Dynamic Partitioning Strategy to Enhance Symbolic Execution

Brendan A. Marcellino and Michael S. Hsiao
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA 24061 USA

***Abstract*—Software testing is a fundamental part of the software development process. In the context of embedded-software applications, testing can find defects which cause unprecedented risks. The path explosion problem often necessitates one to consider an extremely large number of paths in order to reach a specific target. Symbolic execution can reduce this cost by using symbolic values and heuristic exploration strategies. Although various exploration strategies have been proposed in the past, the number of SMT solver calls for reaching a target is still large, resulting in long execution times for programs containing many paths. In this paper, we present a dynamic partitioning strategy in order to mitigate this problem, consequently reducing unnecessary SMT solver calls as well. Using this strategy on SSA-applied code, the code sections are analyzed in a non-consecutive order guided by data dependency metrics within the sections. Experimental results show that our dynamic strategy can achieve significant speedups in reducing the number of unnecessary solver calls in large programs. More than 1000× speedup can be achieved in large programs over conflict-driven learning techniques.**

***Index Terms*—Symbolic execution, software testing, static analysis, embedded software, partitioning strategies**

I. Introduction

Modern embedded software does not include recursion, in order to avoid stack overflow. Software testing plays a critical role in embedded software particularly for the development process. According to [1], testing makes up roughly 50% of the total development cost. With increasing code sizes, both scalability and computational costs pose tremendous challenges for testing. In order to tackle these issues, symbolic execution have offered some relief by exploring the program state space with symbolic values and heuristic search strategies.

Several symbolic execution techniques [2], [6], [7] have been proposed for improving coverage analysis. Depth-First Search (DFS) strategy, for instance, is a simple and popular technique. It serves as the basis for several exploration strategies as it systematically traverses each path in a program. However, this strategy is generally not scalable to large programs due to the path explosion problem, in which the number of paths grows exponentially with an increasing number of conditional statements. Another popular technique is the Breadth-First Search (BFS) strategy. This strategy has a "Domino Effect" with the previous sections affecting the subsequent sections. As a result, having fewer number of feasible paths in the earlier sections may be advantageous to analyzing those programs that have infeasible path segments early in the code. Another work proposed by Jaffar [7] showed that using interpolation for assisting concolic execution process can prune redundant paths. Such paths can be subsumed if the interpolant is implied as they can be guaranteed to not be buggy. One issue with this approach is that path subsumption is used to skip those paths which can only be guaranteed in programs annotated with assertions.

A DFS-based strategy with conflict backtracking was proposed in [2] using a conflict-driven learning strategy. This strategy makes use of the unsatisfiable (UNSAT) core from the SMT solver to derive any useful information about the conflicting nodes. The idea is when an infeasible path is encountered, it will nonchronologically backtrack to the nearest conflicting nodes from its current position. Using this strategy, many infeasible paths can be skipped to achieve better performance in terms of a reduced number of calls as compared to the basic DFS strategy. Even though such a strategy has better performance than the first technique, extraction of UNSAT cores may be expensive, and the one returned (from possibly several UNSAT cores) may not be optimal.

In this paper, we propose a dynamic partitioning strategy in order to overcome the aforementioned challenges. The idea of partitioning is to divide the paths in a program into several sections so that we can better identify the root cause of any conflict. We introduce metrics to analyze data dependencies among the sections to determine those sections that should be grouped first in our analysis. The sections may not be adjacent or consecutive in their order. As a result, we can potentially eliminate a significant number of unnecessary solver calls and better handle large programs. Because the original code is completely represented in static-single-assignment (SSA) form, our method will not call a feasible path infeasible, even after some intermediate sections are removed. We use Roslyn [8], a .NET compiler platform for static analysis. As for the constraint solver, we use Z3 from Microsoft Research [3]. Results show that our strategy can achieve significant speedups. More than 1000× speedup can be achieved in large programs over conflict-driven learning techniques.

