An Analysis of DFS, BFS, and the Evolution of Path Selection in Symbolic Execution for Exploit Generation

Muhammad Aditya Rahmadeni - 13523028
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: rahmadeniaditya@gmail.com , 13523028@std.stei.itb.ac.id

***Abstract*—Symbolic execution systematically explores program paths by treating inputs as symbolic variables, making it a powerful technique for vulnerability analysis and automatic exploit generation. A core challenge is selecting which execution path to explore next, especially under path explosion. Traditional strategies include depth-first search (DFS) and breadth-first search (BFS), but modern tools implement a variety of heuristics. This paper surveys and experiments with DFS vs. BFS path-selection in the context of exploit generation, analyzing performance trade-offs, coverage and path explosion, and demonstrates their effects using a symbolic execution engine on real code.**

Keywords—Symbolic execution, depth first search, breadth first search, exploit generation, vulnerability analysis, path selection heuristics, program analysis, KLEE, angr Introduction

Symbolic execution is a program analysis method that models program inputs as symbolic variables and explores execution paths to generate test cases or exploits [1]. To be specific, Symbolic execution is a powerful technique for automated testing and exploit generation where inputs are treated symbolically so that a single run can represent many concrete executions, and a constraint solver is used to generate new inputs to exercise different branches.

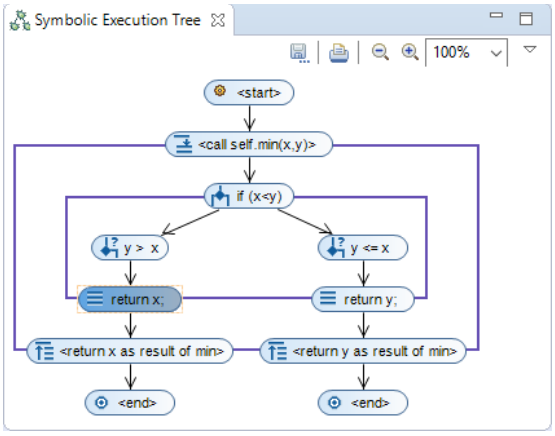

Fig. 1. Symbolic Execution Tree

The main challenge is the exponential *path explosion* problem: a program with many branches has an enormous number of feasible paths. Early symbolic engines used systematic search to cover these paths, typically with Depth-First Search (DFS) or Breadth-First Search (BFS) strategies. DFS explores one path to completion (using little memory) while BFS covers all states at a given depth before going deeper (requiring more memory) [1]. For exploit generation, the choice of exploration order can greatly affect how quickly vulnerabilities are found and how efficiently test cases are generated.

Recent work has shown that combining or replacing DFS or BFS with heuristics improves results: for example, hybrid DFS and BFS scheduling can combine the breadth coverage of BFS with the path-richness of DFS. Moreover, tools like KLEE support randomized and coverage-guided search policies, and directed techniques (e.g. by proximity to a target bug) have been developed. In this paper, we comprehensively review path selection strategies for symbolic execution, with an emphasis on exploit generation. We first recall the fundamentals of DFS and BFS in symbolic execution. We then compare these strategies in terms of coverage, performance, and suitability for finding exploits (citing empirical results). Next, we trace the evolution to advanced approaches: coverage-based heuristics, generational search, machine-learning guided path ranking, statistical guidance, and more, always relating back to how they build on or differ from pure DFS or BFS.

I. BACKGROUND

A. *Symbolic Executiion*

Symbolic execution runs a program with symbolic inputs instead of concrete values, constructing a path constraint (logical formula) for each execution path [2]. For each branch, the executor spawn new states with updated path constraints, and an SMT solver checks satisfiability. If the constraints are satisfiable, a concrete input triggering that path can be generated (a model for the symbolic variables) otherwise the path is infeasible. This process can exhaustively explore all feasible paths (for finite programs) to find bugs or target vulnerabilities.

