A Survey of Symbolic Execution Tools

Salahaldeen Duraibi *dura6540@vandals.uidaho.edu*
[a] *Computer Science Department*
University of Idaho
Moscow, ID, 83844, USA
[b] *Computer Science Department*
Jazan University
Jazan, 45142, Saudi Arabia

Abdullah Mujawib Alashjaee *alas0145@vandals.uidaho.edu*
[a] *Computer Science Department*
University of Idaho
Moscow, ID, 83844, USA
[b] *Computer Science Department*
Northern Borders University

Jia Song *jsong@uidaho.edu*
Computer Science Department
University of Idaho
Moscow, ID, 83844, USA

Abstract

In the software development life cycle (SDLC), testing is an important step to reveal and fix the vulnerabilities and flaws in the software. Testing commercial off-the-shelf applications for security has never been easy, and this is exacerbated when their source code is not accessible. Without access to source code, binary executables of such applications are employed for testing. Binary analysis is commonly used to analyze on the binary executable of an application to discover vulnerabilities. Various means, such as symbolic execution, concolic execution, taint analysis, can be used in binary analysis to help collect control flow information, execution path information, etc. This paper presents the basics of the symbolic execution approach and studies the common tools which utilize symbolic execution in them. With the review, we identified that there are a number of challenges that are associated with the symbolic values fed to the programs as well as the performance and space consumption of the tools. Different tools approached the challenges in different ways, therefore the strengths and weaknesses of each tool are summarized in a table to make it available to interested researchers.

Keywords: Symbolic Execution, Concrete Execution, Concolic Execution, Binary Analysis.

1. INTRODUCTION

In cases where applications are analyzed for defects and source code is not available, software analysts have to conduct analysis at the binary code level of the application. Engaging binary code for software analysis is referred to as binary analysis. It is commonly used for error identification, reverse engineering, and security analysis. In addition, binary analysis is well known for its use for discovering vulnerabilities in software, and this paper focuses on that aspect of the binary analysis. To reveal vulnerabilities, disassembly of the binary executable needs to be done first and then the vulnerability patterns, such as buffer overflow, can be recognized.

Conducting binary analysis is challenging, because a great deal of useful information, such as symbolic information, data types, program structures, is not carried to the binary code. What is more, in the early days of using binary code analysis, analysts used to have difficulty distinguishing between data and code in binary, because data fragments and executable code are mixed in the binaries [1]. Therefore, researchers in the domain of software testing have proposed a number of binary analysis techniques, including taint analysis, symbolic execution, and concolic execution.

Taint analysis is used for information flow tracking, data entering from some specific sources such as user input, application APIs or network interfaces are marked as tainted (untrusted). Then the propagations of the tainted data are tracked throughout the program and the uses of the tainted data are carefully checked. There are two ways of performing taint analysis, static taint analysis and dynamic taint analysis. Static taint analysis tools usually conduct the analysis in a controlled environment where the data are monitored before run time [2]. Tools developed based on static taint analysis can offer better code coverage in their analysis compared to dynamic taint analysis [2]. However, such tools suffer in that they cannot detect runtime security defects of applications. Static taint analysis can be used for different aspects of security analysis including data leak analysis [3], digital forensics [4], web application vulnerability analysis [5, 6]. On the other hand, dynamic taint analysis is a principled approach for tracking information flow during program execution. Different from the static taint analysis that needs for its analysis to run in a confined environment, dynamic taint analysis usually conducts analysis when the application is running in its intended environment. Moreover, dynamic taint analysis can be implemented within the hardware level of a system to conduct analysis [7-10], or at the software stack by either using the source code [11-15] or binary code [16]. However, since dynamic taint analysis conducts applications security analysis at the runtime, it can only find flows that are executed. Hence, it has less code coverage compared to the static taint analysis [2].

Symbolic execution remains one of the favored techniques when it comes to error detections [17]. It is usually used for testing applications for defects and security matters [18]. Symbolic execution has been proposed as a solution to the concrete execution that explores a specific actual data input and a single control flow path at a time. Instead in symbolic execution, a program is explored for the different paths it can take when fed with different inputs. To that end, to accomplish this, symbolic execution does not take actual data as input, rather, it uses symbolic input values. As a result, the output is given as a function of the symbolic value and is considered as a sound analysis compared to the concrete. Symbolic execution has suffered from execution path explosion for large or complex programs [19].