The rest of the paper is organized as follows. Section II discusses some backgrounds and fundamental theories. Section III illustrates the motivation of our work. Section IV describes our dynamic partitioning strategy. Section V discusses the performance of the proposed strategy to previous works. Finally, Section VI provides the concluding summary.

supported in part by NSF grant 1422054.

II. Background

In this section, we provide an overview of symbolic execution and sequential partitioning strategy.

A. Symbolic Execution

In symbolic execution [4], a program is executed using symbolic values instead of concrete values. Upon the execution of a program, a trace consisted of symbolic expressions along the paths will be generated. The conjunction of all symbolic expressions within a trace will form a path constraint. This path constraint can be solved using a number of constraint solvers, such as the Satisfiability Modulo Theories (SMT) solver. If the SMT solver returns SAT (satisfiable), then there would be a set of inputs that satisfy the current path. On the other hand, if the solver returns UNSAT (unsatisfiable), there does not exist any valid inputs that can simultaneously satisfy all constraints along the given path. Even when the path in infeasible, we can still derive useful information to find the root cause of the problems by extracting the UNSAT core which consists of a subset of clauses that the solver has determined to be unsatisfiable. There may exist multiple UNSAT cores for an infeasible path. Typically, a solver will return only one UNSAT core.

Since a path might involve the same variable multiple times, we need to use Single Static Assignment (SSA), which is commonly used in modern compilers, into our work. Using SSA, we can differentiate variables that are being used in different program points. This is also necessary because without SSA, the solver will always return UNSAT for an expression with conflicting constraints involving the same variable.

B. Sequential Partitioning

Sequential partitioning strategy uses BFS technique in order to explore the program state space systematically. In this strategy, code sections are analyzed sequentially which take advantage of infeasible paths discovered in earlier sections. By focusing on the limited set of sections, the reason for any infeasible paths is confined to those sections in question. This is in contrast to conventional learning strategies which consider an entire undivided path and might not return the unsat core that is nearest to the top of the path because there might exist several reasons for an infeasible path. Thus, the sequential partitioning helps the pruning process by indirectly pushing the UNSAT core to near the top of the path. Any conflicting segments in earlier sections will help to reduce a significant number of solver calls for the subsequent sections.

III. Motivation

In order to tackle the scalability challenges, a strategy has been proposed in [2] which uses reachability graph of a program to determine whether a branch needs to be explored and introduces a conflict-driven backtracking strategy. In their approach, an UNSAT core is extracted for every infeasible path encountered. Since all paths that involve the same conflict will be infeasible, we can backtrack immediately to the conflicting node that is at the highest decision level. This can significantly improve the basic DFS strategy by allowing for non-chronological backtracking, eliminating many unnecessary solver calls.

However, there are limitations on such an approach for programs with many data dependencies. Note that the solver may not return the minimum nor the best unsatisfiable core. It will also not generate all possible sets of conflicting clauses from the original formula. Hence, the UNSAT core returned is just one among several from the infeasible path.

An example of CFG is shown in Figure 1 to illustrate the aforementioned problem. Note that nodes with a single successor are assignments, while those nodes having two successors are conditionals. Suppose we are interested in analyzing a particular path $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6$ as highlighted in bold in Figure 1(a). Suppose this path is infeasible, the solver will return UNSAT and produce a set of conflicting clauses (UNSAT core) which is denoted as node 1 and 4 (grayed out) shown in Figure 1(b). Using the non-chronological backtracking strategy, it will jump backward multiple levels to the node at the highest decision level according to the unsat core, which is node 4. The next step is to store all conflicting nodes into a database so that it will skip any paths that contain any of the unsat constraints. It will then change the direction by negating node 4 in order to reach node 5 and continue the searching process from this point onward.

