Speculative Symbolic Execution

Yufeng Zhang, Zhenbang Chen, Ji Wang
National Laboratory for Parallel and Distributed Processing
Department of Computing Science, National University of Defense Technology
Changsha, China
Email: {yufengzhang, zbchen}@nudt.edu.cn, jiwang@ios.ac.cn

***Abstract*—Symbolic execution is an effective path oriented and constraint based program analysis technique. Recently, there is a significant development in the research and application of symbolic execution. However, symbolic execution still suffers from the scalability problem in practice, especially when applied to large-scale or very complex programs. In this paper, we propose a new fashion of symbolic execution, named *Speculative Symbolic Execution* (SSE), to speed up symbolic execution by reducing the invocation times of constraint solver. In SSE, when encountering a branch statement, the search procedure may speculatively explore the branch without regard to the feasibility. Constraint solver is invoked only when the speculated branches are accumulated to a specified number. In addition, we present a key optimization technique that enhances SSE greatly. We have implemented SSE and the optimization technique on Symbolic Pathfinder (SPF). Experimental results on six programs show that, our method can reduce the invocation times of constraint solver by 20.7% to 48.7% (with an average of 29.9%), and save the search time from 23.6% to 43.6% (with an average of 30%).**

***Keywords*-symbolic execution; speculative symbolic execution; constraint solving; Java PathFinder;**

I. Introduction

Symbolic execution (SE) is a basic program analysis technique that was proposed more than thirty years ago [1]. Recently, SE draws renewed interests both from academia and industry partly due to the impressive progress in constraint solving, related algorithms and computation power [2][3][4]. Instead of executing programs with concrete inputs, symbolic execution feeds programs with symbolic ones, meaning that a symbolic input could initially take any value of the specific type. Assignment statements are interpreted as the manipulations of symbolic expressions. When encountering a branch statement, the process forks and both of the branches are taken. On each path, the process maintains a set of constraints called *path condition* which must hold along that path. For each branch, the path condition is updated according to the corresponding condition and submitted to a constraint solver to check the satisfiability. In the context of test generation, when a path ends or a bug is found, the path condition can be solved to get a test case. For deterministic programs, the same execution path or the same bug can be replayed by feeding such test case as input. Basically, symbolic execution attempts to achieve automatic code comprehension by walking through the path space of a program. Providing that all the path conditions can be solved successfully, symbolic execution could cover all the behaviors of the program.

In the past years, symbolic execution has shown a great promise in the application to automated test generation, proving program properties, bug detection and so on [3]. However, in practice, the scalability problem is still one of the main obstacles in applying symbolic execution to large-scale programs. This issue mainly stems from two closely related reasons: path explosion phenomenon and constraint solving overhead. There exists an exponential relationship between the number of conditions and the paths of the program, making exploring the whole path space infeasible for large-scale programs. Constraint solving is the most dominant in the running time of SE. When exploring deep paths, the path condition may be very complex, and even unsolvable in an acceptable time. In addition, constraint solving overhead is almost always aggravated by the path explosion phenomenon.

To alleviate the constraint solving overhead of SE, many techniques have been proposed. In many symbolic execution systems, query optimization techniques are employed to reduce the complexity of queries and query times. For example, *counterexample caching* stores unsatisfiable path conditions as counterexamples to reuse previous solving results [5]. *Constraint independence* splits a constraint set into independent ones, aiming to get the related constraint set and increase the cache hit rate [5][6][7][8]. *Concretization* reduces complex constraints (such as nonlinear constraints [9]) into simpler ones, and is heavily used in concolic execution [7][8][10].

Although these effective techniques improve the performance of symbolic execution greatly in practice, constraint solving is still the most dominant in symbolic execution. According to the experiments of KLEE [5], $40\% \sim 90\%$ of the whole running time is spent on constraint solving. In the experiments of Cloud9 [11], constraint solving consumes more than half of the total execution time. In some experiments in S^2E [12], almost all the running time is dominated by the constraint solving.

This paper proposes a new fashion of symbolic execution, named *Speculative Symbolic Execution* (SSE), which speeds up symbolic execution by reducing the invocation times

of constraint solver, and hence improves the scalability of symbolic execution. Unlike pure symbolic execution, which invokes the constraint solver immediately when a path condition is updated, in SSE, when a branch instruction is encountered, the path condition is updated accordingly, but the constraint solver is not necessarily invoked. The search procedure may advance along the path without the determination of feasibility until the unsolved path conditions are accumulated to a specified number. If the current visiting path is feasible, the procedure continues; otherwise, it backtracks.

Intuitively, SSE takes branches optimistically as feasible ones. Path conditions are submitted to constraint solver in batches, not one by one as in pure symbolic execution. When speculation succeeds, multiple invocations of constraint solvers are replaced by one invocation. When speculation fails, a backtracking mechanism will find the first bad branch that makes the speculation fail. Basically, the more feasible branches in the path space, the better SSE performs.

In this paper, we give out the details of SSE algorithm and discuss its effectiveness. We also propose an optimization technique, named *Absurdity Based Optimization*, which is simple but very effective in practice. For programs with a high ratio of infeasible branches in the path space, this optimization can reduce the times of invoking constraint solver significantly. To some extent, our optimization is complementary to SSE, and can also be applied to pure symbolic execution.

The contribution of this paper is three-fold.

Firstly, we propose speculative symbolic execution, a new fashion of symbolic execution, to extend the scalability of classical symbolic execution by reducing the invocation times of constraint solver.

Secondly, we propose the absurdity based optimization technique to improve the scalability of symbolic execution. This optimization technique can be used both to pure and speculative symbolic execution.

Finally, we have implemented SSE and the optimization on top of Symbolic Pathfinder [13] to extend the scalability of this symbolic execution system. We have conducted several experiments and find a new characteristic of the path spaces of programs. The experimental results show that our approach can save the search time from 23.6% to 43.6% (with an average of 30%). Based on these results, we also investigate how to make our approach work best when applied to real world programs.

