BOOSTING SYMBOLIC EXECUTION FOR VULNERABILITY DETECTION

HAOXIN TU

SINGAPORE MANAGEMENT UNIVERSITY
2025

Boosting Symbolic Execution for Vulnerability Detection

by

Haoxin Tu

Submitted to School of Computing and Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Dissertation Committee:

Lingxiao JIANG (Supervisor / Chair)
Prof. of School of Computing and Information Systems
Singapore Management University

Xuhua DING (Co-Supervisor)
Prof. of School of Computing and Information Systems
Singapore Management University

David LO
Prof. of School of Computing and Information Systems
Singapore Management University

Marcel BÖHME
Prof. of Max Planck Institute for Security and Privacy (MPI-SP)

Singapore Management University
2025

I hereby declare that this PhD dissertation is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in this dissertation.

This PhD dissertation has also not been submitted for any degree in any university previously.

Haoxin Tu

9th May 2025

Boosting Symbolic Execution for Vulnerability Detection

Haoxin Tu

Abstract

Software systems written by humans tend to be unreliable and insecure, hence bugs or vulnerabilities in them are inevitable. Symbolic execution has shown considerable potential in detecting diverse types of software bugs and also vulnerabilities that have severe security implications. However, existing symbolic execution engines still suffer from at least three fundamental limitations in *memory modeling*, *path exploration*, and *structured input generation*, which significantly impede existing engines from efficiently and effectively detecting software bugs and vulnerabilities.

The objective of this dissertation is to boost existing symbolic execution engines, by designing *a new memory model*, *two new path exploration strategies*, and *a new test input generation solution* to alleviate three key limitations, to facilitate automatic vulnerability detection. Specifically, in the first work, we propose SYMLOC, a vulnerability detection system that designs a new symbolic memory model. In the second work, we propose FASTKLEE, a faster path exploration solution achieved by reducing redundant bound checking during execution. In the third work, we propose VITAL to perform vulnerability-oriented path exploration for effective vulnerability detection. In the fourth work, we propose COTTONTAIL, a LLM-driven concolic execution engine that could effectively generate highly structured test inputs for detecting vulnerabilities in parsing test programs.

The prototypes implemented in the dissertation have found many unknown vulnerabilities (e.g., *buffer overflow* and *memory leakage*) in widely used software systems, some of which are assigned with CVE (e.g., CVE-2024-55061).

Acknowledgements

The past four years at SMU have been an invaluable and transformative chapter of my life. As my first extended experience living and studying outside of China, this journey in Singapore has been both intellectually enriching and personally rewarding. Each day brought new challenges, yet the process of conducting research and overcoming obstacles has been immensely fulfilling. I feel truly privileged to have had the opportunity to pursue this PhD, a journey that would not have been possible without the unwavering support and guidance of many individuals. To them, I extend my deepest gratitude.

First and foremost, I extend my deepest gratitude to my advisors, Prof. *Lingxiao Jiang* and Prof. *Xuhua Ding*. Prof. Jiang has imparted essential lessons on conducting rigorous research—teaching me how to identify meaningful research questions, formulate novel ideas, write effectively, and deliver compelling presentations. His dedication to research has been truly inspiring, and I feel privileged to have engaged in insightful discussions with him, reviewed numerous papers, and explored diverse research topics under his guidance. I was also fortunate to be supervised under Prof. Ding during the first two years of my PhD on a project involving symbolic execution for kernel analysis. Through this collaboration, I gained invaluable insights into his meticulous approach to problem-solving and his philosophical perspective on addressing practical research challenges. I am deeply grateful for their unwavering guidance, patience, and encouragement, which have made my PhD journey both intellectually fulfilling and personally rewarding. Their mentorship has been instrumental in shaping my research and academic growth, and this dissertation would not have been possible without their support.

In addition, I would also like to express my heartfelt gratitude to my supervisor Prof. *He Jiang* at Dalian University of Technology (DUT) for his unwavering support in my pursuit of the dual degree program between DUT and SMU. His encouragement and guidance played a pivotal role in enabling me to embark on

this academic journey, which has been both intellectually enriching and personally transformative. The opportunity to study in this program has provided me with a broader perspective, exposed me to diverse research environments, and allowed me to engage with scholars from different backgrounds. This experience has not only strengthened my academic foundation but also shaped my personal and professional growth in ways I could not have imagined.

Furthermore, I would like to extend my sincere gratitude to my dissertation committee members: Profs. *Marcel Böhme*, *David Lo*, *Xuhua Ding*, and *Lingxiao Jiang*. Their invaluable suggestions and insightful feedback throughout my Qualifying Exam, Oral Proposal Defense, and Final Dissertation Defense have significantly contributed to shaping and refining my research. Their expertise and guidance have been instrumental in strengthening the quality and depth of my dissertation, and I deeply appreciate their time and effort in supporting my academic journey. I would also like to express my heartfelt appreciation to Ms. *BOO Chui Ngoh*, Ms. *Caroline TAN*, Prof. *Baihua Zheng*, Ms. *ONG Chew Hong*, Ms. *FONG Wei Ling Pauline*, and Ms. *Felicia TAN Sze Ngee* for their unwavering support in administrative and technical matters. Their assistance has been crucial in ensuring a smooth and efficient research experience, allowing me to focus on my studies without unnecessary obstacles. I am truly grateful for their dedication and support.

Beyond that, during my PhD journey, I was fortunate to collaborate with many outstanding researchers beyond my advisors. I would like to express my sincere gratitude to whom I have had the privilege of co-authoring papers: Prof. *Debin Gao*, Prof. *Marcel Böhme*, Dr. *Seongmin Lee*, Dr. *Jiaqi Hong*, *Pansilu Pitigalaarachchi*, *Imam Nur Bani Yusuf*, *Haiqing Qiu*, Prof. *Zhilei Ren*, Prof. *Xiaochen Li*, Dr. *Zhide Zhou*, Dr. *Dong Liu*, and Dr. *Yixuan Tang*. Their valuable contributions, discussions, and guidance have been instrumental in shaping my academic growth. I am also deeply grateful to my close friends at SMU: *Lei Wang*, *Xin Zhou*, *Ling Cheng*, *Qian Shao*, *Fengzhu Zeng*, *Xiaosen Zheng*, *Xiaoguo Li*, and *Zhi Chen* and at MPI-SP in Germany: *Jing Liu*, *Changyang He*, and *Fengyang Lin*, who provided unwavering

support and helped me navigate the challenges and stresses of PhD life. Their companionship made this journey far more manageable and enjoyable. Additionally, I extend my heartfelt appreciation to my friends at DUT—*Xu Zhao*, *Zun Wang*, *Yue Ma*, *Hao Lin*, and *Yue Gao*—who stood by my side during my master's studies and the early stages of my PhD. Their encouragement have been invaluable.

Moreover, I would like to express my deepest gratitude to my mother and brother for their endless love and unwavering support throughout the past four years of my PhD journey. Their encouragement and steadfast presence, even from afar, have been a constant source of strength for me. The one or two video calls we shared each week have been our only way of maintaining our close bond over these years, and their warmth and reassurance have meant the world to me. I sincerely hope that my PhD graduation will make my family proud and bring them peace of mind, knowing that they no longer need to worry about my studies and future career.

Lastly, I would like to express my heartfelt gratitude to my girlfriend, Dr. *Yuxian*, for her unwavering love and support throughout my PhD journey. The time we spent together—cooking, traveling, and studying—has been filled with joy and has provided me with invaluable companionship amidst the challenges of academic life. Her remarkable talent in creating presentation slides has always impressed me, and I remain her most devoted admirer. I am immensely grateful for her patience, sacrifices, and dedication. I truly appreciate everything she has done for me, and I am forever grateful for her presence in my life.