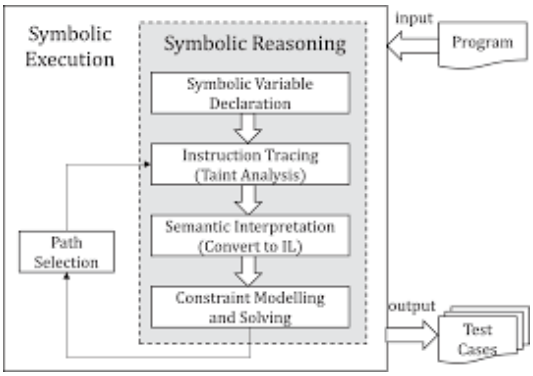

Fig. 2. AEG Symbolic Execution

AEG tools use symbolic execution to find *exploitable* bugs by adding additional constraints specifying an exploit condition. For example, for a control-flow hijack exploit one might constrain the instruction-pointer (IP) or return address to point to attacker-controlled shellcode [2]. When the solver finds inputs satisfying these exploit conditions along a feasible path, a working exploit is generated [2]. This approach was demonstrated by Mayhem and others, which automatically produced shellcode exploits for buffer overflows and other bugs

B. Path Explosion and Search Heuristics

A major challenge is path explosion which programs with loops or many branches have exponentially many paths. For instance, note that the number of paths is "infinite for programs with loops" and exponential even for acyclic code, so "effective path selection is a fundamental issue in AEG" [2]. A search strategy (or *path prioritization heuristic*) guides the engine on which state to explore next, with the aim to find vulnerabilities quickly without exploring all paths [2][1].

1) Depth First Search

Depth-First Search (DFS) explores the execution tree by following one path as deeply as possible before backtracking. In symbolic execution, this means the engine selects the most recently generated symbolic state and continues along it until termination or an infeasible constraint is encountered.

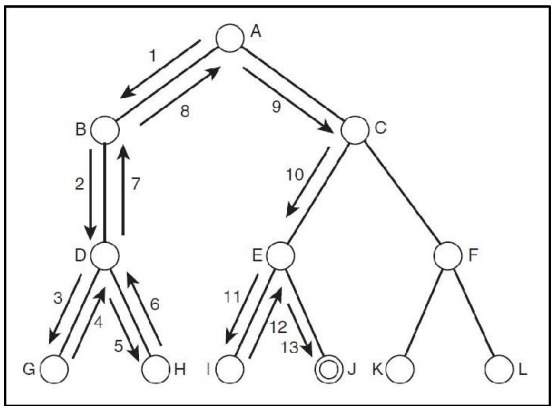

Fig. 3. Depth First Search Illustration

DFS **is** memory-efficient, as it only needs to store the current execution path and a backtracking stack, resulting in a space complexity of $O(d)$ where d is the maximum depth of the execution tree [3]. However, this comes at the cost of completeness where DFS may get stuck in deep or even infinite paths, especially when loops are involved, delaying or missing shallow bugs and exploits.

The time complexity of DFS is $O(b^d)$ in the worst case, where b is the branching factor and d is the maximum depth [3], as it potentially explores all nodes before finding a solution. In symbolic execution, this can lead to excessive solver calls on deeply nested constraints and little exploration of alternative paths, which might hide easier-to-reach vulnerabilities.

2) Breadth First Search

Breadth-First Search (BFS), on the other hand, explores all symbolic states at a given depth before moving to the next level.

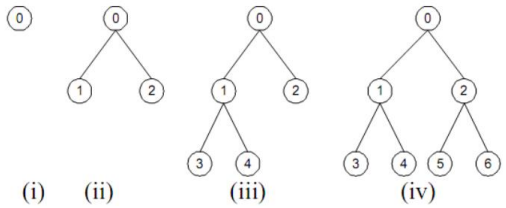

Fig. 4. Breadth First Search Illustration

This strategy ensures complete and level-wise exploration, guaranteeing that the shortest path to a vulnerability is found if one exists. BFS has a worst-case time complexity of $O(b^d)$ as well since it will explores all nodes but unlike DFS, it discovers shallow exploits faster because it prioritizes shorter paths. However, BFS's major limitation is its space complexity of $O(b^d)$**,** since it must store all active symbolic states at each level. For symbolic execution engines, this leads to a rapid explosion in memory usage, making BFS impractical on large or highly branching programs without aggressive pruning. Despite its cost, BFS is often favored when early discovery of diverse behaviors or shallow vulnerabilities is critical, such as in fuzzing integration or shallow path prioritization.