However, there may exist other reasons that can explain the original infeasible path. One of which is the combination of node 0 and 1 as shown in Figure 1(c). If this UNSAT core was returned by the solver instead, it will allow the search to backtrack more levels to node 1. After negating this node, it will start analyzing in another direction towards node 8. The difference in the number of solver calls between Figure 1(b) and 1(c) can be significant. This situation will become even pronounced if we add more data dependencies in the program because there might be nested problems within the analyzed path. Tackling the problems in data dependencies in large programs becomes our motivation since the SMT solver may not return the best UNSAT core.

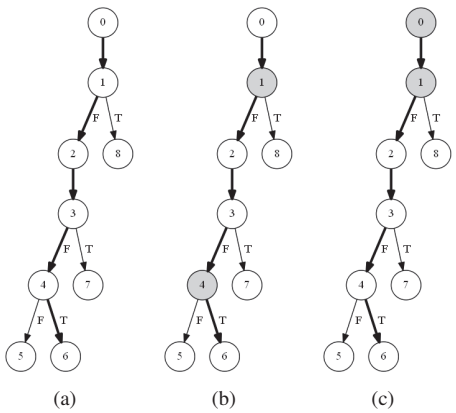


Fig. 1. An example of UNSAT core problems related to data dependencies

IV. Dynamic Partitioning Technique

Partitioning a program is critical in this work because our dynamic strategy will rely on the partition results. Poor partitions will affect the results as they are ineffective in separating valuable information. Partitioning is performed based on the primary control statements of the programs. Any loops are unrolled and function calls are inlined.

In order to make the technique effective, we need to determine those sections that are best to analyze together to maximize pruning opportunities. We introduce metrics to determine which sections should be grouped and analyzed together. The metrics use the amount of data dependencies because there might be a higher chance of getting infeasible paths in the parts of the code that have many data dependencies. In addition, our strategy uses the extracted UNSAT cores generated by the SMT solver to obtain the infeasible segments. All infeasible segments will be recorded to avoid making the same mistake in the future search.

In our work, data dependencies are defined as the relationships between the assignment and conditional expressions of a variable in a program. A path that assigns some variables in certain statements and uses them in other conditional expressions along the path is more likely to be infeasible, when compared to paths that do not have data dependencies. Static analysis must be used to analyze data dependencies which utilizes the abstract syntax tree (AST) created by Rosyln [8].

First, data dependency is computed along each path in a given section in the initialization phase. If the amount of data dependency in a section is above a given threshold, then the corresponding section will be analyzed, and only those feasible paths are kept. Note that conflict-driven learning is used in this initialization phase in order to remove any paths that have conflicting segments. The next step is to compute data dependency for each section pair. The sections with the highest levels of data dependency will be chosen and merged into a single group. Note that the original code is already in SSA form, so the merged code has the correct versions of the variable instances in the original context. Since the union of both sections is considered as a new section, data dependency should be re-calculated. The complete algorithm of dynamic partitioning is described in Algorithm 1.

Algorithm 1 Dynamic Partitioning

```
1: Initialize and transform programs to CFG
2: Partition programs into sections
3: for each section do
4:    PathComb ← store all path combinations
5:    α ← store its metric along with PathComb
6: end for
7: for each α > Threshold do
8:    FilterPath ← store PathComb within α
9:    if FilterPath does not contain UnsatPath then
10:      solve() and update all feasible paths
11:   end if
12: end for
13: Ω ← list all sections
14: while |Ω| > 1 do
15:   for each pairwise combinations in Ω do
16:      β ← store metrics for all possible combinations
17:   end for
18:   SectionComb ← find the largest metrics in β
19:   FilterPath ← store path combinations in SectionComb
20:   solve() and update all feasible paths
21:   Merge all sections in SectionComb and update Ω
22: end while
```

A. Individual Sections

Dynamic partitioning aims to group sections in a non-sequential order. To do that, we need to determine which sections should be analyzed first. One approach is to determine which section has the potential for having the largest number of infeasible paths. Data dependency plays an important role in determining such predictions since the chance of getting infeasible paths increases as the expressions are more dependent in each other along the paths.