As I conclude this journey, I am filled with immense gratitude for all the people who have supported, guided, and encouraged me along the way. My PhD experience has been shaped not only by academic challenges and achievements but also by the unwavering kindness and generosity of my advisors, collaborators, friends, and family. While this dissertation marks the completion of one chapter, the lessons I have learned and the relationships I have built will continue to inspire and guide me in the future. To all who have been a part of this journey, I extend my deepest appreciation—this achievement is as much yours as it is mine.

Table of Contents

1 Introduction **1**

1.1 Background 1

1.2 Motivations 2

1.3 Dissertation Statement and Research Objectives 5

1.4 Overview of Dissertation Works 6

1.4.1 A New Memory Model 6

1.4.2 Two New Path Exploration Strategies 7

1.4.3 A New Test Case Generation Solution 8

1.5 Technical Contributions and Research Impact 8

1.6 Authorship Attribution Statement 10

1.7 Dissertation Organization 13

2 Literature Review **14**

2.1 Memory Modeling for Symbolic Execution 14

2.1.1 Full Symbolic Memory 14

2.1.2 Partial Symbolic Memory 15

2.2 Path Exploration for Symbolic Execution 17

2.2.1 Efficient Path Exploration 17

2.2.2 Effective Path Exploration 20

2.3 Highly Structured Test Input Generation for Symbolic Execution . . 23

2.4 Other Related Work 25

3 Concretely Mapped Symbolic Memory Locations for Memory Error Detection **28**

3.1 Objective . . . 28

3.2 Preliminaries and Motivating Examples . . . 31

3.2.1 Common Memory Errors . . . 31

3.2.2 Illustrative Examples . . . 33

3.3 Design of SYMLOC . . . 38

3.3.1 Address Symbolization . . . 39

3.3.2 Symbolic Memory Operations and Tracking . . . 43

3.3.3 Implementation of SYMLOC . . . 50

3.4 Evaluation . . . 52

3.4.1 Spatial Memory Errors Detection Capability . . . 52

3.4.2 Temporal Memory Errors Detection Capability . . . 60

3.5 Case Studies . . . 66

3.5.1 Case 1: Single *NULL* Pointer Dereference in `rm` . . . 66

3.5.2 Case 2: Consecutive *NULL* Pointer Returns in `Make` . . . 69

3.5.3 Can existing tools detect the errors? . . . 71

3.6 Discussion . . . 72

3.6.1 Comparison with Other Existing Approaches . . . 72

3.6.2 Integration with Other Techniques . . . 78

3.6.3 Threats to Validity . . . 79

3.6.4 Limitations of SYMLOC . . . 79

3.7 Summary . . . 80

4 Faster Path Exploration via Reducing Redundant Bound Checking of Type-Safe Pointers **81**

4.1 Objective . . . 81

4.2 Usage Example . . . 84

4.2.1 Phase I: Perform Type Inference . . . 84

4.2.2 Phase II: Conduct Faster Symbolic Execution 84
4.3 Design of FASTKLEE 85
4.3.1 Type Inference System 86
4.3.2 Customized Memory Operation 87
4.3.3 Implementation 88
4.4 Evaluation 88
4.4.1 Experimental Setup 88
4.4.2 Evaluation Results 89
4.5 Discussion 90
4.6 Summary 92

5 Vulnerability-Oriented Path Exploration via Unsafe-Pointer Guided Monte Carlo Tree Search 93

5.1 Objective 93
5.2 Preliminaries and Motivating Example 97
5.2.1 Preliminaries 97
5.2.2 Motivating Example 99
5.3 Design of VITAL 101
5.3.1 Acquisition of Unsafe Pointers via Type Inference System . 101
5.3.2 Type-unsafe Pointer-guided Monte Carlo Tree Search 102
5.3.3 Implementation of COTTONTAIL 110
5.4 Evaluation 111
5.4.1 Capability of Covering Unsafe Pointer 112
5.4.2 Capability of Vulnerability Detection 115
5.4.3 Ablation Study 118
5.4.4 Practical Application of VITAL 119
5.5 Discussion 121
5.5.1 Overhead of Pointer Type Inference 121
5.5.2 Impact of Different Configurations 121

5.5.3 Threats to Validity . . . 122
5.5.4 Limitations . . . 122
5.6 Summary . . . 123

6 Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation 124

6.1 Objective . . . 124
6.2 Background and Motivation . . . 129
6.2.1 Concolic Execution . . . 129
6.2.2 LLMs for Test Input Generation . . . 130
6.2.3 Investigation Study . . . 130
6.3 Design of COTTONTAIL . . . 133
6.3.1 Structure-aware Path Constraint Selection . . . 133
6.3.2 Smart LLM-driven Constraint Solving . . . 138
6.3.3 History-guided Seed Acquisition . . . 141
6.4 Evaluation . . . 143
6.4.1 RQ1: Comparison with Baseline Approaches . . . 143
6.4.2 RQ2: Ablation Studies . . . 147
6.4.3 RQ3: Vulnerability Detection Capability . . . 152
6.5 Discussion . . . 155
6.5.1 Implications . . . 155
6.5.2 API Costs for Running Experiments . . . 156
6.5.3 Threats to Validity . . . 156
6.5.4 Limitations . . . 157
6.6 Summary . . . 158

7 Conclusion and Future Work 159

7.1 Summary of Contributions . . . 159
7.2 Future Work . . . 160

List of Figures

3.1 Existing symbolic executors (e.g., KLEE) are only able to cover `Path-B` while missing `Path-A`, `C`, `D`, and the memory error in `Path-E` 33

3.2 Missing the detection of UAF errors 36

3.3 High-level design of SYMLOC 38

3.4 Overlapping allocation example 1: overlapping at the beginning . . 43

3.5 Overlapping allocation example 2: overlapping in the middle 43

3.6 Distribution of address-specific spatial memory errors detected by comparative approaches 55

3.7 Unique line coverage (measured by `gcov`): SYMLOC vs KLEE . . 58

3.8 Unique line coverage (measured by `gcov`): SYMLOC vs *symsize* . 58

3.9 Comparison of different symbolization modes in SYMLOC (*Full* symbolization mode VS *Random* symbolization mode) 59

3.10 Completeness of UAF error detection among static/dynamic analysis-based approaches (137 in total). 62

3.11 Completeness of DoF error detection among static/dynamic analysis-based approaches (283 in total. 63

3.12 Completeness of UAF error detection among symbolic execution-based approaches (137 in total). 63

3.13 Completeness of DoF error detection among symbolic execution-based approaches (283 in total). 64

3.14 A simple DoF error missed by KLEE [28] and Valgrind [146] 65

3.15 Case 1: memory error at `cycle-check.c:60` in `rm` 67
3.16 Execution flow of *NULL-pointer dereference* in Case 1 68
3.17 Case 2 : memory error at `output.c:605` in `Make-4.2` 69
3.18 Execution flow of *NULL-pointer dereference* in Case 2 70
3.19 Example compared with RAM [186] 73
3.20 Speedups comparison between MEMSIGHT-KLEE and SYMLOC . 76

4.1 Overview of IR-based *Traditional* SE and *Fast* SE 82
4.2 Scatter plot of the improvement in speedups 90
4.3 Improvement of speedups and time spent on type inference 91

5.1 Correlation between unsafe pointers and memory errors (Pearson's coefficient [46]: 0.93). 96
5.2 Motivating example (adapted from `Chopper` [189]) 99
5.3 CFG of function `f` shown in Figure 5.2(a) illustrating how to get the score of true/false states 107
5.4 Numer of detected *unique* memory errors by VITAL compared to comparative search strategies 114
5.5 Memory consumption comparison results of VITAL against `Chopper` [189] and CBC [209]. 116