Hence, in order to mitigate the path explosion issue, concolic execution is proposed. Concolic execution combines concrete execution and symbolic execution in order to overcome inherent defects identified in the symbolic execution including path explosion and handling calls to native libraries [20]. That is, the program is executed on some concrete input values provided by the analyst and then symbolic path constraints are generated for that specific execution. Concolic execution was first proposed in 2001 by Eric Larson and Todd Austin [21].

The rest of the paper is organized as follows: Section 2 is the background of the study providing basics of how symbolic execution works. The studies of different symbolic execution tools are presented in Section 3. Section 4 discusses the lessons learned from reviewing the common tools, and Section 5 concludes the paper.

2. BACKGROUND

Conventionally, symbolic execution is used for analyzing sequential programs with integer variables [22]. Symbolic Execution uses symbolic values as input data rather than actual data and symbolic expressions as program variables. Different from the concrete execution approach that tests programs on specific input with a single control flow path, symbolic execution rather tests programs with different inputs against multiple execution paths. In symbolic execution, programs are fed with symbolic values instead of concrete input values [23]. The approach uses an execution engine that collects a set of constraints combined and formulas across each explored path. Once instructions are evaluated the formula is updated accordingly. The execution forks when a branching instruction is encountered. A constraint solver - typically one suited for satisfiability modulo theories (SMT) - is used to evaluate expressions involving symbolic values, as well as for generating concrete inputs that can be used to run the program concretely along the desired path [24].

```
      int x, y;
 1:  if (x > y) {
 2:     x = x + y;
 3:     y = x - y;
 4:     x = x - y;
 5:     if (x - y > 0)
 6:        assert (false);
          }
```

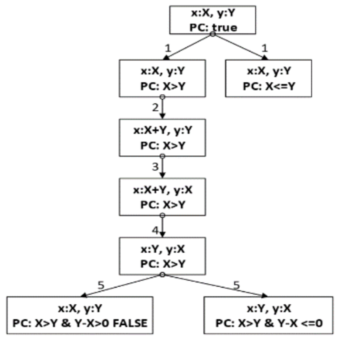

FIGURE 1: Code that swaps two integers and the corresponding symbolic execution tree [20].

The state of the symbolically executed program is usually depicted in a symbolic execution tree that shows the execution paths the input followed during the analysis. The nodes of the tree are the states of the program. Each execution path between states is represented by an arc labelled with a transition number. For example, in Figure 1, the code segment which swaps the value of integer variables x and y is shown to the left of the figure. A corresponding symbolic execution tree can be built and is depicted on the right of Figure 1. According to the symbolic execution tree, in the initial path condition, x and y have the symbolic values X and Y. In each transition, based on the input values, the path condition is updated. Following the execution of the initial statement, both 'then' and 'else' alternatives of the 'if' statement are possible, and the path condition is updated accordingly. Where inputs do not satisfy the path condition (false), it means the symbolic state of the program is not reachable, and as a result, the symbolic execution will not continue for that path of the program. For instance, statement number six (6) is unreachable in the symbolic execution tree in Figure 1.

In order to perform symbolic execution analysis, the program has to exercise a large set of paths through its execution tree that is because whenever more paths are explored, the higher the coverage of examined codes. Nevertheless, such an enumeration of execution paths is computationally expensive. Traditionally symbolic execution uses exhaustive exploration of the possible execution paths. However, this makes the analysis process to remain applicable only to small applications, causing analysts to aim for less ambitious goals. Having said that, a number of approaches that can ease the process of path exploration are employed by most of the recent symbolic execution tools. For example, some researchers proposed standard model checking tools for Java programs in order to perform the path selection process [25, 26].

Generally, symbolic execution challenges are related to four different areas that have been studied including memory, environment, state-space exploration, and constraint solving [27]. Memory related challenges are about the way the symbolic engine manipulates pointers, arrays and other complex objects that may give rise wrong symbolic values or expressions. Environment-related challenges are about the external call that may cause side effects to the execution. State-space exploration is about the control flow path that the execution engine should explore within a reasonable amount of time. Finally, the constraint solving related challenges are simply about the issues pertaining to the scalability of the constraint solver of a tool.

