S2PF: Speculative Symbolic PathFinder

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

ABSTRACT

Recently, symbolic execution gains a significant progress in its techniques and applications. However, in practice, scalability is still a key challenge for symbolic execution. In this paper, we present S^2PF, which improves the scalability of Symbolic PathFinder by integrating speculative symbolic execution with the general heuristic search framework. In addition, two optimizations are proposed to improve the speculative symbolic execution in S^2PF. Experimental results on six programs show that, S^2PF can reduce the solver invocations by 36.4% to 49% (with an average of 40.3%), and save the search time by 30.6% to 43.5% (with an average of 35%).

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Program Verification

General Terms

Performance, Verification

Keywords

symbolic execution, constraint solving, Symbolic PathFinder, speculative symbolic execution

1. INTRODUCTION

Symbolic Execution (SE) [22] is a fundamental program analysis technique proposed more than three decades ago. SE executes a program with symbolic variables and computes the predicates of the input and output variables. Up to now, SE has been used in many topics, including test generation [7], bug detection [8], software verification [19], data structure repair [20], *etc.*

Symbolic PathFinder (SPF) [25] is a symbolic execution engine targeting Java programs. SPF extends the software model checker Java PathFinder (JPF) [31] systematically and provides a generalized SE platform for Java bytecode programs. Since its birth, SPF has been improved in many aspects, such as data structure abstraction [1], string handling [21] and parallelization [29]. These improvements make SPF more scalable and feasible.

Although leveraged by the increasing computation power, SE is still enslaved to many aspects. One key challenge faced by SE is *scalability*, which is mainly posed by path explosion and the overhead of constraint solving. In fact, constraint solving is the most dominative in the execution time of SE. For example, KLEE spends 40%-90% of the execution time in constraint solving [7]. In S^2E, most execution time is dominated by constraint solving [11]. For SPF, according to the experiments we conducted, this rate is 79%-96% [34].

To reduce the overhead of the constraint solving in SE, in [34] we proposed a new fashion of symbolic execution, named *speculative symbolic execution* (SSE). In SSE, the branch statements are speculatively executed without regard to the feasibility. The constraint solver is invoked when the number of the speculated statements on the current path reaches a specific number, say k. If the speculation succeeds, $k-1$ solver invocations are saved; otherwise, SSE needs to backtrack to the last feasible branch. We have implemented SSE with the depth first search (DFS) engine on SPF, and the experimental results indicate that SSE can have an average reduction of 30% in the execution time and the solver invocations.

Though SSE is implemented on SPF in [34], SPF is not systematically extended to incorporate speculation mechanism. For example, how to combine speculation with more general search styles, *e.g.*, the heuristic search framework, is not discussed. In addition, the backtracking mechanism can be optimized further. In this paper, we introduce Speculative SPF (S^2PF), which systematically extends SPF with SSE by incorporating speculation within the heuristic state exploration framework. Besides, the backtracking is improved by using the *unsatisfiable core* (UC) generation technique [12]. We also present a Yices specific optimization to speed up S^2PF further.

The main contributions of this paper are three folds: (1) we present S^2PF, which systematically extends SPF to implement speculative symbolic execution; (2) we propose two optimizations to speed up SSE further: using UC generation technique and a Yices specific feature to reduce solver invocations during backtracking; (3) we have conducted several experiments to justify the correctness and effectiveness of S^2PF, and the results indicate that S^2PF can have an average reduction of 35% in the search time.

The following of this paper is organized as follows: Section 2 briefly introduces SSE by a motivating example and describes the details of the algorithm used in S^{2}PF; Section 3 presents the implementation of S^{2}PF; Section 4 reports the experiments to evaluate S^{2}PF; Section 5 discusses the related work and finally the conclusion is drawn in Section 6.

2. SPECULATIVE SYMBOLIC EXECUTION

In this section, we illustrate the basic idea of SSE by a motivating example, and then describe the speculative heuristic search algorithm that combines speculation and the general heuristic search framework.