6.1 Parser checking passing rate comparison between traditional constraint solver (i.e., `Z3`) and LLM-driven solver (designed in COTTONTAIL). 130
6.2 The *Solve-Complete* paradigm and LLM's response. In the upper box, the underlined text represents different formats, and the colored text enclosed indicates either path constraints or marked test input string. 131
6.3 High-level design of COTTONTAIL concolic execution engine 133
6.4 Sample parsing implementation code from MuJS 135

6.5 Duplicated path constraints in parsing logic (those path constraints aim to cover the same branch at Line 3 in Figure 6.4) 137
6.6 CoT prompts for LLM-driven seed generation 142
6.7 Line coverage comparison among COTTONTAIL and baseline approaches in 12 hours (*x-axis* indicates line coverage while *y-axis* represents the time) 145
6.8 Comparison results of parser checking passing rate (% in *y-axis*) against SYMCC and MARCO 146
6.9 Comparison results of normal and CoT prompts for constraint solving 149
6.10 Comparison results of w/ or w/o test case validator 149
6.11 Line coverage comparison among COTTONTAIL and variant approaches in 12 hours (*x-axis* indicates line coverage while *y-axis* represents the time) 150
6.12 Vulnerable function and vulnerability triggering input in Case Study 154

List of Tables

3.1 The benchmarks used in the evaluation, with their version, size, and the source lines of code (SLOC) 54

3.2 Results of the overall number of detected errors 55

3.3 Results of branch coverage (measured by `klee-stats`) and line coverage (measured by `gcov`) 56

5.1 Comparison results with existing search strategies 113

5.2 Results of time on detecting CVE vulnerabilities. 116

5.3 Comparison results with variant approaches 118

6.1 Open source libraries cross four different formats used in the evaluation (LOC: lines of code; Stars: GitHub stars) 144

6.2 Line and branch coverage comparison results against existing concolic execution engines SYMCC [160] and MARCO [102] 144

6.3 Results of path constraint selector design in COTTONTAIL 148

6.4 New Vulnerabilities Detected by COTTONTAIL 153

Chapter 1

Introduction

1.1 Background

Assuring the quality of software systems is of great importance for maintaining the software reliability, efficiency, and security. First, reliable software functions correctly implement all expected program behavior, increasing trust in its use for critical tasks. Second, well-optimized software that performs efficiently can handle more users or tasks, saving developing time and resources. Third, secure software protects against data breaches and other cyber threats, which is essential for user privacy and trustworthiness.

Detecting software defects is a key activity in the software development process to assure software quality. A defect is a broad term that refers to any deviation of the software from its expected behavior or requirements, such as bugs, errors, vulnerabilities, or any other issue that prevents the software from functioning as intended [18]. To be specific, a bug or an error is a flaw or mistake in a software program's code or design that causes it to produce an incorrect or unexpected result, or to behave in unintended ways (e.g., wrong compiler optimizations). A vulnerability (e.g., memory corporation) is a specific type of weakness in a software system that can have security implications, e.g., to be exploited by an attacker to perform unauthorized actions. Compared with vulnerabilities, some errors or bugs

can still be exploited, although with a lower possibility compared with vulnerability [14]. For the above reason and following existing study [64], in this dissertation, we will use the word *error*, *bug*, and *vulnerability* interchangeably.

Symbolic execution is a promising technique that has shown considerable increasing capabilities in bug and vulnerability detection. Technically, unlike dynamic execution, programs are executed with symbolic inputs instead of real ones, enabling the analysis of program behavior over a wide range of inputs in a single run. This approach can systematically explore possible execution paths within the program, helping to uncover hard-to-find errors and security vulnerabilities by constructing a mathematical representation of the paths that the program can take. Its strength lies in its ability to identify problematic inputs that lead to bugs or security flaws, which might be missed by conventional testing methods that use a limited set of test cases.

Benefiting from the advanced capabilities in automating the exploration of paths and generating test cases, symbolic execution has emerged as a widely used technique in several areas. This includes its application in generating test cases [28, 29, 155], identifying bugs [28, 86, 165], and aiding in debugging and repair processes [109, 137, 138, 147, 215]. It is also effectively employed in cross-checking [48, 112], conducting side-channel analysis [19, 20, 94], and developing exploits [9, 31, 99].

1.2 Motivations

Although the applications of symbolic execution are promising, existing symbolic execution engines still suffer from certain key fundamental limitations in *memory modeling*, *path exploration*, and *test input generation*, which obstruct the engines from detecting important bugs and vulnerabilities effectively and efficiently. Those limitations motivated us to propose advanced techniques and practical solutions to boost symbolic execution for better vulnerability detection.

Motivation 1: Why existing symbolic execution engines can not detect vulnerabilities effectively? Memory modeling issues restricts comprehensive program

semantics modeling, missing the opportunities of vulnerability detection.

Most existing symbolic execution engines, such as KLEE [28] and *symsize* [190], usually model the locations of dynamically allocated memory objects (e.g., `malloc` in C/C++) with concrete values, where each object is located at a fixed address. However, those address values should be *non-deterministic* during actual executions as they are dynamically allocated based on different run-time environments. The limitation may cause existing symbolic execution engines to miss detecting certain kinds of memory errors. First, since the engines do not encode memory addresses into path constraints, they may not be able to cover certain lines of code whose executions are dependent on the memory addresses and miss certain address-specific spatial memory errors such as *buffer overflow*. Second, without maintaining and tracking the constraints among the addresses of different memory objects, they may fail to reliably check for unsafe uses of memory objects and miss certain temporal memory errors such as *use-after-free* or *double-free*. Even though the engines allow symbolization of the memory addresses, without proper handling of read/write operations involving symbolic addresses, the engines can face memory state explosion and cannot explore the programs effectively.

Therefore, we need better ways to model the addresses of dynamically allocated memory objects for more effective detection of memory errors in programs.

Motivation 2: Now we have a more complete memory modeling of program semantics. Can we detect vulnerabilities effectively? No, path explosion challenges prevent the efficient and effective path exploration.

(1) Inefficient (redundant) Path Exploration

Many SE engines (e.g., KLEE [28] or Angr [173]) need to interpret certain Intermediate Representations (IR) of code during execution, which may be slow and costly. First, the number of interpreted instructions tends to be stupendous (usually billions only in one hour's run), and reducing overhead for the most frequently interpreted

ones (i.e., read/write) could potentially accelerate the execution. Second, type-safe is an important property in the program for preventing certain errors such as memory out-of-bound assessment. As proven by prior studies [104, 140, 145], a large portion of pointers in C programs to be read/written can be statically verified to be type-safe.

However, most of the existing IR-based SE engines treat all the pointers (memory addresses) equally and perform the bound checking for every pointer when interpreting read/write instructions during symbolic execution. Thus, a plethora number of bound checks performed during the interpretation is unnecessary, which may induce performance downsides. Therefore, a faster path exploration strategy is needed to speed up the execution performance.

(2) Ineffective (non-vulnerability-oriented) Path Exploration

Effective path exploration in symbolic execution plays a key role in covering more lines of code or detecting more bugs. Previous studies target random, breadth-first search, depth-first search or various heuristics, e.g., coverage-guided, uncovered-lines guided, and program spectrum-guided, to alleviate the path explosion challenge [28, 97, 120]. In short, path exploration strategies in most of the existing symbolic execution engines are not vulnerability-oriented.

However, covering more lines of code does not mean more bugs will be detected. Since unsafe pointer operations have a high risk of inducing vulnerabilities, a new path search strategy with the help of pointer checking is needed for effective path exploration toward vulnerability detection.