Likewise, according to Xu et al., challenges pertaining to the symbolic execution tools (software and program testing) can be referred to as *symbolic-reasoning* and *path-explosion* challenges [28]. Symbolic-reasoning challenges are related to the problems that cause symbolic execution tools to generate incorrect test results for particular control flows. These include a symbolic variable declaration, symbolic jumps, symbolic memories, contextual symbolic values, floating-point numbers, buffer overflows, and arithmetic overflows. Likewise, path explosion challenges are related to the problems that introduce increased control flows to analye a program [29]. In other words, these are problems that cause symbolic execution tools to require increased time and resources on exploring the paths needed for analysis [30]. These include external function calls, loops, and crypto functions that may cause path-explosion issues to both large-sized and small-sized programs. In this paper, these challenges are considered and solutions from each tool in relation to the challenging area are discussed.

3. SYMBOLIC EXECUTION TOOLS

This section discusses some of the famous software testing tools which utilize symbolic execution techniques. The tools proposed in the literature are mostly used for testing input generation [31], regression testing [32], program deobfuscation [33], and dynamic software updates [34]. In addition, there is another group of tools that uses symbolic execution to guide exploit generation [30], vulnerability finding [35], and fuzzing [36].

DART [37] is one of the early works with automated unit testing (concrete execution) technique. It combines three approaches in order to conduct software analysis. It uses static source code parsing for code inspection of C programs. It performs automatic random testing in order to find software bugs specifically inter-procedural bugs and bugs caused by the use of library functions. Finally, it conducts dynamic analysis in order to test how the program behaves under random testing. It tests programs for standards errors such as crashes, assertion violations, and non-termination. As a concrete execution tool, DART does not employ path selection mechanisms because it uses specific input with a single path testing scenario. However, the random choosing of the value over the domain of potential inputs (random testing) followed by DART may lead to the same observation behaviour that may cause redundancy. Likewise, in random testing, the chance of selecting inputs that cause buggy behaviour may be small [38].

CUTE [39] is a software testing tool that uses the concolic execution technique that combines concrete and symbolic executions. Different from DART, CUTE tests programs with first trying NULL, and then, in a subsequent execution, a concrete address, rather than making random choices. CUTE uses concretization of address to maintain consistency across different executions and due to efficiency in constraint solving. In this tool, a logical input map is used to generate memory input graphs for the unit under test. Sen, K. et al. reported that the CUTE works efficiently in exploring paths in C code to expose software bugs resulting in assertion violations, segmentation faults, or infinite loops [39]. The main reason why CUTE uses combined symbolic and concrete execution is to generate test inputs to explore different execution paths with which the execution proceeds [27]. In addition, the tool uses a constraint solver tool that facilitates the incremental generation of the input. Sen et al. proposed an implementation of CUTE in finding algebraic security attacks in cryptographic protocols and security breaches in unsafe languages, but have never published their work in this regard [40]. In another study jCUTE, the tools have been extended for Java programs [41].

Cadar et al. proposed EXE, an effective bug-finding tool [42]. EXE uses the concolic execution technique that runs symbolic inputs to track the constraints in memory locations. The tool uses real code in finding bugs and capitalizes on the effect of running a single code path by automatically generating concrete inputs that can run into multiple program execution paths. What makes EXE different is that once a path hits a bug it automatically generates a test case using the value that has triggered the bug as concrete values. The tool uses search heuristics for path selection. The researchers use two performance optimizations including cashing constraints to avoid calling Simple Theorem Prover (STP) solver and removing irrelevant constraints from the queries the tools send to STP solver.

SAGE proposed by Godefroid et al. is a concolic execution software testing tool that is internally used by Microsoft [43]. SAGE is considered as a general tool because it works at the instruction level to track integer constraints (bit-vectors). In another research, SAGE has been utilized as a security testing tool [44]. The tool uses the random test style firstly employed by DART but mutates well-formed inputs using grammars. SAGE introduced a generational search as a constraint solver to explore the state space of large applications executed with large inputs. In addition, the tool uses a number of optimization techniques to improve the performance and memory usage of the constraint generation, such as tag caching where structurally equivalent tags are mapped to the same physical object, and local constraint caching. Moreover, the tool also uses the constraint subsumption optimization technique for structured-file parsing applications.

According to Tillmann and De, PEX is a software testing tool developed for .NET that produces a small test suite with high code coverage [45]. PEX performs analysis using dynamic symbolic execution. To reason about the feasibility of execution paths, PEX uses constraint solver Z3. Z3 [46] is an efficient satisfiability modulo theory solver which is commonly used in software verification and application analysis. By using Z3, PEX is able to reason operations such as substring, concatenation, and replacement. Z3 also offers binding for certain programming languages [46].

