A Survey of Search Strategies in the Dynamic Symbolic Execution

Yu LIU*, Xu ZHOU[a] and Wei-Wei GONG[b]

National University of Defense Technology, Changsha, China
[a]zhouxu@nudt.edu.cn, [b]IssacGong@outlook.com
**Corresponding author: liuyunudt@gmail.com*

Abstract: Dynamic symbolic execution (DSE) is an important way to discover software vulnerabilities. One key challenge in DSE is to find proper paths in the huge program execution space to generate effective inputs. Currently, the main search strategies used for DSE include classical search strategy, heuristic search strategy, and pruning redundance strategy. This paper reviews and compares the main search strategies of DSE in recent years, including the Generational strategy, CarFast, Control-Flow Directed Search, Fitness-Guided Search strategy, Context-Guided Search strategy, RWset technique and Veritesting.

1 Introduction

Recent years have witnessed a widespread concern over the dynamic symbolic execution for software vulnerability analysis. The key idea behind dynamic symbolic execution was put forward by King, James C [1] 40 years ago. Dynamic testing tools [2,4,12,13,14,16,17,18,19] based on it have been coming out with the remarkable development of constraint solving technology [6]. Meanwhile, DSE has become a key approach to discover the program vulnerabilities.

As an approach of software testing, DSE is supposed to explore as many program nodes and paths as possible, which contributes to the discovery of more bugs in a given time budget. DSE starts with the user-given or random input, through which an execution path can be discovered; by calculating the constraints on this path, it is able to gain new inputs covering more different paths. Every change of path condition makes more optional branches available to DSE. With the expanding of the test case set, DSE will gain a large exploration space, which will bring challenge for selecting the best path [10].

Even a simple program may have a huge exploration space (e.g. the program in Fig. 1). In a given time budget, selecting an appropriate path directly affects the efficiency of DSE. Classical path selection approaches include BFS, DFS and Random Search, since each path has the same importance, it is difficult to avoid the path explosion problem. In order to address that, heuristic search approaches such as the Generational strategy [14], CarFast [5], Control-Flow Directed Search [23] and Fitness-guided Search Strategy [21] are applied to DSE to enhance its efficiency. Heuristic approaches make good use of program execution context but fail to take full advantage of DSE's high semantic insight which can be used to prune redundant paths and find the effective execution path for DSE.

```
line1  example(char* input,int num)
line2  {
line3        int counter = 0,value = 0;
line4        if(num == 200)
line5              exit();
line6        if(num >= 100){
line7              for(int i = 0;i < 100;i++){
line8                    if(input[i] == 'S')
line9                    {
line10                         counter ++;
line11                         value += 2;
line12                   }
line13             }
line14             if(counter == 50)
line15                   bug();
line16       }
line17       if(num < 50){
line18             value = 10;
line19       }
line20       else{
line21             value = 20;
line22       }
line23       counter = value + 10;
line24       return counter;
line25 }
```

Figure 1. An example program under test

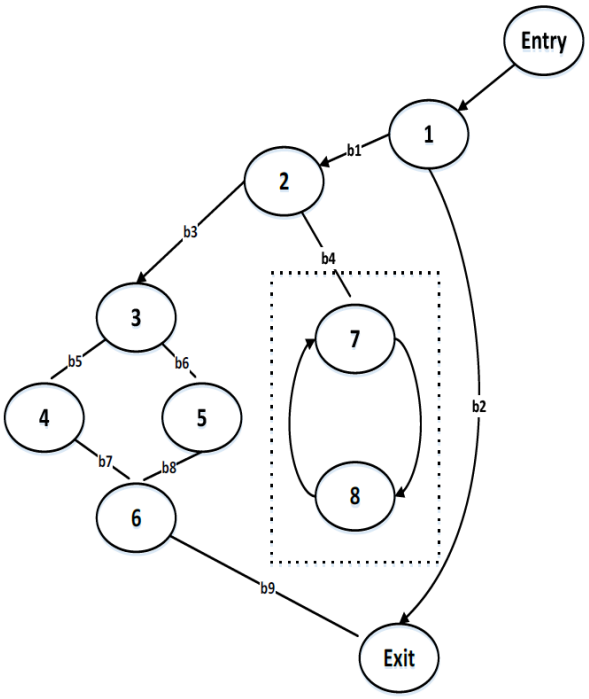

Figure 2. The topology of the example program

The remainder of the paper is organized as follows. Section 2 introduces the dynamic symbolic execution. We explain path search strategy in the dynamic symbolization execution in Section 3. Section 4 makes a comparison of theses search strategies. We conclude in Section 5.