The remainder of this paper is organized as follows. Section 2 introduces the background and shows the basic idea of SSE by motivating examples. Section 3 elaborates the algorithm of SSE and the absurdity based optimization technique. Section 4 presents our implementation on SPF and reports the experimental results. Finally, Sections 5 and 6 discuss the related work and conclude, respectively.

II. Overview

In this section, we describe how SSE works and why SSE is better than pure SE by motivating examples.

A. Background: Symbolic Execution

Essentially, symbolic execution feeds programs with symbolic values as inputs and outputs the result as functions of symbolic values. A search procedure is employed to systematically traverse the path space of a program by maintaining symbolic program states. A symbolic state includes the symbolic values of program variables, a path condition and a program counter [1]. The path condition is a boolean formula that contains the constraints which the inputs should satisfy if they drive the program along the current path. Operations of variables are interpreted as the manipulations of symbolic expressions. When encountering a branch instruction, both of the branches are taken. For each branch, the corresponding condition is added into the path condition and a constraint solver is invoked to check the satisfiability of the new path condition. The process advances along feasible branches until the path ending is reached. Finally, the generated symbolic states form a *symbolic execution tree*.

Take the program in Figure 1 for example. It computes the sum of the absolute values of two integers and outputs the result if the sum is greater than 2. Initially, the inputs are represented as two symbols: X and Y, and the path condition is $\langle true \rangle$. Execution path forks when meeting the branch statement `if(x<0)`. The constraints $\langle X \geq 0 \rangle$ and $\langle X < 0 \rangle$ are added to the path conditions of the two paths respectively. A constraint solver is invoked to check the feasibility of these two paths, both of which here are feasible. Figure 1 shows the left part of the final execution tree, in which symbolic states are represented as nodes.

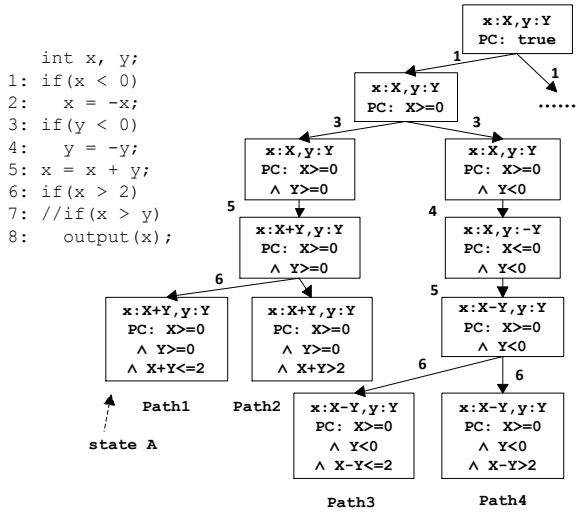

Figure 1. An Example Program and Its Execution Tree

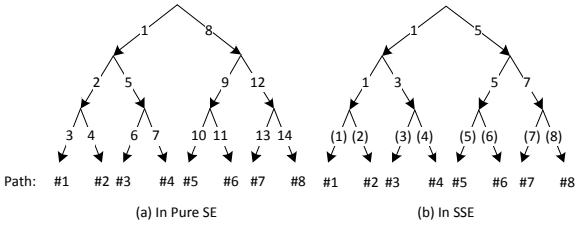

Figure 2. Constraint Solving in DFS

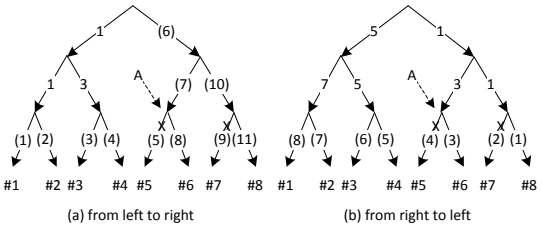

Figure 3. Constraint Solving in SSE by DFS With Backtracking

In this paper, we focus on how the constraint solver is invoked during symbolic execution. We choose the commonly used *depth first search* (DFS) in our illustration.

Figure 2(a) shows the path space of the example program with the same layout in Figure 1. The left side of a node corresponds to the *false* side of the branch statement. The number n marked on a branch means that the feasibility of the branch is determined in the *n*-th invocation of the solver. Totally, 14 times of constraint solving are needed.

B. Motivating Examples of SSE

When encountering a branch statement, SSE may advance along the two branches without checking the feasibility. The constraint solver is invoked only when the number of unchecked branches reaches a specific number, say *max speculation depth*. If the constraint solver gives a positive result, it means that the speculation succeeds. Otherwise, we need backtrack to the last feasible branch. Now we present how speculation reduces solving times in a DFS manner with the example in Figure 1.

The initial symbolic state of the program under SSE is the same as that under SE. Assuming that the max speculative depth is set as 3, for branch statement `if(x<0)`, the procedure advances along the `else` side without checking feasibility. Branches of the statement `if(y<0)` are handled similarly. When the procedure takes the `else` branch of statement `if(x>2)` speculatively, the max speculation depth is reached, therefore a constraint solver is invoked. Since the path segment from root to *state A* (in Figure 1) is executed speculatively, we call this segment a *speculation segment*. As a result, only one time of constraint solving is enough to know the feasibility of the three branches on path #1. As shown in Figure 2(b), the number n associated on a branch demonstrates the feasibility of the branch is known in the *n*-th solving. The invocations of constraint solver only occur at the branches marked with bracketed numbers. In all, only 8 queries are needed, saving nearly half of that in pure SE.

Now consider commenting line 6 and uncommenting line 7 in the example program in Figure 1. Path #5 and #7 would be infeasible. In Figure 3, they are marked with a cross. In this case, the number of constraint solving under pure SE is still 14. In SSE, as shown in Figure 3(a), the fifth invocation to the solver with path condition $\langle X < 0 \wedge Y \geq 0 \wedge (-X + Y) \leq Y \rangle$ results in *unsat*. In the sequel, the backtracking mechanism analyzes the current speculation segment (*i.e.*, from root to *point A*) using binary search to find the first infeasible branch, which results in two extra calls (6th and 7th) to the constraint solver. Then the procedure backtracks to *point A* and continues on path #6. Constraint solving on path #7 is similar to that on path #5. Finally, the constraint solver is invoked for 11 times (with 9 *sat* and 2 *unsat* results), saving 3 invocations in comparison with the 14 invocations in pure SE.