Motivation 3: Now we have better path exploration strategies. Can we detect vulnerabilities effectively? No, tricky vulnerabilities can only be triggered by highly structured test inputs, but existing engines can hardly generate them.

Testing certain software systems (e.g., compilers and interpreters) typically relies on structural inputs. Specifically, the main functionality of such software systems only can be examined after the successful parsing of structural test inputs. For

example, a build system such as Apache Maven first parses its input as an XML document and checks its conformance to a schema before invoking the actual build functionality. Document processors, Web browsers, compilers and various other programs follow this same check-then-run pattern. Although a few studies [85, 86] are focusing on addressing this problem, current symbolic execution engines are still far from generating highly structured test inputs practically at the scale.

Thus, the effective and efficient generation of structured inputs is needed for assuring the security of important software systems such as parsing test programs.

In short, to boost symbolic execution for vulnerability detection, we need to design new solutions for memory modeling, path exploration, and test input generation.

1.3 Dissertation Statement and Research Objectives

The primary goal of this dissertation is to *boost symbolic execution by designing novel techniques to address key limitations in memory modeling, path exploration, and test input generation, thereby improving its capabilities of vulnerability detection.*

We target the following three specific research objectives:

1. We aim to improve the memory modeling capabilities in symbolic execution to facilitate memory error (e.g., buffer overflow and use after free) detection capabilities of symbolic execution engines.
2. We target to improve both the efficiency and effectiveness of path exploration in symbolic execution engines by combining static program analysis techniques (i.e., static pointer analysis) with symbolic execution.
   (a) We first aim to investigate how the path exploration efficiency of symbolic execution can be improved by reducing the overhead of redundant bound checking of safe pointers.
   (b) We then plan to leverage the same idea, i.e., by combining the results of static pointer analysis and a search algorithm to assist symbolic execution for effective vulnerability-oriented path exploration.

3. We focus on boosting symbolic execution to generate structured test input for parsing test programs. The main idea is to unleash the power of Large Language Model (LLM) for smart constraint solving, where the solving can not only stratify the path constraints but also meet the validity of input syntax.

1.4 Overview of Dissertation Works

In the following, we summarize the dissertation works into three parts and describe the key research ideas in each part.

1.4.1 A New Memory Model

SYMLOC. We propose SYMLOC, a symbolic execution-based approach that uses concretely mapped symbolic memory locations to alleviate the limitations mentioned above. Specifically, a new integration of three techniques is designed in SYMLOC: (1) the symbolization of addresses and encoding of symbolic addresses into path constraints, (2) the symbolic memory read/write operations using a symbolic-concrete memory map, and (3) the automatic tracking of the uses of symbolic memory locations. We build SYMLOC on top of the well-known symbolic execution engine KLEE and demonstrate its benefits in terms of memory error detection and code coverage capabilities. Our evaluation results show that: for address-specific spatial memory errors, SYMLOC can detect 23 more errors in `GNU Coreutils`, `Make`, and `m4` programs that are difficult for other approaches to detect, and cover 15% and 48% more unique lines of code in the programs than two baseline approaches; for temporal memory errors, SYMLOC can detect 8%-64% more errors in the Juliet Test Suite than various existing state-of-the-art memory error detectors. We also present two case studies to show sample memory errors detected by SYMLOC along with their root causes and implications.

1.4.2 Two New Path Exploration Strategies

(1) Towards Efficient (Faster) Path Exploration

FASTKLEE. We propose FASTKLEE, a faster SE engine that aims to speed up execution via reducing redundant bound checking of type-safe pointers during IR code interpretation. Specifically, in FASTKLEE, a type inference system is first leveraged to classify pointer types (i.e., safe or unsafe) for the most frequently interpreted read/write instructions. Then, a customized memory operation is designed to perform bound checking for only the unsafe pointers and omit redundant checking on safe pointers. We implement FASTKLEE on top of the well-known SE engine KLEE and combined it with the notable type inference system `CCured`. Evaluation results demonstrate that FASTKLEE is able to reduce by up to 9.1% (5.6% on average) as the state-of-the-art approach KLEE in terms of the time to explore the same number (i.e., 10k) of execution paths.

(2) Towards Effective (Vulnerability-oriented) Path Exploration

VITAL. We propose VITAL, a new vulnerability-oriented path exploration for symbolic execution via a *type-unsafe* pointer-guided Monte Carlo Tree Search (MCTS). A pointer that is *type-unsafe* cannot be statically proven to be safely dereferenced without memory corruption. Our key hypothesis is that a path with more *type-unsafe* pointers is more likely to be vulnerable. To this end, VITAL drives a guided *MCTS* to prioritize paths in the symbolic execution tree that contain a larger number of *unsafe* pointers and to effectively balance the exploration-exploitation trade-off. We built VITAL on top of KLEE and compared it with existing path search strategies and chopped symbolic execution. In the former, the results demonstrate that VITAL could cover up to 90.03% more unsafe pointers and detect up to 57.14% more unique memory errors. In the latter, the results show that VITAL could achieve a speedup of up to 30x execution time and a reduction of up to 20x memory consumption to automatically detect known CVE vulnerabilities without prior expert knowledge. In

practice, VITAL also detected one previously unknown vulnerability (a new CVE ID is assigned), which has been fixed by developers.

1.4.3 A New Test Case Generation Solution

COTTONTAIL. We propose COTTONTAIL, a new Large Language Model (LLM)-driven concolic execution engine, to mitigate the limitations in existing concolic executors. A more complete program path representation, named Expressive Structural Coverage Tree (ESCT), is first constructed to select structure-aware path constraints. Later, an LLM-driven constraint solver based on a *Solve-Complete* paradigm is designed to solve the path constraints smartly to get test inputs that are not only satisfiable to the constraints but also valid to the input syntax. Finally, a history-guided seed acquisition is employed to obtain new highly structured test inputs either before testing starts or after testing is saturated. We have implemented COTTONTAIL on top of SymCC and evaluated eight extensively tested open-source libraries across four different formats (XML, SQL, JavaScript, and JSON). The experimental result is promising: it shows that COTTONTAIL outperforms state-of-the-art approaches (SYMCC and MARCO) by 14.15% and 14.31% in terms of line coverage. Besides, COTTONTAIL found 6 previously unknown vulnerabilities (six new CVEs have been assigned to them). We have reported these issues to developers, and 4 out of them have been fixed so far.

1.5 Technical Contributions and Research Impact

The technical contributions of this dissertation are summarized as follows:

- We propose SYMLOC, a novel approach that integrates three key techniques—address symbolization, a symbolic-concrete memory map, and symbolic memory tracking—to enhance symbolic execution for memory allocation modeling, improving memory error detection capabilities. SYMLOC outper-

forms state-of-the-art detectors on real-world benchmarks and identifies two previously overlooked address-specific memory errors with significant implications. To facilitate further research, we provide a replication package to support the development of optimized symbolic execution engines and advanced memory error detection techniques.

- We propose FASTKLEE, the first symbolic execution engine designed to accelerate IR-based symbolic execution by reducing interpretation overhead. To achieve this, we leverage a type inference system to classify pointers and implement a customized memory operation that eliminates redundant checks for type-safe pointers, thereby improving efficiency. We open-source FASTKLEE to promote further research and demonstrate its effectiveness, highlighting its potential to facilitate valuable path exploration and enhance symbolic execution performance.

- We propose VITAL, the first approach to perform vulnerability-oriented path exploration in symbolic execution for effective and efficient vulnerability detection. VITAL introduces a novel indicator—the number of type-unsafe pointers in a path—to approximate vulnerability proneness and leverages an unsafe pointer-guided Monte Carlo Tree Search algorithm to navigate vulnerability-focused path exploration. Extensive experiments demonstrate its superior performance in unsafe pointer coverage and memory error/vulnerability detection compared to state-of-the-art approaches. To foster further research, we release a replication package, including source code, benchmarks, and experiment scripts, enabling reproducibility and advancements in symbolic execution for vulnerability detection.

