International Journal of Computer Applications (0975 - 8887)
Volume 186 - No.37, August 2024

Symbolic Execution-based Code Coverage Framework for Augmented Software Testing

Rachel Glockenmeier
Dakota State University
820 N. Washington Ave.
Madison, SD, USA 57042

Varghese Vaidyan, PhD
Dakota State University
820 N. Washington Ave.
Madison, SD, USA 57042

Yong Wang, PhD
Dakota State University
820 N. Washington Ave.
Madison, SD, USA 57042

ABSTRACT

Although extensive research has been done on automated software comprehension, no analysis framework exists that is free of limitations and constraints to address the challenge of comprehending software without manually investigating it. This introduces several challenges. One important task to software researchers is to identify all the sets of paths inside a target program. Answering this will offer further information about the target program and allow for understanding, and ultimately further security and quality analysis. Introducing a comprehensive framework for code coverage analysis, a notable gap in the existing works, is addressed by offering empirical evidence and an analysis framework to evaluate the impacts of enhancements to symbolic execution techniques in target programs. Using angr as the symbolic execution engine, several code exploration approaches based on prior research and angr's capabilities are implemented. To analyze the implications of these changes on code coverage, the proposed approach performs a comparative investigation over a wide variety of binary programs, accounting for varying complexity levels and memory restrictions. The experimental findings show a wide coverage range ranging from 0.3185% to 16.7093%, depending on the testing circumstances. By developing a benchmark for code coverage under symbolic execution, the framework not only elucidates the interaction of testing variables, but also offers a full analytical framework for assessing coverage expectations in respective contexts.

General Terms

software analysis, code coverage

Keywords

symbolic execution, angr, AEG

1. INTRODUCTION

Despite extensive research on automated software comprehension, no tool exists without caveats and limitations to address the challenge of understanding software without manual study. There are still many challenges facing researchers attempting to perform this work. One of the main challenges is to identify the full set of paths within a target program. The answer to this question can then be used to determine further information about the target program. For example, this research is the first step in automating exploit development for heap vulnerabilities by using the paths within a program to find the allocation primitives.

To use the set of program paths discovered with symbolic execution, the amount of code coverage that can be achieved with symbolic execution strategies must be understood. The metric of code coverage allows for an approximation of what percentage of paths within the target program were able to be discovered with symbolic execution, and ultimately, the lower bound for how complete the analysis using these paths can possibly be.

There are many works attempting to use and create tools using symbolic execution with improvements for the specific applications in related areas research [3, 4, 6]. However, none of the works compare different search strategies based on the code coverage that they can achieve. The results of this work will enable other researchers to use these code coverage metrics as a starting point for tool development. These results will also be used as the underpinning of further research planned to build a tool that uses the paths that can be extracted from a target program to find heap primitives. To complete this work and other work that relies of the coverage of symbolic execution, the results of this experiment are required to understand what the maximum possible performance of such a tool could be.

To address the specific goal of determining the code coverage that can be achieved with symbolic execution, this work contributes the following:

—An analysis of what coverage can be expected using symbolic execution on a binary under different test conditions

—An analysis platform to evaluate code coverage for binaries

2. BACKGROUND

The concept of symbolic execution has been studied since its first major finding in the 1970s and is a powerful static analysis technique [11]. Symbolic execution is a technique for analyzing programs to determine which inputs cause effects in the later portions of the program. This is done by allowing an interpreter to use symbolic values for arguments or inputs and maintaining these values as the interpreter traces through the program. Over the years there has been work to improve the technique and make it more applicable to real world software and to attempt to address challenges with the technique [5]. The most common of these challenges in-

clude state explosion, constraint solving limitations, representing different memory types, and complex software environments with external dependencies [9].Symbolic execution has applicability to several domains including software security [1].
Symbolic execution can be done in several ways depending on how the engine treats different conditions that it encounters. Some of the policies that determine how the engine will behave include:

—State merging from different execution paths.
—How program loops are handled (including while and for loops).
—Whether function summaries are used.
—How program paths are determined to be infeasible and when they are discarded [12].