2 Overview of Dynamic Symbolic Execution

2.1 Symbolic Execution

The main idea of symbolic execution is to represent variable values in the program with symbolic values. The symbolic execution maintains a symbolic state consistent with the program execution state. The symbolic state consists of a symbolic path constraint *PC* and a symbolic memory state σ. In classical symbol execution, σ is initialized to empty and *PC* is initialized to true. σ and *PC* are being updated during symbol execution. For example, for an input variable v, the symbol value s_0 is introduced into the mapping $\{v \mapsto s_0\}$; at each assignment statement, σ is updated, representing variables in the program as symbolic expressions over the symbolic values. In the statement if (e) S1 else S2, the condition *e* is evaluated under the current symbolic memory state as σ (e), and *PC* is updated to $PC \wedge \sigma(e)$, generating a new path constraint *PC'* with the initial value being $PC \wedge \neg\sigma(e)$. When the symbolic execution of a path comes to an end, the concrete input values generated by *PC* using a constraint solver will be executed, steering the execution of the program towards the path of *PC*, and the same execution result is obtained.

2.2 Dynamic Symbolic Execution

DSE is based on classical symbolic execution [1,26], and combines concrete execution with symbolic execution to make the latter faster and more accurate. DSE includes three main strategies.

The first one is concolic symbolic execution adopted in DART [4] and CUTE [18]. Dynamic symbol execution maintains two states, one is the concrete state and the other is the symbolic state. The concrete state maps all variables to their concrete values, while a symbolic state maps only variables without concrete values to their symbolic values. Concolic execution starts from a specific input and collects symbolic constraints at the conditional statements along the execution path. When the program execution reaches a conditional branch, concolic execution fork two new states where it generates two corresponding inputs respectively by solving constraints and then explore both branches independently. In theory, concolic execution can cover all program paths, while in practice it usually stops when a user defined coverage criteria are met, or the time budget expires.

The second one is called Execution-Generated Testing. The EGT [27] approach which is used by EXE and KLEE is a technique of generating testcases by solving the constraints on concrete values of program. EGT starts with a symbolic input rather than a concrete one. When program read external variables, EGT replaces it with a symbolic variables. All the variables depending on these symbolic variables will be operate symbolically. If the operation based on concrete values, it should be executed with concrete values. In the process of symbolic execution, the EGT adds constraints to the symbolic variables. When the symbol execution of a path comes to an end, we can generate input by solving these constraints on this path.

The last one is symbolic guided fuzzing, which is used by SAGE [14] and Driller [2]. As a combination of symbolic execution techniques and fuzzing techniques [2,8,14,15,20], symbolic-guided fuzzing utilizes the rapid test generating of fuzzing techniques and the good semantic insight into the program of symbolic execution. Symbolic-guided fuzzing use the dynamic symbol execution to guide how to mutate inputs [20], or just generate input to supplemental fuzzing[2]. The testcases generated by the Symbolic-guided fuzzing can trigger previously-unexplored code, which is not reachable by the fuzzing approach. Besides. Symbolic-guided fuzzing has better execution efficiency than pure dynamic symbolic execution.

2.3 The Importance of Search Strategy

In reality, it is not feasible to traverse all execution paths under a fixed time budget. This is referred to as the path explosion [10]. Therefore, each dynamic symbol execution is inevitable to face an important problem of how to choose a proper path in the huge path space. Some search strategies are adopted to the dynamic symbol execution to solve this problem. These strategies improve the coverage of symbolic execution,

which enables dynamic symbol execution to detect bugs in a more efficient way.

3 Search Strategies in the Dynamic Symbolization Execution

The search strategies used in the implementation of DSE can be divided into three categories: classical search strategy, heuristic search strategy, and pruning redundance strategy.

3.1 Classical Search Strategy

The classical search strategy uses the typical tree traversal algorithms which are independent of any prior knowledge of the execution tree. Typical tree traversal algorithms include DFS, BFS and Random Search.

