0), \quad \pi_3^2(a = x_0), \quad \pi_3^3(y_0 \leq 0) \quad \text{and} \quad \pi_3^4(b = -y_0).
\]

Consider other two previously explored paths \( \pi_1 \) and \( \pi_2 \). The symbolic values at \( \pi_1^1 \) and \( \pi_2^1 \) are equivalent because they both are \( x_0 > 0 \). Similarly, there exist other three equivalent pairs: \( \langle \pi_1^2, \pi_2^2 \rangle, \langle \pi_2^3, \pi_3^3 \rangle \) and \( \langle \pi_3^4, \pi_2^4 \rangle \). In this case, \( \pi_3 \) needs not to be explored as the symbolic values at its all instances have been collectively covered by previously explored paths \( \pi_1 \) and \( \pi_2 \).

In summary, we believe that program paths collectively cover the program behavior. However, some program paths may not exhibit unique program behavior.

1 Introduction

Symbolic execution [1] [2] [3] [4] [5] [6] has recently regained the prominence as a technique for various software engineering tasks [7] [8]. It uses the symbolic inputs instead of concrete inputs to drive program execution. Through encoding the path condition as a quantifier-free, first-order logic formula and then deciding the formula with a constraint solver, symbolic execution can systematically explore the paths of a program and generate the corresponding test inputs. The desire for exhaustive path exploration is because program paths capture the underlying program behavior [4]. Covering more paths usually means more program behavior coverage and is more rigorous software testing. However, in practice such goal is often not achievable due to the path explosion problem, that is, the number of paths in a program usually grows exponentially with the increase of program size. Even for a medium-size program, systematically exploring the paths is also prohibitively expensive.

In this paper, we argue that exploring more paths does not necessarily reveal more program behavior. If the program paths is also prohibitively expensive.

Index Terms—symbolic execution, path coverage, program dependence

---

* T. Liu is with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), School of Electronic and Information Engineering, Xi’an Jiaotong University, Xi’an, 710049, China. Email: tingliu@mail.xjtu.edu.cn
* H. Wang, X. Guan and C. Shen are with MOEKLINNS, School of Electronic and Information Engineering, Xi’an Jiaotong University, Xi’an, 710049, China. Email: {hjwang, xguan, cshen}@sei.xjtu.edu.cn
* Q. Zheng is with MOEKLINNS, School of Electronic and Information Engineering, Xi’an Jiaotong University, Xi’an, 710049, China. Email: qzheng@mail.xjtu.edu.cn
* Z. Yang is with the Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA, and with MOEKLINNS, School of Electronic and Information Engineering, Xi’an Jiaotong University, Xi’an, 710049, China. Email: zijiang.yang@umich.edu
* Our tool and the benchmarks are available at: http://labs.xjtu.edu.cn/labs/wlaq/hjwang/toolbench.html
void Test(int x, int y) {
    int a, b;
    1: if(x>0) 4: if(y>0)
    2: a=x; 5: b=y;
    3: else 6: b=y;
    4: a=-x;
    5: if(y>0) 7: else
    6: b=y;
    7: else
    8: b=-y;
    9: }

    Fig. 1: Intuition behind Path Reduction based on Symbolic Value.

and such paths can be considered redundant. Therefore, it is very promising for the path reduction based on symbolic value. However, a straight-forward implementation that stores and compares the symbolic values is impractical due to the following two main obstacles. First, checking the equivalence of symbolic values is prohibitively expensive. The large overhead may neutralize the gain achieved by the path reduction. Second, we must be able to predict which paths are redundant. Identifying the redundant paths after they have been explored brings no benefit.

In this paper, we develop a practical implementation called Dependence Guided Symbolic Execution (DGSE) to soundly approximate our approach. Instead of directly checking the equivalence of symbolic values, we rely on the analysis of program dependencies to determine whether different visits to a statement instance produce the same symbolic value. Therefore, there is no need to store the symbolic values in DGSE. In order to address the second obstacle, we exploit the program dependencies to guide path exploration so that the redundant paths can be predicted and eliminated. However, traditional program dependencies that include control, data and potential dependence are not sufficient for the soundness of DGSE. Therefore, we introduce a new type of program dependence called interactive dependence, which describes the relationship that two program statements interact at the third statement. DGSE is sound if it does not miss any path with unique symbolic values.

We can prove that DGSE has the equivalent fault detection capability as traditional symbolic execution, as long as the potential faults are modeled as conditional abort statements. For example, the statements assert(c) and x=y/z can be modeled as if(!c) abort and if(z==0) abort; else x=y/z, respectively. Therefore, we claim that exhaustive path exploration is not necessary for the fault detection. We also show through empirical study that even though the potential faults are not modeled, DGSE explores significantly fewer paths but has almost equivalent fault detection capability as traditional symbolic execution.

We have implemented a prototype for DGSE based on Symbolic PathFinder (SPF) [6]. The evaluation on Siemens suite [10] and several medium-size benchmarks shows that, compared with traditional symbolic execution, DGSE explores only 6.96% to 96.57% of the paths and achieves a speedup of 1.02X to 49.56X. Additional experiments show that, given the same amount of time, DGSE explores more distinctive paths than traditional symbolic execution does. Furthermore, we demonstrate that DGSE can be applied to regression testing. The initial version of our research has been presented in [11]. The main improvements of this paper include: (1) we have added a theoretical model; (2) we have significantly revised the core algorithms; (3) we have conducted more larger experiments, etc. The contributions of this paper include the following.

- We propose a new symbolic execution approach to mitigate the path explosion problem by predicting and eliminating the redundant paths based on symbolic value. Our approach can achieve the equivalent fault detection capability as traditional symbolic execution without exhaustive path exploration, as long as the potential faults are modeled as conditional abort statements.
- We develop a practical implementation called Dependence Guided Symbolic Execution (DGSE) to make our new approach feasible. Through exploiting program dependence, DGSE is able to predict and eliminate the redundant paths at a reasonable computational cost.
- We define a new type of program dependence called interactive dependence. Together with traditional program dependencies they enable dependent guided symbolic execution to avoid the redundant paths. Without interactive dependence some distinctive paths may be missed, which leads to an unsound optimization.
- We have implemented a prototype based on Symbolic PathFinder. The experiments conducted on Siemens suite and three medium-size benchmarks show that the redundant paths are abundant and widespread, and DGSE is effective in reducing the number of explored paths as well as the time usage.
- DGSE leverages static dependence analysis and symbolic execution to enable efficient path exploration. The static dependence analysis of DGSE can be adapted to accomplish various software engineering tasks. In this paper, we demonstrate that DGSE can be adapted for regression testing.

The remainder of this paper is organized as follows. In Section 2, we review the related work. After giving the motivating example in Section 3, we introduce
traditional symbolic execution with test generation and the relevant definitions in Section 4. Then, we present DGSE in Section 5, followed by experimental results in Section 6. After discussing the threats to validity in Section 7, we demonstrate the application of DGSE in Section 8. Finally, Section 9 provides the conclusion and future work.

2 Related Work

In recent years, we have seen a significant growth in the research of symbolic execution, first proposed by Clarke [1] and King [2]. The aims of the research can be roughly divided into three directions: (1) to improve its efficiency [9] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26]; (2) to improve its effectiveness [3] [4] [27] [28] [29] [30] [31] [32] [33] [34]; and (3) to improve its application [35] [36] [37] [38] [39] [40] [41] [42] [43]. Since the goal of DGSE is to improve the efficiency of symbolic execution, we mainly discuss the research in this direction, followed by brief discussion of other two directions.

2.1 Efficiency of Symbolic Execution

The research to improve the efficiency of symbolic execution can be roughly divided into five categories. The first category is to attack the path explosion problem, e.g., [12], by integrating abstraction with symbolic execution to reduce the search space. In the second category (e.g., [13] [14]), researchers perform compositional symbolic execution to enable more efficient constraint solving. It extends symbolic execution by testing functions in isolation and encodes test results as function summaries expressed using input preconditions and output postconditions. Then, when testing high-level functions, those function summaries can be re-used. The research of the third category, e.g., [9] [16] [19] [21], adopts the notion of path equivalence to avoid exhaustive path exploration. However, which paths are considered equivalent depends on the goal of the research. For example, the research [9] is to guarantee the same input-output relations. Therefore, two program paths are equivalent if they have the same symbolic expression for the output. In contrast, the research [19] is to explore all possible program states. Based on this goal, two paths are equivalent if the symbolic states of all live variables are the same. The fourth category, e.g., [15] [20] [22] [24] [25] [44], limits the analysis scope of a program to avoid irrelevant path exploration. It is a natural idea and can be used for the target-oriented software engineering tasks, such as regression testing. For example, the research [22] first classifies the branches of a program into three categories $B_E$, $B_I$ and $B_P$. The category $B_E$ represents the branches that lead to the changes would not be reached. The category $B_I$ indicates the branches that lead to the program states would not be infected. The category $B_P$ includes the branches that lead to the infectious states would not be propagated to the output. Then, any path that executes these three categories of branches is irrelevant to the changed program behavior and thus pruned away. Finally, the research, e.g., [17] [18], in the fifth category exploits the parallelism of a program to parallelize symbolic execution. For example, the research [18] presents ranged symbolic execution, a novel technique to parallelize symbolic execution. The key insight of the research [18] is that the state of symbolic execution can be encoded succinctly by a test case. By defining a fixed branch exploration order, e.g., taking the true branch before taking the false branch at each non-deterministic branch point during the exploration, a linear order among the test inputs is determined. Therefore, two test inputs can define a range, which facilitate to distribute the path exploration—both in a sequential setting with a single worker node and in parallel setting with multiple worker nodes.

DGSE improves the efficiency of symbolic execution by predicting and eliminating the redundant paths, which shares the similarity with the research in the third category. The research that is most similar to DGSE is path exploration based on symbolic output [9], named as PESO in this paper. In fact, DGSE is also inspired by the research of PESO. PESO presents a mechanism to partition the paths based on symbolic output. Two paths are placed in the same partition if the symbolic expression connecting the output with the inputs is the same in both paths. Since symbolic output cannot be directly used to guide symbolic execution, PESO uses the relevant slice instead. If two paths have the same relevant slice with respect to the output, they are considered equivalent and placed into the same partition. A relevant slice is derived from the transitive closure of dynamic control, data and potential dependence of the output. When exploring the paths of a program, PESO only executes one path in each partition and thus greatly improves the efficiency of symbolic execution. Although PESO and DGSE both exploit program dependence to guide symbolic execution, they have different goals. DGSE aims to achieve the equivalent fault detection capability as exhaustive path exploration without systematically exploring the paths. However, PESO only tries to explore those distinctive paths relevant to the output. In other words, PESO only considers the fault detection that is relevant to the output. As a result, some faults that are irrelevant to the output such as program crash can be detected by DGSE, however, may be missed by PESO. In addition, the algorithms of PESO and DGSE are different, which makes DGSE further tackle the potential threats of PESO pointed out in the research [9]: (1) only focusing on the output may miss some faults that are irrelevant to the output; and (2) the efficiency of PESO may decrease significantly when there are multiple outputs. The motivating example in Section 3 further illustrates the differences between DGSE and PESO.
2.2 Effectiveness of Symbolic Execution

The research to enhance the effectiveness of symbolic execution mainly focuses on two areas. First, researchers develop various domain-specific solvers that are effective for some special operations and library functions [28] [29] [30]. For example, Z3-str [28] can be used for reasoning about string operation. The second area is to work around the traditional limitation of path condition through the mixed techniques [3] [4] [27] [32] [34] [45], such as mixed symbolic-concrete execution [3] [4] [27], Concolic walk [32], symmetric execution [34]. For example, the research [32] provides a simple combination of linear constraint solving and heuristic search to overcome the complex path condition. The technique first splits the path condition into linear and non-linear constraints, then finds a point in the polytope induced by the linear constraints, and last utilizes adaptive search within the polytope, guided by the constraint fitness functions, to find a solution to the whole path condition.