2.1 Challenges with Symbolic Execution

Despite complexities, symbolic execution remains a powerful way to understand software, and there are many attempting to use and overcome the challenges with symbolic execution [2]. Of particular interest to this line of work are the strategies to overcome state explosion when attempting to find feasible execution paths within a program. State explosion is the first challenge cited in many works, as well as one that many papers attempt to find solutions for. When attempting to symbolically execute a complex program with many loops, it becomes difficult (or impossible) to track and explore all possible states for the program due quantity.

2.2 Symbolic Execution Path Detection

Within symbolic execution, differing search strategies can be used to find paths:

—Depth-first search (DFS)
—Breadth-first search (BFS)
—Random path selection
—Coverage optimized or subpath-guided search by using edge weights to prioritize under-explored paths
—Shortest-distance search
—Goal-based path search – This strategy can refer to many different search algorithms that have more specific goals in mind. For example, the buggy-first path search which, is introduced as a concept for prioritizing paths where there are more likely to be bugs, such as those with complex pointer arithmetic [15].

2.3 angr

The most popular tool for symbolic execution is angr. This Python tool was developed by UC Santa Barbara and published in 2016. angr allows for static and dynamic symbolic execution [13]. Python has been found to be suitable in many applications due to the configurable libraries [16, 17]. Python is suitable in this work because angr is a highly configurable python library that allows a developer to have control over aspects of the constraint solver, state manager, and most importantly for the proposed work, the path search algorithm. By default, angr's execution engine uses a breadth-first search, but angr has several path search strategies that can be used by changing the path exploration technique [8].

2.4 angrD

angrD is an example of a symbolic execution tool that is built on top of angr [8]. It is similar to many other tools in this area of investigation because it is built on top of the engine that angr provides to improve vulnerability detection. For example, to detect heap overflow vulnerabilities, angrD stores the associated addresses for allocation functions, and checks this list as additional allocations are made.

3. RELATED TOOLS

There are several tools that are built on the concept of symbolic execution. Some of the ways that these tools differ is discussed in the Background section(Section 2) through examining attempted solutions to avoid challenges with symbolic execution. There are many tools that have been written using symbolic execution, but this section serves to highlight some of the most well-known and recent tools. Many of the tools developed to use symbolic execution, focus on discovering a specific path or the feasibility of a given constraint, rather than code exploration. The tools selected below are tools attempting to improve the code coverage through symbolic or concolic execution.

3.1 MAYHEM

MAYHEM applies symbolic execution to security challenges and Automated Exploit Generation (AEG). This tool focuses on finding exploitable bugs by prioritizing search paths based on those that involve symbolic memory (pointers or memory addresses). In this work, Mayhem is successfully used to detect vulnerabilities in Windows and Linux software [6].

3.2 KLEE

KLEE uses path metrics to bias symbolic execution toward exploring paths that are under-explored. obtaining better coverage. The weight of each state is determined by how recently the code was explored and the distance between the current point and unexplored instructions [3].

3.3 EXE (Execution generated Executions)

EXE is a tool to find bugs in software by generating test cases causing the program to crash. As the tool runs, it uses symbolic execution to determine when input values are symbolic. When symbolic expressions are found, EXE forks and constrains that argument to execute as true in one execution and false in the forked execution and continues until a path ends or fails. From these failure cases, EXE is able to use a constraint solver to locate the full path and generate a test case for the failure [4].

3.4 SAGE (Scalable Automated Guided Execution)

SAGE is a tool utilizing symbolic execution to perform whitebox fuzzing. Symbolic execution is a limiting factor in terms of the speed and depth into the code that can be achieved. To combat this limitation, SAGE implements a search algorithm which negates each constraint along the path to generate test cases. Traditional search methods only negate a small number of constraints, which leads to fewer test cases and poorer fuzzing results [10].