2.1 Motivating Example

Take the program in Figure 1 for example. The program computes the sum of the absolute values of two integers, x and y, and outputs the sum if the sum is larger than y. The path space of this program is shown in Figure 2(a), where the left side of a branch represents the false side of the corresponding branch statement and the right side for the true side. The path space includes 8 paths, including 2 infeasible ones (#5 and #7) and 6 feasible ones. In pure SE, whenever the search procedure encounters a branch, the path condition is updated and then submitted to a constraint solver to check its satisfiability. Thus, the constraint solver is invoked 14 times in total.

In SSE, the search procedure strides over branch statements without checking the feasibility until the unchecked branches are accumulated to a particular number, say the *max speculation depth*. Accordingly, a path segment consisted by unchecked branches is called a *speculation segment*. For the program in Figure 1, assuming that the max speculation depth is 3 and the path space is traversed in DFS, the `else` branches of the statements `if(x<0)`, `if(y<0)` and `if(x>y)` are taken speculatively. At the end of the path #1, the number of uncheck branches reaches 3 and therefore the constraint solver is invoked. Here the result is `sat` and thus 2 solver invocations are saved. In Figure 2, the bracketed numbers show where the constraint solver is invoked and the number n tagged on a branch indicates the feasibility of the branch is known in the n-th invocation of the constraint solver.

When the search procedure reaches the end of the path #5, the path condition $\langle x < 0 \wedge y \geq 0 \wedge y - x \leq y \rangle$ is unsatisfiable, so we need to backtrack to the last feasible branch. In [34], we use binary search in backtracking, which needs two solver invocations, as indicated by (6) and (7) in Figure 2(a). Totally, the constraint solver is invoked by 11 times, saving 3 invocations compared with pure SE. Actually, the solver invocations in SSE are also related to the exploration order over the path space. Figure 2(b) shows how the constraint solving is performed when the path space is traversed from the other side, which only needs 8 solver invocations.

```
     int x, y;
if(x < 0)
   x = -x;
if(y < 0)
   y = -y;
x = x + y;
if(x > y)
   output(x);
```

Figure 1: A Motivating Example

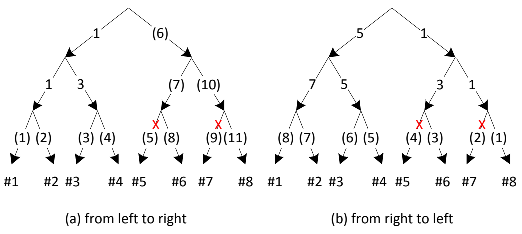

Figure 2: Constraint Solving in SSE with DFS

2.2 Algorithm

2.2.1 *Speculative Heuristic Search*

We have proposed the speculative DFS algorithm that combines speculation and DFS in [34]. In this section, we present a speculative heuristic search algorithm that integrates speculation with the general heuristic search framework.

Generally, a heuristic search procedure uses an `open` list to keep track of the fringe of the search and a `closed` list to record the states that have already been accessed [24]. Each state in the `open` list is associated with a heuristic value as its priority. The search procedure iteratively fetches out from the `open` list a state with the highest priority and explores its successors. The successors not in the `closed` list are saved to the `open` list in order of their heuristic values for future expansion. The procedure continues until the `open` list is empty.

Our *Speculative Heuristic Search* (SHS) algorithm is a variant of the general heuristic search algorithm. SHS algorithm has the following three features: first, the heuristic search (choosing which state to expand) and DFS (the generation of speculation segments) are interweaved; second, the `open` list is used to store the *open states* (*i.e.*, states with unexplored successors), and a state may be fetched from and saved into the `open` list for multiple times; third, each time when a state is fetched out from the `open` list, only one of its direct successors is accessed.

Figure 3 briefly shows the main procedure of the SHS algorithm. The procedure `SpeculativeHeuristicSearch()` performs heuristic search by maintaining an `openlist` to store the states that need to be propagated in the future. At the beginning, the initial state is added to the `openlist`. A `while` loop iteratively fetches a state from the `openlist` to propagate until the `openlist` is empty. In line 5, the most prioritized state in the `openlist` is selected by the function `getNextState()`. Then the speculation starts from this state (line 6) and continues to forward until a speculation segment is fully propagated, *i.e.*, the max speculation

```
 1:SpeculativeHeuristicSearch() {
 2:  openlist = {initial state};
 3:  while(openlist != empty set) {
 4:    specuSegment = empty set;
 5:    state s = getNextState(openlist);
 6:    propagateSegment(s, specuSegment);
 7:    if(failed speculation)
 8:      backtrack(specuSegment);
 9:    saveOpenStates(specuSegment);
10:  }
11:}
```

