IJCSE International Journal of Computer Sciences and Engineering Open Access
Survey Paper Vol-6, Special Issue-8, Oct 2018 E-ISSN: 2347-2693

A BRIEF SURVEY ON SYMBOLIC EXECUTION TEST-SELECTION TECHNIQUES

[1*]K.K. Nivethithaa, [2]V. Krishnapriya

[1,2]Sri Ramakrishna College of Arts and Science for Women Tamil Nadu, India

Available online at: www.ijcseonline.org

Abstract- Symbolic execution techniques decrease the cost of path redundancy by choosing a separation of an existing test suite to use in retesting a customized program. Over the history, Eliminating Path Redundancy via Postconditioned Symbolic Execution techniques has been described in the literature. This paper aims to present a brief survey on symbolic executions in black-box and white-box regression testing under the Software testing and learning techniques that are in use in today's software engineering of verification and validation tasks. Number of comparative study has been performed to evaluate the performance of predictive accuracy on the test cases and the outcome discloses that Bidirectional Symbolic Analysis for Effective Branch Testing method outperforms having better performance other predictive methods are not performing well.

Keywords: Symbolic execution, testing and debugging, Parallel symbolic execution, AEG.

I. INTRODUCTION

Testing software is a very important and challenging activity. Nearly half of the software production development cost is spent on testing. The main objective of software testing with clustering approach is to eliminate as many errors as possible to ensure that the tested software meets an acceptable level of quality.

Software bugs pervade every level of the modern software stack, degrading both stability and security. Current practice attempts to address this challenge through a variety of techniques, including code reviews, higher-level programming languages, testing, and static analysis. While these practices prevent many bugs from being released to the public, significant gaps remain [3].

Symbolic execution has been shown to be largely successful in program verification, testing and analysis [1-2]. It is a method for program reasoning that uses symbolic values as inputs instead of actual data, and it represents the values of program variables as symbolic expressions on the input symbolic values. As symbolic execution reaches each program point along different paths, different symbolic states' are created. For each symbolic state, a path condition is maintained, which is a formula over the symbolic inputs built by accumulating constraints that those inputs must satisfy in order for execution to reach the state. A symbolic execution tree depicts all executed paths during the symbolic execution.

The symbolic execution suffers from the well-known path explosion problem since the number of distinct execution paths through a program is often exponential in the number of if-statements or, in the worst case, infinite. Consequently, while symbolic execution often examines orders of magnitude more paths than traditional testing, it typically fails to exhaust all interesting paths. In particular, it often fails to reach code deep within a program due to complexities earlier in the program. Even when the tool succeeds in reaching deep code, it considers only the input values satisfying the few paths that manage to reach this code.

Attaining the target required facing key challenges in scalability across several dimensions:

- **Symbolic execution:** how to efficiently perform symbolic execution on x64 execution traces with billions of instructions and tens of thousands of symbolic variables for applications with millions of lines of code.
- **Constraint generation and solving:** how to generate, solve and manage billions of constraints.
- **Long-running state-space searches:** how to perform systematic state-space explorations effectively for weeks or months at a time.
- **Diversity:** how to easily configure, check and monitor white box testing so that it is applicable to hundreds of diverse applications.
- **Fault tolerance and always-on usage:** how to manage hundreds of machines running white box testing 24/7 with as little down-time as possible.

This paper explores strategies for Symbolic Execution test cases based on eliminating path Redundancy class decision models. Such representations have been usually used to partition the input domain of the system being tested, which in turn is used to choose and create system test cases so as to attain certain strategies for partition coverage. Such models are widely applied for black-box system testing for database applications and are therefore a natural and practical choice in our context.

II. RELATED WORK

***M. Staats and C. S. Pasareanu* (2010)** [4] proposed a technique, Simple Static Partitioning, for parallelizing symbolic execution. The technique uses a set of pre-conditions to partition the symbolic execution tree, allowing us to effectively distribute symbolic execution and decrease the time needed to explore the symbolic execution tree. The proposed technique requires little communication between parallel instances and is designed to work with a variety of architectures, ranging from fast multi-core machines to cloud or grid computing environments.