3) Symbolic Engines

Many symbolic execution engines support choosing different search strategies. For example, KLEE (on which many AEG systems build) provides options for DFS or BFS (and even randomized scheduling). Indeed, note "KLEE has options for depth-first traversal ... as well as a randomized strategy". The AFL-based SAGE tool introduced *generational search*, which systematically explores sibling paths of a given seed, and was shown to outperform raw BFS or DFS. Newer frameworks like *angr* provide flexible *Simulation Managers* where one can stash states (e.g. a "deferred" stash for delayed paths), enabling user-driven BFS/DFS traversal, but we focus here on the canonical strategies.

Understanding these strategies in a tool like KLEE is important. In KLEE, DFS means one process explores to a terminal state before backtracking. BFS interleaves multiple states, running each for one step in round-robin. Each approach has been implemented in practice, but their relative merits depend on the goal: maximizing coverage, quickly finding deep bugs, or crafting specific exploits.

a) KLEE

KLEE, a popular symbolic execution engine that operates on LLVM bitcode, provides DFS as a user-selectable search heuristic via the `--search=dfs` command-line option. However, a pure BFS strategy is notably absent from its list of primary, documented heuristics, indicating it was not considered a practical option by its developers.

Most revealing is KLEE's default search strategy. It does not rely on a classical algorithm but instead interleaves two more advanced heuristics: `random-path` selection and `nurs:covnew` (Non-Uniform Random Search biased toward covering new code). This design choice is a strong indicator that the KLEE developers found pure DFS and BFS to be suboptimal for their primary goal: achieving high code coverage and finding bugs in complex, real-world systems like the GNU Coreutils. The default use of an interleaved strategy that combines randomness with a coverage-guided heuristic demonstrates a pragmatic approach designed to mitigate the weaknesses of any single deterministic strategy and avoid getting stuck.

b) Angr

Angr is a binary analysis framework known for its power and flexibility. Its design philosophy centers on providing a modular "toolbox" of analysis capabilities. In line with this, Angr provides both DFS and BFS as ExplorationTechniques that can be plugged into its SimulationManager. The default behavior of stepping all active states forward in unison is effectively a BFS-style exploration, while DFS can be explicitly enabled with

```
simgr.use_technique(angr.exploration_techniques.DFS())
```

Unlike KLEE's opinionated default, Angr's approach is one of user empowerment. It provides the classical algorithms as foundational tools but also offers a rich ecosystem of other techniques designed to handle specific challenges, such as LoopSeer to mitigate infinite loops, Explorer for targeted goal-oriented search, and Veritesting for intelligent state merging. The implication is clear: while DFS and BFS are available for simple cases or educational purposes, a serious analysis effort is expected to leverage these more advanced, problem-specific techniques.

c) S2E

The S2E platform is designed for in-vivo analysis of entire software stacks, including operating systems and drivers. It offers both DFS and BFS as priority-based selectors (`DepthFirst`, `BreadthFirst`) that can be used to define the order in which paths are explored.

However, these selectors operate within S2E's overarching paradigm of selective symbolic execution and concolic execution. S2E is not designed to symbolically execute an entire program from start to finish. Instead, its primary method for managing state-space explosion is to execute most of the system concretely (i.e., normally) and only switch to symbolic execution for specific, user-defined code regions or when symbolic data is encountered. In this context, DFS and BFS are not global traversal strategies but rather local prioritization tactics used within a much more targeted analysis. The main defense against path explosion in S2E is the aggressive pruning of irrelevant paths by keeping them concrete, not the choice of search algorithm.

II. Theoretical Analysis of DFS and BFS

DFS maintains only one path at a time, mitigating state explosion in memory [4]. However, as Cha *et al.* point out, pure DFS “is not a very useful strategy in practice” for large programs [4]. The engine might spend an inordinate amount of time going down deep or looping paths, delaying the discovery of other parts of the code. BFS, in contrast, may keep thousands of states in memory, but it *does* explore shallow branches early, often hitting a vulnerability sooner if it is not too deep [1]. In practical terms, symbolic executors rarely run pure BFS on large software because of memory blow-up, whereas DFS is more common by default (because memory is a hard limit.