Figure 3: Speculative Heuristic Search Algorithm

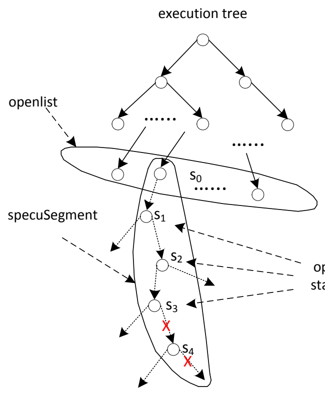

Figure 4: A Snapshot of Executing SHS Algorithm

depth is reached or the path ending or an exception is encountered. Line 7 checks the feasibility of the speculation segment, after which the unreachable states are trimmed by function `backtrack()`. Sequentially, in line 9, open states in the speculation segment are saved in the `openlist`.

Figure 4 illustrates a snapshot of executing the SHS algorithm. State s_0 is selected from the `openlist` to propagate. The speculation segment includes 5 states, $s_0 \sim s_4$. At the end of the segment, the constraint solver is invoked and returns `unsat`. So the `backtrack` procedure analyzes the segment and marks out the last two infeasible branches (indicated by a cross in Figure 4). After that, the states s_1, s_2 and s_3 are saved into the `openlist` for future propagation because they have unexplored successors. State s_4 is abandoned because it is guarded by an infeasible branch.

Note that, in our algorithm we do not use the closed list as the general heuristic search algorithm because there is no cycle in the path space of a program.

2.2.2 Unsatisfiable Core Based Backtracking

In [34], we propose to perform binary search on a failed speculation segment to find the backtracking point. This method needs approximately log_2^k solver invocations for a speculation segment with k branches. Although binary search is stable when k is large, backtracking can still be optimized, especially when k is small or the first infeasible branch is close to the start point of the speculation segment. Thereby, here we propose to use the *unsatisfiable core based backtracking* to reduce the solver invocations.

In mathematical logic, the *unsatisfiable core* (UC) is a small unsatisfiable subset of unsatisfiable set of clauses [13]. Currently, there are a few modern SMT solvers supporting UC generation [3, 4, 33, 15]. In SSE, UC can be used to help backtracking when a speculation fails.

For a failed speculation segment with k branches, assuming that $p_1, ..., p_k$ are the corresponding path conditions and C_i be the set of constraints contained by p_i, where $C_i \subseteq C_{i+1}$ $(1 \leq i \leq k-1)$. Suppose that C_j^u is the generated UC when solving p_j, then we can conclude that p_i is unsatisfiable if $C_j^u \subseteq C_i$.

However, the tools mentioned above (except MathSat) are not extracting minimal unsatisfiable cores (MUC), *i.e.*, there may exist an unsatisfiable proper subset of a generated UC. Besides, a constraint set may contain multiple UCs. This means that $C_j^u \nsubseteq C_i$ does not imply the satisfiability of C_i. Therefore, after getting C_k^u when speculation fails, we check $C_1, ..., C_k$ sequentially to find the smallest C_i that includes C_k^u, then we solve C_{i-1} recursively (if $i > 0$) until the last satisfiable set (*i.e.*, the last feasible branch) is located. In fact, computing all the MUCs can help to locate the last feasible branch with less invocations, but the cost is high [13].

2.2.3 Discussion

Essentially, the SHS algorithm is a hybrid of the heuristic search and DFS. The global state selection is determined by the heuristic value while the speculation segment is extended in the DFS style. The proof of the correctness of the SHS algorithm is similar to that of the speculative DFS algorithm in [34].

It is worth noting that the bug point guarded by an unsatisfiable path condition, *e.g.*, `if(a<a){b = b/0;}`, may be touched by SSE. The solution is to check the reachability of a bug point before reporting it. In practice, other types of false alarms can be eliminated in the same way.

The theoretical upper bound of the effectiveness of the SHS algorithm is specified by the following proposition.