DFS. Applied in symbolic execution tools such as DART[4] and CUTE[18] in the early days, DFS is the earliest search algorithm that used in dynamic symbolic execution. DFS has a good memory model that applies to dynamic symbolic execution: it first explores a path until the end, which provides an incremental path constraints. The previously used symbol constraint states are stored in memory for reuse by the solver. However, DFS may get trapped in loops and iterations that rely on symbolic input for conditional evaluation in the program. In addition, as the execution path becomes longer, the constraints will become larger, which will make the constraint solving more difficult.

BFS. BFS algorithms are rarely used in dynamic symbolic execution because it tends to choose a new path execution every time, which will make the dynamic symbol execution to continually switch the program states and the symbol states, resulting in a very poor memory utilization. What is worse, BFS algorithm will rapidly increase the exploration space of the program, making the path explosion problem more obvious.

Random Search. The random search algorithm executes from the root node of the program execution tree, and it will randomly select a path to continue the execution when a conditional branch appears. This selection method may provide a quick access to the deep of the program execution tree, however, it is of little stability and hard to guarantee a reliable path coverage. Hybrid concolic testing[11] interleaves random testing with concolic execution and utilizes random testing to rapidly increase path coverage. When the random testing saturate, it switches to the symbolic mode and uses the concolic execution to reach deeper uncovered nodes. Hybrid concolic testing shows a better coverage than DFS and pure random search.

3.2 Heuristic Search Strategy

Heuristic search strategy uses DSE to obtain the context information, select metrics (e.g., high statement coverage, node coverage, path coverage), and calculate the weight for the candidate path. Different heuristic algorithms focus on the different criteria, which makes a great difference of the results.

The Generational Strategy. The Generational strategy was introduced by SAGE [14], which is a whole-program white-box file fuzzing tool for x86 Windows applications. The generational search starts with an input seed and maintains a worklist. Each time generational search fetches an input from the *workList* for symbolic execution at a time. The generational search fetches an input from the *workList*, then starts the symbolic execution. After a single execution, *ExpandExecution* function based on path constraints is applied to generate new input. These *childInputs* are symbolically executed respectively, and scored according to the incremental block coverage. Then the *childInputs* will be inserted into the *workList* according to the score. Each new input limits the backtracking through the *bound* parameter. This search method can maximize the number of new test cases generated per symbolic execution, at the same time avoid any redundant search. Heuristics enables the generational search to maximize code coverage in a short time.

CarFast. CarFast [5] algorithm is a kind of heuristic algorithm which is based on the information of CFG graph. The main idea is that the number of program states contained in each branch in the program is different. In order to cover more states, branches containing more states should take the priority in dynamic symbolic execution. For the purpose of calculating the state information contained in the branches, CarFast algorithm needs to analyze the CFG graph of the program and sort the branches. CarFast algorithm works by analyzing the program CFG, which may result in a low efficiency of DSE when the program scale becomes large.

Control-Flow Directed Search. Control-Flow Directed(CFD) Search [7,23] uses the static structure of the program to direct the dynamic symbolic execution to explore path space of the program. Firstly, Control-Flow Directed Search generates the control flow graph of the program, then assigns values to each edge. The two branch edges of the conditional jump have a weight of 1, the remaining edges (including the function calls) have a weight of 0. For undiscovered program nodes, the total weight of the reachable path is computed. The path of the least weight is selected for exploration. Control-Flow Directed Search always selects the path with the fewest constraint, which reduces pressure of the solver.

Fitness-Guided Search Strategy. Fitness functions were originally used in search-based software [22], and introduced by Pex [24] as Fitnex strategy, a guide strategy of the dynamic symbolic execution. Fitness-Guided Search (FGS) Strategy flips a branch according to the fitness value of each branching node. Fitness value for the branch b is calculated by the formula (F (p) - FGain (b)). F (p) is the fitness function of path p. Fitness functions are derived from the boolean binary predicates [28]. The fitness function calculates the distance from a path to a target, which can be expressed as a fitness value. For example, for the if (counter == 50) in Fig.1 line 14, the fitness function is

"if (|50 – x| == 0) then 0 else |50 – x|". FGain (b) is the fitness gain of the branch *b*, which indicates the increase (or the decrease) of the path's fitness value when the branch is flipped. For example, when we flip the branching node code for the false branch of Fig.1 line 8 (input[i] == 'S') to the true branch, the fitness gain is 1.For each branch node, composite fitness value is (F(p) - FGain(b)). The Fitness-Guided Search Strategy always selects the branch with the smallest composite fitness value for the execution, which makes each symbolic execution closer to the target. At the same time, in order to achieve better results, Fitness-Guided Search Strategy also integrates other simple search strategies.