It is worth noting that constraint solving in SSE is related to the order in which the path space is explored. Consider exploring the path space from right to left, *i.e.*, exploring the true side of a branch statement first. As shown in Figure 3(b), it only needs 8 invocations to the constraint solver.

III. Speculative Symbolic Execution

One can imagine using different search styles in SSE. In this section, we present the *speculative DFS* algorithm that combines speculation and DFS, and the absurdity based optimization. Then we discuss the effectiveness of our approach.

A. Speculative DFS Algorithm

Figure 4 shows the algorithm of speculative DFS, including the main `search` procedure and the `backtrack` procedure. The algorithm traverses the path space of program by DFS and performs speculation with a specially designated backtracking mechanism. A `StateStack` is maintained to store the symbolic states on the current path. Initially, the initial symbolic state of the program is pushed into the `StateStack`. The `while` loop expands the top element of the `StateStack` until the stack is empty. The procedure proceeds by symbolically executing the next statement of the top state in the `StateStack` repeatedly. For a non-branch statement, our algorithm performs identically with pure SE. When processing a branch statement, if the current speculation depth has not reached the `maxSpeculationDepth`, the branch is taken without checking feasibility and a new state with updated path condition is pushed into the `StateStack` directly as shown in line 10. Otherwise, as shown in line 12, function `checkFeasibility()` checks the satisfiability of the current path condition. If the result is *sat*, the current state is pushed into the `StateStack` and a new speculation segment starts. If the result is *unsat*, the `backtrack()` procedure cuts the infeasible branches away. The procedure backtracks when reaching the end of a path. According to the feasibility of the last speculation segment on the path, the `backtrack()` procedure performs

```
 1: search(int maxSpeculationDepth) {
 2:   StateStack = {initial state};
 3:   while(StateStack not empty) {
 4:     s=get next statement;
 5:     if(s is non-branch statement)
 6:       perform as pure symbolic execution;
 7:     else {
 8:       choose one unexplored branch;
 9:       if(not reach maxSpeculationDepth){
10:         pushState();
11:       } else {
12:         checkFeasibility();
13:         if(feasible) {//speculation succeeds
14:           pushState();
15:           start new speculation segment;
16:         } else { // speculation fails
17:           backtrack();
18:         }
19:       }
20:     }
21:     if(path ends)  {// path end
22:       if(in speculation segment)
23:         checkFeasibility();
24:       backtrack();
25:     }
26:   }
27: }
28: backtrack() {
29:   if(speculation fails) {
30:     binarySearchFirstBadBranch();
31:     pop unreachable states;
32:     backtrack to the last feasible branch;
33:   } else {
34:     backtrack to the last unexplored branch;
35:   }
36: }
```

Figure 4. Speculative DFS Algorithm

differently. Note that, when the `maxSpeculationDepth` is set as 1, this algorithm is equivalent to pure SE.

The backtracking procedure performs differently in different cases to suit for the context of speculation. For a failed speculation with k branches, the speculation segment before the last branch (already known as infeasible) is analyzed to find the first infeasible branch (line 30). We adopt the binary search strategy for its stable performance in different cases, which needs approximately $\lceil \log_2(k-1) \rceil$ times of constraint solving. Line 34 deals with another case when the path ends with a reachable state, the procedure backtracks to the last unexplored branch.

We define the correctness of SSE as

"*Speculative symbolic execution generates the same execution tree as pure symbolic execution in the end*".

For the sake of space, the proof of the correctness of speculative DFS algorithm is available in the accompanying long version of this paper [14].

B. Eliminating False Alarms

Although SSE generates the same execution tree as pure SE, in practice, bugs located in dead code may cause SSE to yield analysis results that differ from that of pure SE. Consider the example shown in Figure 5, line 5 contains a 'divide-by-zero' bug; however, it is unreachable since its path condition $\langle a = b \wedge a \neq b \rangle$ is unsatisfiable. In SSE,

```
1: int a, b;
2: if(a == b) {
3:   if(a != b) {
4:     // unreachable bug
5:     a = a/0;
6:   }
7: }
```

Figure 5. Code With an Unreachable Bug Point

providing that the two branch statements in line 2 and line 3 are in a same speculation segment, line 5 would be executed without a prior determination of its reachability. Hence a false alarm will be reported.

Technically, this issue can be simply addressed via checking the reachability of the potential bug point just before generating the bug report. Note that, checking the feasibility at the potential bug point is equivalent to ending the speculation segment at the potential bug point, which would not bring extra computation to SSE.

It is necessary to point out that the exceptions caused by constraint solving (such as caused by the constraints beyond the ability of constraint solver) should be handled carefully, because a repeated reachability checking would trigger the same exception again. In this situation, the speculation segment should be checked carefully to find the first solvable and feasible branch, if any.

C. Feasibility

What brings risk to the feasibility of SSE is that SSE may execute dead code which is never executed in pure SE. Our backtracking mechanism guarantees that all the infeasible states will be cut away from the execution tree. However, in practice, there may exist some speculatively executed dead codes that bring influence to the unbacktrackable components of the system (such as updating a database). When the program behaviors are impacted by these components, SSE may get a different result from pure SE. In fact, for such kind of programs, pure SE may not work either.

This issue can be addressed by blocking the influence of speculatively executed instructions. One typical technique is providing appropriate support for symbolic execution (such as environment modeling [5]) to make the system more backtrackable.