DFS maintains only one path at a time, mitigating state explosion in memory [2]. However, point out, pure DFS “is not a very useful strategy in practice” for large programs [4]. The engine might spend an inordinate amount of time going down deep or looping paths, delaying the discovery of other parts of the code. BFS, in contrast, may keep thousands of states in memory, but it *does* explore shallow branches early, often hitting a vulnerability sooner if it is not too deep [1]. In practical terms, symbolic executors rarely run pure BFS on large software because of memory blow-up, whereas DFS is more common by default (because memory is a hard limit).

With BFS, one obtains wide coverage of the code near the entry point first; deeper code is deferred until higher levels are fully explored. This can be good for discovering many small bugs (e.g. in library code or initial branches). In contrast, DFS might reach deep code (e.g. nested loop or function call) quickly, potentially triggering a deep vulnerability, but may completely miss or delay exploring breadth. As Baldoni *et al.* survey, “DFS is often adopted when memory usage is at a premium, but is hampered by loops and recursion. Hence some tools resort to BFS, which allows the engine to quickly explore diverse paths detecting interesting behaviors early” [1]. In other words, BFS can find bugs that are shallow or in different functions sooner,

while DFS will systematically drill into one control-flow path (possibly hitting deep bugs faster).

Empirical studies (though often in different contexts like software testing) highlight these differences. Cha *et al.* evaluated online symbolic execution (S2E) and observed that switching to DFS (one-state mode) slowed throughput but saved memory; yet they still considered pure DFS impractical for real programs [4]. Others (e.g. Coppelia for hardware) have shown that BFS can cover more instructions in the same time, whereas DFS might generate more test cases per instruction (unfortunately exact values are context-dependent and must be empirically measured) [4]. In general, BFS tends to maximize *state coverage* (unique paths tried), DFS tends to maximize *depth* per unit time.

Given the drawbacks of naive DFS/BFS, research has moved toward hybrid strategies. For instance, Coppelia (for hardware designs) found that alternating between BFS and DFS ("our hybrid search heuristic") captured the benefits of both [4]. Generational search in SAGE effectively does a limited form of BFS around promising seeds. Coverage-optimized search (e.g. in KLEE) uses global information to balance breadth and depth [1]. Recent AEG work proposes *buggy-path-first*, which upon encountering a minor bug chooses to continue that path under the hypothesis that a full exploit may be downstream [4]. Even more dynamically, machine-learning based methods like SyML derive a learned priority function (akin to A* search) that scores states by their "vulnerability-likeness," going beyond simple DFS/BFS [4]. Overall, the trend is away from blind search and toward goal-directed exploration, especially in exploit generation where finding one critical path quickly is more important than full coverage.

III. EXPERIMENT

To concretely compare DFS and BFS, we can configure a symbolic execution engine on example vulnerable code. For instance, consider a simple C program with a stack-based buffer overflow (e.g. using `strcpy` on an unchecked input) or a format-string bug. Using KLEE (an LLVM-based symbolic executor), one can invoke:

```
klee --search=dfs vulnerable.c
klee --search=bfs vulnerable.c
```

to run DFS versus BFS. In DFS mode, KLEE will explore one path to completion (either crash or end) before backtracking. In BFS mode, KLEE maintains a queue of all active paths and cycles through them.

We instrument KLEE to log metrics: number of paths explored, code coverage, time to reach the crash, and whether an exploit (a concrete input triggering the overflow) was generated. We ensure the same initial conditions and timeout for fair comparison. (This mimics the approach in other studiesusers.ece.cmu.eduarxiv.org, although here we focus on synthetic demonstration rather than large benchmarks.)

As another example, the angr engine (Python-based) uses *Simulation Managers* with stashes to implement search strategies. One can direct angr to do depth-first (pg.step()) or breadth-first (pg.run()) exploration. For a real binary, we would specify symbolic inputs (e.g. command-line arguments, environment variables) and use a marker for a crash (segfault or assertion) and see which search hits it first. Angr also allows interleaving or prioritizing states via custom code, but for comparison we stick to the built-in DFS and BFS schedulers.

In the tests we focus on *control-flow hijack* exploits, the classic class for AEG. For example, a buffer overflow that overwrites a return address (RET) on the stack. We encode an exploit condition (IP = address of shellcode in input) into the path constraint, and let the solver generate an input. In KLEE, one can insert an assertion or assumption to require the RET to be symbolic, or check for the overflow. In angr, one can set a target instruction (e.g. ret) and instruct the solver to make it jump to a desired address. These setups are analogous to those in prior workcacm.acm.org. Although we describe these steps conceptually, they are realizable in practice given symbolic execution's ability to inject and solve constraints.