2.3 Application of Symbolic Execution

With the advance of symbolic execution, it has been applied to various software engineering tasks, such as regression verification [35] [36], program debugging [37] [39] [46] [47], and dynamic discovery of invariants [40] [41]. The research [35] presents partition-based regression verification. The technique symbolically groups inputs of two versions, and creates partitions for a certain subset of inputs, which either guarantee behavioral equivalence or expose behavioral difference. In program debugging, DARWIN [37] proposes an automatic approach for debugging evolving programs. The technique [37] works in two phases. In the first phase, the technique collects and composes the path constraints of the failed test case in two versions to generate the alternative test case. In the second phase, the technique compares the traces of alternative test case and failed test case to produce a bug report. Symbolic execution as well can be used to dynamically discover the program invariants. The research [41] introduces iDiscovery, which leverages symbolic execution to improve the quality of invariants computed by Daikon [48]. The candidate invariants generated by Daikon are synthesized into assertions and instrumented into the program. The instrumented code is then executed symbolically to generate new test cases that are fed back to Daikon to help further refine the candidate invariants.

3 A Motivating Example

Figure 2 presents an example program with three input variables and three conditional statements. The right-hand table of Figure 2 lists all eight paths \( p_1 \) to \( p_8 \) of the program. Since the branch sequence can uniquely identify a path, we utilize it to represent the corresponding path in the paper. For example, we use \( p_8 = [2^F, A^F, 8^F] \) to represent \( p_8 = [1, 2^F, A^F, 7^F, 8^F, 11, 12, 13, 14] \). We can see that there are two errors in the program, which are raised at Lines 13 and 14, respectively. When the values of variables \( a \) and \( b \) are both 2 at Line 12, no memory is allocated for \( s \). This further leads to a segmentation fault at Line 13. If the values of variables \( a \) and \( c \) are both 4 at Line 14, a division-by-zero error will happen at Line 14. Column Error in the table indicates the errors that can be manifested by the corresponding paths.

For the given program, traditional symbolic execution systematically explores all eight paths, as indicated by column TraSE in the table. Although traditional symbolic execution is able to detect both errors, exhaustive path exploration is prohibitively expensive in practice.

Column DGSE lists the paths explored by our approach. Although paths \( p_4 \) and \( p_8 \) are pruned away, DGSE is still able to detect both errors of the program. At the first glance, it may be surprising that our approach eliminates \( p_4 \) and \( p_8 \) because both are error-revealing paths. We take \( p_8 \) to explain the rationality of such reduction, and \( p_4 \) is pruned away based on the similar reason. In fact, pruning away \( p_8 \) does not affect the fault detection capability because the same error has been manifested in \( p_6 \). Two paths \( p_6 \) and \( p_8 \) only differ at Line 4 where \( p_6 \) takes \( 4^T \) and \( p_8 \) takes \( 4^F \). Although taking different branches leads to different values of \( b \), such variable actually has no impact on Line 14. As a result, \( p_6 \) and \( p_8 \) produce the same symbolic value at Line 14 and thus have the same capability to detect the division-by-zero error at Line 14. Even so, it does not mean that \( p_6 \) is exactly equivalent to \( p_8 \) because \( p_8 \) may cover other symbolic values that are not covered by \( p_6 \). For example, the symbolic values at Line 13 are \( s = "abcd" \) in \( p_6 \) and \( s = "ab" \) in \( p_8 \). The reason that \( p_8 \) is pruned away is because its all symbolic values are collectively covered by previously explored paths \( p_6 \) and \( p_7 \).

In the remaining of this section, we further use the example to illustrate the differences between DGSE and PESO [9]. The path partition in PESO is based on the output. An obvious choice of the output is Line 14 where the result is returned from the function. Column PESO14 lists the explored paths in PESO by treating Line 14 as the output. Although PESO explores fewer paths than DGSE does, it cannot detect the error at Line 13 because such error is irrelevant to the output. In order to remedy this problem, a straight-forward solution is to designate both Lines 13 and 14 as the outputs so that the error at Line 13 can also be detected by PESO. However, when multiple outputs are considered, the efficiency of PESO significantly decreases. As shown in column PESO13&14, PESO has to explore all eight paths. Another solution is to apply PESO multiple times and each time is based on a different output. The results of such solution are shown in column PESO13&PESO14, where PESO is applied twice: once for Line 13 and the other for Line 14. It can be observed that paths \( p_4 \) and \( p_8 \) are pruned away, same as DGSE. However, paths \( p_4 \) and \( p_5 \) are explored twice, making the total number of explored paths still be eight.
char s = null;
int Test(int x, int y, int z) {
  0: int a, b, c;
  1: a = 4;
  2: if (x > 1)
  3:   a = 2;
  4: if (y < 1)
  5:   b = 0;
  6: else
  7:   b = 2;
  8: if (z < 2)
  9:   c = z;
 10: else
 11:   c = 4;
 12: s = (char *)malloc(a - b);
 13: strncpy(s, "abcd", a - b);
 14: return 1 / (a - c);
 15:}

Algorithm 1 Traditional Symbolic Execution with Test Generation

1: SymbolicExec(Program P)
2:   T ← ∅
3:   while (stack ≠ ∅) do
4:     (pcon, ψ) ← stack.pop()
5:     if (pcon is satisfiable) then
6:       let t be an input that satisfies pcon
7:       T ← T ∪ {t}
8:     PathExec(P, t, ψ)
9:   end if
10: end while
11: return T
12: end procedure
13: PathExec(P, t, ψ)
14:   execute t in P and compute path condition pcon
15:   let pcon ← ψ₁ ∧ . . . ∧ ψₘ
16:   d ← 0 or index of ψᵢ in pcon that matches ψ
17:   for (i ← d + 1; i ≤ m; i++) do
18:     negate ψᵢ and obtain path condition pconᵢ
19:     stack.push((pconᵢ, ¬ψᵢ))
20:   end for
21: end procedure

When PESO is applied multiple times, it may lead to the repetitive path exploration.

4 Preliminaries

In this section, we first introduce traditional symbolic execution with test generation, and then present the relevant definitions.

4.1 Traditional Symbolic Execution with Test Generation

In this section, we review traditional symbolic execution with test generation, as shown in Algorithm 1. Given a program P and an initial task (true, true) (Line 3), the algorithm systematically explores the paths of P and generates the corresponding test inputs T. In the algorithm, a task is a pair (pcon, ψ), where pcon is a conjunction of constraints used to guide symbolic execution to explore a particular path, and ψ is the last constraint of pcon used to avoid repetitive path exploration. The term ψ is called the initiating constraint in the paper.

Presented at Lines 4-11, the main procedure of the algorithm iteratively generates test inputs based on the tasks until there is no to-be-explored task in stack. During each iteration, a to-be-explored task (pcon, ψ) is popped from stack (Line 5). If pcon is satisfiable, the algorithm outputs a test input that can drive P to follow a path prefixed by pcon (Lines 7-8). Note that the first input is generated through the path condition true. In theory, the path condition true leads to a random input, however, in practice it is not that case in many symbolic execution implementations such as SPF [6]. SPF takes priority to execute the true (false) branch at a non-deterministic conditional point in the exploration. As a result, the first path in SPF is not random and thus the first input is also not random.

The to-be-explored tasks, which drive the future path exploration, are generated in the procedure PathExec. When executing the test t in P (Line 15), a path condition pcon = ψ₁ ∧ . . . ∧ ψₘ (Line 16) corresponding to the execution is obtained. Line 17 indicates which constraints in pcon can be negated to generate the to-be-explored tasks. If the initiating constraint ψ is true, which indicates that it is the first path exploration, the algorithm should negate all the possible constraints and thus d is assigned 0. Otherwise, ψ must match a constraint ψᵢ ∈ pcon. In this case, the algorithm only negates the constraints behind ψᵢ in pcon. At Lines 18-21, the algorithm negates the constraints to generate the to-be-explored tasks.

Table 1 shows the path exploration of Algorithm 1 in the program of Figure 2. The first column lists the index of each explored path. Column From gives the
task from which the path is initiated. The next column shows the path, followed by the corresponding test input. Column `pcon` represents the path condition. The following column `ψ` indicates the initiating constraint. Only those constraints behind `ψ` in `pcon` can be negated to generate the to-be-explored tasks. The last column lists the to-be-explored tasks that are generated under the present path. Note that we use the branch to represent its corresponding constraint in a path condition. Assume that the first path is \( \pi_1 = [2^T, 4^T, 8^T] \). At the end of its exploration, a test input \((2, 0, 1)\) is obtained. Meanwhile, three to-be-explored tasks are generated and pushed into stack: \( (2^F, 4^F), (2^T \land 4^F \land 8^F) \) \( \) and \( (2^T \land 4^T \land 8^F, 8^F) \). The path \( \pi_2 \) is initiated from \( (2^T \land 4^T \land 8^F, 8^F) \). It does not generate any to-be-explored task because there is no constraint behind \( 8^F \) in its `pcon`. The to-be-explored task \( (2^T \land 4^T \land 8^F, 8^F) \) leads to path \( \pi_3 = [2^T, 4^F, 8^T] \). Through negating \( 8^T \) to \( 8^F \), the algorithm generates the new to-be-explored task \( (2^T \land 4^F \land 8^F, 8^F) \) under \( \pi_3 \). When Algorithm 1 terminates, all eight paths are explored and the corresponding test inputs are also obtained.

### 4.2 Program Dependence

The control flow graph (CFG) of a program can be formally represented by a tuple \( (N, E) \), where \( N \) is a set of program nodes and \( E \subseteq N \times N \) represents the execution flow between the nodes. If \( n_i \) is a conditional statement, we use \( n_i^T \) or \( n_i^F \) to represent \( n_i \) depending on whether the true or false branch of \( n_i \) is taken. We also use \( br_i \) or \( br_i' \) to represent \( n_i \) when we do not care whether the branch of \( n_i \) is true or false. We consider that \( n_i^T \) and \( n_i^F \) (also \( br_i \) and \( br_i' \)) are opposite, and they themselves are also considered as the nodes. Let \( Br \subseteq N \) be a set of branch nodes.

**Definition 1.** Control Dependence is a map \( controlD : Br \times N \rightarrow \{T, F\} \) that returns true for a pair of nodes \( \langle br_i, n_j \rangle \) if \( br_i \) must lead to the execution of \( n_j \) while \( br_i' \) may result in \( n_j \) not being executed; otherwise it returns false.

Our definition of control dependence is slightly different from its standard definition. The branch information is added in our definition to differentiate the different branches of a conditional statement, which is the same as the research [16]. Consider the program in Figure 2. In standard definition, node 3 is control-dependent on node 2. However, in our definition we have \( controlD(2^T, 3) = T \) while \( controlD(2^F, 3) = F \).