***T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley* (2011)** [5] discussed the automatic exploit generation challenge is given a program, automatically find vulnerabilities and generate exploits for them. The authors presented AEG, the first end-to-end system for fully automatic exploit generation. They used AEG to analyze 14 open-source projects and successfully generated 16 control flow hijacking exploits. Two of the generated exploits (expect-5.43 and htget-0.93) are zero-day exploits against unknown vulnerabilities. The AEG challenge consists of six components:

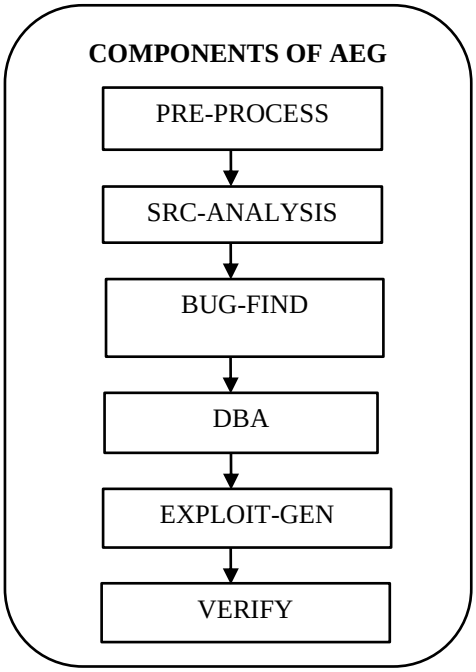


Fig.1: Components of AEG

***Duc-Hiep Chu and Joxan Jaffar* (2012)** [6] presented a general method for its application, restricted to verification of safety properties, but without any prior knowledge about global symmetry. They started by using a notion of weak symmetry which allows for more reduction than in previous notions of symmetry. This notion is relative to the target safety property. The key idea is to perform symmetric transformations on state interpolation, a concept which has been used widely for pruning in SMT and CEGAR. The method naturally favors "quite symmetric" systems: more similarity among the processes leads to greater pruning of the tree.

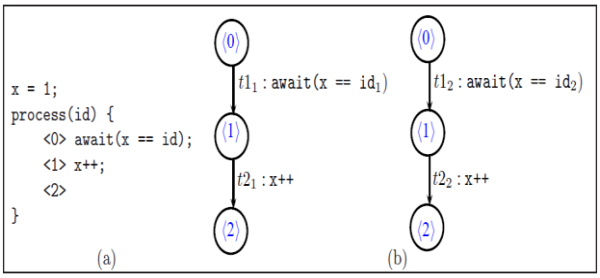


Fig.2: (a) A parameterized system (b) Its 2-process concretization

In Figure 2(a). Note the (local) program points in angle brackets. Figure 2(b) "concretizes" the processes explicitly. Note the use in the first process of a local variable id1 which is not writable in the process, and whose value is 1. Similarly for id_2 in the other process.

***E. Bounimova, P. Godefroid, and D. A. Molnar* (2013)** [7] described the key challenges with running whitebox fuzzing in production. To given principles for addressing these challenges and describe two new systems built from these principles: SAGAN, which collects data from every fuzzing run for further analysis, and JobCenter, which controls deployment of our whitebox fuzzing infrastructure across commodity virtual machines. The architecture of SAGE (Scalable, Automated, Guided Execution) in figure 3.