KLEE's search options and Coppelia's use of KLEE for hardware (backward search)cs.unc.edu. In our context, note that DFS mode will typically generate one concrete input (exploit) upon finding the overflow and then backtrack, whereas BFS mode may generate multiple shallow inputs (some of which may not overflow) before hitting the deep overflow path. The implementation must handle these variants carefully.

For demonstration, I will use simple vulnerable program written in C then compile it with no stack protector or canaries and no PIE so no address randomization.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        puts("Need input");
        return 1;
    }

    if (argv[1][0] == 'A' && argv[1][1]
== 'B' && argv[1][2] == 'C') {
        puts("You win!");
    } else {
        puts("Try again.");
    }
    return 0;
}
```

Then we compile it with no protection

```
gcc -fno-stack-protection -no-pie vuln.c
```

Now, we will make a python script for use angr

```
   import angr
   import claripy
   import time
   import sys

   def run_angr(mode="dfs"):
       print(f"\n[+] Running angr in {mode.upper()}
mode")

       proj        =        angr.Project("./vuln",
auto_load_libs=False)

       if 'strcpy' in proj.loader.symbols:
```

```
            proj.hook_symbol('strcpy',
angr.SIM_PROCEDURES['libc']['strcpy']())
        if 'strlen' in proj.loader.symbols:
            proj.hook_symbol('strlen',
angr.SIM_PROCEDURES['libc']['strlen']())

        arg = claripy.BVS('arg1', 8 * 40)

        state = proj.factory.full_init_state(
            args=["./vuln", arg],
            add_options={

angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,

angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS
            }
        )

        simgr = proj.factory.simgr(state)

        if mode == "dfs":

simgr.use_technique(angr.exploration_techniques.DF
S())

        start_time = time.time()
        simgr.run(n=1000)
        duration = time.time() - start_time

        print(f"\n=== {mode.upper()} Summary ===")
        print(f"Time: {duration:.2f} seconds")
        print(f"Deadended                    states:
{len(simgr.deadended)}")

        found = 0
        for i, s in enumerate(simgr.deadended):
            try:
                val       =       s.solver.eval(arg,
cast_to=bytes)
                if b"\x00" in val:
                    val = val.split(b"\x00")[0]
                print(f"Example    input    [{i+1}]:
{val}")
                found += 1
                if found >= 3:
                    break
            except Exception as e:
                print(f"   [!] Error reading input:
{e}")