A recently-developed tool, Tracer, is another software testing tool that is developed based on symbolic execution approach [47]. Tracer is a verification tool for the finite-state of sequential C programs. The tool uses constraint logic programming (CLP) as a resolver. In addition, the tool uses interpolation methods including the strongest postconditions and weakest preconditions.

Identifying vulnerability in binary code is a complicated task. BitBlaze is one of the projects that focused on the analysis of binary codes for vulnerability analysis [2]. The BitBlaze project contributed to the community three tools each focusing different approaches of preforming binary analysis. The tools include Vine, a static taint analysis component, TEMU, a dynamic taint analysis component, and Ruder, a Concolic execution component. Rudder has core utilities and interfaces that enable users to take a snapshot and reload the exploration state providing user-specific path selection policies. The tool uses the ‘Lazy’ approach which collects necessary information in the symbolic machine during the execution. Moreover, the tool uses STP for symbolic-reasoning and breadth-first search approach for path selection [2].

BAP is one of the early binary analysis tools that is developed based on the symbolic execution approach [48]. BAP is a redesigned type of Vine [2] with the goal of including useful analysis and verification techniques that may be appropriate for binary code analysis and allowing user-level analysis. It assembles binary code into an optimized intermediate language (IL) and subsequently performs analysis at the IL level.

Automatic Exploit Generation (AEG) is developed based on concrete and symbolic execution approaches [49]. The tool identifies exploitable paths in a program. Hence to address the path-reasoning challenge, the tool employs a novel technique called preconditioned symbolic execution with which it targets paths that are more likely to be exploitable. In the report, the researchers have proposed five challenging areas where tools like AEG should focus on doing their analysis. The five challenging areas include the state space explosion problem, the path selection problem, the environment modelling problem, the mixed analysis challenge, and the exploit verification problem.

Different from the AEG, Mayham is a concolic execution tool that finds exploitable bugs in binary code without debugging information [50]. There are four design principles adopted by Mayham that make it different from the tools discussed previously. The tool makes forward symbolic execution with arbitrary time, it does not repeat work for maximized performance, the tool keeps the works of previous analysis for reusability, and finally, the tool can reason about symbolic memory. In addition, the tool is known for its hybrid way of combining offline and online executions. Another work similar to that provided in Mayham is proposed in Veritesting [51]. Veritesting is a binary only symbolic execution tool targeting large scale testing of commodity off-the-shelf software. It uses dynamic symbolic execution for testing and static symbolic execution for verification.

A recent tool, Firmalice, is proposed for the analysis of privacy-sensitive and security-critical applications installed on the IoT devices specifically [52]. The tool mainly focuses on the identification of the existence authentication bypass activities and existing backdoor. The tool uses concretizing user input as a constraint solver.

Driller, developed by Stephens et al. is a hybrid vulnerability excavation tool that uses concolic execution to guide fuzzing [53]. The concolic execution component analyses the program, traces user input and utilizes its constraint-solving engine to guide fuzzing too take different paths, therefore it finds bugs located deeper in the code. Helping with a concolic execution component, Driller can detect more vulnerabilities, however, it requires a lot of computing power and may quickly run into the path explosion problem [53].

Table 1 summarizes the tools reviewed in this paper together with their techniques used to overcome challenges associated with symbolic-reasoning, path-reasoning, and the optimization approaches the tools employed in order to boost the performance or reduce the space required for the analysis. Only three tools have used optimization techniques (EXE, SAGE, and BAP). However, SAGE seems relatively more efficient in path reasoning and symbolic reasoning, while EXE is only good in path reasoning and BAP has shown less accurate. Four tools are language-dependent including CUTE, DART, EXE, and Tracer. Nevertheless, most of the tools reviewed in this paper are language independent and employ binary codes for their analysis. There has been an increase since 2012, which may show that symbolic execution tools are becoming more robust and main vulnerability analysis. The Concolic execution techniques have gained a higher bar of acceptance for the past decade. The tools have used different constraint solvers, however, the most used are SMT solvers. Of the 13 symbolic execution tools reviewed in this paper, 6 use Concolic approaches that combine symbolic with concrete executions.

The first five columns provided in Table 1 capture the nature of the tools and little do they say about the evaluation of the tools; the last column capture the performance of the tools. Most of the tools do not perform quantitative evaluation of their results. However, based on the reviewing we were able to quantitatively compare different claims reported in each of the papers. As a result, the last column of Table 1 discusses the overall performance of each of the tools in relation to other similar works reviewed here.