D. Absurdity Based Optimization

SSE treats an unexplored branch as feasible one at its first glance and backtracks when a speculation fails. This feature implies that the more feasible branches in the execution tree, the better SSE performs. Meanwhile, this feature also implies that SSE is not good at handling the programs with a high ratio of infeasible branches since too many backtrackings might negate the benefits brought by successful speculation. To address this problem, we propose a simple but effective optimization, *absurdity based optimization*, which is complementary to SSE for its effectiveness on

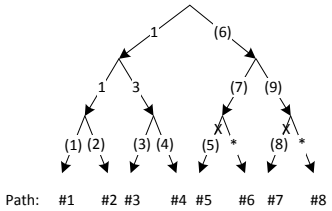

Figure 6. Speculative DFS With Absurdity Based Optimization

the programs with a high ratio of infeasible branches. This optimization is based on the following proposition.

Proposition 1: Regardless of runtime errors, given a reachable branch statement, at least one of its branches is feasible.

This proposition comes from the well-known *Reductio AD Absurdum* in first order logic [15], which says that if $\Gamma; \varphi$ is inconsistent, then $\Gamma \models \neg\varphi$, where Γ is a set of well-formed formulae (wff) and φ is a wff. In the context of symbolic execution, for instance, let state s be a reachable state with a satisfiable path condition $\langle\varphi_1 \wedge ... \wedge \varphi_n\rangle$. Suppose the next statement is a branch statement with two choices, say `if(`ϕ`)`, where ϕ is a boolean condition. If the search procedure has explored the `then` branch and find that it is infeasible, *i.e.*, the constraints set $\{\varphi_1, ..., \varphi_n\}$ and ϕ are inconsistent, then we can deduce that $\varphi_1, ..., \varphi_n \models \neg\phi$. Therefore, without querying the constraint solver, we know that the `else` branch is feasible.

This simple optimization is applicable both to pure and speculative symbolic execution. In practice, most of the branch instructions used in programs only have two choices. Therefore, as soon as an infeasible branch is explored before its counterpart, one invocation time of constraint solver can be saved. A high ratio of infeasible branches in the path space can provide many chances to perform this optimization. Consider the example in Figure 1 (comment line 6 and uncomment line 7), if applied with our optimization, the 8th and 11th calls to the constraint solver are unnecessary. As shown in Figure 6, branches where constraint solving is saved are marked with asterisks.

Absurdity based optimization is also related to the order in which the execution tree is explored. If the path space in Figure 6 is explored from right to left, since no infeasible branch is explored before its counterpart, no information can be used to perform optimization. In Section 4, we show how the effectiveness of absurdity based optimization is impacted by the exploration order over the path space.

E. Discussion

In this subsection, we first explain the benefits and cost of SSE, then we discuss what factors influence the effectiveness of SSE, and finally, we take a theoretical analysis on the speculative DFS algorithm.

The benefit brought by SSE is the saved constraint solvings when speculation succeeds. A successful speculation on a speculation segment of length k only needs once constraint solving, saving $k-1$ times compared with pure SE.

The cost of our approach lies in failed speculations. Consider a speculation segment with k branches $b_1, ..., b_i, ..., b_k (1 \leq i \leq k)$, where branches after b_i (including b_i) are infeasible ones, the corresponding path conditions are $p_1, ..., p_i, ..., p_k$. In SSE, the instructions between b_i and b_k are executed speculatively, which consumes extra time and memories. In addition to the first time of solving on p_k, binary search between $p_1 \sim p_{k-1}$ to find backtracking point needs at most $\lceil \log_2(k-1) \rceil$ times of queries. This may be more expensive than solving for path conditions $p_1 \sim p_i$ in pure SE when i is small.

The effectiveness of our approach is influenced by the characteristics of the program under analysis. Specifically, there are the following factors:

- *The ratio of infeasible branches in the path space*. SSE is suitable for the programs with a high ratio of feasible branches. For the programs with a high ratio of infeasible branches, SSE can be improved by the absurdity based optimization. Generally, this factor is the most important one.
- *The shape of the path space*. SSE is also related to the shape of the path space. For example, the continuous branches on the same direction (*i.e.*, all left turning or right turning) in the execution tree could increase the success rate of speculation.
- *The exploration order over the path space*. As discussed before, both SSE and optimization depend on the exploration order.
- *The complexity of path conditions*. The reduction of constraint solving for complex constraints can make SSE more useful.
- *The proportion of the constraint solving time in the total running time of SE*. We only attack the constraint solving part of SE. Therefore, the proportion of constraint solving time in the total running time of SE influences our ultimate goal.

The upper bound of speculative DFS algorithm is specified by the following proposition.

Proposition 2: The number of invocations of the constraint solver in speculative depth first search are larger than half of that in pure symbolic execution.

The proof of Proposition 2 is available in [14]. Specially, when a path space is a perfect binary tree with height n (the number of branches in the longest path), in pure SE, the times of constraint solving is $2^{n+1}-2$ (equal to the number of branches in the tree). While in speculative DFS, let k be the max speculation depth, then the times of constraint solving T_n^k can be quantified by the following equation:

$$T_n^k = \begin{cases} 2^n & (n \leq k) \\ 2^n + \dfrac{2^n - 2^{(n\%k)}}{2^k - 1} & (n > k) \end{cases} \quad (1)$$

The proof of Equation (1) is shown in [14]. For a perfect binary tree, speculative DFS performs best when $n \leq k$, saving nearly a half of the constraint solving times. When $n > k$, our algorithm gets better with the increase of k.

Speculative DFS performs worst when the execution tree only consists of a single path. In this case, although too many backtrackings affect the performance, our optimization technique can help to improve SSE. It is hard to take a precise analysis for the worst case because of the irregularity of the path spaces of programs.

In fact, both of the best case and worst case hardly happen in practice, more experimental evaluation is described in the next section.

IV. Implementation and Experimental Evaluation

A. Implementation

We have implemented the speculative DFS algorithm and the absurdity based optimization on top of Symbolic PathFinder (SPF) [13] with Java PathFinder (JPF) v6.0 [16][17]. JPF is an open source model checker for Java bytecode. It mainly consists of a Java Virtual Machine to support state storing, state matching and backtracking, as well as an adaptive search engine to systematically explore program states. Symbolic PathFinder (SPF) is built as an extension of JPF. SPF implements symbolic version semantics for Java bytecode instructions and uses JPF to systematically explore the execution tree of program under analysis. The features of our implementation are as follows.