Recall that data dependency exists in a path when some variables are used in conditional expressions and the same variables are assigned in other statements along this path. The calculation is based on the percentage of paths having data dependencies involving more than one variable. As the percentage increases, there is more data dependencies affecting those sections. Users can control the minimum threshold. A lower threshold will allow more sections for merger, while a higher threshold will increase the prediction accuracy by considering fewer sections at a time. In our experiments, the optimum threshold to achieve better performance was 80%.

B. Section Pairs

Analysis in individual sections gives a good start in reducing the number of feasible paths. We now combine those feasible paths in a section with other feasible paths from different sections. However, which other section should be brought in? In this work, we consider all possible section pairs since it is only quadratic in complexity.

Section pairs that have the largest data dependency will most likely contain infeasible paths. Thus, this step is to merge these sections into a group consisted of all path combinations with their components as well. The merged group is named after the earliest section in the group. For example, suppose we have four sections, and let sections 1 and 3 have the greatest data dependency. After the analysis, they are merged into one single group, called section 1. Section 3 is then deleted from the database, as it is absorbed into section 1. Now, the data dependency needs to be computed for section pairs (1, 2), (1, 4), and (2, 4). Since the group ordering is important, we created a novel algorithm to divide and merge all assignments in an orderly manner when analyzing any selected pair of sections. When merging the sections, all the involved sections should be ordered correctly (after SSA is applied). Hence, the analysis will still be accurate even though the sections are added non-sequentially.

C. Coverage Analysis

Our dynamic approach can achieve a full coverage similar to existing methods. However, we can achieve full coverage much more quickly. Again, we note that all programs should be translated into SSA form before performing the analysis.

Theorem 4.1: If a contiguous path in the original code is feasible, our dynamic approach will never declare any subset of this path as infeasible.

Proof Given a feasible contiguous path, the conjunction of all symbolic expressions along the path (in SSA form) will form its corresponding path constraint. If this path constraint is feasible, then there exists a set of valid inputs that can satisfy all constraints on this path. Thus, any subset of this path constraint will also be satisfied by this valid input set.

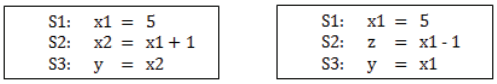
Suppose there is a formula $f = (\text{S1, S2, S3})$ which intersects with 3 sequential sections (S1, S2, and S3) in a program. If f is satisfiable, then any subset of sections along this path is also satisfiable, even if an intermediate section is not considered in f.

Theorem 4.2: If a contiguous path is infeasible, our dynamic approach may declare any subset of this path as either feasible or infeasible.

Proof Given an infeasible contiguous path, its path constraint would be unsatisfiable. For every unsatisfiable instance, there exist one or more unsatisfiable cores. If an unsat core is fully contained in the set of constraints for a subset of the path in question, then this subset would also be unsatisfiable. However, if no unsat core is contained in this subset, this subset may be declared as feasible.

Consider an infeasible contiguous path whose path constraint is $f = (\text{S1, S2, S3})$. Suppose an UNSAT core involves nodes in S1 and S2 as they cannot be satisfied together. Hence, a path analysis involving S1 and S3 might yield satisfiable since there is not enough information to make such a decision. This is not a bad result, since this shows that our approach is conservative. That is, we will never mistake a feasible path to be an infeasible one.

In order to understand how arbitrary sections can be merged, we will analyze two cases as shown in Figure 2. In both examples shown in the figure, S1 and S3 will be intentionally merged into a single section to be analyzed first, leaving S2 to be analyzed later. In both cases, variable x is either reassigned or used in all sections, making this variable is heavily dependent across the sections. Before partitioning is performed, the program is first converted to SSA form. In Figure 2(a), the path constraint after merging S1 and S3 would be $(x1 = 5) \wedge (y = x2)$. In this case, $x2$ is treated as an independent input variable, since its definition was in S2 (currently excluded in the partition). As a result, the analysis will be based on an over-approximation because the real $x2$ is not being considered. On the other hand, Figure 2(b) shows that variable $x1$ is being used in both sections S2 and S3. The use of $x1$ in S2 will have no effect on the merging S1 and S3 since there is no data dependency among all sections. Thus, our approach is sound as it will eventually identify all infeasible paths as well.