Sources	Tools	SE or CE	Targeting Language or Binary	Constraint solvers	Used Oprimization technique	Strengths & weaknesses
[39]	CUTE	CE	Language	approximate pointer constraints		its weakness is that it targets only C language programs
[37]	DART	SE	Language	depth first exploration		It lucks constraint solver for path selection, as it uses concrete inputs, that is it is time consuming
[42]	EXE	CE	Language	best-first search (BFS) heuristic and depth-first search	Constraint caching and Constraint independence	It is relatively more efficient in path reasoning.
[45]	PEX	CE	Binary	Z3 for both symbolic and path reasoning		its effectiveness relies on good run-time checks in the code or the run-time system.
[43]	SAGE	CE	Binary	code-coverage maximizing heuristic, compositionally (function summaries), Generational Search	tag caching, local constraint caching, and constraint subsumption.	It is relatively more efficient in path reasoning and symbolic reasoning cause it optimizes using caching.
[47]	Tracer	SE	Language	constraint logic programming		relatively less pupolar because language specific and does not use known way of symbolic reasoning
[2]	Rudder	CE	Binary	STP as the solver for symbolic reasoning, and breadth-first search for path reasoning.		one of the most famous among security testing concolic execution tools
[48]	BAP	SE	Binary	SMT solvers	Optimizes intermediate language (IL), making syntaxdirected analysis possible	It does not support floating point and privileged instructions, hence lass accurate
[49]	AEG	CE	Binary	preconditioned symbolic execution and path prioritization technique for path selection.		resistant to buffer overflows, and it an end-to-end fully automated tool
[50]	Mayham	SE	Binary	SMT solver		MAYHEM does not have models for all system/library calls
[51]	Veritesting	SE	Binary	SMT solver for path reasoning		Can do some modern defenses such as canaries
[52]	Firmlice	SE	Binary	concretizing user input		used for modern application testing such mobile and IoT applications
[53]	Driller	SE	Binary	Mutated inputs		supports fuzzing with sysmbolic execution

TABLE 1: Summary of Symbolic Execution Tools.

4. DISCUSSIONS AND FUTURE DIRECTIONS

In this section, potential directions to enhance the state of art of symbolic execution tools are discussed. According to our review, the scalability of such reviewed tools is an open direction that can be taken as future research. Researchers have only slightly worked on the optimizations of both performance and memory space of the tools. Investigating new optimization methods to lower the overhead of symbolic execution and concolic execution tools could make the tools more useable. The well-known path explosion problem is still a main concern of the symbolic execution tools. Therefore, finding a way to limit the paths or possibly reduce the number of less important paths may be helpful to slow down the path explosion problem.

Taking symbolic execution tools that may detect problems, such as authentication bypass, towards cloud and mobile applications could be an interesting future direction. In addition to the symbolic execution engines, SMT solvers are decision procedures that solve problems that arise from the use of logic formulas. SMT solvers are predominately used in the security testing tools, however, software testing tools make little use of these solvers, and their support for non-linear real and integer arithmetic is still in its infancy.

5. CONCLUSION

Without access to source code, binary analysis becomes an effective method for finding vulnerabilities from programs. Researchers have proposed and developed many techniques to help with the binary analysis process, for example, taint analysis, symbolic and concolic executions. In this paper, symbolic and concolic execution techniques are discussed in detail. Tools utilize symbolic and/or concolic execution are reviewed as well. These tools mostly focus on software security testing, and they usually use symbolic execution or concolic execution to help with the test generation and program analysis. The work related to this area is vast and cannot be covered in a single review paper. However, this survey paper discusses well-known and usually referenced tools that cannot be overlooked while studying this area. The comparison table built from the review can be used by researchers in this area to provide a guide to the commonly used tools which employs symbolic and/or concolic execution techniques.

6. REFERENCES

[1] D Andriesse. "*Practical Binary Analysis*", no starch press. 2019.

[2] D Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Prateek Saxena. "*BitBlaze: A new approach to computer security via binary analysis*." in International Conference on Information Systems Security. 2008. Springer.

[3] M von Maltitz, C Diekmann, G Carle. "*Privacy Assessment Using Static Taint Analysis (Tool Paper)*." in International Conference on Formal Techniques for Distributed Objects, Components, and Systems. 2017. Springer.