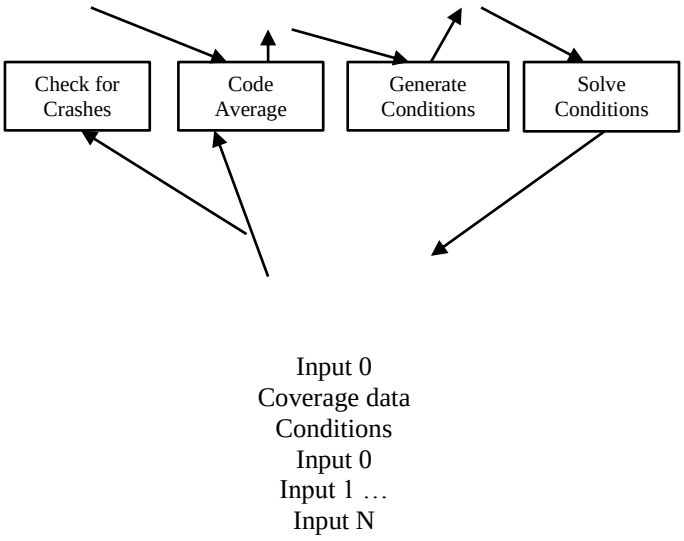


Fig.3: Scalable, Automated, Guided Execution

***Duc-Hiep Chu et.al* (2014)** [8] presented a systematic method which speculates that infeasibility may be temporarily ignored in the pursuit of better information about the path in question. This speculation does not lose the intrinsic benefits of symbolic execution because its operation shall be bounded. They argued that the trade-off between this 'enhanced learning' and incurring additional cost (which in principle may not be productive) is in fact in favor of speculation. Finally, they demonstrated with a state-of-the-art

system on realistic benchmarks that this method enhances symbolic execution by a factor of 2 or more.

***D. A. Ramos and D. R. Engler* (2015)** [9] presented UC-KLEE, a novel, scalable framework for checking C/C++ systems code, along with two use cases. First, we use UC-KLEE to check whether patches introduce crashes. To checked over 800 patches from BIND and OpenSSL and find 12 bugs, including two OpenSSL denial-of-service vulnerabilities. They also verified (with caveats) that 115 patches do not introduce crashes. Second, we use UC-KLEE as a generalized checking framework and implement checkers to find memory leaks, uninitialized data, and unsafe user input. To evaluate the checkers on over 20,000 functions from BIND, OpenSSL, and the Linux kernel, find 67 bugs, and verify that hundreds of functions are leak free and that thousands of functions do not access uninitialized data.

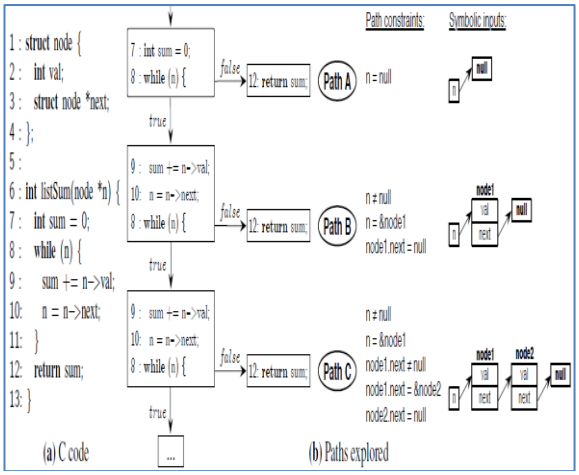
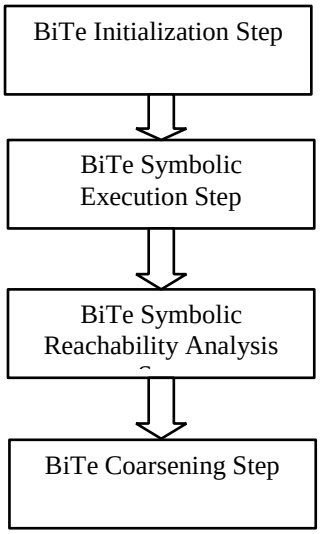


Fig. 4: Example code fragment analyzed by UC-KLEE.