(a) x is reassigned in S2 (b) x is used in both S2 and S3

Fig. 2. An example of how arbitrary sections can be merged after translating into SSA form

Our dynamic approach also orders the sections to be analyzed. This must be done because the program runs in a sequential order. Changing the order of the sections must not change the structure of the programs. Given an example in Figure 2, after merging S1 and S3, we will later add S2 into our analysis and place it between S1 and S3. Therefore, when this analysis is conducted, the path will traverse from S1 to S3 sequentially as in their original order.

V. Experimental Results

The experiments were performed on a 3.2GHz Intel Xeon machine with 8GB of RAM. They were tested on a number of control-intensive C# test cases. Any loops are unrolled and we analyze only the body of the loop to demonstrate the scalability of the proposed approach. The first set of test cases are handcrafted with random assignments and the number of conditionals as the controlling factor. The values of several variables are modified to be used in later parts of the programs to create data dependencies among those variables. A second set of test cases consisted of real-world C# programs are also reported. Tables I and II consist of the same test cases that are automatically extracted into 3 and 6 sections, respectively, using the partitioning technique described in Section IV. # **conds** denotes the number of conditionals in the test cases. We compared the number of SMT solver calls by our partitioning strategy with the traditional Depth-First Search (DFS) strategy, the sequential partitioning strategy and the previous work which uses conflict-driven backtracking strategy (DFS + CDL) [2]. The speedup was calculated to see the impact of our dynamic strategy and the sequential strategy to (DFS + CDL) [2]. In order to make a fair comparison, we used the same SMT solver, Z3 [3], to evaluate all strategies in each test cases.

Table I compares the results between our partitioning strategies (code divided into 3 sections) with (DFS + CDL). The DFS strategy is used for evaluating the total number of paths available in each test cases. In other words, the computation cost for DFS would be the worst among others due to the lack of search optimization and will not be considered for computing the speedup. Instead, the speedup achieved is compared only against DFS + CDL.

It can be seen from Table I that even sequential partitioning can outperform the conflict backtracking strategy in most cases because it helps to restrict the UNSAT core to near the top of the infeasible path. Our dynamic partitioning achieves another substantial speedup over sequential partitioning for many instances. The first test, tc1, is one of the smallest programs

TABLE I
RESULTS FOR 3-SECTION PARTITIONS

Test Case	# conds	DFS	DFS+CDL		Sequential			Dynamic		
			calls	Time (s)	calls	Time (s)	Speedup	calls	Time (s)	Speedup
tc1	24	5184	129	5.4	88	5.42	-	47	4.28	1.26
tc2	24	5184	471	13.8	350	12.36	1.16	114	6.34	2.17
tc3	32	50176	7612	138	204	8.1	17.03	77	5.21	26.48
tc4	44	321489	25269	606	639	22.07	27.45	140	6.7	90.44
tc5	48	531441	29695	648	405	13.95	46.45	127	6.19	105
tc6	48	531441	33515	792	729	24.86	31.85	153	7.06	112
tc7	52	810000	44604	1044	850	30.62	34.09	169	7.26	144
tc8	54	705600	68755	1554	1320	32.39	47.97	212	8.72	178
tc9	58	1.1M	87880	1992	1491	57.58	34.59	225	8.87	225
tc10	62	1.8M	145742	3690	1944	80.08	46.07	261	9.71	380
tc11	64	2.2M	186005	4578	2166	95.09	48.14	286	10.58	433
tc12	68	3.4M	-	-	2386	123	> 58.53	350	12.94	> 556