Context-Guided Search Strategy. Context-Guided Search(CGS) Strategy selects the branch with new context for the dynamic symbolic execution based on the context information of the executed program, so as to obtain better branch coverage. Context-Guided Search Strategy uses *k-context* to represent context information and determine whether to execute a new branch through the comparison of *k-contexts*. *k-context* is calculated from the information of preceding branches and dominator and defined as a sequence of k preceding branches in an execution path. For example in Fig.2, *2-context* of b5 is (b3,b5). Under ∞*-context*, CGS becomes BFS. In the CGS strategy, k incrementally increases. While computing the *k-context*, CGS needs to take the dominator information into account. In CFG, node d dominates node n, written as d dom n, if every path from the entry node to node n must go through node d[8]. It can be applied to edges (e.g. In Fig.2, b1 dominates b5 since all the execution paths heading for b5 must go through b1). In the calculation of *k-context*, CGS considers non-dominating branches only in the context information. For example in Fig.2, *2-context* of b5 is (b1,b5) instead of (b1,b3,b5), since b3 is the dominator of b5.When the dynamic symbolic execution encounters branch selection, the CGS algorithm compares the *k-context* of the current branch with those in the context cache. If the *k-context* did not appear before, the branch is selected for the next input. CGS algorithm makes good use of the context information, and has a very good efficiency.

3.3 Pruning Redundance Strategy

Strictly speaking, the strategy of pruning redundance strategy is not a search strategy. However, by reducing the search space, this method can effectively control the path explosion problem, and select the path suitable for DSE.

RWset. The main idea of the RWset [25] technique is that when the dynamic symbolic execution reaches a branch that has the same state as some previous execution, the branch can be pruned. At the same time, if two states that only differ in program values that are not subsequently read can be seen as identical, which can also be combined into a state. RWset truncates a path as soon as possibl e, thereby enhancing the efficiency of the dynamic symbolic execution, and reducing the path explosion.

Veritesting. Veritesting [9] is a combination of DSE and SSE, and has been integrated into the angr [29] platform as an effective selecting method to avoid path explosion. Veritesting starts with DSE, when the program encounters a branch and needs to fork new executions, it switches to an SSE-style approach. In SSE mode, Veritesting uses SSE to analyze a dynamically recovered CFG, removes some unexecutable paths from CFG, and unrolls loops. Finally, it merges the states obtained by SSE with the previous DSE states, and then continues the dynamic symbolic execution. This method solves the DSE path explosion overhead with SSE solution overhead, which gains better node coverage and path coverage in most cases.

Table 1. The comparison of Search strategies

	Main Cost	Criteria	Usage of Semantic Insight	Tools
DFS/BFS	--	--	--	DART CUTE
Random Search	--	--	--	DART
Generational Search	ExpandExecution	Code coverage	The hierarchy of the program topology diagram	SAGE
CarFast	Analysis of CFG	Statement coverage	The state information of the CFG	CarFast
CFD	Calculating the weight of edge	Shortest path	Topological structure of CFG graph	CREST
FGS	Solving fitness function	Closer to the target	Fitness value of conditional branch	Pex
CGS	Comparing k-context	Path coverage Node coverage	The context dependency of the program path	--
RWset	Pruning statement	Merge Statements	Program status using DSE	RWset
Veritesting	SSE	Reduce invalid paths	DSE status and SSE status	Angr MergePoint

4 The Comparison of Search Strategies

No matter what search strategy it is, its main purpose is to choose the best path for the dynamic symbol execution within a limited time and help it achieve a better performance. The comparison of these strategies is in Table 1.

In theory, DFS and BFS enable a full traversal of the execution tree. However, this approach is not feasible because of the huge search space. The heuristic search algorithm abandons completeness, selects specific metrics, and uses the program context, CFG and other information to prioritize the paths of the dynamic symbolic execution, in order to achieve the optimal performance with the limited time cost. The method of pruning redundant paths reduces the state space of DSE, so as to improve the efficiency of DSE.