[4] X Lin, T Chen, T Zhu, K Yang, F Wei, "*Automated forensic analysis of mobile applications on Android devices.*" Digital Investigation, 2018. 26: p. S59-S66.

[5] Z Xing, Z Bin, F Chao, Z Quan. "*Staticly Detect Stack Overflow Vulnerabilities with Taint Analysis.*" in ITM Web of Conferences. 2016. EDP Sciences.

[6] C Feng, X Zhang. "*A Static Taint Detection Method for Stack Overflow Vulnerabilities in Binaries.*" 4th International Conference on Information Science and Control Engineering (ICISCE). 2017. IEEE.

[7] S Chen, J. Xu , N. Nakka, Z. Kalbarczyk, R.K. Iyer. "*Defeating memory corruption attacks via pointer taintedness detection.*" in 2005 International Conference on Dependable Systems and Networks (DSN'05). 2005. IEEE.

[8] GE Suh, JW Lee, D Zhang, S Devadas. "*Secure program execution via dynamic information flow tracking*." in ACM Sigplan Notices. 2004. ACM.

[9] G Venkataramani, Ioannis Doudalis, Yan Solihin, Milos Prvulovic. "*Flexitaint: A programmable accelerator for dynamic taint propagation.*" in 2008 IEEE 14th International Symposium on High Performance Computer Architecture. 2008. IEEE.

[10] J Shin, Hongce Zhang, Jinyong Lee, Ingoo Heo, Yu-Yuan Chen, Ruby Lee, Yunheung Paek. "*A hardware-based technique for efficient implicit information flow tracking.*" in 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2016. IEEE.

[11] VP Kemerlis, G Portokalidis, K Jee, AD Keromytis. "*libdft: Practical dynamic data flow tracking for commodity systems.*" in Acm Sigplan Notices. 2012. ACM.

[12] W Xu, S Bhatkar, R Sekar. "*Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks*." in USENIX Security Symposium. 2006.

[13] V Ganesh, T Leek, M Rinard. "*Taint-based directed whitebox fuzzing.*" in Proceedings of the 31st International Conference on Software Engineering. 2009. IEEE Computer Society.

[14] TR Leek, GZ Baker, RE Brown, MA Zhivich. "*Coverage maximization using dynamic taint tracing.*" 2007, MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB.

[15] R Wang, G Xu, X Zeng, X Li, Z Feng. "*TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting*." Journal of Parallel and Distributed Computing, 2018. 118: p. 100-106.

[16] J Clause, W Li, A Orso. "*Dytan: a generic dynamic taint analysis framework.*" in Proceedings of the 2007 international symposium on Software testing and analysis. 2007. ACM.

[17] C Cadar, K Sen. "*Symbolic execution for software testing: three decades later.*" Commun. ACM, 2013. 56(2): p. 82-90.

[18] S Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, Prateek Saxena. "*Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints*." in NDSS. 2019.

[19] Q Yi ; Zijiang Yang ; Shengjian Guo ; Chao Wang ; Jian Liu ; Chen Zhao. "*Eliminating path redundancy via postconditioned symbolic execution.*" IEEE Transactions on Software Engineering, 2017. 44(1): p. 25-43.

[20] CS Păsăreanu and W. Visser. "*A survey of new trends in symbolic execution for software testing and analysis.*" International journal on software tools for technology transfer, 2009. 11(4): p. 339.

[21] E Larson, T Austin. "*High Coverage Detection of Input Related Security Faults*," 12th USENIX Sec. 2001. Symposium.

[22] CS Pasareanu, R Kersten, K Luckow, QS Phan. "*Symbolic Execution and Recent Applications to Worst-Case Execution,*" Load Testing and Security Analysis.

[23] C Cadar, P Godefroid, S Khurshid. "*Symbolic execution for software testing in practice: preliminary assessment.*" in 2011 33rd International Conference on Software Engineering (ICSE). 2011. IEEE.

[24] R Baldon, iEmilio Coppa, Daniele Cono D'Elia "*Assisting malware analysis with symbolic execution: A case study.*" in International Conference on Cyber Security Cryptography and Machine Learning. 2017. Springer.

[25] S Khurshid, C.S. Păsăreanu, and W. Visser. "*Generalized symbolic execution for model checking and testing*." in International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2003. Springer.

[26] X Deng, J Lee "*Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems.*" in 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). 2006. IEEE.