4. SYMEXCOV ARTIFACT

To achieve the goal of determining the code coverage that can be achieved with symbolic execution, the design science methodology utilizing a single-case experiment mechanism has been selected to study, perform, and validate this work. The artifact, named SymExCov, will follow the path outlined in Figure 1 below. The following sub-sections provide additional details on the most notable of these phases including the design and evaluation of the artifact.

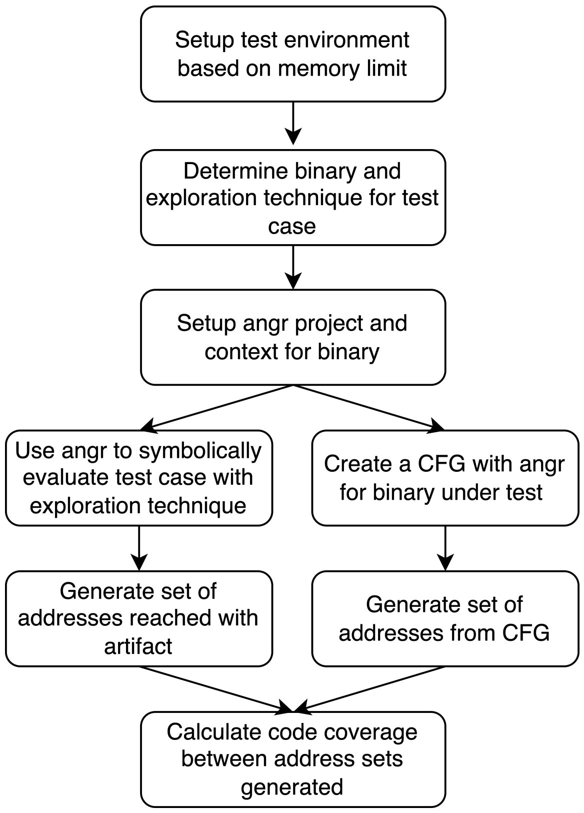
Setup test environment based on memory limit
Determine binary and exploration technique for test case
Setup angr project and context for binary
Use angr to symbolically evaluate test case with exploration technique
Create a CFG with angr for binary under test
Generate set of addresses reached with artifact
Generate set of addresses from CFG
Calculate code coverage between address sets generated

Fig. 1. Design diagram for research artifact

4.1 *Artifact Design and Development*

To address the research goals from Section 1, an artifact was developed. This artifact is written in Python utilizing the angr library. At a high level, the artifact uses symbolic execution (through angr) to traverse as much of the code as possible and keep track of the addresses within the code that have been reached.
angr implements many exploration techniques that can be selected to modify the way that angr behaves. Each of these exploration techniques are designed to improve the performance of angr for specific use cases. To achieve the goal of understanding the coverage that can be achieved with different search biases, the following set of exploration techniques were selected:

—Default – angr's default exploration technique is a version of a breadth-first search (BFS). As angr steps through the program, it collects states encountered as active states and attempts to step each of these active states at the same time.

—DFS – DFS focuses on getting as deep into the code as possible before expanding the search out. To do this, only one active state is kept at a time (placing all other states as deferred states until the one active state is completed).

—Stochastic – Stochastic exploration is commonly referred to as the "random" technique. This technique keeps only one path as the active path at a time. For each path, the basic blocks are assigned a random weight that is used as a probability distribution for determining the likelihood that a state still exists after a fork in the path occurs.

—Loopseer – Loopseer's strategy tries to ensure that execution does not spend an inordinate amount of time within the same loop. To do this, it keeps track of the number of passes that have been made within the same loop and stashes the state for later evaluation if too much time is being spent on the same loop.

—Oppologist – Oppologist helps to handle cases where angr encounters an unsupported operation. In these cases, it can concretize the arguments to the unsupported instruction and use angr's unicorn engine to emulate the unknown instruction so the execution of the binary can continue beyond this operation.