- **New search strategy**. We have implemented the speculative DFS algorithm as a new search strategy, named `SpeculativeSegmentDFSearch`, to explore the execution tree of a program speculatively. The backtracking mechanism in Figure 4 is employed in the new search strategy.
- **New choice generator**. We have designed a new class `SpecuPCChoiceGenerator`, which is inherited from `PCChoiceGenerator`. The new choice generator is utilized to help backtracking and performing the absurdity based optimization.
- **New semantics of branch instructions**. To Support speculative execution, the semantics of branch instructions are adapted. Each branch instruction generates an instance of class `SpecuPCChoiceGenerator`. Speculation is performed according to the current speculation depth as shown in Figure 4.
- **Eliminating false alarms**. There exist four kinds of false alarms caused by SSE in SPF: runtime errors in the analyzed program, property violations, user defined exceptions and crashes caused by the program under analysis. We have handled all these issues in our implementation.

To use SSE in SPF, users need to configure SPF to use the speculative DFS strategy (using the property `search.class`) and specify the max speculation depth (using the property `symbolic.speculative.depth`). More details are omitted for the sake of space.

B. Experiments

To evaluate SSE, we have conducted some experiments. The objective of the experiments is to investigate the following research questions.

a. Effectiveness and cost. What are the effectiveness and cost of SSE compared with pure SE?

b. Speculation depth. How does the value of the max speculation depth influence the results and what is the optimal speculation depth for a real-world program?

c. Exploration order. In speculative DFS, the execution tree can be explored from two directions, false-side-first and true-side-first order, which one is better?

1) Experimental Setup: We choose five programs that are often used in the experiments related to JPF [13][18][19]. `WBS`, the Wheel Brake System, comes from the automotive domain [19]. The rest are all Java data structure programs: red-black tree (`TreeMap`), binary search tree (`BinTree`), binomial heap (`BinHeap`) and Fibonacci heap (`FibHeap`) [18]. In addition, we write a data structure program `List`, which implements a double linked list with sorted elements. For data structure programs, we use parameterized testing [18][20] to generate random call sequences of a limited length. The lines of these programs range from 230 (for `BinTree`) to 477 (for `TreeMap`). The ratios of the infeasible branches in the path spaces range from 0% (for `WBS`) to 42% (for `List`). We choose these programs in our experiments for two reasons. Firstly, these programs are often used in the experiments related to JPF. It is reasonable to choose these programs as the benchmark to evaluate SSE. Secondly, the effectiveness of SSE is heavily influenced by the ratio of infeasible branches in the path space of a program. For our selected programs, the ratios of the infeasible branches cover different levels. In fact, 42% (for `List`) is relatively high. Since each reachable branch has at least one feasible side, this ratio can never be higher than 50% if each branch only has two sides.

We conduct different experiments to investigate the aforementioned research questions. For each program, we perform four kinds of analysis: pure SE with/without optimization and SSE with/without optimization. In each kind of analysis, we vary the value of the max speculation depth and the exploration order independently. For each program, the max speculation depth is increased from 2 to the execution depth of the program. In fact, setting the max speculation depth larger than the execution depth yields the same analysis results as setting that as the execution depth, because in such cases speculation segments always end because of path ending. We use Yices [21] as the constraint solver because of its high performance and good usability. Under each configuration, we run three times to get the average results.

All of the experiments are carried out on an Intel Core i7 2.80GHz computer with 8 GB of RAM.

2) Results:

a. Effectiveness and Cost

Table I shows part of the experimental results of three kinds of analysis: pure SE, SSE and SSE with optimization. The first column shows the name of each program associated with its corresponding call sequence length if any. We only list the best case and the worst case of SSE (measured by the search time) when the max speculation depth varies from 2 to the maximum value. The corresponding max speculation depth is shown after the notation 'B.' and 'W.'. The third and fourth columns show the numbers of different constraint solving results and the percentage of unsat results respectively. Columns 5 and 6 show the total search time and the percentage of the time spent on constraint solving. The executed instructions are presented in the last column to show the cost of SSE. All the results shown in Table 1 are collected under the true-side-first exploration order.

The best results of SSE are from `WBS`, which has no infeasible branches in the execution tree. SSE reduces the times of constraint solving by 48.7% in the best case and 35.3% in the worst case. The search time is saved by 43.6% and 32% respectively. The worst results of SSE are from `List`. In the best case, SSE reduces 4.5% of the times of constraint solving and 6% of the search time. In the worst case, SSE brings extra 7.6% of the times of constraint solving and 7.8% of the search time. The reason is that, the high ratio of infeasible branches (42%) causes too many failed speculations, which negate the benefit brought by successful speculations. In average, SSE (without optimization) reduces the search time by 16.8% in the best case and by 8.8% in the worst case.

SSE with optimization outperforms SE and SSE for all programs. The optimization brings the most benefits for `List`, making SSE reduce 32.5% of the times of constraint solving and 37.6% of the search time in the best case. This is because the high ratio of unsat branches provides a lot of chances to perform optimization. As expected, for `WBS`, our optimization brings no benefit because no infeasible branches can be used. In average, SSE with optimization reduces 29.9% of the times of constraint solving and 30% of the search time in the best case, and 25.2% of the times of constraint solving and 24.6% of the search time in the worst case.

The results in column 7 shows that, in both of pure SE and SSE, constraint solving dominates most of the search time. The percentage of the time spent on constraint solving in the search time is reduced slightly by SSE.

The last column shows the number of executed instructions in different analysis. We can see that, despite executing a plenty of extra instructions, SSE is still faster than pure SE. Another important point is that SSE nearly does not consume extra memories than pure SE. The reason is that speculative DFS only spends extra memories to store the states in the current failed speculation segment, which can be ignored in our experiments.