The authors illustrated lazy initialization by explaining how UC-KLEE executes the example function listSum in Figure 4(a), which sums the entries in a linked list. Figure 4(b) summarizes the three execution paths we explore. For clarity, we elide error checks that UC-KLEE normally performs at memory accesses, division/remainder operations, and assertions.

***M. Baluda, G. Denaro, and M. Pezze* (2016)** [10] proposed a new approach that combines symbolic execution and symbolic reachability analysis to improve the effectiveness of branch testing. The approach embraces the ideal definition of branch coverage as the percentage of executable branches traversed with the test suite, and proposed a new bidirectional symbolic analysis for both testing rare execution conditions and eliminating infeasible branches from the set of test objectives. The approach is centered on a model of the analyzed execution space. The model identifies the frontier between symbolic execution and symbolic reachability analysis, to guide the alternation and the progress of bidirectional analysis towards the coverage targets.

Bidirectional Test Case Generation (BiTe) exploits bidirectional symbolic analysis to improve and refine branch coverage, by both covering not-yet-covered branches and identifying infeasible branches to be pruned from the coverage domain.



To execute the many program branches that depend on the inputs that match the value of the keyword, the symbolic execution must solve the path condition that leads to executing the then branch of the if statement at line after returning from the invocation of strcmp(kw, word), for each invocation of function GetKeyword.

III. COMPARISON ANALYSIS

This paper aims to collect and consider papers that deal with different Symbolic Execution testing techniques. Our objective is not to undertake a constraints review, but quite to provide a broad state-of-the-art view on these related fields. Many different approaches have been projected to assist Symbolic execution, testing and debugging, testing tools, which has mentioned in a body of literature that is spread over a wide variety of fields and periodical locations. The majority of comparison study has been available in the software engineering domain, and particularly in the software testing and software maintenance literature. However, the Eliminating Path Redundancy via Post-conditioned Symbolic Execution testing literature also overlaps with those of programming language analysis, empirical software engineering and software metrics.

Table 1: SUMMARY TABLE FOR COMPARISON OF ELIMINATING PATH REDUNDANCY VIA POSTCONDITIONED SYMBOLIC EXECUTION TECHNIQUES

Title	Algorithm	Key-Idea	Techniques	Results	Performance
Parallel symbolic	Parallelizing Java	Parallel symbolic	Simple Static	Analysis time and	Speedups of 30% to

execution for structural test generation (2010) [4]	Pathfinder	execution	Partitioning.	Modified Condition/Decision Coverage test generation time using six case examples.	90% of the maximum speed up.
AEG: Automatic Exploit Generation (2011) [5]	AEG exploit generation algorithm and Stack-Overflow Return-to-Stack Exploit Predicate Generation Algorithm.	Four different preconditions for efficient exploit generation. (**None**, **Known Length**, **Known Prefix** and **Concolic Execution**)	Traditional Symbolic Execution for Bug Finding.	Successfully generated 16 controlflow hijack exploits, two of which were against previously unknown vulnerabilities.	The quickest generated an exploit was 0.5s for iwconfig (with a length precondition)
A Complete Method for Symmetry Reduction in Safety Verification (2012) [6]	Symmetry Reduction Algorithm.	To exploit weak symmetry completely.	Top-down techniques.	The coverage relation as the form of backward learning in a recursive manner.	The technique is more efficient both in space and time.
Billions and Billions of Constraints: Whitebox Fuzz Testing in Production (2013) [7]	White box fuzz testing.	Several data analyses which led to short-term, concrete actions that improved white box fuzzing.	Scalable, Automated, Guided Execution (SAGE)	To run whitebox fuzz testing for weeks, and now even months, with low effort and cost.	99.21% of tasks took less than 2000 seconds to solve all their constraints.
Lazy Symbolic Execution for Enhanced Learning (2014) [8]	Lazy algorithm.	Forward symbolic execution.	Symbolic Execution and Enhanced Learning.	The number of decisions seems to be a good possibility.	The verification time 10919 (in seconds)
Under-Constrained Symbolic Execution: Correctness Checking for Real Code (2015) [9]	Under-constrained symbolic execution.	Finding bugs, symbolic execution, considers all possible inputs to a program.	Path pruning.	To examine a program's execution at runtime and flag errors.	To found a total of 79 bugs, including two OpenSSL denial-of-service vulnerabilities.
Bidirectional Symbolic Analysis for Effective Branch Testing (2016) [10]	Symbolic Execution Step and Symoblic Reachability Analysis Step.	To measure the thoroughness of test cases.	Generalized Control Flow Graph (GCFG).	To increase the likelihood of revealing subtle failures and to provide consistent data for managing the overall quality process.	To cover most feasible branches (from 66% to 99%).