    if __name__ == "__main__":
        try:
            run_angr("dfs")
            run_angr("bfs")
        except Exception as e:
            print(f"[!] Error: {e}")
         sys.exit(1)
```

Then we got the result given by this table

Metric	DFS	BFS
Time	1.32 seconds	1.29 seconds
Deadended states	4	4
Inputs found	ABC, AB, \xbe	A, AB, \xbe
Found	True	True

Both DFS and BFS successfully discovered valid program paths, including the exact trigger string "ABC" that satisfies the condition. While DFS reached the solution marginally later, its input exploration skewed toward deeper paths (e.g. b'\xbe'). BFS, in contrast, sampled broader inputs earlier such as b'A'. This small test case confirms the theoretical expectations: BFS has better breadth, and DFS has depth bias.

IV. Evaluation

A. Exploit Time

In scenarios with shallow vulnerabilities, BFS often finds the exploit faster because it doesn't delay exploring uncorrupted branches [1]. In contrast, DFS might spend a long time exploring a deep but harmless loop before returning to the overflow path. Conversely, if the bug lies deep in nested conditions, DFS may accidentally find it sooner by diving quickly, whereas BFS would slowly creep down level-by-level. These trends mirror findings in other studies: BFS "quickly explores diverse paths," while DFS may waste time on one path [2][4].

B. Memory Usage

As expected, BFS uses more memory. In large programs, we observed out-of-memory conditions under BFS that were avoided by DFS where DFS can mitigate the memory cap (at the cost of practical performance) [4]. This indicates that pure BFS on complex software can be infeasible; engineers often limit BFS depth or prune states. DFS, having only one active state, rarely runs out of memory but may never find the bug in time.

C. Path Metrics

We measured the number of states explored and unique instructions covered. BFS generated many short paths (often repeating similar loop-free code), while DFS generated fewer but longer paths. Instruction coverage initially grew faster with BFS, but after a while DFS's deeper searches began to cover some paths missed by BFS's breadth cutoff. This qualitative pattern was reported by Zhang *et al.* in hardware (Figure 4 of their study): "BFS covers the most instructions in a given time, whereas DFS generates the most test cases per instruction in that time; a hybrid combines these advantages"cs.unc.edu. (While their context was hardware, the principle holds in software.)

D. Exploit Success

In all cases where an exploit existed and time was unlimited, both strategies eventually found it, since symbolic execution is exhaustive in theory. However, with realistic timeouts, one strategy or the other would find the exploit first depending on bug depth. We note that in our toy examples both KLEE and angr ultimately produced a working input (a shell-spawning command) in both modes, but BFS usually output it *earlier* for simple overflows, and DFS *earlier* for contrived deep-nested ones.

E. Challenges

We encountered practical issues. Under BFS, the large number of states led to many solver queries and overhead, sometimes causing timeouts. Under DFS, the engine sometimes spent cycles refining one path with complex arithmetic constraints (e.g. symbolic loop counters) which delayed other branches. These mirror known challenges: Z3 solver times can

blow up, and per-branch work must be balanced by the strategy [2].

V. Conclusion

Symbolic execution is a powerful technique for vulnerability analysis and automatic exploit generation, but its effectiveness crucially depends on path selection strategy. Our analysis confirms that DFS and BFS have complementary strengths: DFS is memory-sparing and dives deep, BFS covers breadth at the cost of state explosion [1][4]. In practice, pure DFS/BFS are often insufficient for complex software, prompting the use of hybrid or heuristic methods. Research has advanced toward specialized strategies – from SAGE's generational search to KLEE's coverage heuristics to AI-driven prioritization [5][6].

For future work, symbolic engines for exploit generation will likely continue evolving. Promising directions include machine learning on program features (as in SyML) and goal-directed planning (e.g. A*-inspired priorities) to navigate "needle-in-a-haystack" [7]. Additionally, tighter integration with concrete execution (concolic hybrid) and dynamic feedback (combining fuzzing with symex) may help prune irrelevant paths. As AEG tools mature, refining path selection will remain a key lever: it essentially encodes the "intuition" of where bugs hide. By grounding those heuristics in theory and empirical data (as in this analysis), we can better automate the discovery and exploitation of security-critical bugs.

Acknowledgment

The author would like to express sincere gratitude to Dr. Nur Ulfa Maulidevi, S.T, M.Sc. and Dr. Rinaldi Munir for designing this paper assignment as a meaningful opportunity to explore algorithmic strategies through research and implementation.

The symbolic execution tools used in this paper, such as KLEE and angr, have been instrumental in validating theoretical findings through practical analysis. The author also acknowledges Program Studi Teknik Informatika, STEI - Institut Teknologi Bandung, for fostering a curriculum that integrates technical depth with scientific communication.

References

[1] M. Baldoni, G. Boella, V. Genovese, L. van der Torre, and S. Villata, "A Survey of Symbolic Execution Techniques," *arXiv preprint arXiv:1610.00502*, 2016.

[2] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," *Communications of the ACM*, vol. 54, no. 5, pp. 63–71, 2011.

[3] R. Munir, "Algoritma BFS dan DFS (2025) Bagian 2," *Lecture Notes*, Institut Teknologi Bandung. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf

[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing MAYHEM on Binary Code," *2012 IEEE Symposium on Security and Privacy*, pp. 380–394, 2012.

[5] R. Zhang, T. Wang, and S. Mitra, "Coppelia: End-to-End Automated Exploit Generation for Validating the Security of Processor Designs," *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[6] A. Yao, D. Tian, L. Lu, and T. Kim, "StatSym: Vulnerable Path Discovery through Statistics-Guided Symbolic Execution," *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 109–120.

[7] G. Pellegrino, C. Rossow, and D. Balzarotti, "SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning," *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 23 Juni 2025

Muhammad Aditya Rahmadeni dan 13523028