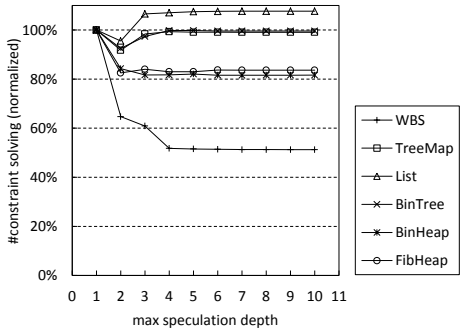

Figure 7. Impact of Max Speculation Depth in SSE

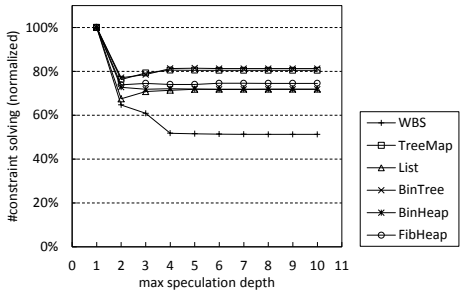

Figure 8. Impact of Max Speculation Depth in SSE With Optimization

b. Speculation Depth

Figure 7 shows how the max speculation depth impacts the times of constraint solving in SSE (without optimization). Results for larger speculation depths are omitted since they are nearly the same as the tails of the lines. For the program P, let T_P^p be the times of constraint solving in pure SE, and let T_P^k be the times of constraint solving in SSE with the max speculation depth k. Figure 7 shows the result of $T_P^k/T_P^p \times 100\%$. For `List`, `TreeMap`, `BinTree` and `FibHeap`, the optimal speculation depth is 2. Particularly, for `List`, SSE brings benefit only when the max speculation depth is 2. This is because the high ratio of infeasible branches causes too many backtrackings. For `BinHeap`, the optimal speculation depth is 6. For the program without infeasible branches (`WBS`), the results decrease monotonously with the increase of max speculation depth since the speculations never fail. Figure 8 shows the impact of the max speculation depth in SSE with optimization, in which the optimal speculation depths for different programs are the same as that in Figure 7. Generally, regardless of the tiny fluctuation in the tail, the optimal speculation depth ranges from 2 to 6 and shifts from small to big when the ratio of infeasible branches decreases.

We can see that, the optimization technique improves SSE significantly. Another interesting observation is that the results become stable when the max speculation depth reaches a threshold. This also demonstrates that our backtracking mechanism is quite efficient. Besides, the impacts of the

Table I
EXPERIMENTAL RESULTS (SPECU. DEP.=MAX SPECULATION DEPTH, B.=BEST, W.=WORST)

Program (call seq. length)	Analysis (specu. dep.)		#sat/unsat/all (Savings)	% unsat	Search Time(s) (Savings)	Solving Time(s) (Savings)	Solving Time ratio	#Instruction (extra)
WBS	pure SE		27646/0/27646	0%	66.2	62.9	95%	1382246
	SSE	B.(10)	14174/0/14174(48.7%)	0%	37.5(43.4%)	34.3(45.4%)	91%	1382246(0.0%)
		W.(2)	17886/0/17886(35.3%)	0%	44.9(32.2%)	41.8(33.5%)	92%	1382246(0.0%)
	SSE+ Opi.	B.(10)	14174/0/14174(48.7%)	0%	37.3(43.6%)	34.1(45.7%)	91%	1382246(0.0%)
		W.(2)	17886/0/17886(35.3%)	0%	45.0(32.0%)	42.0(33.2%)	93%	1382246(0.0%)
TreeMap (5)	pure SE		27005/17261/44266	39%	80.0	74.7	93%	855119
	SSE	B.(2)	18569/22045/40614(8.3%)	54%	72.2(9.7%)	65.6(12.2%)	91%	1077553(26.0%)
		W.(5)	20096/23772/43868(0.9%)	54%	79.4(0.8%)	71.5(4.3%)	90%	1548222(81.0%)
	SSE+ Opi.	B.(2)	11527/23561/35088(20.7%)	67%	61.1(23.6%)	54.6(27%)	89%	1159619(35.6%)
		W.(5)	13187/23829/37016(16.4%)	64%	66.4(17.0%)	59(21.0%)	89%	1549553(81.1%)
BinTree (5)	pure SE		22381/15589/37970	41%	76.6	72.2	94%	381092
	SSE	B.(2)	15913/19215/35128(7.5%)	55%	70.5(8.1%)	65.4(9.4%)	92%	578416(51.8%)
		W.(6)	16841/20975/37816(0.4%)	55%	77.0(−0.5%)	70.8(2.0%)	92%	980918(157.4%)
	SSE+ Opi.	B.(2)	9191/20086/29277(23.0%)	69%	57.7(24.7%)	52.4(27.4%)	91%	677685(77.8%)
		W.(10)	9860/20998/30858(18.7%)	68%	61.6(19.6%)	55.7(22.8%)	90%	984040(158.2%)
BinHeap (6)	pure SE		164116/23576/187692	13%	410.0	371.0	90%	21809086
	SSE	B.(21)	114948/38188/153136(18.4%)	25%	335.9(18.1%)	292.3(21.2%)	87%	29950152(37.3%)
		W.(2)	125178/32932/158110(15.8%)	21%	345.6(15.7%)	306.8(17.3%)	89%	24138598(10.7%)
	SSE+ Opi.	B.(21)	96600/38202/134802(28.2%)	28%	300(26.8%)	257.5(30.6%)	79%	29950152(37.3%)
		W.(2)	102410/34164/136574(27.2%)	25%	303.9(25.9%)	264.9(28.6%)	81%	25766426(18.1%)
FibHeap (6)	pure SE		58014/9142/67156	14%	148.5	133.0	90%	8098034
	SSE	B.(2)	44498/10898/55396(17.5%)	20%	125.2(15.7%)	110.0(17.3%)	88%	8416826(3.9%)
		W.(8)	40302/15848/56150(16.4%)	28%	130.0(12.5%)	112.6(15.3%)	87%	10731504(32.5%)
	SSE+ Opi.	B.(2)	37694/11906/49600(26.1%)	24%	113.0(23.9%)	97.5(26.7%)	86%	8859148(9.4%)
		W.(10)	33896/16160/50056(25.5%)	32%	117.0(21.2%)	100.0(24.8%)	85%	10731504(32.5%)
List (6)	pure SE		128076/94380/222456	42%	520.6	501.5	96%	2842969
	SSE	B.(2)	104299/108116/212415(4.5%)	51%	489.3(6.0%)	467.7(6.7%)	96%	3832245(34.8%)
		W.(7)	118384/121056/239440(−7.6%)	51%	561.4(−7.8%)	533.6(−6.4%)	95%	7311109(157.2%)
	SSE+ Opi.	B.(2)	33488/116635/150123(32.5%)	78%	325.0(37.6%)	303.0(39.6%)	93%	5371823(88.9%)
		W.(20)	38705/121176/159881(28.1%)	76%	354.1(32.0%)	327.7(34.7%)	93%	7333909(157.9%)