- We propose COTTONTAIL, the first LLM-driven concolic execution engine designed for highly structured test case generation in a white-box manner. To enhance its effectiveness, COTTONTAIL integrates three key techniques: structure-aware path constraint selection, smart LLM-driven constraint solving,

and history-guided seed generation. Extensive experiments demonstrate its superiority over existing concolic execution engines, with the ability to uncover five previously unknown vulnerabilities. To advance research at the intersection of program analysis and LLM, we open-source COTTONTAIL, providing a foundation for further innovation in ensuring software security.

Research Impact. This dissertation not only introduces methodological innovations but also demonstrates practical impact in program analysis and software testing. The implemented tools, SYMLOC, VITAL, and COTTONTAIL, have reported more than 10 important memory-related vulnerabilities over widely used software systems. In many cases, our reports have received positive acknowledgment from developers, encouraging further submission of high-quality bug reports. Notably, a significant number (7 so far) of newly discovered memory-related vulnerabilities have been assigned new CVE identifiers, underscoring the security relevance and real-world impact of our findings. Furthermore, the proposed tools are lightweight to deploy and requires minimal human effort, enabling automated detection of software bugs/vulnerabilities. All tools developed in this dissertation have been open-sourced to foster further research on improving software reliability and security.

1.6 Authorship Attribution Statement

This thesis includes material from 4 papers (two published and two under review) in which I am the project leader and listed as the first author.
Chapter 3's contents are from the following *published* paper [193]:

- **Haoxin Tu**, Lingxiao Jiang, Jiaqi Hong, Xuhua Ding, and He Jiang. *Concretely Mapped Symbolic Memory Locations for Memory Error Detection*. IEEE Transactions on Software Engineering (**TSE** 2024).

The contributions of the co-authors are as follows:

- I came up with the key idea, designed all experiments, implemented all the source code and conducted all experiments. I prepared the manuscript drafts.
- Prof. Lingxiao Jiang, Prof. Xuhua Ding, Prof. He Jiang, and Dr. Jiaqi Hong improved the idea, suggested the design of the experiments.
- Prof. Lingxiao Jiang and Prof. Xuhua Ding provided suggestions to improve the presentation and revised the manuscript.

Chapter 4's contents are from the following *published* paper [192]:

- **Haoxin Tu**, Lingxiao Jiang, Xuhua Ding, and He Jiang. FASTKLEE*: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers.* The Tool Demonstrations Track of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (**ESEC/FSE** 2022)

The contributions of the co-authors are as follows:

- I came up with the key idea, designed all experiments, implemented all the source code and conducted all experiments. I prepared the manuscript drafts.
- Prof. Lingxiao Jiang, Prof. Xuhua Ding, and Prof. He Jiang improved the idea, suggested the design of the experiments.
- Prof. Lingxiao Jiang provided suggestions to improve the presentation and revised the manuscript.

Chapter 5's contents are from the following *under review* paper:

- **Haoxin Tu**, Lingxiao Jiang, and Marcel Böhme. VITAL*: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search*. (**Under Review**, 2025)

The contributions of the co-authors are as follows:

- I came up with the key idea, designed all experiments, implemented all the source code and conducted all experiments. I prepared the manuscript drafts.
- Prof. Lingxiao Jiang and Prof. Marcel Böhme improved the idea, suggested the design of the experiments, and revised the manuscript.

Chapter 6's contents are from the following *under review* paper:

- **Haoxin Tu**, Seongmin Lee, Yuxian Li, Peng Chen, Lingxiao Jiang, and Marcel Böhme. COTTONTAIL*: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation*. (**Under Review**, 2025)

  The contributions of the co-authors are as follows:

- I came up with the key idea, designed all experiments, implemented all the source code and conducted all experiments. I prepared the manuscript drafts.
- Prof. Lingxiao Jiang, Prof. Marcel Böhme, and Dr. Seongmin Lee improved the idea, suggested the design of the experiments, and revised the manuscript.
- Dr. Yuxian Li and Peng Chen provided suggestions to improve the presentation and revised the manuscript.

Other Publications during My PhD Study.

- **Haoxin Tu**, Lingxiao Jiang, Debin Gao, and He Jiang, *Beyond a Joke: Dead Code Elimination Can Delete Live Code*, in Proceedings of 46th IEEE ACM International Conference on Software Engineering (ICSE-NIER 2024).
- Pansilu Pitigalaarachchi, Xuhua Ding, Haiqing Qiu, **Haoxin Tu**, Jiaqi Hong, and Lingxiao Jiang, *KRover: A Symbolic Execution Engine for Dynamic Kernel Analysis*, in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS 2023).
- **Haoxin Tu**, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang, *RemGen: Remanufacturing A Random Program Generator for Compiler*

Testing, in the 33rd IEEE International Symposium on Software Reliability Engineering (ISSRE 2022).

- **Haoxin Tu**, Zhide Zhou, He Jiang, Imam Nur Bani Yusuf, Yuxian Li, and Lingxiao Jiang, *Isolating Compiler Bugs by Generating Effective Witness Programs with Large Language Models*, in IEEE Transactions on Software Engineering, 2024.

- **Haoxin Tu**, He Jiang, Zhide Zhou, Yixuan Tang, Zhilei Ren, Lei Qiao, and Lingxiao Jiang, *Detecting C++ Compiler Front-end Bugs via Grammar Mutation and Differential Testing*, in IEEE Transactions on Reliability, 2022.

1.7 Dissertation Organization

The dissertation is organized as follows. Chapter 1 introduces of the dissertation, including the motivations, research objects, overview of dissertation works, and technical contributions. Chapter 2 review the literature about existing strategies and highlight research gaps along with our research goals. Chapter 3 decipts SYMLOC, a system that detects memory errors via boosted memory models. Chapters 4 and 5 deliver new solutions for faster and effective path exploration. Chapter 4 covers FASTKLEE, a fast symbolic execution engine with efficient path exploration by reducing the overhead of interpreting safe pointer operations. Chapter 5 introduces a new vulnerability-oriented path exploration strategy VITAL, which adopts a smart targeted path exploration based on Monte Carlo Tree Search. Chapter 6 presents COTTONTAIL, an LLM-driven approach that is capable of generating highly structured test inputs for discovering vulnerabilities hidden in the deep code regions. Chapter 7 summarizes dissertation's contributions and suggests potential future directions.

Chapter 2

Literature Review

2.1 Memory Modeling for Symbolic Execution

As highlighted in existing studies [27, 28, 173], the choice of a memory model is a crucial aspect in the design of a symbolic execution engine, impacting both the extent of coverage during exploration and the efficiency of constraint solving. One notable challenge is the symbolic memory address problem [169], which occurs when the address used in an operation is represented by a symbolic expression. In the following sections, we will delve into several widely recognized solutions to this problem. According to different strategies for handling symbolic addresses, there are two prominent solutions for modeling memory in symbolic execution, namely full and partial symbolic memory.

2.1.1 Full Symbolic Memory

At a general level, some engines opt for handle memory addresses as entirely symbolic. This method has been adopted in various studies, such as BitBlaze [174] and BAP [23]. Two foundational approaches, including state forking and If-Then-Else (ITE) formulas, are proposed to handle the full symbolic memory processing [114].

In the realm of state forking, when an operation involves reading from or writing to a symbolic address, the state undergoes forking. This process involves creating

multiple possible states that could result from the operation. For each of these forked states, the path constraints are updated to reflect the new conditions.