PROPOSITION 1. *The number of solver invocations in the speculative heuristic search algorithm is larger than half of that in the non-speculative heuristic search algorithm.*

The proof is similar to that of the Proposition 2 in [34]. More details are eliminated for space sake.

In the worst case, the size of the open list of SHS algorithm is approximately k times larger than that of non-speculative heuristic search algorithm, where k is the max speculation depth. The reason is that the speculation from an open state can add at most k states to the open list.

2.3 Absurdity Based Optimization

Absurdity based optimization (AB optimization) aims to reduce the solver invocations by utilizing the information between different paths. Given a reachable state s with path condition P, suppose its next instruction to execute is a branch instruction with n branches $b_1, ..., b_n$, and the corresponding conditions are $\varphi_1, ..., \varphi_n$, where $\varphi_1 \vee ... \vee \varphi_n \Leftrightarrow true$. Assuming that all the branches except b_k $(1 \leq k \leq n)$ are infeasible, *i.e.*, $P \wedge \varphi_i$ $(1 \leq i \leq n \wedge i \neq k)$ is unsatisfiable. Let $\mathfrak{A}$ be the assignment that makes P true, from $\varphi_1 \vee ... \vee \varphi_n \Leftrightarrow true$, we can easily deduce that φ_k is true under $\mathfrak{A}$. Thus, branch b_k is feasible.

Take the two branches below state s_3 in Figure 4 for example, after we know the infeasibility of the right branch below s_3, the feasibility of the left branch can be deduced without invoking the solver.

3. IMPLEMENTATION

3.1 Speculative Symbolic PathFinder

We have implemented the SHS algorithm with UC based backtracking as well as the AB optimization in S^{2}PF. The prime features of our implementation are as follows.

The speculative heuristic search engine. The SHS algorithm and the UC based backtracking is implemented in a new class `SpeculativeHeuristicSearch`, which is inherited from the class `HeuristicSearch` provided by JPF. The fields `queue` and `specuStack` are maintained to store the open list and the speculation segment, respectively. The search() method of HeuristicSearch is overridden to implement the SHS algorithm. A static variable currentSpeculationDepth is used to indicate the speculation depth of the current JVM state. We use Yices [33] as the constraint solver. A new class ProblemYicesWithUC is created as the interface between S^{2}PF and Yices to support UC backtracking. After a speculation segment is generated, the reachable states in the speculation segment with more choices in the choice generator are saved in the queue for further exploration.

The new semantics for branch instructions. The semantics of branch instructions are modified to support speculative execution. For each branch instruction, we generate a SpecuPCChoiceGenerator, which is a subclass of PCChoiceGenerator. The speculation depth of the current JVM state is associated to this new choice generator. The feasibility of its each choice is also recorded, in order to perform AB optimization. The path condition is updated according to the current choice of the choice generator. The solver is invoked if the speculation depth reaches the max speculation depth. If the result is unsat, the UC based backtracking procedure is invoked to cut off the infeasible branches from the current speculation segment.

Eliminating false alarms. As discussed before, we handle all the false alarms introduced by SSE in S^{2}PF, including the runtime errors in the analyzed program, property violations, user defined exceptions, the crashes caused by the analyzed program, *etc.* In fact, most false alarms are eliminated by overriding the JVM.forward() function, in which the statement executeNextTransition() is encapsulated with a try/catch block. In the catch block, the path condition is solved and the exception is re-thrown out only when the solving result is sat.

To use S^{2}PF, users need to configure new properties for SPF. The most important ones are search.class and symbolic.speculative.depth, which specify the search class and the max speculation depth, respectively.

3.2 Yices specific Optimization

We also have implemented a Yices specific optimization to improve SSE further. When Yices is invoked at the end of a speculation segment, the constraints in the path condition are translated into the input commands of Yices. Originally in SPF, the satisfiability of a constraint set is checked (by Yices command (check)) after all the constraints are fed to Yices. In S^{2}PF, we use Yices in an interactive way. The constraints from the root of a path to the leaf are fed to Yices sequentially. The consistency of the added constraints is checked whenever a new constraint is fed to Yices. Once Yices finds an inconsistency in the fed constraint set, the branches corresponding to the unprocessed conditions are trimmed from the speculation segment because of the infeasibility. However, in the interactive mode, Yices is not always able to find inconsistency (without command (check)) when the added constraint set is unsatisfiable. As a consequence, after using this feature of Yices to help to trim a speculation segment, we use the UC based backtracking to find the last feasible branch.