max speculation depth on the search time are not shown because they are nearly the same as that on the times of constraint solving.

c. Exploration Order

Figure 9 illustrates the difference of the search time under two different exploration orders in SSE without optimization. For a program P, let t_f^p be the search time of pure SE in false-side-first order, t_t^k be the search time of SSE with true-side-first order and t_f^k be the search time of SSE with false-side-first order, where k is the max speculation depth. Figure 9 shows the result of $(t_f^k - t_t^k)/t_s^p \times 100\%$. We can observe that there exists a distinct advantage of false-side-first order. Figure 10 shows the same calculation under SSE with optimization. In this case, the true-side-first order is slightly better than the false-side-first order, especially when the speculation depth is set as the optimal value.

To find the reasons, we collect the constraint solving results of two sides of the branches in pure SE and find that the true side has a much higher probability to be infeasible. This finding implies that, in general, the execution tree of a program tends to incline to the false sides of the branches. As a result, the higher rate of feasible false side branches endows SSE in false-side-first order with a higher success rate of speculation, while the lower rate of feasible true side branches provides more information to the optimization. In

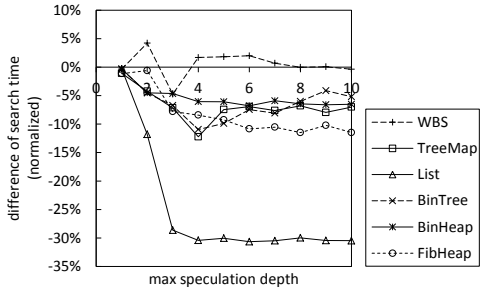

Figure 9. Difference of Search Time Between Different Exploration Orders in SSE

summary, the dissymmetry of the path space of programs makes SSE with optimization in the true-side-first order the optimal in our experiments, since the shape of the execution tree can be leveraged to the utmost extent. A more detailed analysis can be referred to [14].

C. Threats to Validity

The main validity problems need to consider in our experiments are threats to the external validity, which include two aspects: the chosen programs and the implementation platform.

We chose 6 programs in our experiments, 5 of which are often used in the experiments related to JPF. The characteristics of the path spaces of these programs influence

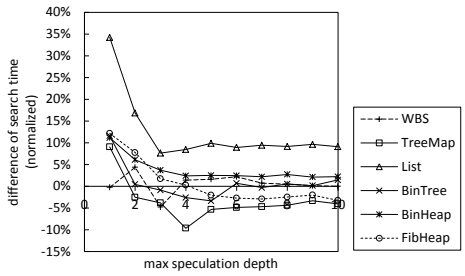

Figure 10. Difference of Search Time Between Different Exploration Orders in SSE With Optimization

the results definitely and our selected programs may not be representative. The ratios of the infeasible branches in our chosen programs range from 0 to 42%. From this perspective, our subjects are quite representative. We limit the call sequences length for data structure programs to control the running time of the experiments. Longer bounds would make constraint solving more time-consuming and may affect the results. The conditions of some branches in data structure programs are heap constraints. We do not perform speculation for heap constraints because the subsequent instructions heavily depends on the condition. We believe that for other types of programs, such as numerical programs or control programs (`WBS` in our experiments), the results may be better.

We selected SPF as the implementation platform and Yices as the constraint solver. A different selection may yield different running time, whereas the times of constraint solving would not change.

V. Related Work

Our work is inspired by the speculation execution used in pipelined processors [22], which predicts the outcome of a branch and issues the subsequent instructions before the actual branch outcome is known. SSE also executes the instructions after a branch before the feasibility of the branch is known. That is why we use the term *speculative symbolic execution* in this paper.

Speculation is used to improve performance in many other systems, such as operating systems [23], distributed file systems [24]. An essential difference between these systems and the work proposed in this paper is that, the performance improvement brought by our method stems from the decrease of the execution times of special operations, rather than the better parallelization brought by speculation in other systems mentioned above. To the best of our knowledge, we are the first to conduct systematic research on using speculation in symbolic execution.

Our work is also related to the large body of work on the scalability problem in symbolic execution, which stems from two reasons: path explosion problem and constraint solving overhead. To attack the path explosion problem, researchers have proposed to use path pruning [25][26][27], compositional method [28], abstraction [29], state merging [30], parallelism [11][19] and so on to improve path exploration. To alleviate the constraint solving overhead, a plenty of work have been proposed [5][6][7][8][9][10][20][31]. Generally, the optimization techniques employed in current symbolic execution systems attack the constraint solving overhead by query simplification, reusing previous results or fast checking before constraint solving [6][5][7][8].

The work proposed in this paper is an orthogonal and complementary approach. SSE employs a new fashion of path exploration technique, aiming to attack the constraint solving overhead by reducing the invocation times of constraint solver. SSE neither reduces the complexity of the queries submitted to the solver, nor caches constraints to reuse previous constraint solving results. SSE reduces the constraint solving overhead from a unique perspective.