'M': millions '-': timeout of 2 hours

TABLE II
RESULTS FOR 6-SECTION PARTITIONS

Test Case	# conds	DFS	DFS+CDL		Sequential			Dynamic		
			calls	Time (s)	calls	Time (s)	Speedup	calls	Time (s)	Speedup
tc1	24	5184	129	5.4	47	3.66	1.59	44	3.63	1.61
tc2	24	5184	471	13.8	201	6.9	1.99	127	5.49	2.52
tc3	32	50176	7612	138	69	4.15	33.25	59	4.01	34.41
tc4	44	321489	25269	606	159	6.02	101	111	5.14	118
tc5	48	531441	29695	648	99	4.66	139	105	4.7	138
tc6	48	531441	33515	792	171	6.39	124	111	5.13	154
tc7	52	810000	44604	1044	184	6.65	157	147	6.24	167
tc8	54	705600	68755	1554	296	9.36	166	202	7.18	216
tc9	58	1.1M	87880	1992	313	9.84	202	222	7.97	250
tc10	62	1.8M	145742	3690	378	11.68	316	256	8.65	427
tc11	64	2.2M	186005	4578	412	12.6	363	235	8.42	544
tc12	68	3.4M	-	-	428	13.59	> 530	255	9.1	> 791

'M': millions '-': timeout of 2 hours

in our experiments with only 24 conditional statements. The conflict backtracking approach performs worse than our dynamic strategy but better compared to the sequential strategy. Results for all other test cases showed speedups from our approach, indicating that the non-partitioning strategy does not scale for larger programs. With three sections, the sequential partitioning strategy can achieve up to nearly 2 orders of magnitude speedup while the dynamic partitioning strategy was able to achieve up to nearly 3 orders of magnitude speedup over DFS + CDL. In fact, as the programs get bigger and data dependencies starts playing an important role, dynamic approach achieved greater speedups over the DFS + CDL strategy compared to the sequential approach. It can also be seen that the differences on the number of solver calls between our strategy and DFS + CDL grew wider. The largest speedup was seen in tc12, in which the dynamic approach was faster more than $556\times$ than DFS + CDL.

Table II reports results for the same test cases that are divided into 6 sections. Partitioning strategies showed effectiveness in dealing with large programs by having significant speedups in most cases. The biggest speedup was achieved by dynamic approach in tc12 that has more than 3.4 million paths which is able to perform $791\times$ faster than DFS + CDL. For this test case, the DFS + CDL timed out after 2 hours.

In some cases, even when our partitioning strategies have the same number of solver calls as DFS+CDL, the execution times can be quite different. This is because with partitioning, many of the paths that need to be solved involve only a subset of the sections rather than the complete path, which result in smaller formula with fewer path-predicates. In tc8 from Table I and tc9 from Table II, for instance, the total number of solver calls for both test cases in our dynamic strategy were 212 and 222, respectively. However, tc9 had a smaller execution time compared to tc8 because more predicates were included on average in Table I than in Table II due to larger section sizes in the 3-section case.

TABLE III
RESULTS FOR REAL-WORLD C# PROGRAMS

Test Case	# conds	# sections	DFS	DFS+CDL		Sequential			Dynamic		
				calls	Time (s)	calls	Time (s)	Speedup	calls	Time (s)	Speedup
score	11	11	2048	276	8.83	22	2.99	2.95	22	3.3	2.67
score_2	22	22	4.1M	-	-	44	5.37	$>$ 1340	44	5.2	$>$ 1387
sparsemtx	16	16	65536	145	10.08	32	3.63	2.77	31	3.37	2.99
sparsemtx_2	32	32	4.3B	-	-	64	5.27	$>$ 1366	63	4.89	$>$ 1472
sequence	42	7	823543	5538	143	49	3.6	39.72	42	3.42	41.81
sequence_2	84	14	$>$ 500B	-	-	98	6.89	$>$ 1044	91	5.94	$>$ 1212
bubblesort	28	28	268M	77	4.61	56	3.68	1.25	55	3.76	1.22
bubblesort_2	56	56	$>$ 1000B	229	14.47	112	9.23	1.56	111	8.87	1.63
shellshort	61	61	536M	218	12.85	122	8.94	1.43	121	7.65	1.68
shellshort_2	122	122	$>$ 1000B	751	62.51	244	34.87	1.81	243	32.38	1.93