III. CONCLUSION

This paper presents a brief survey about Symbolic Execution techniques for Eliminating Path Redundancy testing discussed with the different categories. This survey can be classified into Parallel symbolic execution, AEG, Symmetry Reduction in Safety Verification, Whitebox Fuzz Testing, Lazy Symbolic Execution Under-Constrained Symbolic Execution and Bidirectional Symbolic Analysis for Effective Branch Testing. To concluded the discussion on symbolic execution algorithms with Path Redundancy testing by a comparative study with black-box and white box regression category. It also discussed the concept of finding bugs and symbolic execution, considers all possible inputs to a program measures which proves to be the most important criteria for Eliminating Path Redundancy.

The further work enhanced and expanded for the Symbolic execution technique for heuristics based on static program analysis can make the pruning more efficient algorithm.

IV. REFERENCES

[1] J.Jaar, A. E. Santosa, and R. Voicu. An interpolation method for clp traversal. In CP, 2009.
[2] K. L. McMillan. Lazy annotation for program testing and verification. In CAV, 2010.
[3] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex sys tems programs. In Proc. of Symp. on Operating Systems Design and Impl (OSDI) (2008).
[4] M. Staats and C. S. Pasareanu, "Parallel symbolic execution for structural test generation," in Proc. Int. Symp. Softw. Testing Anal., 2010, pp. 183–194.
[5] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in Proc. USENIX Symp. Netw. Distrib. Syst. Secur., Feb. 2011, pp. 283–300.

[6] D.-H. Chu and J. Jaffar, "A complete method for symmetry reduction in safety verification," in Proc. Int. Conf. Comput. Aided Verification, 2012, pp. 616–633.
[7] E. Bounimova, P. Godefroid, and D. A. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in Proc. 35th Int. Conf. Softw. Eng., 2013, pp. 122–131.
[8] D. Chu, J. Jaffar, and V. Murali, "Lazy symbolic execution for enhanced learning," in Proc. 5th Int. Conf. Runtime Verification, 2014, pp. 323–339.
[9] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in Proc. 24th USENIX Secur. Symp., 2015, pp. 49–64.
[10] M. Baluda, G. Denaro, and M. Pezze, "Bidirectional symbolic analysis for effective branch testing," IEEE Trans. Softw. Eng., vol. 42, no. 5, pp. 403–426, May 2016.

Authors Profile

Ms. K.K.Nivethithaa, Pursuing Mphil Research Degree in Sri Ramakrishna College of Arts and Science for Women at Coimbatore. She did her UG and PG degree in Sri Ramakrishna College of Arts and Science for Women at Coimbatore.

Dr. V.KrishnaPriya is presently, the Professor and Head-PG,School of Computing at Sri Ramakrishna College of Arts and Science at Coimbatore. She received her Ph.D. in Computer Science from Mother Theresa University, Kodaikanal. She holds her Masters in Computer Science (MCA) and Bachelors in Chemistry from Bharathiar University. She has published papers in 15 International journals and 15 National Journals and has more than 15 presentations in International and National conferences to her credit. Has 21 years of academic experience and has held various positions at Sri Ramakrishna College of Arts and Science for Women.