Lei Bu *et al.* use the idea of speculation in [32] for the reachability checking of linear hybrid automata. Different from their work, the speculation in SSE is limited by a specific number, but the speculation in [32] stops when a target location is reached, which is more like a target driven 'slicing'. This is caused by different contexts of using speculation. Another difference is the backtracking mechanism. We use binary search to find backtracking points, whereas the irreducible infeasible set technique [33] is employed in [32] for backtracking. For SSE, binary search is stable and effective. Nevertheless, the minimal unsatisfiable core extraction technique [34] may also be used in the backtracking of SSE to reduce the times of constraint solving further.

At the time of this writing, EPFL released S^2E v1.2 [35]. An optimization named *speculative forking* is used in the concolic execution, where symbolic states are forked without regard to the feasibility at the branch that depends on symbolic values. These speculatively generated states are used as backtracking points (if feasible) to avoid re-execution from scratch when new inputs are generated. Although we both use the similar terms, speculation is used to achieve different goals.

The philosophy of SSE is a little similar to that used in many static analysis systems, which consider extra program behaviors and eliminate false alarms in the end [36][37]. The difference is that SSE prunes the infeasible behaviors in an appropriate chance to keep results precise and make a good tradeoff between the cost and the benefit as well.

VI. Conclusion and Future Work

We have proposed a new fashion of symbolic execution named *speculative symbolic execution* to reduce the invocation times of constraint solver, and hence extend the scalability of symbolic execution. SSE attacks the constraint solving overhead, which is almost always the most dominant in the running time of symbolic execution. We have proposed the speculative DFS algorithm and discussed its effectiveness. We also propose a key optimization technique,

named *absurdity based optimization*, to further improve SSE. This optimization is very effective especially for the programs with a high ratio of infeasible branches.

We have implemented SSE and our optimization technique on top of SPF. Experiments have been conducted to investigate several important research questions. The experimental results on six programs show that, SSE can reduce the invocation times of constraint solver by 20.7% to 48.7% (with a median of 29.9%), and save the search time from 23.6% to 43.6% (with a median of 30%). For future work, we plan to research on different search styles and use existing query optimization techniques to enhance SSE further.

ACKNOWLEDGMENT

This work is supported by the projects NSFC-61272140, NSFC-61120106006, SRFDP-20114307120015 and National 863 project 2011AA010106. We would like to thank Corina Păsăreanu and Willem Visser for their helps and discussions on using JPF and SPF.

REFERENCES

[1] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] C. Cadar, P. Godefroid, S. Khurshid, C. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *ICSE*, 2011, pp. 1066–1071.

[3] C. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *STTT*, vol. 11, no. 4, pp. 339–353, 2009.

[4] ——, "Symbolic execution and model checking for testing," in *HVC*, 2007, pp. 17–18.

[5] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[6] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: automatically generating inputs of death," *TISSEC*, vol. 12, no. 2, p. 10, 2008.

[7] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE*, 2005, pp. 263–272.

[8] P. Godefroid, M. Levin, D. Molnar *et al.*, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[9] C. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA*, 2011, pp. 34–44.

[10] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[11] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *ACM SIGOPS OSR*, vol. 43, no. 4, pp. 5–10, 2010.

[12] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 265–278, 2011.

[13] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *ISSTA*, 2008, pp. 15–26.

[14] Y. Zhang, Z. Chen, and J. Wang, "Speculative symbolic execution," http://arxiv.org/abs/1205.4951, 2012.

[15] H. Enderton, *A mathematical introduction to logic, second edition*, 2001.

[16] JPF, http://babelfish.arc.nasa.gov/trac/jpf.

[17] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[18] W. Visser, C. Păsăreanu, and R. Pelánek, "Test input generation for java containers using state matching," in *ISSTA*, 2006, pp. 37–48.

[19] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *ISSTA*, 2010, pp. 183–194.

[20] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *TAP*, 2008, pp. 134–153.

[21] Yices, http://yices.csl.sri.com/.

[22] J. E. Smith, "A study of branch prediction strategies," in *ISCA*, 1981, pp. 135–148.

[23] B. Wester, P. Chen, and J. Flinn, "Operating system support for application-specific speculation," in *EuroSys*, 2011, pp. 229–242.

[24] E. Nightingale, P. Chen, and J. Flinn, "Speculative execution in a distributed file system," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, 2005, pp. 191–205.

[25] S. Bardin and P. Herrmann, "Pruning the search space in path-based test generation," in *ICST*, 2009, pp. 240–249.

[26] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," in *HotDep*, 2009.

[27] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, 2008, pp. 443–446.

[28] P. Godefroid, "Compositional dynamic test generation," in *ACM SIGPLAN Notices*, vol. 42, no. 1, 2007, pp. 47–54.

[29] S. Anand, C. Păsăreanu, and W. Visser, "Symbolic execution with abstraction," *STTT*, vol. 11, no. 1, pp. 53–67, 2009.

[30] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *PLDI*, 2012, pp. 193–204.

[31] I. Erete and A. Orso, "Optimizing constraint solving to better support symbolic execution," in *ICSTW*, 2011, pp. 310–315.

[32] L. Bu, Y. Yang, and X. Li, "IIS-guided DFS for efficient bounded reachability analysis of linear hybrid automata," in *HVC*, 2011.

[33] J. Chinneck and E. Dravnieks, "Locating minimal infeasible constraint sets in linear programs," *ORSA Journal on Computing*, vol. 3, no. 2, pp. 157–168, 1991.

[34] A. Cimatti, A. Griggio, and R. Sebastiani, "Computing small unsatisfiable cores in satisfiability modulo theories," *JAIR*, vol. 40, no. 1, pp. 701–728, 2011.

[35] S2E, https://s2e.epfl.ch/.

[36] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, 2004, pp. 97–106.

[37] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis," *SAS*, pp. 203–217, 2005.