**Definition 2.** Data Dependence is a map \( dataD : N \times N \rightarrow \{T, F\} \) that returns true for a pair of nodes \( \langle n_i, n_j \rangle \) if there exists a path \( \pi \) from \( n_i \) to \( n_j \) where \( n_i \) defines a variable \( v \), \( n_j \) uses \( v \) and no \( n_k(1 < k < j) \) in \( \pi \) redefines \( v \); otherwise it returns false.

**Definition 3.** Potential Dependence is a map \( potentialD : Br \times N \rightarrow \{T, F\} \) that returns true for a pair of nodes \( \langle br_i, n_j \rangle \) if (1) there exists a path \( \pi \) from \( br_i \) to \( n_j \) where \( n_j \) uses a variable \( v \) and the definition of \( v \) occurs before \( br_i \); (2) \( n_j \) is not control-dependent on \( br_i \) and \( br_i' \); and (3) a different definition of \( v \) could potentially reach \( n_j \) if \( br_i' \) instead of \( br_i \) is taken; otherwise it returns false.

Our definition of potential dependence is adapted from its original definition [9] [49]. The original potential dependence is defined with respect to a specific path, which actually is dynamic potential dependence. However, our definition only requires the existence of such a path, which can be considered as static potential dependence. When our definition is applied to a specific path to consider dynamic potential dependence, it can obtain the same result as the original definition. Note that our definition also includes the branch information. Consider a pair of nodes \( (2^F, 14) \) in Figure 3: (1) there exists a path \( [1, 2^F, 4^F, 7, 8^F, 11, 12, 13, 14] \) where node 14 uses variable \( a \) and the definition of \( a \) is at node 1 that is before \( 2^F \); (2) \( controlD(2^T, 14) = controlD(2^F, 14) = F \); and (3) a different definition, which is node 3, of a reaches node 14 if \( 2^T \) instead of \( 2^F \) is taken. According to Definition 3, \( potentialD(2^T, 14) = T \).

In order to facilitate our discussion, we define Traditional Dependence as a map \( traditionalD : N \times N \rightarrow \{T, F\} \) that returns true for a pair of nodes \( \langle n_i, n_j \rangle \) if \( controlD(n_i, n_j) = T \lor dataD(n_i, n_j) = T \lor potentialD(n_i, n_j) = T \); otherwise it returns false.

Although traditional program dependencies have been widely used in various software engineering tasks,

<table>
<thead>
<tr>
<th>No.</th>
<th>From</th>
<th>Path</th>
<th>Input</th>
<th>pcon</th>
<th>ψ</th>
<th>To-be-Explored Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>π₁</td>
<td>(true, true)</td>
<td>[2^T, 4^T, 8^T]</td>
<td>(2, 0, 1)</td>
<td>2^T ∧ 4^T ∧ 8^T</td>
<td>true</td>
<td>(2^F, 2^F) ( (2^T ∧ 4^F, 4^F) ), (2^T ∧ 4^T ∧ 8^F, 8^F)</td>
</tr>
<tr>
<td>π₂</td>
<td>(2^T ∧ 4^T ∧ 8^F, 8^F)</td>
<td>[2^T, 4^T, 8^T]</td>
<td>(2, 0, 2)</td>
<td>2^T ∧ 4^T ∧ 8^F</td>
<td>8^F</td>
<td>–</td>
</tr>
<tr>
<td>π₃</td>
<td>(2^T ∧ 4^F, 8^F)</td>
<td>[2^T, 4^F, 8^T]</td>
<td>(2, 1, 1)</td>
<td>2^T ∧ 4^F ∧ 8^T</td>
<td>4^F</td>
<td>(2^T ∧ 4^F ∧ 8^F, 8^F)</td>
</tr>
<tr>
<td>π₄</td>
<td>(2^T ∧ 4^F ∧ 8^F, 8^F)</td>
<td>[2^T, 4^F, 8^T]</td>
<td>(2, 1, 2)</td>
<td>2^T ∧ 4^F ∧ 8^F</td>
<td>8^F</td>
<td>–</td>
</tr>
<tr>
<td>π₅</td>
<td>(2^F, 2^F)</td>
<td>[2^F, 4^T, 8^F]</td>
<td>(1, 0, 1)</td>
<td>2^F ∧ 4^T ∧ 8^T</td>
<td>2^F</td>
<td>(2^F ∧ 4^F, 4^F) ( (2^T ∧ 4^F ∧ 8^F, 8^F) )</td>
</tr>
<tr>
<td>π₆</td>
<td>(2^F ∧ 4^F, 4^F)</td>
<td>[2^F, 4^F, 8^T]</td>
<td>(1, 0, 2)</td>
<td>2^F ∧ 4^F ∧ 8^T</td>
<td>4^F</td>
<td>–</td>
</tr>
<tr>
<td>π₇</td>
<td>(2^F ∧ 4^F, 4^F)</td>
<td>[2^F, 4^F, 8^T]</td>
<td>(1, 1, 1)</td>
<td>2^F ∧ 4^F ∧ 8^T</td>
<td>4^F</td>
<td>(2^F ∧ 4^F ∧ 8^F, 8^F)</td>
</tr>
<tr>
<td>π₈</td>
<td>(2^F ∧ 4^F ∧ 8^F, 8^F)</td>
<td>[2^F, 4^F, 8^T]</td>
<td>(1, 1, 2)</td>
<td>2^F ∧ 4^F ∧ 8^F</td>
<td>4^F</td>
<td>–</td>
</tr>
</tbody>
</table>
they are insufficient for DGSE to explore all the distinctive paths. Because they cannot guarantee the path-prefixing property as traditional symbolic execution. In traditional symbolic execution, if \( \psi_1 \land \cdots \land \psi_{i-1} \land \psi_i \) is the prefix of a path condition, the path explored based on \( \psi_1 \land \cdots \land \psi_{i-1} \land \neg \psi_i \) still has \( \psi_1 \land \cdots \land \psi_{i-1} \land \neg \psi_i \) as the prefix. In order to remedy this problem, we propose interactive dependence to complement traditional program dependencies. An example to explain the necessity of interactive dependence will be given in Table 4.

Definition 4. Interactive Dependence is a map interactive: \( N \times N \rightarrow \{ T, F \} \) that returns true for a pair of nodes \( \langle n_i, n_j \rangle \) if there exists a path \( \pi = [\ldots n_i \ldots n_j \ldots n_k \ldots] \) where node \( n_k \) is simultaneously dependent on nodes \( n_i \) and \( n_j \) in terms of either traditional dependence or interactive dependence; otherwise it returns false.

Interactive dependence is recursively defined. The base case is that node \( n_1 \) is simultaneously dependent on nodes \( n_i \) and \( n_j \) in terms of traditional dependence. An example of interactive dependence is shown in Figure 3. There is a path \([1, 2^F, 4^F, 7, 8^F, 12, 13, 14]\) and where node 14 is simultaneously data-dependent on nodes 1 and 11. Therefore, node 11 is interactive-dependent on node 1. In the example, if either node 1 or 11 is modified, there is a chance for the division-by-zero error at node 14 to disappear. Therefore, although node 1 does not impact the value or execution of node 14, it truly impacts the effect of node 11 on node 14. Table 2 presents the complete program dependencies of the program in Figure 2.

5 Practical Path Reduction Approach

A strict implementation of our path reduction based on symbolic value is impractical due to two obstacles:

- It is prohibitively expensive to check the equivalence of symbolic values, because we have to store and compare all the symbolic values.
- It is extremely hard to predict the redundant paths if we do not execute these paths and compute the symbolic values exhibited in these paths.

In this section, we first present DGSE that soundly approximates our path reduction based on symbolic value. DGSE addresses the above two obstacles through exploiting program dependence. Next, we describe how to compute program dependence via static analysis and illustrate why we propose interactive dependence. At last, we prove the soundness of DGSE.

5.1 Dependence Guided Symbolic Execution

The optimization achieved by DGSE is sound in the sense that it does not miss any path that can exhibit unique symbolic values. DGSE adopts the following techniques to optimize symbolic execution:

- DGSE utilizes the analysis of program dependencies to check the equivalence of symbolic values, instead of the direct logic comparison.
- DGSE abandons negating some branches based on the program dependencies so that it can predict and eliminate the redundant paths.

In order to explain the above practical techniques, we first introduce the concept of relevant path slice and relevant path condition.

Definition 5. Relevant Path Slice and Relevant Path Condition. Let \( n \) be a node in the path \( \pi \). The relevant path slice \( \pi_{\rho^R}[n] \) in \( \pi \) with respect to \( n \) contains all nodes \( n' \) in \( \pi \) such that transitive.D(\( n', n \)) = \( T \), where transitive.D denotes the transitive closure of dynamic control, data, potential and interactive dependence. The relevant path condition \( \pi_{\rho^R}[n] \) in \( \pi \) with respect to \( n \) consists of all the branch nodes in \( \pi_{\rho^R}[n] \).

Given a path \( \pi \) and a branch \( br \) that is executed in \( \pi \), both relevant path slices \( \pi_{\rho^R}[br] \) and \( \pi_{\rho^R}[br'] \) are well defined in \( \pi \) even though branch \( br' \), the opposite branch of \( br \), is not executed in \( \pi \). Take path \( \pi_5 = [1, 2^F, 4^T, 5, 8^T, 9, 12, 13, 14] \) in Figure 2 for example. Since branch \( 8^T \) is executed in \( \pi_5 \), \( \pi_{\rho^R}[8^T] \) is defined with value \([1, 2^F, 8^T]\) in \( \pi_5 \). Although \( 8^F \) is not executed in

### Table 2: Program Dependencies of Example in Figure 2

<table>
<thead>
<tr>
<th>Type</th>
<th>Program Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlD</td>
<td>((2^F,3)(4^T,5)(4^F,7)(8^T,9)(8^F,11))</td>
</tr>
<tr>
<td>dataD</td>
<td>((1,12)(1,13)(1,14)(3,12)(3,13)(3,14)(5,12))</td>
</tr>
<tr>
<td>potentialD</td>
<td>((2^F,12)(2^F,13)(2^F,14))</td>
</tr>
<tr>
<td>interactiveD</td>
<td>((1,2^F)(1,4^F)(1,4^F)(1,5)(1,7)(1,8^F)(1,8^F))</td>
</tr>
<tr>
<td></td>
<td>((1,9)(1,11)(1,12)(2^F,4^T)(2^F,4^T)(2^F,4^T)(2^F,5))</td>
</tr>
<tr>
<td></td>
<td>((2^F,7)(2^F,8^F)(2^F,8^F)(2^F,9)(2^F,11))</td>
</tr>
<tr>
<td></td>
<td>((2^F,12)(3,4^F)(3,4^F)(3,5)(3,7)(3,8^F))</td>
</tr>
<tr>
<td></td>
<td>((3,8^F)(3,9)(3,11)(3,12)(5,12)(7,12))</td>
</tr>
</tbody>
</table>
$\pi_5$, $\pi_5^{RPS}[8^F]$ is also well defined in $\pi_5$. Three program dependence pairs, as shown in Table 2, are relevant to $8^F$ in $\pi_5$: $\langle 1, 8^F \rangle$, $\langle 2^F, 8^F \rangle$ and $\langle 1, 2^F \rangle$. Therefore, we have $\pi_5^{RPS}[8^F] = \{1,2^F, 8^F\}$. Note that $\pi_5^{RPS}[8^F] \neq \pi_5^{RPS}[8^F]$. Such mechanism is required, because Algorithm 2 needs to negate branches and compute their relevant path slices and relevant path conditions.