—Lengthlimiter – This technique allows specifying a maximum path length before marking that path as complete and continuing to other paths.

—Spiller – Spiller keeps track of the system memory usage and offloads some active states to disk when memory usage is high. These states can be resumed when some active states have completed execution [7, 14].

This set is a subset of exploration techniques available within angr. Some techniques that were not selected for this study are not applicable for this work. This is because they are intended to find whether a single path exists between an entry function and a specific sink function or if a specific condition can be satisfied. In this work, the focus was on attempting to measure code coverage, so the goal was to reach all the code, not find a specific portion of the code.

Algorithm 1 FindBasicBlocksDFS

```
$project \leftarrow angr.Project(binary)$
$state \leftarrow project.entryState()$
$simgr \leftarrow project.simulationManager(state)$
$simgr.useTechnique(angr.exploreTechniques.DFS())$

while $simgr.active \neq \emptyset$ do
    $simgr.step()$
    for each $activeState \in simgr.active$ do
        $currentAddr \leftarrow activeState.addr()$
        if $currentAddr \ni entireSet$ then
            $entireSet.add(currentAddr)$
        end if
    end for
end while
```

For each exploration technique listed above, a function was created within the artifact to symbolically execute the binary under test. Algorithm 1 shows the method for symbolically executing and finding all of the addresses reached for the DFS exploration technique. This algorithm first initializes the angr project and state, then sets up the simulationManager with the information needed to symbolically execute the binary, including setting the starting state and the execution technique that will be used. After the initialization is complete, the simulationManager is used to maintain all states and context as the binary is symbolically executed. The step() function is used to progress execution through the binary. The exact functionality of stepping will vary based on the execution technique employed, but this function is responsible for progressing each active state forward and adding newly discovered states to the appropriate stash. As execution is stepped through the binary, this algorithm

keeps track of the addresses for the basic blocks that have been reached for determining coverage during the evaluation phase.
Algorithm 1 is a specific example. However, other techniques were similar, replacing the portion where the exploration technique is set as well as any setup required for the specific technique being used.

4.2 Artifact Evaluation

To evaluate the artifact created, two experiments were performed. In the first experiment, all exploration techniques were run for each binary within the test suite using a fixed amount of memory for each test. In the second experiment, one of the binaries from the test suite was selected and run for each exploration technique, but with a variable amount of memory to understand the impact of memory on code coverage. Each test requires running the artifact to completion or memory exhaustion for each of the execution techniques selected across the relevant binaries.
This effort aims to determine the code coverage possible with symbolic execution. More specifically, code coverage for this work refers to the number of basic blocks (code segments in which there are no branches present) that can be reached with symbolic execution relative to the total number of basic blocks that exist within the binary under test.
To determine the total number of basic blocks, and establish a baseline to compare against, angr's Control Flow Graph (CFG) functionality is used (specifically CFGFast). A CFG is a graph created for the binary where basic blocks are represented as nodes on the graph, and the edges are the control transitions between these blocks. As a result, collecting the address contained within each node of the CFG gives the set of basic blocks present within a binary.

Algorithm 2 CalculateCodeCoverage

```
for each $binary \in binaries$ do
    $project \leftarrow angr.Project(binary)$
    $cfgAddrSet \leftarrow project.CFGFast()$

    for each $technique \in explorationTechniques$ do
        $explorationSet \leftarrow addrsFromAngrExecution$
        $commonAddrs \leftarrow cfgAddrSet \cap explorationSet$
        $percentCoverage \leftarrow 100\frac{|commonAddrs|}{|cfgAddrSet|}$
    end for
end for
```

Algorithm 2 shows the calculation of code coverage for each of the test cases. For each case, the set of addresses found from symbolic execution was compared against the set collected from the CFG. The coverage calculation shown divides the number of addresses that are in both sets by the total number of addresses within the set from the CFG. This number is then multiplied by 100 to determine the percentage of the code that was reached.