The ITE involves encoding the uncertainty associated with the possible values of a symbolic pointer directly into the expressions maintained in the symbolic store and the path constraints. This is done without creating any new states. The central concept here is to utilize the ability of certain solvers to handle formulas containing if-then-else expressions. These expressions are typically in the format of `ite(c, t, f)`, which yields 't' (true) if condition 'c' is true, and 'f' (false) otherwise. This method is applied distinctly for memory read and write operations.

A significant amount of studies (e.g., EXE [29], KLEE [28], and SAGE [71]) utilize the comprehensive capabilities of certain SMT solvers for modeling fully symbolic pointers. This is achieved using a theory of arrays [81], which allows for the representation of array operations as integral elements in constraint formulas.

The broad applicability of a fully symbolic memory model enables it to offer the most precise representation of a program's memory behavior, covering every conceivable memory manipulation. In numerous practical instances, the range of potential addresses that a memory operation might target is relatively small [174]. Generally, though, a symbolic address has the potential to reference any location in memory. This can result in an overwhelming increase in the number of possible states and complex constraints, posing challenges in terms of tractability.

2.1.2 Partial Symbolic Memory

Mayhem [31] addresses the scalability issues of fully symbolic memory and the accuracy loss in memory concretization by introducing a partial memory model, which lies in the middle of the spectrum. The central concept of this model is to concretize written addresses while symbolically modeling read addresses, but only when the contiguous interval of potential values they can assume is sufficiently small. This approach achieves a good trade-off between effectiveness and efficiency: it

employs more complex formulas than those used in concretization by encoding multiple pointer values per state, but it does not attempt to represent all possible values like fully symbolic memory [8].

One basic method to limit the range of potential values an address might take involves testing various concrete values and assessing their compliance with the current path constraints. This process helps to eliminate large sections of the address space with each test until a narrow range is established. However, this algorithm has several drawbacks, such as the high cost of solver queries for each symbolic dereference, the possibility of non-continuous memory ranges, and the structured nature of values within a symbolic pointer's memory region.

To enhance efficiency, Mayhem implements several optimizations. These include value-set analysis [11] and forms of query caching. These techniques aim to refine ranges more effectively. If the range size still exceeds a predetermined threshold (like 1024) after these steps, the address is then concretized. Similarly, Angr [173] also employs the partial memory model concept and expands it by optionally allowing to write operations on symbolic pointers within small, contiguous intervals.

Note that in scenarios where the combinatorial complexity of analysis becomes unmanageable due to the inability to restrict pointer values to sufficiently small ranges, address concretization emerges as a frequently used solution. This process involves converting a symbolic pointer into a single, specific address. Such concretization can effectively reduce both the number of states and the complexity of the formulas presented to the solver, thereby enhancing runtime efficiency. However, this approach might lead the engine to overlook certain paths, particularly those that depend on specific values of some pointers.

Unlike existing approaches, our goal is to leverage a symbolic execution-based approach to facilitate the detection of memory errors. Unlike existing static/dynamic analysis-based memory detectors, our approach should not only can detect more memory errors but could also provide useful test cases for those detected errors to facilitate the debugging and fixing process during software development.

2.2 Path Exploration for Symbolic Execution

2.2.1 Efficient Path Exploration

Existing studies contributing to efficient path exploration can be broadly classified into three categories, namely improving constraint solving, intermediate representation code optimization, and compilation-based symbolic execution.

(1) Improving Constraint Solving

Constraint satisfaction problems are crucial in various areas such as analysis, testing, and verification of software programs. Constraint solvers act as decision-making tools for issues expressed in logical formulas. A classic example is the boolean satisfiability problem (SAT), which determines if there is a solution to interpret a formula's symbols that makes it true. Despite SAT's NP-complete status, recent advancements have significantly expanded its feasibility in practical scenarios [56] due to the following advantages of SMT solvers.

SMT solver algorithms are versatile, handling complex constraint combinations. They are incremental, allowing backtracking as constraints change, and can explain inconsistencies. Theories can be mixed in many ways, like reasoning about arrays of strings. Moreover, decision procedures often work in tandem to optimize efficiency, for instance, by first solving simpler parts of a complex formula. This versatility enables SMT solvers to tackle large, complex problems.

In symbolic execution, constraint solving is vital for path feasibility checks, symbolic variable assignments, and assertion verifications. Different solvers have been used over time, chosen based on their supported theories and performance. For instance, the STP [178] solver, supporting bit-vector and array theories, has been utilized in systems like EXE [29], KLEE [28], and AEG [9]. Other executors, like Java PathFinder [155], have combined SMT solving with extra decision procedures and heuristics to handle complex constraints. Z3 [214], developed at Microsoft

Research, has become a leading SMT solution. It offers top-tier performance and supports numerous theories, including bit-vectors, arrays, and various arithmetic forms. Recent symbolic executors like Mayhem [31], SAGE [88], and Angr [173] predominantly use Z3, often eliminating the need for additional decision procedures due to its extensive theory support.

Despite these advances making symbolic execution more practical [27], constraint solving still poses scalability challenges for symbolic execution engines. It struggles with costly theories or complex library calls. Solutions include simplifying constraints and easing solver load through techniques like caching or deferring queries to handle problematic constraints.

(2) Intermediate Representation Code Optimization

Dong *et al.* [65] study the influence of standard compiler optimizations on symbolic execution and examine how compiler optimizations affect the performance of symbolic execution. Focusing on LLVM optimizations, the study uses KLEE, a symbolic execution engine, to analyze their impact on Unix Coreutils and NECLA benchmarks. The research reveals that some optimizations can negatively affect symbolic execution, contradicting their benefits in standard execution. This leads to the conclusion that standard compiler optimizations need careful consideration in symbolic execution to achieve high code coverage and efficiency.

Chen *et al.* [35] propose LEO and further leverage machine-learning-based compiler optimization tuning to select a set of optimizations to accelerate SE. LEO operates by separately adjusting compiler optimizations for individual code segments and libraries within a program. This tailored approach allows for more effective symbolic execution, demonstrated by empirical studies on GNU Coreutils programs using the KLEE engine. The results show significant improvements in line coverage and execution efficiency compared to default configurations, highlighting the potential of machine learning in optimizing symbolic execution.

Jonas *et al.* [196] design a stand-alone optimization (-OVERIFY) for optimizing

programs for fast verification, which includes accelerating symbolic execution, such an optimization switches developed to optimize programs specifically for automated testing and verification tools like symbolic execution engines. By modifying traditional compiler transformations, it significantly streamlines the verification process, making it more efficient. Their experiments demonstrate how this approach can lead to substantial reductions in verification time, with tests on the Coreutils suite showing time reductions up to 95 times.

(3) Compilation-based Symbolic Execution

In recent years, compilation-based symbolic execution solutions, e.g., SYMCC [160] and SYMQEMU [161], significantly enhance performance compared to traditional methods. SYMCC operates by embedding symbolic processing into the target program, creating binaries that perform concolic execution and are capable of generating test cases without multiple path executions. SYMCC operates as a drop-in replacement for clang/clang++, offering higher efficiency and coverage, as demonstrated by its successful application to the OpenJPEG project, where it identified vulnerabilities. This novel approach combines the ease of implementation with the speed of execution, showcasing significant performance improvements in both benchmarks and real-world software analysis.

SymQEMU shares the same philosophy as SYMCC, which is a novel approach to symbolic execution for binary-only software analysis. SymQEMU enhances performance by integrating symbolic execution capabilities directly into binary code through a compilation-based method, leveraging QEMU's CPU emulation. This approach achieves high performance while maintaining platform independence. SymQEMU is demonstrated to outperform state-of-the-art binary symbolic executors like S2E and QSYM, and it successfully identified a previously unknown vulnerability in `libarchive` test program, showcasing its effectiveness in real-world software testing and software quality assurance.