[27] R Baldoni, E Coppa, DC D'elia, C Demetrescu. "*A survey of symbolic execution techniques.*" ACM Computing Surveys (CSUR), 2018. 51(3): p. 50.

[28] H Xu, Z Zhao, Y Zhou, MR Lyu. "*Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs*." IEEE Transactions on Dependable and Secure Computing, 2018.

[29] A Arusoaie, D Lucanu, V Rusu. "*Symbolic execution based on language transformation*." Computer Languages, Systems & Structures, 2015. 44: p. 48-71.

[30] AJ Kahn, Y Drougas, AP Shendarkar. "*Symbolic execution for web application firewall performance.*" 2019, Google Patents.

[31] L Arquint, M Schwerhoff. "*Profiling Symbolic Execution.*" 2019.

[32] T Kuchta, H Palikareva, C Cadar. "*Shadow symbolic execution for testing software patches.*" ACM Transactions on Software Engineering and Methodology (TOSEM), 2018. 27(3): p. 10.

[33] M Liang, Z Li, Q Zeng, Z Fang. "*Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization.*" in Information and Communications Security: 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings. 2018. Springer.

[34] S Guo "*Efficient Symbolic Execution of Concurrent Software*." 2019, Virginia Tech.

[35] G Wang, S Chattopadhyay, AK Biswas, T Mitra. "*KLEESPECTRE: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution.*" arXiv preprint arXiv:1909.00647, 2019.

[36] C Chen, B Cui, J Ma, R Wu, J Guo, W Liu. "*A systematic review of fuzzing techniques.*" Computers & Security, 2018. 75: p. 118-137.

[37] P Godefroid, N. Klarlund, and K. Sen. "*DART: directed automated random testing."* in ACM Sigplan Notices. 2005. ACM.

[38] AJ Offutt, JH Hayes. "*A semantic model of program faults.*" in ACM SIGSOFT Software Engineering Notes. 1996. ACM.

[39] K Sen, D Marinov, G Agha "*CUTE: a concolic unit testing engine for C.*" in ACM SIGSOFT Software Engineering Notes. 2005. ACM.

[40] R Ahmadi, K Jahed, J Dingel. "*mCUTE: A Model-level Concolic Unit Testing Engine for UML State Machines.*" in 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019), Demonstration Track, page to appear. ACM. 2019.

[41] K Sen, G Agha. "*CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools.*" in International Conference on Computer Aided Verification. 2006. Springer.

[42] C Cadar, V Ganesh, PM Pawlowski, DL Dill. "*EXE: automatically generating inputs of death.*" ACM Transactions on Information and System Security (TISSEC), 2008. 12(2): p. 10.

[43] P Godefroid, MY Levin, DA Molnar. "*Automated Whitebox Fuzz Testing.*" in NDSS. 2008. Citeseer.

[44] P Godefroid, MY Levin, D Molnar. "*SAGE: whitebox fuzzing for security testing*." Communications of the ACM, 2012. 55(3): p. 40-44.

[45] N Tillmann, J De Halleux. "*Pex–white box test generation for. net.*" in International conference on tests and proofs. 2008. Springer.

[46] L De Moura, N Bjørner. "*Z3: An efficient SMT solver.*" in International conference on Tools and Algorithms for the Construction and Analysis of Systems. 2008. Springer.

[47] J Jaffar, V Murali, JA Navas, AE Santosa. "*TRACER: A symbolic execution tool for verification.*" in International Conference on Computer Aided Verification. 2012. Springer.

[48] D Brumley, I Jager, T Avgerinos. "*BAP: A binary analysis platform.*" in International Conference on Computer Aided Verification. 2011. Springer.

[49] T Avgerinos, SK Cha, BLT Hao, D Brumley. "*AEG: Automatic exploit generation*." 2011.

[50] SK Cha, T Avgerinos, A Rebert. "*Unleashing mayhem on binary code*." in 2012 IEEE Symposium on Security and Privacy. 2012. IEEE.

[51] T Avgerinos, A Rebert, SK Cha, D Brumley. "*Enhancing symbolic execution with veritesting.*" in Proceedings of the 36th International Conference on Software Engineering. 2014. ACM.

[52] Y Shoshitaishvili, R Wang, C Hauser, C Kruegel. "*Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.*" in NDSS. 2015.

[53] N Stephens, J Grosen, C Salls, A Dutcher, R Wang. "*Driller: Augmenting Fuzzing Through Selective Symbolic Execution*." in NDSS. 2016.