5. IMPLEMENTATION AND RESULTS

The following section provides details on the specific configuration under which this work was executed as well as the results of the experiments performed.
Two experiments were performed using the created artifact. The first set of experiments were performed across a set of diverse binaries with a fixed amount of memory available to the artifact. The second set of experiments were performed with a single binary using varying amounts of memory across test cases.

5.1 Setup

To evaluate the artifact developed (Section 4.2), a suite of test binaries was compiled. For the purposes of this work, the binaries needed to meet the requirements below:

—Binaries compiled for an x86 system architecture given that angr is more feature-rich for this architecture.
—Publicly available or easy to obtain binaries.
—Binaries with varying complexity to ensure that simple and more realistic test software was represented.

The set of test binaries used was binutils for the Ubuntu version within the test setup. The set of binaries used: addr2line, ar, as, c++filt, dwp, elfedit, gold, gprof, ld, ld.bfd, ld.gold, nm, objcopy, objdump, ranlib, readelf, size, strings, and strip.
The experiments run in this work each used a virtual machine configured with the system attributes in Table 1 below.

Table 1. Experimental Setup

Operating system	Ubuntu 23.04 Desktop AMD64 (Lunar Lobster)
RAM	Experiment 1 - 16 GB for each test Experiment 2 - 2, 4, 8, 16, 32, 64, 128, and 256 GB
CPU Count	8 Cores
System software	Python 3.11 (including development headers) angr v9.2.76 (released October 2023) Additional dependencies for angr (make, C compiler, archinfo, pyvex, cle, claripy, and ailment)
Test software	Experiment 1 - Entire test suite described above Experiment 2 - objcopy binary for each test

Ultimately, the experimentation setup facilitated the ability to collect data from each of the binaries in the test suite. Each binary was run through the artifact several times in order to evaluate every binary against all of the exploration techniques within angr that were selected to be a part of the artifact.

5.2 Results

The collection of results for each of the binaries tested can be displayed as average coverage across the different binaries or exploration technique, as shown in Table 2 and Table 3, allowing a discussion of the results in a more generic sense across differing binaries and software complexities.

Table 2. Average Code Coverage (Percent) Per Binary

Binary	Number of Basic Blocks	Average Coverage (%)
strings	949	16.7093
ar	2015	9.4080
objcopy	7572	3.2695
strip	7572	1.9866
objdump	16549	1.4882
readelf	32603	0.3185

The contents of Tables 4, 5, and 6 are the results for the objcopy binary across all of the different memory limits and a sampling of the techniques available that are representative of the different kinds of results seen across all methods. The objcopy binary was

Table 3. Average Code Coverage (Percent) By Exploration Technique

Exploration Technique	Average Code Coverage (%)
DFS	5.1735
default	5.1981
random	7.9506
loopseer	5.2431
oppologist	5.1981
lenlimiter	5.1981
spiller	4.7483

selected for this experiment because it was one of the binaries with medium complexity from the initial test suite based on the number of basic blocks present.

Table 4. Code Coverage (Percent) For Loopseer Technique

Memory (GB)	Basic Blocks Reached	Code Coverage (%)
2	120	1.5715
4	120	1.5715
8	120	1.5715
16	120	1.5715
32	120	1.5715
64	120	1.5715
128	120	1.5715
256	120	1.5715

Table 5. Code Coverage (Percent) For Oppologist Technique

Memory (GB)	Basic Blocks Reached	Code Coverage (%)
2	282	3.7110
4	295	3.8827
8	322	4.2393
16	322	4.2393
32	322	4.2393
64	381	5.0184
128	381	5.0184
256	381	5.0184

Table 6. Code Coverage (Percent) For Spiller Technique

Memory (GB)	Basic Blocks Reached	Code Coverage (%)
2	349	4.5958
4	260	3.4204
8	244	3.1695
16	336	4.4374
32	355	4.6751
64	469	6.1806
128	396	5.2165
256	431	5.6920

Table 7 and Figure 2 are based on the full table of results for experiment 2, but have the results across each memory limit averaged to show the impact of the memory limit on the code coverage possible regardless of the exploration technique used.