The relevant path slice $\pi^{RPS}[n]$ allows us to evaluate the symbolic value at $n$ in $\pi$. Based on Definition 5, the relevant path slice $\pi^{RPS}[n]$ contains all the nodes in $\pi$ that possibly affect the symbolic value at $n$ in $\pi$. If $\pi_1^{rps}[n] = \pi_2^{rps}[n]$, two paths $\pi_1$ and $\pi_2$ will have the same symbolic value at $n$. Therefore, we can utilize relevant path slice to check the equivalence of symbolic values. As a result, if the relevant path slices of all the nodes in $\pi$ are collectively covered by previously explored paths, $\pi$ will not exhibit any new symbolic value and thus is redundant. Definition 6 presents the formal definition of redundant path.

**Definition 6.** Redundant Path. Let $\Pi$ be a set of program paths. Given a path $\pi \notin \Pi$, let $\pi^{RPS} = \{\pi^{rps}[n]|n \in \pi\}$ be the set of relevant path slices with respect to all the nodes in $\pi$. The path $\pi$ is redundant if $\pi^{RPS} \subseteq \bigcup_{\pi_1 \in \Pi}(\pi_1^{RPS})$; otherwise $\pi$ is a distinctive path.

Algorithm 2 presents the pseudo-code for DGSE. For ease of presentation, we do not differentiate a branch and its corresponding constraint. The main procedure DGSymbolicExec of Algorithm 2 is the same as that of Algorithm 1 except the call to DGPathExec instead of PathExec at Line 9. The procedures DGPathExec and PathExec have two main differences. One is at Lines 20-21 of Algorithm 2, where DGSE utilizes relevant path condition instead of path condition of Algorithm 1. The other is at Lines 25-28. In Algorithm 1, all the possible constraints behind $\psi_d$ in pcon are negated, while Algorithm 2 negates the constraints that are not only behind $\psi_d$ in pcon but also transitively dynamically dependent on $\psi_d$. We explain these differences by comparing Table 3, where Algorithm 2 is applied to the program in Figure 2, against Table 1, where Algorithm 1 is applied.

In traditional symbolic execution, the to-be-explored tasks generated under the first path $\pi_1 = [2^F, 4^T, 8^T]$ are $[2^F, 2^F]$, $[2^T \wedge 4^F, 4^F]$ and $[2^T \wedge 4^T \wedge 8^F, 8^F]$, as shown in Table 1. However, DGSE generates different to-be-explored task $[2^T \wedge 8^F, 8^F]$ as shown in Table 3, which corresponds to $[2^T \wedge 4^T \wedge 8^F, 8^F]$, through Lines 20-21 in DGPathExec. We know that traditional symbolic execution utilizes path condition as the first term of its to-be-explored task, and the path condition includes its all preceeding constraints. Therefore, the path condition in $\pi_1$ with respect to $8^F$ is $2^T \wedge 4^T \wedge 8^F$. However, DGSE utilizes relevant path condition instead. Because $8^F$ is not transitively dynamically dependent on $4^F$ in $\pi_1$, the relevant path condition in $\pi_1$ with respect to $8^F$ is $2^T \wedge 8^F$. As a result, DGSE generates a different to-be-explored task under $\pi_1$. This is an improvement that DGSE achieves compared with traditional symbolic execution, since each to-be-explored task leads to a path exploration and the to-be-explored task $[2^T \wedge 8^F, 8^F]$ in DGSE intuitively matches two to-be-explored tasks $[2^T \wedge 4^T \wedge 8^F, 8^F]$ and $[2^T \wedge 4^F \wedge 8^F, 8^F]$ in traditional symbolic execution. Therefore, DGSE explores fewer paths than traditional symbolic execution does.

Starting from $[2^T \wedge 4^F, 4^F]$, traditional symbolic execution explores $\pi_3 = [2^T, 4^F, 8^T]$. Meanwhile, it negates $8^T$ to $8^F$ and generates the to-be-explored task $[2^T \wedge 4^F \wedge 8^F, 8^F]$ in DGSE intuitively matches two to-be-explored tasks $[2^T \wedge 4^T \wedge 8^F, 8^F]$ and $[2^T \wedge 4^F \wedge 8^F, 8^F]$ in traditional symbolic execution. Therefore, DGSE explores fewer paths than traditional symbolic execution does.

Starting from $[2^F, 2^F]$, traditional symbolic execution explores $\pi_5 = [2^F, 4^F, 8^T]$ and generates two to-be-explored tasks $[2^F \wedge 4^T \wedge 8^F, 4^F]$ and $[2^F \wedge 4^F \wedge 8^F, 8^F]$, as shown in Table 1. In DGSE, we first check whether $4^F$
and $8^F$ are transitively dynamically dependent on $2^F$, as indicated by Line 25 in DGPPathExec. Since both pass the check, DGSE also generates two to-be-explored tasks as shown in Table 3. Due to the reason of relevant path condition, the second to-be-explored task $(2^F \land 4^T \land 8^T)$ also becomes $(2^F \land 8^F, 8^F)$ in DGSE.

In summary, DGSE improves traditional symbolic execution in two ways. First, DGSE utilizes relevant path condition instead of path condition. We know that the path condition includes its all preceding constraints and is a complete path prefix. This is also the reason that the number of explored paths in traditional symbolic execution is usually exponential to the number of conditional statements of a program. The relevant path condition, on the other hand, only includes its relevant preceding constraints and may be an incomplete path prefix. Second, DGSE abandons negating some branches based on the program dependencies to predict and eliminate the redundant paths. We know that the relevant path condition may be an incomplete path prefix. If DGSE negates all the possible branches as traditional symbolic execution, it may generate the identical tasks, which is not necessary. These two main differences provide the potential for DGSE to achieve the path reduction, even maybe an exponential path reduction. Consider the program shown in Figure 4 with $N$ independent if-else blocks. Traditional symbolic execution explores totally $2^N$ paths. However, DGSE achieves an exponential path reduction through exploring only $N + 1$ paths: $[1^T, \ldots, N^T]$ and $[1^T, \ldots, k^F, \ldots, N^T] (1 \leq k \leq N)$.

### 5.2 Program Dependence Analysis

DGSE leverages static dependence analysis and symbolic execution in synergy to enable efficient path exploration. In this section, we introduce the static dependence analysis. Since control, data and potential dependence have been widely studied [9] [16] [50], we focus on newly proposed interactive dependence. First, we utilize the program in Figure 2 to illustrate that traditional dependencies are insufficient for DGSE to explore all the distinctive paths. Table 4 presents the path exploration of DGSE without considering interactive dependence, as indicated by column $transTraD(\psi_d, \neg \psi_i)$. In the paper, $transTraD$ denotes the transitive closure of dynamic control, data and potential dependence, especially not including interactive dependence. Assume that the first path is still $\pi_1 = [2^T, 4^T, 8^T]$. At the end of its exploration, three to-be-explored tasks $(2^F, 2^F)$, $(4^F, 4^F)$ and $(8^F, 8^F)$ are generated. Compared with those in Table 3, the to-be-explored tasks $(4^F, 4^F)$ and $(8^F, 8^F)$ no longer contain the constraint $2^T$ because $4^F$ and $8^F$ are not transitively dynamically dependent on $2^T$ without interactive dependence. The discrepancy appears in $\pi_5$. As shown in Table 3, DGSE generates the to-be-explored tasks $(2^F \land 4^F, 4^F)$ and $(2^F \land 8^F, 8^F)$ under $\pi_5$. However, it does not generate any to-be-explored task as shown in Table 4, because both $4^F$ and $8^F$ are no longer transitively dynamically dependent on $2^F$ without interactive dependence. As a result, DGSE terminates prematurely without interactive dependence and the division-by-zero error at Line 14 does not be detected.

In the rest of this section, we discuss how to compute interactive dependence. Note that we conduct static program dependence analysis at the inter-procedural level through adopting the techniques proposed in [51], even though the computational procedure introduced in this section does not make this explicit. The computation is divided into two phases: we first conduct pair reaching definition analysis and then leverage the results to compute interactive dependence.

We now introduce the simultaneous reachability of a pair of nodes. An important concept in the reachability analysis is under what situation a node is killed by another node. There are three cases. A definition node $n_i$ is killed by node $n_j$ when the variables defined at
n_i and n_j are the same. This case is for the analysis of date dependence. The next two cases happen when n_i is a branch node, and they are for the analysis of control and potential dependence, respectively. If n_i no longer directly or indirectly controls node n_j, n_i is killed by n_j. In the last case, we first identify the set of definition nodes N that are directly or indirectly controlled by the opposite branch n'_i of n_i. Then, we compute the set of variables V that are defined at the set of nodes N. When the variable v ∈ V is the same as the variable defined at n_j, n_i is killed by n_j.

Definition 7. Pair Reaching Definition. Two nodes n_i and n_j simultaneously reach node n_k if there exists a path π = [⋯n_i⋯n_j⋯n_k⋯] such that n_i is not killed between n_i and n_k, and n_j is not killed between n_j and n_k in π.

We conduct pair reaching definition analysis based on Equations (1), (2) and (3). In the equations, In[n_j] represents the set of pairs that can reach n_j, and Out[n_j] denotes the set of pairs that can flow from n_j. The set Gen[n_j] indicates the set of pairs that are generated at n_j, and Kill[n_j] is the set of pairs that are killed at n_j. We use pre(n_j) to denote the set of nodes that are immediate predecessors of n_j in CFG. In fact, Equations (1), (2) and (3) are the classical data flow analysis algorithm [52], which is implemented via graph reachability [53]. The complexity of graph reachability is in the order of polynomial time, thus the pair reaching definition analysis is also in the order of polynomial time.

\[ \text{In}[n_j] \leftarrow \{ \text{Gen}[n_j] \cup (\text{In}[n_j] - \text{Kill}[n_j]) \]  
\[ \text{Out}[n_j] \leftarrow \text{Gen}[n_j] \cup (\text{In}[n_j] - \text{Kill}[n_j]) \]  
\[ \text{In}[n_j] \leftarrow \cup_{n_i \in \text{pre}(n_j)} \text{Out}[n_i] \]  (3)

Algorithm 3 leverages the results of pair reaching definition analysis to compute interactive dependence. The procedure PairReachingDef at Line 2 represents pair reaching definition analysis. Due to the recursive nature of Definition 4, we first compute the base case using the for loop at Lines 3-9 and then compute the inductive cases using the while loop at Lines 11-28. At Lines 3-9, we compute interactive dependence based on traditional dependence. For each n_k in CFG, we obtain the pair reaching definition ⟨n_i, n_j⟩ ∈ In[n_k]. According to Definition 4, if n_k is simultaneously traditional-dependent on both n_i and n_j, n_j is interactive-dependent on n_i.

The set of all the interactive dependence pairs obtained in the base case are then added to the worklist at Line 10, which initiates further analysis. For each ⟨n_i, n_k⟩ such that n_k is interactive-dependent on n_i, we locate another n_j such that n_k is also traditional-dependent or interactive-dependent on n_j. Then we check whether ⟨n_i, n_j⟩ ∈ In[n_k]. If so, n_i is interactive-dependent on n_j. Similarly, if ⟨n_j, n_i⟩ ∈ In[n_k], n_i is interactive-dependent on n_j. The complexity of Algorithm 3 is \(O(n^3)\), where n is the number of nodes in CFG.