'M': millions 'B': billions '-': timeout of 2 hours

From Tables I and II, we see that increasing the number of sections from 3 to 6 generally will reduce the total number of solver calls, which also significantly reduce the execution times needed. By having fewer sections, the total number of individual paths that need to be analyzed will be larger, thus increasing the overall analysis time. In tc12, for instance, the number of solver calls in Table II was only 428 (sequential strategy), which is $5\times$ smaller than 2386 in Table I. This resulted in a $9\times$ faster execution than the 3-section partition.

The execution times taken for DFS + CDL for most test cases are generally longer than our strategy. In Table II, the partitioning strategies took less than 15 seconds for all test cases while DFS + CDL took less than a minute for only 2 test cases (tc12 and tc22). Starting from tc32, the number of calls performed by DFS + CDL increased significantly which also significantly increased the execution times.

In order to see the effect of our strategy in real programs, we chose five C# programs as shown in Table III and report the performance of both the previous work (DFS + CDL) and sequential and dynamic strategies. These programs consist of arrays, intensive loops and arithmetic/relational operations. Since our strategy analyzes the loop-free portions of the code, any loops need to be unrolled before being able to start the analysis. We also duplicated their structures to assess the performance of all strategies in large programs. # **sections** denotes the number of partitions in the test cases.

It can be seen from Table III that both sequential and dynamic strategies can outperform previous approaches in all cases. In some cases, such as score, sparsemtx, and sequence, when the loop unrolling is doubled, DFS + CDL was timed out resulting in a speedup of more than $1000\times$ with both strategies. This shows that the amount of savings could increase exponentially with an increasing size of the program.

Score consists of multiple cascaded IF-ELSE statements without any nested conditions. Sequential and dynamic strategies have the same number of solver calls because any unreachable IF condition will result in skipping the same number of subsequent unnecessary solver calls. However, DFS + CDL was not able to produce the same result as this strategy does not always get the best UNSAT core for an infeasible path. In addition, there is no nested conditions in bubblesort and shellsort making sequential and dynamic strategies to produce similar results. The speedup was similar even after duplicating the structures since the previous work was still able to reduce many solver calls and avoid many unsat core problems.

VI. CONCLUSION

In this paper, we presented a dynamic partitioning strategy for improving symbolic execution by reducing the number of unnecessary solver calls while still covering all paths. We have shown that our strategy is able to achieve significant speedups especially in large programs. Our approach performs better among all strategies due to its ability to perform in non-sequential ordering of program sections based on data dependency metrics. The number of solver calls were significantly reduced, resulting in more than three orders of magnitude speedup in some cases.

REFERENCES

[1] G. J. Myers, *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.

[2] S. Krishnamoorthy, M. Hsiao, and L. Lingappan, "Tackling the Path Explosion Problem in Symbolic Execution-driven Test Generation for Programs" *Asian Test Symposium*, pp. 59-64, 2010.

[3] L. de Moura and N. Bjorner, "Z3: An Efficient SMT Solver," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 4963, pp. 337-340, 2008.

[4] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, vol. 19, pp. 385394, 1976.

[5] L. de Moura and N. Bjorner. Satisfiability modulo theories: introduction and applications. *Comm. of the ACM*. Vol. 54, pp. 69-77, 2011.

[6] K. L. McMillan, "Lazy Annotation for Program Testing and Verification," in *Proc. Computer Aided Verification*, pp. 104-118, 2010.

[7] J. Jaffar, V. Murali and J. A. Navas, "Boosting concolic testing via interpolation," in *Proc. Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering,* pp. 48-58, 2013.

[8] ".NET Compiler Platform ("Roslyn")". https://roslyn.codeplex.com.