Table 7. Average Code Coverage (Percent) By Memory Limit

Memory (GB)	Average Code Coverage (%)
2	3.0658
4	2.9714
8	3.0884
16	3.2695
32	3.3035
64	3.8528
128	3.7147
256	3.7827

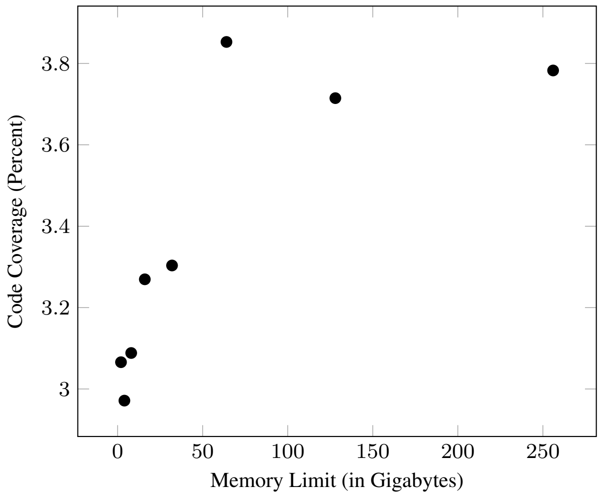

Fig. 2. Coverage as a Function of Memory

6. DISCUSSION OF RESULTS

These results give a baseline for understanding roughly what code coverage can be expected by employing symbolic execution. In addition, this artifact can be utilized against another binary sample to help understand what coverage they can expect and what the highest performing exploration technique or potential benefit to increasing the memory limit is for specific use case.

Looking broadly at the results for experiment 1, there was not an execution technique that performed best across all binaries used. The highest performer varied across the different binaries tested. This is likely the result of having selected a diverse set of binaries with differing functionality and complexity. For example, a binary with very long code paths will perform worse than one with shorter paths when using lengthlimiter to limit the depth of the search performed but will likely have higher coverage with DFS.

Table 2 highlights the relationship between the average code coverage that was achieved for each binary and the complexity of the binary. Unsurprisingly, the amount of coverage that was possible to achieve was inversely proportional to the complexity of the binary. There are several factors that make this the expected outcome. First, as the number of basic blocks increases within a program, so does the likelihood that there are difficult conditions for symbolic execution to deal with, such as nested loops and complex state requirements to keep track of. Additionally, the tests for this experiment were run under the conditions stated for the experimental setup. Most notably, the RAM was limited to 16 GB, which biases the results against the more complicated programs. Symbolic execution requires significant memory resources, and in nearly all the test cases, the evaluation of the binary was stopped when the system ran out of RAM, rather than the execution completing.

Table 3 displays the coverage that was achieved for each of the techniques chosen averaged, to avoid biasing the results by the complexity of the binary that was under test. The data shows that the average coverage achieved by this metric varied between 4.7483% and 7.9749% depending on the exploration technique, with random/stochastic exploration having the highest amount of coverage and the spiller technique having the lowest. Despite the range of coverage that was achieved, all these coverage numbers are low relative to what would be required to use symbolic execution as a complete solution or the underlying method for another exhaustive tool.

The second experiment was performed to understand the effects that memory limits have on code coverage achieved. The results in Tables 4, 6, and 5 show data collected in Experiment 2 and highlight the behavior of individual exploration techniques across memory limits. The expectation was that as the memory limit was increased, the code coverage achieved would also increase until the point that state explosion was achieved. The experiment showed that for the binary under test state explosion occurs early enough in the exploration for some techniques that modifying the memory limits did not have an effect on coverage. An example of this is the DFS exploration technique. For each of the memory limits tested, the DFS technique achieves the same coverage.