<table>
<thead>
<tr>
<th>No.</th>
<th>From</th>
<th>Path</th>
<th>Input</th>
<th>pcon</th>
<th>(\psi)</th>
<th>transTraD((\psi_d, \neg\psi_i))</th>
<th>To-be-Explored Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>\pi_1</td>
<td>(true, true)</td>
<td>([2^p, 4^p, 8^p])</td>
<td>(2, 0, 1)</td>
<td>(2^p \land 4^p \land 8^p)</td>
<td>true</td>
<td>–</td>
<td>(\langle 2^p, 2^p \rangle)</td>
</tr>
<tr>
<td>\pi_2</td>
<td>(8^p, 8^p)</td>
<td>([2^p, 4^p, 8^p])</td>
<td>(2, 0, 2)</td>
<td>(2^p \land 4^p \land 8^p)</td>
<td>(8^p)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>\pi_3</td>
<td>(4^p, 4^p)</td>
<td>([2^p, 4^p, 8^p])</td>
<td>(2, 1, 1)</td>
<td>(2^p \land 4^p \land 8^p)</td>
<td>(4^p)</td>
<td>transitiveD(4^p, 8^p) = (\neg)</td>
<td>–</td>
</tr>
<tr>
<td>\pi_4</td>
<td>(2^p, 2^p)</td>
<td>([2^p, 4^p, 8^p])</td>
<td>(1, 0, 1)</td>
<td>(2^p \land 4^p \land 8^p)</td>
<td>(2^p)</td>
<td>transitiveD(2^p, 4^p) = (\neg)</td>
<td>–</td>
</tr>
</tbody>
</table>

**Algorithm 3 Interactive Dependence Computation**

1: InteractiveD(CFG \(cfg\))
2: \( In \leftarrow \text{PairReachingDef}(cfg) \)
3: for (each \(n_k \in \text{CFG} \)) do
4: for (each \(\langle n_i, n_j \rangle \in \text{In}[n_k] \)) do
5: if (traditionalD(n_i, n_k) \land traditionalD(n_j, n_k)) then
6: \( S_{id} \leftarrow S_{id} \cup \langle n_i, n_j \rangle \)
7: end if
8: end for
9: end for
10: worklist \( \leftarrow S_{id} \)
11: while (worklist \( \neq \emptyset \)) do
12: \( \langle n_i, n_k \rangle \leftarrow \text{worklist}.\text{remove}() \)
13: \( N \leftarrow \{ \langle n_j | \text{traditionalD}(n_j, n_k) \land \text{interactiveD} \} \)
14: for (each \(n_j \in N\)) do
15: if (\(\langle n_j, n_i \rangle \in \text{In}[n_k]\)) then
16: \( S_{id} \leftarrow S_{id} \cup \langle n_j, n_i \rangle \)
17: if (\(\langle n_j, n_i \rangle \) has not been added) then
18: worklist \( \leftarrow \text{worklist} \cup \langle n_j, n_i \rangle \)
19: end if
20: end if
21: if (\(\langle n_j, n_i \rangle \in \text{In}[n_k]\)) then
22: \( S_{id} \leftarrow S_{id} \cup \langle n_j, n_i \rangle \)
23: if (\(\langle n_j, n_i \rangle \) has not been added) then
24: worklist \( \leftarrow \text{worklist} \cup \langle n_j, n_i \rangle \)
25: end if
26: end if
27: end for
28: end while
29: return \(S_{id}\)
30: end procedure

**TABLE 4: Incomplete Symbolic Execution without Interactive Dependence**
5.3 Proof of Soundness

In this section, we prove that DGSE has the equivalent fault detection capability as traditional symbolic execution, as long as the potential faults are modeled as conditional abort statements. We also assume that the constraint solver is sound. If a path condition cannot be decided by the constraint solver, there is no guarantee that the paths explored by traditional symbolic execution and DGSE are complete, thus their equivalence cannot be proved. Furthermore, we assume that the program dependencies in a path can be precisely determined.

In the proof, we first prove that any two adjacent nodes of a relevant path slice have a transitive-dependence relation in Lemma 5.1. Then, based on Lemma 5.1 and Algorithm 2, we prove in Lemma 5.2 that if a relevant path slice is covered by traditional symbolic execution, the same relevant path slice is also covered by DGSE. Finally, based on Lemma 5.2, we prove that DGSE and traditional symbolic execution are equivalent in terms of reachability of conditional abort statements.

Lemma 5.1. Given a relevant path slice \( \pi_{\text{rps}}[n_m] = [n_1, n_2, \ldots, n_m] \) in the path \( \pi \), for any \( n_i \in \pi_{\text{rps}}[n_m] \) \((1 \leq i \leq m)\), we have \( \text{transitive}D(n_i-1, n_i) = T \).

Proof. We prove the lemma by induction.

1) Base Step: We know that \( \pi_{\text{rps}}[n_m] \) is the relevant path slice in \( \pi \) with respect to \( n_m \). According to the definition of relevant path slice, we have \( \text{transitive}D(n_i, n_i) = T \) \((1 \leq i \leq m)\). Therefore, we have \( \text{transitive}D(n_{m-1}, n_m) = T \).

2) Inductive Step: The inductive hypothesis is that we have \( \text{transitive}D(n_i-1, n_i) = T \) \((2 < i \leq m)\). We prove that we also have \( \text{transitive}D(n_i, n_{i-1}) = T \). Note that \( n_{i-2} \) is in \( \pi_{\text{rps}}[n_m] \) and therefore we have \( \text{transitive}D(n_i, n_m) = T \). Assume that there is a dependence chain from \( n_m \) to \( n_{i-2} \), which is \( \text{transitive}D(n_j, n_m) = T \land \text{transitive}D(n_{i-2}, n_j) = T \). There are two cases to consider: (I) If \( j = i-1 \), it has proved that \( \text{transitive}D(n_{i-2}, n_{i-1}) = T \); (II) If \( j > i-1 \), we have \( \text{transitive}D(n_{j-1}, n_j) = T \) based on the inductive hypothesis. In addition, we also have \( \text{transitive}D(n_i, n_{j-2}) = T \) based on the assumed dependence chain from \( n_m \) to \( n_{i-2} \). Therefore, we have \( \text{transitive}D(n_{i-2}, n_{i-1}) = T \) based on the definition of transitive dependence. As a result, we obtain another dependence chain from \( n_m \) to \( n_{i-2} \), which is \( \text{transitive}D(n_j, n_m) = T \land \text{transitive}D(n_{i-2}, n_j) = T \land \text{transitive}D(n_{i-2}, n_{j-1}) = T \). Repeating such process, we eventually get a dependence chain \( \text{transitive}D(n_j, n_m) = T \land \text{transitive}D(n_{i-1}, n_j) = T \land \ldots \land \text{transitive}D(n_{i-2}, n_{i-1}) = T \). Therefore, we also have \( \text{transitive}D(n_{i-2}, n_{i-1}) = T \).

Lemma 5.2. Given a program \( P \), for any relevant path slice \( \pi_{\text{rps}}[n] \) that lies on a feasible path explored by traditional symbolic execution, DGSE is also able to explore a feasible path in \( P \) that covers \( \pi_{\text{rps}}[n] \).

Proof. We prove the lemma by induction.

1) Base Step: Let \( \pi_{\text{rps}}[n] = [n] \). According to the definition of relevant path slice, \( n \) is not control-dependent on any other node \( n_c \), otherwise \( n_c \) is also in \( \pi_{\text{rps}}[n] \). If \( n \) is not a branch node, \( n \) is reachable in the first path exploration and thus \( \pi_{\text{rps}}[n] = [n] \) is covered by the first path. If \( n \) is a branch node, we assume that \( \pi_{\text{rps}}[n] = [n] = [b_r] \) without loss of generality. If the first path exploration takes branch \( b_r \) at \( n \), \( \pi_{\text{rps}}[n] = [n] = [b_r] \) is covered by the first path. Otherwise, the first path exploration must take the opposite branch \( b_r' \) of \( b_r \). According to Algorithm 2, DGSE negates all the branches in the first path exploration, and therefore DGSE negates \( b_r' \) to \( b_r \) and generates the to-be-explored relevant path condition \( b_r \). Based on \( b_r \), DGSE explores a path to cover \( \pi_{\text{rps}}[n] = [n] = [b_r] \) in Algorithm 2.

2) Inductive Step: The inductive hypothesis is that any relevant path slice \( \pi_{\text{rps}}[n_i] = [n_1, \ldots, n_i] \) with length less than or equal to \( i \) has been covered by the path in DGSE. Assume that \( \pi_{\text{rps}}[n_i] \) is covered by the path \( \pi_i \) and its corresponding relevant path condition is \( \pi_{\text{rps}}[n_i] \). Consider a longer relevant path slice \( \pi_{\text{rps}}[n_{i+1}] = [n_{i+1}] \) with length \( i+1 \). If \( n_{i+1} \) is not control-dependent on any node, based on the same reason in the base step, \( \pi_{\text{rps}}[n_{i+1}] \) can be covered by DGSE. In the following we consider the case that \( n_{i+1} \) is control-dependent on a node \( n_c \in [n_1, \ldots, n_i] \). If \( n_{i+1} \) is not a branch node, \( n_{i+1} \) is reachable in \( \pi_i \) and thus \( \pi_{\text{rps}}[n_{i+1}] \) is also covered by \( \pi_i \). The reason is that the control dependence in DGSE includes the branch information and thus \( n_c \) and \( n_{i+1} \) are simultaneously reachable in \( \pi_i \). If \( n_{i+1} \) is a branch node, we assume \( n_{i+1} = b_{i+1} \) without loss of generality. If \( b_{i+1} \) is taken by \( \pi_i \), the relevant path slice \( \pi_{\text{rps}}[n_{i+1}] \) is covered by \( \pi_i \). Otherwise, the path \( \pi_i \) takes the opposite branch \( b_{i+1}' \) of \( b_{i+1} \). As Lemma 5.1, we know that \( b_{i+1}' \) is transitively dynamically dependent on \( n_i \). Therefore, DGSE will negate \( b_{i+1}' \) to \( b_{i+1} \) in \( \pi_i \) and generate a to-be-explored relevant path condition \( \pi_{\text{rps}}[n_i'] \), which leads to that \( \pi_{\text{rps}}[n_{i+1}] \) will be covered in Algorithm 2.

\[\square\]

Theorem 5.3. Assume a program \( P \) and its potential faults are modeled as conditional abort statements. Any fault that can be detected by traditional symbolic execution can also be detected by DGSE.

Proof. Let a random potential fault be \( \text{if } (\neg\phi) \) abort; and \( n \) represents the if statement. Since traditional symbolic execution offers exhaustive path coverage, it can explore a path \( \pi = [\ldots n T \ldots] \) that leads to abort statement if \( \pi \) is feasible. Assume that the
relevant path slice with respect to \( n^T \) in \( \pi \) is \( \pi^{ps}[n^T] \). According to Lemma 5.2, DGSE would also explore a path to cover \( \pi^{ps}[n^T] \). Therefore, if there exists a feasible path that executes \( n^T \) and leads to abort statement, DGSE also explores a feasible path that executes \( n^T \) and leads to abort statement.