Unlike the existing approaches, our goal is to make IR-based SE more efficient

by reducing the internal interpretation overheads, i.e., we aim to reduce redundant bound checking of type-safe pointers during IR code interpretation. It is worth noting that our approach can be complementary to existing approaches and further boost faster SE by combing them with our proposed approach.

2.2.2 Effective Path Exploration

Enumerating every path in a program is often excessively costly. When it comes to software engineering tasks like testing and debugging, path exploration priority is given to examining the most likely paths first. It is important to note that path exploration methods are frequently customized to assist the symbolic execution engine in meeting particular objectives, such as detecting overflows. However, the quest for a universally ideal strategy is still an unresolved challenge.

Depth First Search (DFS), which thoroughly explores a path before moving to the deepest unexplored branch, and breadth First Search (BFS), expanding all paths concurrently, are widely used strategies. DFS is preferred for its lower memory consumption but struggles with paths involving loops and recursive calls. Consequently, despite its higher memory demand and longer duration for specific path explorations, some tools opt for BFS. DFS and BFS enable the engine to rapidly traverse various paths, identifying intriguing behaviors early on. Another common technique is random path selection, which has evolved into multiple variations. For example, KLEE [28] calculates path probabilities based on their length and branch visiting frequency, prioritizing less frequently explored instructions or paths. This strategy helps avoid issues like loop-induced starvation that causes path explosion.

Several studies, including EXE [29], KLEE[28], Mayhem [31], and S2E [42], have focused on developing certain path searching heuristics to enhance code coverage. For example, KLEE [20] introduces a coverage optimize search that assigns a weight to each state for random selection. This weight considers factors like the proximity to the nearest uncovered instruction, recent code coverage by the state,

and the state's call stack depth. Similarly, Li *et al.* [120] present a subpath-guided search heuristic. It aims to explore less frequented program areas by choosing the least explored subpath in the control flow graph. This method tracks the frequency of explored subpaths, where a subpath is a consecutive sequence of a specified length n from a complete path. Notably, the choice of n significantly influences the code coverage attained by a symbolic engine using this method, yet no universally optimal value for n has been identified. The shortest-distance symbolic execution [129] technique, while not primarily focused on coverage, seeks to find program inputs that activate a specific program point. Similar to coverage-based strategies, this approach employs a metric to determine the shortest distance to the target point, calculated as the length of the shortest path in the inter-procedural control flow graph. Here, paths closer to the target are given priority by the engine.

Several search heuristics are designed to prioritize paths that may lead to states relevant to specific goals. For example, AEG [9] introduces two strategies with this focus. The "buggy-path first" strategy selects paths that have previously exhibited minor and non-exploitable bugs, based on the reasoning that such paths (e.g., having contained small errors) might not have been thoroughly tested and could potentially lead to more serious and exploitable bugs in future program states. In a similar vein, the "loop exhaustion" strategy targets paths involving loop iterations, inspired by the common observation that programming errors in loops frequently result in issues like buffer overflows or other memory-related errors. On the other hand, Mayhem [31] prioritizes paths where either memory accesses to symbolic addresses are noted or symbolic instruction pointers are observed, aiming to discover exploitable bugs.

Zhang *et al.* [219] introduce an innovative approach using dynamic symbolic execution to automatically identify a program path that satisfies a regular property, such as file usage or memory safety, which can be represented by a Finite State Machine (FSM). This method guides the dynamic symbolic execution using the FSM, focusing first on branches of an execution path most likely to fulfill the desired property. It combines static and dynamic analysis to determine the priority of a path

for exploration: the states of the FSM already reached by the current execution path are dynamically computed during symbolic execution, while backward data flow analysis is employed to statically predict future states. A path is deemed to satisfy the property if there is a non-empty intersection between these two sets of states.

Fitness functions are extensively utilized in search-based test generation [135]. These functions gauge how close a given explored path is to achieving the desired test coverage. This concept has been adapted to symbolic execution in several studies [27, 204]. For instance, Xie *et al.* [204] present the "fitnex" strategy in the realm of concolic execution. This approach gives priority to paths more likely to execute a specific branch, and allows for the computation of similar fitness values for different types of branch conditions. The symbolic engine then prioritizes the path with the lowest fitness value for that branch. Paths that have not reached the branch are assigned a worst-case fitness value.

However, guided by code coverage, these techniques give the same priority to code that is unlikely to contain vulnerabilities. There are only a few vulnerability-oriented search strategies. `StatSym` [207] first instruments test programs to construct predicates that indicate vulnerable features and then employ a path construction algorithm to select the vulnerable paths. `SyML` [164] guides path exploration toward vulnerable states through pattern learning: it first trains models to learn the patterns of vulnerable paths from certain features (e.g., register/memory accesses), and then leverages the trained model to make predictions to discover interesting execution paths in new programs. However, these approaches require a training set of vulnerabilities previously discovered in the program and try to link patterns in the runtime information to vulnerability. In contrast, VITAL requires neither training nor unspecific runtime information to quantify the vulnerability-proneness of a path.

Csallner *et al.* [52] and Engler *et al.* [73] propose and extend the idea of under-constrained symbolic execution, where the symbolic executor cuts the code (e.g., an interesting function) to be analyzed out of its enclosing system and checks it in an isolation manner. Recent work `Chopper` [189] can be treated as a variant of

under-constrained symbolic execution, where it cuts out uninteresting functions that are vulnerability irrelevant.

Unlike existing solutions, our goal is to maximize the number of unsafe pointers to increase the likelihood of exposing memory unsafe vulnerabilities, without involving prior expert knowledge to decide which functions to skip, but it can still explore vulnerable program paths effectively. Then, we aim to leverage MCTS for vulnerability-oriented path exploration for symbolic execution. Compared with the most related works, the differences can be summarized as follows. (1) The purpose and the target are different. Our goal is to detect vulnerable paths in C/C++ programs, while `canopy` aims to find the costly paths in Java programs and Yeh *et al.*' approach focuses on the path with the largest executed number of basic blocks in binary programs. (2) The node expansion to be designed is guided by the results of static program analysis, i.e., type inference, while both the two compared approaches apply a random expansion strategy. (3) The simulation policy to be designed is optimized by previous simulation outcomes, while `canopy` adopts random simulation with limited optimizations and Yeh *et al.*' approach uses CFG of the binary program to perform simulation (which may yield imprecise reward, as recovering CFG from binary programs is an undecidable problem [173]).

2.3 Highly Structured Test Input Generation for Symbolic Execution

Several studies have explored the use of input grammar to enhance the efficiency of symbolic execution in parsing programs, as seen in works like [85, 131]. Godefroid *et al.* [85] introduced a grammar-based white-box fuzzing approach, which advocates for the use of token symbolization during symbolic execution. The resulting token constraints are then solved using the input grammar. Similarly, CESE [131] utilizes an input grammar to improve Dynamic Symbolic Execution (DSE) for programs

that parse this grammar. CESE starts by generating initial inputs from a symbolic grammar derived from the input grammar. These inputs are subsequently employed in the DSE to facilitate the exploration of deeper execution paths.

To address scalability challenges in Dynamic Symbolic Execution (DSE), Godefroid [84] introduced the SMART approach. This method utilizes DSE to initially create input-output relation summaries for lower-level functions. These summaries are then directly applied when these functions are called during the DSE of higher-level (caller) functions. Anand *et al.* [5] enhance SMART by introducing a demand-driven compositional symbolic execution method. This strategy aims to curtail the number of explored paths through a lazy summary technique, which is based on encoding using uninterpreted functions [20]. FOCAL [113] further refines demand-driven compositional symbolic execution by using Craig interpolants [51] for function summary refinement. FOCAL employs backward analysis to generate system-level inputs for specific failure targets and composes constraints from the context of the target's invoking chain, starting from the entry function. Gillian [167] offers a language-independent framework for compositional symbolic execution, utilizing bi-abductive symbolic analysis to facilitate compositional testing. Pan *et al.* [152] specifically target parsing programs within the scope of compositional symbolic execution. We focus on summarizing only the tokenization code, striking a balance between generalization and efficiency for the analysis of parsing programs. It would be intriguing to incorporate insights from these studies to further enhance the efficiency of our approach, especially in the analysis of application logic code.