4. EVALUATION

4.1 Experimental Setup

To evaluate S^{2}PF, we choose the same programs used in the experiments in [34], including 5 data structure programs: red-black tree (TreeMap), binary search tree (BinTree), binomial heap (BinHeap), Fibonacci heap (FibHeap) and ordered linked list (List), as well as a program from the automotive domain, the Wheel Brake System (WBS). These programs are often used in the experiments related to JPF.

For each program, we perform the following four kinds of analysis.

Mode A. Raw SPF with the heuristic search engine configured equivalent to DFS, *i.e.*, the state with a greater depth has a higher priority in the open list. We run SPF and collect the results as the base line of the experimental results.

Mode B. SPF with the speculative DFS engine and AB optimization. This is the best mode in [34], providing the results of our prior work.

Mode C. S^{2}PF with speculative heuristic search engine (also configured equivalent to DFS) and AB optimization. We perform this analysis to show the correctness of the heuristic search engine.

Mode D. S^{2}PF with all efforts, including AB optimization, UC based backtracking and Yices specific optimization.

When analyzing each program under Mode B, C and D, the max speculation depth is increased from 2 to the execution depth of the program. Besides, we also run S^{2}PF with configuring heuristic search framework as the breadth first search (BFS), *i.e.*, the state with less depth is more prioritized in the open list. All of the experiments are carried out on an Intel Core i7 2.80GHz computer with 8 GB of RAM.

4.2 Experimental results

Table 1 shows part of the experimental results. The first column shows the name of each program with its corresponding call sequence length if any. For each program, we list the number of solver invocations (column 3) and the search time (column 4) in each mode, where the corresponding optimal max speculation depth is associated in column 2. We do not show the results with larger max speculation depths since the results do not change once the max speculation reaches the maximum execution depth of a program.

The search time under Mode A is slightly different from the results of raw SPF in [34] because here we use a different search engine. The results of Mode B are literally from paper [34], in which the savings of the search time are computed under the base line in [34] (marked with asterisks in Table 1). The numbers of the solver invocations in Mode B are the same as that in Model C, which indicate the correctness of the implementation of the SHS algorithm.

The results of Mode D show that, S^{2}PF with all the efforts brings an obvious improvement to SPF. S^{2}PF reduces the solver invocations by from 36.4% (BinHeap) to 49% (WBS), with an average of 40.3%. The search time is reduced by from 30.6% (TreeMap) to 43.5% (WBS), with an average of 35%.

An important observation is that, for all the six programs, S^{2}PF performs strictly better when increasing the max speculation depth. In Table 1, the best results of Mode D are uniformly collected when running with the largest max speculation depth. This feature implies that S^{2}PF is quite efficient in backtracking. More importantly, it also implies that there is no need to find the optimal max speculation depths for different programs as that in [34].

Another result is that the analysis using the heuristic search framework configured as BFS yields the same re-

Table 1: Experimental Results (call seq.=call sequence length, dep.=max speculation depth)

Program (call seq.)	Mode (dep.)	#sat/unsat/all (Savings)	Time(s) (Savings)
WBS	A	27646/0/27646	62
	B(10)	14174/0/14174(49%)	37.3(43.6%)*
	C(10)	14174/0/14174(49%)	35(43.5%)
	D(10)	14174/0/14174(49%)	35(43.5%)
TreeMap (5)	A	27005/17261/44266	75
	B(2)	11527/23561/35088(21%)	61(23.6%)*
	C(2)	11527/23561/35088(21%)	57(24%)
	D(7)	10380/17262/27642(37.5%)	52(30.6%)
BinTree (5)	A	22381/15589/37970	73
	B(2)	9191/20086/29277(23%)	57.7(25.5%)*
	C(2)	9191/20086/29277(23%)	54(26%)
	D(9)	7809/15629/23438(38.3%)	50(31.5%)
BinHeap (6)	A	164116/23576/187692	380
	B(10)	96600/38202/134802(28.2%)	300(26.8%)*
	C(10)	96600/38202/134802(28.2%)	272(28.4%)
	D(10)	95756/23576/119332(36.4%)	262(31%)
FibHeap (6)	A	58014/9142/67156	137
	B(2)	37694/11906/49600(26%)	113(23.9%)*
	C(2)	37694/11906/49600(26%)	105(23.4%)
	D(10)	33456/9142/42598(36.6%)	92(32.8%)
List (6)	A	128076/94380/222456	502
	B(2)	33488/116635/150123(32.5%)	325(37.6%)*
	C(2)	33488/116635/150123(32.5%)	310(38.2%)
	D(7)	30112/94380/124492(44%)	298(41%)