5 Conclusion

DSE is an effective means of finding software vulnerabilities. Search strategies play a major role in the improvement of DSE. In this paper, we discuss and compare the main search strategies of DSE, divide them into three categories: classical search strategy, heuristic search strategy, and pruning redundance strategy. In the future, the research on the combination of pruning redundant paths with a heuristic search approach could possibly enhance the execution efficiency of DSE.

Acknowledgement

This work is partially supported by National High-tech R\&D Program of China (863 Program) under Grants 2015AA01A301, by program for New Century Excellent Talents in University, by National Science Foundation (NSF) China 61272142, 61402492, 61402486, 61379146, 61272483, by the laboratory pre-research fund (9140C810106150C81001)

References

1. King, James C. "Symbolic execution and program testing." *Communications of the ACM* 19.7 (1976): 385-394.
2. Stephens, Nick, et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." *Proceedings of the Network and Distributed System Security Symposium*. 2016.
3. Seo, Hyunmin, and Sunghun Kim. "How we get there: a context-guided search strategy in concolic testing." *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014.
4. Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices*. Vol. 40. No. 6. ACM, 2005.
5. Park, Sangmin, et al. "CarFast: achieving higher statement coverage faster." *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.
6. De Moura, Leonardo, and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications." *Communications of the ACM* 54.9 (2011): 69-77.
7. CREST. Automatic test generation tool for C. https://code.google.com/p/crest/.
8. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison Wesley, 2nd edition, Sept. 2006.
9. Avgerinos, Thanassis, et al. "Enhancing symbolic execution with veritesting." *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.
10. Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." *Communications of the ACM* 56.2 (2013): 82-90.
11. Majumdar, Rupak, and Koushik Sen. "Hybrid concolic testing." *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007.
12. Chipounov, Vitaly, Volodymyr Kuznetsov, and George Candea. "S2E: a platform for in-vivo multi-path analysis of software systems." *ACM SIGPLAN Notices* 46.3 (2011): 265-278.
13. Ciortea, Liviu, et al. "Cloud9: a software testing service." *ACM SIGOPS Operating Systems Review* 43.4 (2010): 5-10.
14. Godefroid, Patrice, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." *NDSS*. Vol. 8. 2008.
15. Caselden, Dan, et al. *Transformation-aware exploit generation using a HI-CFG*. No. UCB/EECS-2013-85. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, 2013.
16. Babic, Domagoj, and Alan Hu. "Calysto." *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008.
17. Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." *OSDI*. Vol. 8. 2008.
18. Sen, Koushik, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." *ACM SIGSOFT Software Engineering Notes*. Vol. 30. No. 5. ACM, 2005.
19. Cadar, Cristian, et al. "EXE: automatically generating inputs of death." *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008): 10.
20. Cha, Sang Kil, Maverick Woo, and David Brumley. "Program-adaptive mutational fuzzing." *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.
21. Xie, Tao, et al. "Fitness-guided path exploration in dynamic symbolic execution." *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009.

22. McMinn, Phil. "Search-based software test data generation: A survey." *Software Testing Verification and Reliability* 14.2 (2004): 105-156.
23. Burnim, Jacob, and Koushik Sen. "Heuristics for scalable dynamic test generation." *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 2008.
24. Tillmann, Nikolai, and Jonathan De Halleux. "Pex–white box test generation for. net." *International conference on tests and proofs*. Springer Berlin Heidelberg, 2008.
25. Boonstoppel, Peter, Cristian Cadar, and Dawson Engler. "RWset: Attacking path explosion in constraint-based test generation." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008.
26. Clarke, Lori A. "A program testing system." *Proceedings of the 1976 annual conference*. ACM, 1976.
27. Cadar, Cristian, and Dawson Engler. "Execution generated test cases: How to make systems code crash itself." *International SPIN Workshop on Model Checking of Software*. Springer Berlin Heidelberg, 2005.
28. Tracey, Nigel, John Clark, and Keith Mander. "Automated program flaw finding using simulated annealing." *ACM SIGSOFT Software Engineering Notes*. Vol. 23. No. 2. ACM, 1998.
29. Shoshitaishvili, Yan, et al. "SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis." *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.