6 Evaluation

We have developed a prototype for DGSE based on two existing tools Indus [50] and Symbolic PathFinder (SPF) [6]. Indus is an inter-procedural analysis tool based on Soot [54]. The control and data dependence can be simply retrieved from Indus. As for the potential and interactive dependence, we computed them using the information flow analysis available in Indus. SPF is a symbolic execution extension to the Java PathFinder framework [6]. We extended SPF with a customized listener to load the results of static program dependence analysis, and used the results to guide symbolic execution in SPF.

We compare DGSE against traditional symbolic execution SPF to answer the following research questions.

1) Compared with SPF, what is the reduction ratio in terms of the number of explored paths? Can such reduction in DGSE lead to speedup even with additional computational overhead?

2) It is often the case that the paths of a program cannot be systematically explored within reasonable amount of time. Does DGSE offer better distinctive path coverage than SPF within a time limit?

3) Search strategies may have an impact on the effectiveness of symbolic execution. How do different search strategies, such as breadth-first, depth-first and random search, affect SPF and DGSE?

4) We have proved that DGSE has the equivalent fault detection capability with traditional symbolic execution as long as the potential faults are modeled as conditional abort statements. What if such modeling is not conducted? Does fewer explored paths in DGSE lead to inferior fault detection capability?

6.1 Experiment Setup

Siemens suite [10] has been widely used for various software engineering tasks, such as software testing and fault localization. The lines of code in the suite range from 173 to 570. Since the programs are written in C language, we have manually translated them to Java language, same as the work [16] [35] [55].

Note that, we omitted two programs Printtokens and Totinfo from the suite in our evaluation. The program Printtokens contains numerous arrays and uses the symbolic variable as the array index, which leads to the imprecision of symbolic execution. The program Totinfo has many double type of variables, which significantly slows the speed of constraint solving and even leads to the timeout of constraint solving. In the paper, we computed the lines of code for these subjects based on their \( C \) versions. In addition, we add a program of similar size called Wheel Brake System (WBS) [15], which has been used as a case study in several papers on symbolic execution. WBS is a synchronous reactive component that is used to provide safe breaking of the aircraft during taxiing, landing, and in the event of a rejected take-off.

Besides above six programs, we choose three larger publicly available programs: Siena, Apache CLI and NanoXML. Siena is an internet-scale event notification middleware for distributed event-based applications. It consists of 94 procedures and 1256 lines of code. In our experiments, we used the procedure encode in the class SENP to serve as the main method as the research [35]. Apache CLI provides an API for parsing command line options. There are 183 procedures and 3612 lines of code in the program. We designed its test driver based on the example shown on its official website [56]. NanoXML is a small XML parser for Java, which has 129 procedures and 4608 lines of code. We used DumpXML distributed with NanoXML as the test driver. Moreover, these three subjects contain the complex data structures such as HashMap, which are not supported by SPF. We have modified them through using the general data structures to equivalently instead. In the paper, the lines of code for these three subjects are computed based on their versions before modification.

All experiments are conducted on a Windows 7 desktop with 3.2 GHz Intel Core i5 CPU and 8 GB memory. The SMT solver Z3 [57] is used for all the subjects except for Siena, in which Z3BitVec [57] is used. The default search strategy in our experiments is depth-first search. Moreover, in theory DGSE is a random process according to the random first input, and there should be an assessment of the statistical power of the empirical results [58]. However, due to the implementation of SPF, DGSE that is implemented based on SPF would also not generate a random first input.

6.2 Number of Explored Paths and Time Usage

This group of experiments empirically study how effective DGSE is in terms of path reduction and whether such path reduction leads to the speedup. Table 5 compares the numbers of paths explored by DGSE and SPF. The first column lists the names of the experimental subjects, followed by their sizes in terms of lines of the code. Column Inputs shows the number of symbolic inputs. Since some subjects have the variable number of symbolic inputs, we conducted two experiments for each: one with the minimum number of symbolic inputs under which the program can accomplish its all functionality, the other with the maximum number of symbolic inputs under which symbolic execution can terminate within 12-hour time limit [59]. The remaining columns in Table 5 list the detailed results between DGSE and SPF. Column Feasible Paths shows the number of
TABLE 5: DGSE and SPF: Number of Explored Paths

<table>
<thead>
<tr>
<th>Subject</th>
<th>LoC</th>
<th>Inputs</th>
<th>DGSE Feasible Paths</th>
<th>SPF Feasible Paths</th>
<th>Ratio (%)</th>
<th>DGSE Infeasible Paths</th>
<th>SPF Infeasible Paths</th>
<th>Ratio (%)</th>
<th>Total Paths</th>
<th>DGSE Total Paths</th>
<th>SPF Total Paths</th>
<th>Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WBS</td>
<td>231</td>
<td>6</td>
<td>324</td>
<td>357</td>
<td>56.25</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>324</td>
<td>357</td>
<td>100</td>
<td>56.25</td>
</tr>
<tr>
<td>Tcas</td>
<td>172</td>
<td>12</td>
<td>344</td>
<td>392</td>
<td>87.76</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>1352</td>
<td>1400</td>
<td>96.52%</td>
<td></td>
</tr>
<tr>
<td>Schedule2</td>
<td>374</td>
<td>3</td>
<td>127</td>
<td>345</td>
<td>37.03</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>127</td>
<td>345</td>
<td>37.03%</td>
<td></td>
</tr>
<tr>
<td>Schedule</td>
<td>412</td>
<td>6</td>
<td>8191</td>
<td>117649</td>
<td>6.96</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8191</td>
<td>117649</td>
<td>6.96%</td>
<td></td>
</tr>
<tr>
<td>Replace</td>
<td>564</td>
<td>5</td>
<td>386</td>
<td>594</td>
<td>64.98</td>
<td>62</td>
<td>278</td>
<td>22.30</td>
<td>448</td>
<td>872</td>
<td>51.38%</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 6: DGSE and SPF: Time Usage

<table>
<thead>
<tr>
<th>Subject</th>
<th>Inputs</th>
<th>Total Time(s)</th>
<th>DGSE</th>
<th>SPF</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Overhead</td>
<td>SE</td>
<td>Total</td>
</tr>
<tr>
<td>WBS</td>
<td>6</td>
<td>1.00</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Tcas</td>
<td>12</td>
<td>1.02</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Schedule2</td>
<td>3</td>
<td>1.83</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Schedule</td>
<td>6</td>
<td>2.87</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Replace</td>
<td>5</td>
<td>1.60</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>PrintTokens2</td>
<td>2</td>
<td>3.96</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Siena</td>
<td>9</td>
<td>4.56</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Apache CLI</td>
<td>9</td>
<td>2.15</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
<tr>
<td>NanoXML</td>
<td>9</td>
<td>-</td>
<td>1.00</td>
<td>2.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>

TABLE 7: Time Usage Breakdown

<table>
<thead>
<tr>
<th>Subject</th>
<th>Inputs</th>
<th>DGSE Solver Calls</th>
<th>SPF Solver Calls</th>
<th>Sat Solver Time(s)</th>
<th>Unsat Solver Time(s)</th>
<th>DGSE Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>WBS</td>
<td>6</td>
<td>3780</td>
<td>6720</td>
<td>1.00</td>
<td>1.4</td>
<td>1.0</td>
</tr>
<tr>
<td>Tcas</td>
<td>12</td>
<td>22490</td>
<td>23598</td>
<td>10.00</td>
<td>10.00</td>
<td>1.0</td>
</tr>
<tr>
<td>Schedule2</td>
<td>3</td>
<td>1126</td>
<td>4312</td>
<td>2.3</td>
<td>5.5</td>
<td>2.3</td>
</tr>
<tr>
<td>Schedule</td>
<td>6</td>
<td>110548</td>
<td>2840383</td>
<td>269</td>
<td>275</td>
<td>1.0</td>
</tr>
<tr>
<td>Replace</td>
<td>5</td>
<td>9460</td>
<td>17491</td>
<td>5.5</td>
<td>5.5</td>
<td>1.0</td>
</tr>
<tr>
<td>PrintTokens2</td>
<td>11</td>
<td>927311</td>
<td>939852</td>
<td>3.99</td>
<td>3.99</td>
<td>1.0</td>
</tr>
<tr>
<td>Siena</td>
<td>9</td>
<td>138631</td>
<td>1386107</td>
<td>&lt;10.00</td>
<td>10.48</td>
<td>201.7</td>
</tr>
<tr>
<td>Apache CLI</td>
<td>9</td>
<td>8063394</td>
<td>15801652</td>
<td>51.03</td>
<td>51.03</td>
<td>1972</td>
</tr>
<tr>
<td>NanoXML</td>
<td>9</td>
<td>12565740</td>
<td>12671295</td>
<td>401</td>
<td>401</td>
<td>10.48</td>
</tr>
</tbody>
</table>

explored paths that are feasible, i.e., the paths that can produce the corresponding test inputs. In contrast, column Infeasible Paths lists the number of infeasible paths that end at an unsatisfiable constraint solving and cannot produce the corresponding test inputs. The last column gives the total number of paths that include both feasible and infeasible paths.

Within 12-hour time limit, SPF and DGSE fail to terminate in three and one subject, respectively. Among the six subjects that both DGSE and SPF terminate, DGSE explores as low as 6.72% of the feasible paths of SPF, and as low as 10.74% of the infeasible paths of SPF. When considering both feasible and infeasible paths, DGSE explores 6.96% to 96.57% of the paths of SPF.

Table 5 confirms that the number of paths explored by DGSE is significantly smaller than that of SPF. However, there is still concern whether the path reduction can compensate the additional overhead in DGSE, which includes program dependence analysis and the guidance of symbolic execution. Table 6 shows our study results on whether DGSE achieves the speedup. We divide the time usage of DGSE into two parts: overhead and symbolic execution. It can be observed that the overhead is not too expensive. Only in WBS it accounts for 50% of total time usage and in fact it is due to the less total time usage. Even with the overhead, DGSE still can achieve speedup from 1.02X to 49.56X in our experiments.

The study results also show that the more paths to be explored, the more significant the speedup is. This is more obvious in the experiments on the same
void Test(int x) {
    1: int a=1;  
    2: int b=2;  
    3: if (x>1)  
    4:     a=2;  
    5: if (x>0)  
    6:     b=x;  
    7: int first_out=a;  
    8: int sec_out=b;  
}

Fig. 5: An Example for Infeasible Path Reduction

subject with different numbers of symbolic inputs. The speedups in the experiments with minimum number of symbolic inputs are 1.83X, 1.83X, 1.6X and 4.6X, while in those with maximum number of symbolic inputs they become 25.87X, 23.91X, 30.9X and 49.56X.

Another interesting result is that the speedup seems more significant than the path reduction. Table 5 shows DGSE explores 6.96% to 96.57% of the paths of SPF, that is, SPF explores 1.04X to 14.36X of the paths of DGSE. However, Table 6 shows DGSE achieves a speedup from 1.02X to 49.56X. Considering the existence of the overhead in DGSE, it is very surprising. In fact, for all the large subjects, the speedup is more significant than the path reduction. Obviously, the number of explored paths alone cannot explain this interesting result.