sults as above. This is because for SHS algorithm, using a different heuristic does not impact the number of solver invocations.

5. RELATED WORK

Speculation is used to improve the performance in many systems, such as pipelined processors [28] and operating systems [32]. S^2E v1.2 [26] uses speculative forking in the concolic execution to generate backtracking points at branches. In [5], Lei Bu *et al.* use a target location-guided search for the reachability problem of linear hybrid automata, where the irreducible infeasible set technique [9] is employed to help backtracking.

Our work is related to the large body of the existing work on the scalability challenge of symbolic execution, including path pruning [2, 10, 6], compositional method [16], abstraction [1], state merging [23], parallelism [14, 29] and all other efforts on alleviating the overhead of constraint solving [7, 8, 27, 18, 17, 30]. SSE is orthogonal and complementary to these approaches.

6. CONCLUSION

In this paper, we present S^2PF, an extension tool of SPF, which implements SSE under the general heuristic search framework. Two optimizations to speed up SSE are also proposed. The experimental results of analyzing six programs show that S^2PF can save 35% of the search time in average. The next step is to conduct extensive experiments on real-world programs and optimize S^2PF further.

7. REFERENCES

[1] S. Anand, C. Păsăreanu, and W. Visser. Symbolic execution with abstraction. *STTT*, 11(1):53–67, 2009.

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

[3] C. Barrett and C. Tinelli. Cvc3. In *CAV*, pages 298–302, 2007.

[4] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The mathsat 4 smt solver. In *CAV*, pages 299–303, 2008.

[5] L. Bu, Y. Yang, and X. Li. IIS-guided dfs for efficient bounded reachability analysis of linear hybrid automata. In *HVC*, 2011.

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, 2008.

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[8] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. *TISSEC*, 12(2):10, 2008.

[9] J. Chinneck and E. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

[10] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *HotDep*, 2009.

[11] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News*, 39(1):265–278, 2011.

[12] A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *SAT*, pages 334–339, 2007.

[13] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *JAIR*, 40(1):701–728, 2011.

[14] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. *ACM SIGOPS OSR*, 43(4):5–10, 2010.

[15] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[16] P. Godefroid. Compositional dynamic test generation. In *ACM SIGPLAN Notices*, volume 42, pages 47–54, 2007.

[17] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223, 2005.

[18] P. Godefroid, M. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, 2008.

[19] J. Jaffar, J. Navas, and A. Santosa. Unbounded symbolic execution for program verification. In *RV*, pages 396–411, 2011.

[20] S. Khurshid, I. García, and Y. Suen. Repairing structurally complex data. *Model Checking Software*, pages 903–903, 2005.

[21] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA*, pages 105–116, 2009.

[22] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[24] G. Luger. *Artificial intelligence: Structures and strategies for complex problem solving.* Addison-Wesley Longman, 2005.

[25] C. Păsăreanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *ASE*, pages 179–180, 2010.

[26] S2E. https://s2e.epfl.ch/.

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

[28] J. E. Smith. A study of branch prediction strategies. In *ISCA*, pages 135–148, 1981.

[29] M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *ISSTA*, pages 183–194, 2010.

[30] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. In *TAP*, pages 134–153, 2008.

[31] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[32] B. Wester, P. Chen, and J. Flinn. Operating system support for application-specific speculation. In *EuroSys*, pages 229–242, 2011.

[33] Yices. http://yices.csl.sri.com/.

[34] Y. Zhang, Z. Chen, and J. Wang. Speculative symbolic execution. In *ISSRE*, 2012, to appear. (http://arxiv.org/abs/1205.4951).