The results for the spiller technique, Table 6, have the highest amount of variation from the expected results. It is likely that this behavior is because as spiller brings a state back to active, as memory usage allows, it is randomly choosing which state should be reactivated and that has a larger effect on coverage than the memory available to the process. Further study would need to be performed to understand the spiller technique's behavior more clearly.

Table 7 and Figure 2 are representations of the average coverage for a memory limit (across exploration techniques). These diagrams show some expected and unexpected behavior that was observed during this experiment. These results show that generally as the memory limit is increased, the coverage increases. However, unexpectedly the increase in coverage does not fit a logarithmic curve because there are points where an increase in memory did not result in an increase in coverage, such as the data when testing with 64GB. It is possible that with more exploration techniques and additional binaries, this data could be better fit to a logarithmic curve. While it was expected that the data would fit a logarithmic curve, and adding memory would have diminishing increases in coverage as more memory was added, the hope was that there could be more significant gains in coverage achieved by increasing the limits from Experiment 1 than what was achieved in Experiment 2. Similar to Experiment 1, in Experiment 2 the coverage achieved could be varied by choosing different memory limits or exploration techniques, but ultimately these coverage results are still low and cannot be used directly as the basis for a tool requiring high code coverage.

7. CONCLUSIONS

The outcomes of this investigation provide an analysis framework for utilizing symbolic execution in tool development. The empirical findings provide insights into the complex relationship of code complexity and coverage, the coverage performance for different angr execution techniques, and general angr code coverage capabilities. This approach delivers unique insights into the predictive potential of symbolic execution inside selected application domains by providing a baseline data and toolkit for producing tailored data.

Future work includes a larger study, with additional memory resources to provide more information on the coverage capabilities for more complex binaries, and additional tooling using symbolic execution as the underlying engine. For example, this research could be used as the basis for further research that would require a high level of code coverage to be accurate. Such as tooling to automatically detect heap primitives within a target binary for application in the space of automated heap exploitation.

8. REFERENCES

[1] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Commun. ACM*, 57(2):74–84, feb 2014.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of High-Coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, dec 2008. USENIX Association.

[4] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), dec 2008.

[5] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, feb 2013.

[6] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[7] Simulation managers. https://docs.angr.io/en/latest/core-concepts/pathgroups.htmlexploration-techniques.

[8] Xueshuai Ge, Tieming Liu, Yaobin Xie, and Yuanyuan Zhang. A vulnerability automation exploitation method based on symbolic execution. In *International Conference on Electroning Information Engineering and Data Processing (EIEDP 2023)*, volume 12700 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, May 2023.

[9] Xueshuai Ge, Tieming Liu, Yaobin Xie, and Yuanyuan Zhang. A survey of automatic exploitation of binary vulnerabilities. In Xiaohao Cai and Badrul Hisham bin Ahmad, editors, *International Conference on Computer Network Security and Software Engineering (CNSSE 2023)*, volume 12714 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, Jun 2023.

[10] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, mar 2012.

[11] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *SIGPLAN Not.*, 47(6):193–204, Jun 2012.

[13] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in

binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[14] Api reference. https://docs.angr.io/en/latest/api.htmlangr.sim_manager.SimulationManager.

[15] Haoxin Tu. Boosting symbolic execution for heap-based vulnerability detection and exploit generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 218–220, 2023.

[16] Varghese Vaidyan and Bhaskar Rimal. Hybrid quantum artificial intelligence electromagnetic spectrum analysis framework for transportation system security. *Journal of Hardware and System Security*, December 2023.

[17] Varghese Mathew Vaidyan and Akhilesh Tyagi. Hybrid classical-quantum artificial intelligence models for electromagnetic control system processor fault analysis. In *2022 IEEE IAS Global Conference on Emerging Technologies (GlobConET)*, pages 798–803, 2022.