In order to investigate this issue, we examine how the time is spent on constraint solving, the most expensive procedure of symbolic execution. Table 7 gives the results on the number of constraint solver calls and the time spent on the constraint solver. The latter is further divided into the constraint solving time for the feasible paths (column Sat Time) and the infeasible paths (column Unsat Time). The results show that although DGSE makes fewer calls to the constraint solver, it achieves more significant speedup. Consider PrintTokens2 with three symbolic inputs. Although DGSE only makes 13.79% of constraint solver calls of SPF, that is, SPF makes 7.25X constraint solver calls over DGSE, SPF spends 55.4X time over DGSE. The first reason that explains the speedup is that DGSE not only conducts fewer constraint solver calls but also produces the simpler relevant path condition. The average number of constraints per constraint solver call is 19.48 in DGSE while 24.00 in SPF. Although not guaranteed, in general the constraint solving is easier with smaller number of constraints. In addition, DGSE explores dramatically smaller number of infeasible paths. As is well known, solving an unsatisfiable formula usually takes longer time as it requires exhaustive traversal of the state space, thus the speedup gained from fewer unsatisfiable constraint solver calls can be significant. To understand why DGSE explores fewer infeasible paths, consider the example program in Figure 5.

Assume that the first path is π₁ = [3ᵀ, 5ᵀ]. At the end of its exploration, traditional symbolic execution generates two to-be-explored tasks (3ᵀ, 3ᵀ) and (3ᵀ ∧ 5ᵀ, 5ᵀ). As illustrated in Table 8, the second to-be-explored task (3ᵀ ∧ 5ᵀ, 5ᵀ) leads to an infeasible path π₂ because the path condition 3ᵀ ∧ 5ᵀ, which corresponds to (x > 1) ∧ (x ≤ 0), is unsatisfiable. In contrast, DGSE generates two to-be-explored tasks (3ᵀ, 3ᵀ) and (5ᵀ, 5ᵀ) after the exploration of π₁, as shown in Table 9. The second to-be-explored task (5ᵀ, 5ᵀ) makes DGSE skip the infeasible path π₂ and execute the feasible path π₄ instead. As a result, the infeasible path π₂ is avoided in DGSE.

### 6.3 Distinctive Path Coverage

Exhaustive path exploration is often not achievable within reasonable amount of time. In this case we wish symbolic execution can cover as many paths as possible. In this group of experiments, we compare the distinctive path exploration between SPF and DGSE within 12-hour time limit. We do not consider the redundant paths as they do not contribute to the unique fault detection.

The subjects used in this group of experiments are Siena, Apache CLI and NanoXML that have been used as experimental subjects in several papers on symbolic execution [22][35][60]. Siena has nine symbolic inputs. The number of symbolic inputs for Apache CLI and NanoXML are variable. We also fix nine symbolic inputs for these two programs in our experiments.

Figure 6 shows the experimental results of distinctive path coverage for Siena, Apache CLI and NanoXML. The blue solid line and red dashed line give the distinctive path coverage of SPF and DGSE over time, respectively. Note that SPF does not complete the full path exploration in all three subjects within 12-hour time limit, while DGSE only does not terminate in NanoXML. In the figure, we can see that DGSE consistently performs better than SPF. As explained in Section 6.2, in general SPF averages spends longer time to explore a path because its path condition is more complicated. A careful study of Figure 6 reveals another more important reason for the lower distinctive path coverage in SPF.
Some flat lines in blue solid show SPF spends significant amount of time to explore the redundant paths, which are avoided in DGSE. Consider the program Apache CLI that provides an API for parsing the command line options. Since most of the options are independent of each other, Apache CLI can be divided into numerous independent parsing modules. In this case, DGSE would explore the paths for each parsing module in isolation, while SPF still combines all parsing modules together to explore the paths. As a result, DGSE reduces the number of explored paths from $O(2^n)$ to $O(2^{m_1} + ... + 2^{m_n})$, where $n$ is the accumulated number of branch conditions, $n_i$ is the number of branch conditions in the $i$-th parsing module, $m$ is the number of parsing modules and we have $n = \sum_{i=1}^{m} n_i$.

6.4 Different Search Strategies

Search strategies may have impact on the effectiveness of symbolic execution technique. For example, postconditioned symbolic execution [61] offers the most benefit under the depth-first search and no benefit at all under the breadth-first search. Although so far we have only discussed the depth-first search, symbolic execution can be easily adapted with different search strategies. In this section, we evaluate the impact of breadth-first, depth-first and random search on SPF and DGSE. For random search, we conduct the experiments five times and then compute the arithmetic mean of the results.

Figure 7 depicts the distinctive path coverage for SPF with three search strategies. The breadth-first, depth-first and random search are depicted in blue solid line, red dashed line and black dotted line, respectively. It can be observed that the search strategies indeed have impact on the distinctive path coverage, although there is no clear winner. That is, in general we cannot know in advance which search strategy works best in a given program. The main reason is that SPF cannot predict when the redundant paths appear. Figure 8 shows three search strategies for DGSE. It can be observed that DGSE is less sensitive to the search strategies. In particular, for the subjects Siena and Apache CLI, there is almost no difference among three search strategies. This can be explained by that DGSE strives to explore the distinctive paths and thus the primary influence is the average time usage for each path exploration. This is clearly shown in Figure 8 (c), which indicates breadth-first search works best while depth-first search works worst. We know that breadth-first search takes the priority to negate the first to-be-negated branch, while depth-first search prefers to negate the last to-be-negated branch. As a result, the average number of constraints per constraint solving is smaller in breadth-first search than that in depth-first search, which further leads to less average time usage for each path exploration in breadth-first search than that in depth-first search.

6.5 Fault Detection Capability

DGSE has the equivalent fault detection capability as traditional symbolic execution if the potential faults are correctly modeled. Consider an assignment statement $x\leftarrow a/b$. After converting the statement to $\text{if } (b==0) \text{ abort; else } x\leftarrow a/b$, DGSE and SPF have the equivalent capability to detect the division-by-zero error. In the case that $x\leftarrow a/b$ is not modeled, what is the capability to detect the error at $x\leftarrow a/b$ for the test suites generated by SPF and DGSE?

In this section, we exploit mutation testing [62] [63] to evaluate the effectiveness of the test suites generated by SPF and DGSE. Mutation testing makes small changes to a program and generates a set of faulty versions called mutants. We say that the test suite kills a mutant if one test input in the suite can give different outputs between the original program and its mutated version. For six subjects that both SPF and DGSE can conduct full path exploration, we let SPF and DGSE thoroughly explore the program paths and generate the corresponding test suite $I_{SPF}$ and $I_{DGSE}$. The kill rates of SPF and DGSE are defined by the percentage of mutants killed by $I_{SPF}$.
and $I_{DGSE}$, and then the fault detection capability can be measured by their kill rates. In our experiments, we utilize the tool MuJava [62] to automatically generate the mutants for our benchmarks. We applied all 15 method-level and all 28 class-level mutation operators available in MuJava. Since not all mutation operators are able to produce mutants, the number of mutants generated by a mutation operator can be different.

The experimental results are given in Table 10. Column Mutation lists the number of mutations, ranging from 615 to 3430. Column Test Cases gives the number of test cases generated by DGSE and SPF, which corresponds to column Feasible Paths in Table 5. Column Kill shows the number of mutants killed by SPF and DGSE. It can be observed that the numbers of killed mutants from SPF and DGSE are very similar. Column Unique Kill shows the number of mutants that are killed only by DGSE or SPF. Note that even though the paths explored by DGSE are the subset of the paths explored by SPF, the test cases generated by DGSE may not be the subset of test cases generated by SPF, because symbolic execution only generates a representative test case for each explored path. Therefore, it is possible that a mutant is killed by DGSE but not by SPF. Finally, column Kill Rate gives the percentage of mutants that are killed. Although the number of test cases generated by DGSE are only 37.03% to 87.76% of those of SPF, the maximal difference in kill rates between DGSE and SPF is less than 0.2%. This group of experiments indicate that the redundant paths rarely detect the unique faults.

Table 10 shows that the kill rates among different programs vary greatly, ranging from about 47% for WBS to about 70% for Schedule2. There are two possible explanations. The first is due to the inherent nature of a program. For a test case to kill a mutant, its execution must meet the PIE model [64]: the execution must not only be able to reach the mutated statements, but also produce different program states and propagate the differences to a manifested location. The second reason is due to the influence of different mutation operators. As discussed in [62] [65], mutation operators can be classified into eight categories: AO, RO, CO, SO, LO, AS, DL, OO, which denote the Arithmetic mutation Operator, the Relational mutation Operator, the Conditional mutation Operator, the Assignment mutation Operator, the Logical mutation Operator, the Arithmetic Symbolic mutation Operator, the String mutation Operator, and the Object-Oriented mutation Operator, respectively.
TABLE 10: Kill Rates of Mutation Testing

<table>
<thead>
<tr>
<th>Program</th>
<th>Inputs</th>
<th>Mutation</th>
<th>Test Cases</th>
<th>Kill</th>
<th>Unique Kill</th>
<th>Kill Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>WBS</td>
<td>6</td>
<td>836</td>
<td>DGSE SPF</td>
<td>392</td>
<td>393</td>
<td>0 1</td>
</tr>
<tr>
<td>DGSE</td>
<td>0</td>
<td>775</td>
<td>DGSE SPF</td>
<td>537</td>
<td>537</td>
<td>1 2</td>
</tr>
<tr>
<td>Tcas</td>
<td>3</td>
<td>1084</td>
<td>DGSE SPF</td>
<td>760</td>
<td>761</td>
<td>0 1</td>
</tr>
<tr>
<td>Schedule2</td>
<td>3</td>
<td>295</td>
<td>DGSE SPF</td>
<td>1108</td>
<td>1108</td>
<td>0 0</td>
</tr>
<tr>
<td>Replace</td>
<td>5</td>
<td>3430</td>
<td>DGSE SPF</td>
<td>1896</td>
<td>1897</td>
<td>0 1</td>
</tr>
</tbody>
</table>

TABLE 11: Kill Rates for Different Types of Mutation Operators

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>AO</th>
<th>RO</th>
<th>CO</th>
<th>LO</th>
<th>DL</th>
<th>OO</th>
<th>Avg. Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGSE</td>
<td>40.5%</td>
<td>49.0%</td>
<td>68.9%</td>
<td>60.7%</td>
<td>44.6%</td>
<td>0.0%</td>
<td>52.5%</td>
</tr>
<tr>
<td>SPF</td>
<td>43.7%</td>
<td>54.5%</td>
<td>84.6%</td>
<td>54.0%</td>
<td>44.6%</td>
<td>0.0%</td>
<td>52.4%</td>
</tr>
<tr>
<td>DGSE</td>
<td>61.0%</td>
<td>75.8%</td>
<td>88.9%</td>
<td>80.8%</td>
<td>68.6%</td>
<td>20.0%</td>
<td>77.8%</td>
</tr>
<tr>
<td>SPF</td>
<td>61.3%</td>
<td>75.8%</td>
<td>88.9%</td>
<td>80.8%</td>
<td>68.6%</td>
<td>20.0%</td>
<td>77.8%</td>
</tr>
<tr>
<td>DGSE</td>
<td>61.4%</td>
<td>82.2%</td>
<td>84.1%</td>
<td>83.6%</td>
<td>70.8%</td>
<td>64.1%</td>
<td>67.8%</td>
</tr>
<tr>
<td>SPF</td>
<td>61.6%</td>
<td>82.2%</td>
<td>84.1%</td>
<td>83.6%</td>
<td>70.8%</td>
<td>64.1%</td>
<td>67.8%</td>
</tr>
<tr>
<td>DGSE</td>
<td>50.2%</td>
<td>65.1%</td>
<td>72.7%</td>
<td>65.4%</td>
<td>74.4%</td>
<td>62.2%</td>
<td>68.7%</td>
</tr>
<tr>
<td>SPF</td>
<td>50.2%</td>
<td>65.1%</td>
<td>72.7%</td>
<td>65.4%</td>
<td>74.4%</td>
<td>62.2%</td>
<td>68.7%</td>
</tr>
<tr>
<td>DGSE</td>
<td>57.1%</td>
<td>65.1%</td>
<td>72.7%</td>
<td>65.4%</td>
<td>74.4%</td>
<td>62.2%</td>
<td>68.7%</td>
</tr>
<tr>
<td>SPF</td>
<td>57.2%</td>
<td>65.1%</td>
<td>72.7%</td>
<td>65.4%</td>
<td>74.4%</td>
<td>62.2%</td>
<td>68.7%</td>
</tr>
</tbody>
</table>

Due to the impricism of the static program analysis on the issues such as aliasing, we conservatively compute the program dependencies. This leads to an overapproximation of the program dependencies. As a result, a redundant path may be wrongly considered distinctive and thus is explored unnecessarily. On the other hand, the over-approximation guarantees that DGSE is sound, which means DGSE would not miss any distinctive path if the resource is enough.

The effectiveness of DGSE depends on the inherent dependence relationship of a program. A tightly coupled program with dense dependence relationship will harvest little benefit from DGSE. However, as confirmed by our experiments, redundancy is usually both abundant and widespread in a program. In the best case, DGSE can provide exponential improvement for the loosely coupled program.

Same as all other symbolic execution techniques, DGSE relies on the soundness of the constraint solver. Although SAT/SMT solving has seen tremendous progress in the last two decades, they may still report UNKNOWN besides two desirable SAT and UNSAT answers, due to either unsolvable theories such as nonlinear logic or resource limit. As a result, DGSE, same as SPF, may miss some paths that should be explored.

8 Application of DGSE: Regression Testing

DGSE leverages static dependence analysis and symbolic execution in synergy to enable efficient path exploration. In fact, the static dependence analysis of DGSE can be adapted so that it can be applied to other software engineering tasks. For example, we can only statically analyse those program dependencies that are relevant to the program changes. If so, DGSE can be applied to regression testing. In this section, we illustrate that DGSE can be adapted to implement a regression testing technique, which is similar to DiSE [15]. Note that our objective is to demonstrate the application of DGSE rather than its effectiveness in this particular application.

7 Threats to Validity

The main internal threat to the validity is the potential presence of errors in our implementation. To minimize this threat, we tested our implementation on various programs and manually checking the results. Another source of threat to validity comes from the experimental subject selection in the evaluation. Studies on more experimental subjects can help better assess the efficiency and effectiveness of DGSE.
public class WBS{
  1:  int AltPress=0;
  2:  int Meter=2;
  3:  public void update(int PedalPos, int BSwith, int PedalCmd){
  4:    if(PedalPos<=0) // if(PedalPos==0)
  5:      PedalCmd=PedalCmd+1;
  6:    else if(PedalPos==1)
  7:      PedalCmd=PedalCmd+2;
  8:    else
  9:      PedalCmd=PedalPos;
 10:    PedalCmd=PedalCmd+1;
 11:    if(BSwith==0)
 12:      Meter=1;
 13:    else if(BSwith==1)
 14:      Meter=2;
 15:    if(PedalCmd==2)
 16:      AltPress=0
 17:    else if(PedalCmd==3)
 18:      AltPress=1
 19:    else
 20:      AltPress=2;
}

Fig. 9: The Example Program for DiSE

DiSE is an efficient symbolic execution technique, which only explores the paths that are affected by the program changes. Therefore, the test cases generated by DiSE can be used for regression testing. DiSE performs a two-phase analysis. The first phase conducts static analysis to compute a set of locations that are impacted by the program changes. The set of locations are then utilized to guide symbolic execution in the second phase. Figure 9 shows the motivating example in DiSE [15], which has totally 24 feasible program paths. If Line 3 is the only statement that has been changed, DiSE only explores eight feasible paths that are impacted by Line 3. The eight feasible paths are formed by the combination of the yellow nodes shown in Figure 10.

We use the same example program to demonstrate that DGSE is able to achieve the same optimization as DiSE [15]. First, we revise the static dependence analysis to only compute those program dependencies that are relevant to the program changes. The relevant program dependencies are then used to guide symbolic execution as described in Section 5.1.

Table 12 shows the program dependencies that are relevant to the changed Line 3 of Figure 9. With the guidance of the table, we see that the program dependencies that are irrelevant to the changed Line 3 are ignored, such as (10^7, 11) and (10^5, 13). Table 13 illustrates the path exploration of dependence guided symbolic execution. Same as DiSE, DGSE also only explores eight feasible program paths.

Although in this example DGSE and DiSE enjoy the same improvement, in general DGSE is more efficient because DiSE does not consider the dependence between the affected locations. For example, if a change happens to Line 2 of the program in Figure 2, DiSE considers all statements except Line 1 are affected and thus explores all eight paths as Table 1. However, DGSE can still determine that the affected Lines 4 and 8 are not dependent and there is no need to consider all the combination of the branches of Lines 4 and 8. Consequently, DGSE only explores six paths. The path exploration of such situation in DGSE is the same as Table 3.

9 CONCLUSION AND FUTURE WORK

In this paper, we propose an optimization to traditional symbolic execution based on symbolic value. We also present a practical implementation called Dependence Guided Symbolic Execution (DGSE) to soundly approximate such optimization. DGSE utilizes static analysis to compute the program dependencies, which then guide symbolic execution to selectively explore program paths. We implemented DGSE based on Symbolic PathFinder and conducted experiments to evaluate the effectiveness of DGSE. The experimental results show that, compared with traditional symbolic execution, DGSE can significantly improve its performance without sacrificing fault detection capability. Moreover, we demonstrate that DGSE can be applied to regression testing. In the future work, we will apply DGSE to more larger subjects, especially those subjects that contain the complex data structures. Moreover, at present we utilize the pairs of nodes to compute the program dependencies. We will further study whether n length tuples are valuable for computing the program dependencies.
TABLE 13: Path Exploration of DGSE for Regression Testing

<table>
<thead>
<tr>
<th>From</th>
<th>Path</th>
<th>pcon</th>
<th>$\psi$</th>
<th>transitiveD($\psi_d$, $\neg \psi_s$)</th>
<th>To-be-Explored Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>($true, true$)</td>
<td>[3F, 10F, 14T]</td>
<td>3F $\land$ 10F $\land$ 14T</td>
<td>true</td>
<td>–</td>
<td>($3F \land 14F, 16F$)</td>
</tr>
<tr>
<td>($3F \land 14F, 16F$)</td>
<td>[3F, 10F, 14F, 16T]</td>
<td>3F $\land$ 10F $\land$ 14F $\land$ 16T</td>
<td>14F</td>
<td>transitiveD(14F, 16F) = T</td>
<td>($3F \land 14F, 16F, 16F$)</td>
</tr>
<tr>
<td>($3F \land 14F, 16F$)</td>
<td>[3F, 10F, 14F, 16T]</td>
<td>3F $\land$ 10F $\land$ 14F $\land$ 16T</td>
<td>16F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>($3F, 3F$)</td>
<td>[3F, 5F, 10F, 14T]</td>
<td>3F $\land$ 5F $\land$ 10F $\land$ 14T</td>
<td>3F</td>
<td>transitiveD(3F, 5F) = T</td>
<td>($3F \land 5F, 16F$)</td>
</tr>
<tr>
<td>($3F \land 5F, 16F$)</td>
<td>[3F, 5F, 10F, 14F, 16T]</td>
<td>3F $\land$ 5F $\land$ 10F $\land$ 14F $\land$ 16T</td>
<td>14F</td>
<td>transitiveD(14F, 16F) = T</td>
<td>($3F \land 5F, 16F$)</td>
</tr>
<tr>
<td>($3F \land 5F, 16F$)</td>
<td>[3F, 5F, 10F, 14F, 16T]</td>
<td>3F $\land$ 5F $\land$ 10F $\land$ 14F $\land$ 16T</td>
<td>16F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>($3F \land 5F, 16F$)</td>
<td>[3F, 5F, 10F, 14F, 16T]</td>
<td>3F $\land$ 5F $\land$ 10F $\land$ 14F $\land$ 16T</td>
<td>16F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>($3F \land 5F, 16F$)</td>
<td>[3F, 5F, 10F, 14F, 16T]</td>
<td>3F $\land$ 5F $\land$ 10F $\land$ 14F $\land$ 16T</td>
<td>16F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Infeasible</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

ACKNOWLEDGEMENT

This work was supported by the National Natural Science Foundation of China (91218301, U1301254, 91418205, 61472318, 61428206, 61532015), Fok Ying-Tong Education Foundation (151067), Key Project of the National Research Program of China (2013BAK09B01), Ministry of Education Innovation Research Team (IRT13035), the Fundamental Research Funds for the Central Universities and the National Science Foundation (NSF) under grant CCF-1500365. Ting Liu and Zijiang Yang are the corresponding authors.

REFERENCES


Haijun Wang received the B.S. degree in computer science and technology from Xi’an Jiaotong University, China, in 2007. He is currently working toward the Ph.D. degree in system engineering from School of Electronic and Information, Xi’an Jiaotong University, China. His research interests include program analysis, regression testing, fault localization and software security.

Ting Liu received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xi’an Jiaotong University, Xi’an, China, in 2003 and 2010, respectively. Currently, he is an associate professor of the Systems Engineering Institute, Xi’an Jiaotong University. His research interests include Smart Grid, network security and trustworthy software.

Xiaohong Guan received the B.S. and M.S. degrees in automatic control from Tsinghua University, Beijing, China, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from the University of Connecticut, Storrs, in 1993. He is currently a Cheung Kong Professor of Systems Engineering and the Dean of School of Electronic and Information Engineering in Xi’an Jiaotong University. His research interests include allocation and scheduling of complex networked resources, network security, and software engineering. He is a fellow of IEEE.

Chao Shen received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xi’an Jiaotong University, China, in 2007 and 2014, respectively. Currently, he is an associate professor of the Systems Engineering Institute, Xi’an Jiaotong University of China. His research interests include insider/intrusion detection, behaviour biometric, and measurement and experimental methodology.

Qinghua Zheng received the B.S. degree in computer science in 1990, the M.S. degree in computer organization and architecture in 1993, and the Ph.D. degree in system engineering in 1997 from Xi’an Jiaotong University, China. He was a post-doctoral researcher at Harvard University in 2002. He is currently a professor in Xi’an Jiaotong University, and the dean of the Department of Computer Science. His research areas include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software. He is a member of IEEE.

Zijiang Yang is an associate professor in computer science at Western Michigan University. He holds a Ph.D. from the University of Pennsylvania, an M.S. from Rice University and a B.S. from the University of Science and Technology of China. Before joining WMU he was an associate research staff member at NEC Labs America. He was also a visiting professor at the University of Michigan from 2009 to 2013. His research interests are in the area of software engineering with the primary focus on the testing, debugging and verification of software systems. He is a senior member of IEEE.