Some related work intersects with the field of input grammar inference. Glade [13] introduces an algorithm that constructs a context-free input grammar from a program's input-output examples, which can then be utilized to enhance fuzzing techniques. REINAM [202] builds upon Glade's foundation, addressing the issue of over-generalization. It creates a probabilistic context-free input grammar, refining the process of grammar synthesis. Skyfire [197] takes a different approach by learning a probabilistic context-sensitive grammar (PCSG), which is designed to accurately

represent the distribution of valid inputs. Such PCSG is then employed to generate seeds for more effective fuzzing. In a unique direction, Mimid [91] focuses on learning a readable context-free input grammar through a white-box approach. It tracks the access of input characters to facilitate the inference of grammar.

In concolic execution, SYMFUZZ [32] identifies dependencies among bit positions in input through symbolic analysis. Driller [177] employs selective concolic execution to navigate those uncovered paths when fuzzing gets stuck. T-Fuzz [157] simplifies the original program by removing complex input checks and then uses symbolic execution to filter out false positives and confirm actual bugs. QYSM [211] alleviates the strict soundness requirements of conventional concolic executors. Intriguer [43] further optimizes QSYM with field-level knowledge. In contrast, Angora [37] and Matryoshka [38] opt for taint analysis. SYMCC [160] first proposes compliation-based concolic execution to further gain performance enhancement. A recent work Marco [102] improves the efficiency and effectiveness of exploring code paths by integrating a directed branch scheduling algorithm.

Unlike existing LLM-based solutions or traditional concolic execution engines, our goal is to combine more precise semantic information (i.e., path constraints) with LLM to improve its test case generation capabilities. Besides, we also aim to utilize a customized CoT (Chain of Thought) prompt which separates the logic for constraint solving, yielding better results.

2.4 Other Related Work

Static Analysis for Memory Error Detection. Static tools typically leverage pattern matching and abstract interpretation techniques to facilitate the detection of memory errors. Coccinelle [151] employs a given pattern to analyze and certify memory errors, mainly in C programs. Cppcheck [134] checks non-standard code to detect potential memory errors. TscanCode [185] extends CppCheck and detects temporal memory errors using specific syntactic patterns. Regarding abstract

interpretation-based approaches, Frama-C [53] implements program safety verification and uses value flow analysis to detect memory errors. Clang static analyzer [4] performs path-sensitive analysis to detect general memory errors.

Dynamic Analysis for Memory Error Detection. Typical dynamic memory detectors can be classified into two types based on their strategy of instrumentation. On run-time-instrumentation, Valgrind [146] uses disassembly and resynthesizing technology to detect memory errors. Purify [96] utilizes object code insertion technology to instrument object files with additional instructions. Dr. Memory [22] applies a copy-and-annotate technique to copy the incoming instructions verbatim. On compile-time instrumentation, Mudflap [70] statically inserts the predicate validity assertion at the pointer deference site (i.e., the pointer's use site) to decide whether the pointer accesses valid memory. AddrSan [170] leverages a direct shadow mapping scheme to detect memory errors.

Symbolic Execution for Memory Error Detection. KLEE [28] is the leading symbolic execution that can detect many memory errors by using a concrete memory model. Recently, David *et al.* propose *symsize* [190] which adopts a bound memory model to model the size of memory allocation. David *et al.* [186] propose a novel symbolic memory model RAM that accesses objects with symbolic offsets which are handled using array theory to facilitate constraint solving in symbolic execution. Matin [148] introduces an enhanced, fine-grain, and efficient representation of memory that mimics the allocations of tested applications. Coppa *et al.* [49] model symbolic memory as a set of tuples, where each tuple associates an address expression to a timestamp-based and a condition-based value expression. Angr [173] and Mayhem [31] share the same index-based memory model that allows a symbolic read under certain conditions.

Applications of Monte Carlo Tree Search. MCTS has been used to solve many problems in the software engineering domain. MCTS was initially applied to enhance heuristics for a theorem prover [75]. Then, it was used to optimize program synthesis [121] and symbolic regression [200]. Furthermore, MCTS has

been previously employed in Java PathFinder [162], where it was used for explicit state model checking, and a heuristic for deadlock detection was implemented. Liu *et al.* [122] adopt MCTS to achieve the best trade-off between concolic execution and fuzzing for coverage-based testing. Zhao *et al.* [220] model the seed scheduling problem in fuzzing as a decision-making problem and use MCTS to select the next seed to test through an optimal path. The works most related to ours are the `canopy` [126] and the approach proposed by Yeh *et al.* [208]. `canopy` uses MCTS to guide the search for costly paths in programs, where the cost is defined as memory consumed and execution time along a path. Yeh *et al.*' approach utilizes MCTS to select valuable paths to explore based on the number of visited basic blocks.

LLMs for Software Security. Recent research has demonstrated their potential in software security tasks such as fuzz testing. ChatFuzz [101] employs an LLM as an input mutator, picking a seed input from the fuzzer's seed pool and mutating it to generate format-conforming input variations. Codamosa [118] uses LLMs to generate example test cases for under-covered functions, addressing coverage plateaus in search-based software testing. LLAMAFUZZ [217] extends LLM mutation capabilities to generate valuable structured binary inputs. ChatAFL [101], CovRL-Fuzz [74], and InputBlaster [123] integrate LLMs to enhance input generation for fuzzing in applications like network protocols, JavaScript engines, and mobile apps, respectively. Beyond input generation, LLMs can also generate fuzz drivers for APIs. oss-fuzz-gen [66, 89] employs zero-shot or few-shot learning techniques to create fuzz drivers based on a given set of APIs. Promptfuzz [127] generates fuzz drivers through various API mutations, and Zhang *et al.* [216] evaluate different generation strategies for LLM-based fuzz driver generation. *TitanFuzz* [57] leverages LLMs to generate both harness programs and arguments for fuzzing deep learning libraries.

Chapter 3

Concretely Mapped Symbolic Memory Locations for Memory Error Detection

3.1 Objective

Memory allocation functions (e.g., `malloc` and `free`) are fundamental operations for managing memory objects in programs written in C/C++ programming languages [7]. However, previous studies [2, 22, 78, 93, 183] show that memory errors such as *buffer overflow* and *use-after-free* caused by misuse of such allocation functions are common and can have catastrophic consequences [54]. To facilitate the detection of memory errors, many techniques have been proposed, including static analysis-based approaches such as Frama-C [53] and Coccinelle [151], dynamic analysis-based approaches such as Valgrind [146] and AddressSanitizer [170], and symbolic execution-based approaches such as KLEE [28] and *symsize* [190].

Symbolic execution could be one of the most automated approaches to detect memory errors since it can automatically explore different paths in programs and generate actionable test cases to assist in validating a reported error [47]. In contrast, static analyses usually do not excel at generating test cases to validate an error

reported for a program and dynamic analyses have trouble covering different paths in the program. Benefiting from its capabilities in automatic path exploration and test case generation, symbolic execution has become a popular technique broadly adopted in many domains, e.g., test case generation [28, 29, 155], bug-detection [28, 86, 165], debugging and repairing [109, 137, 138, 147, 215], cross-checking [48, 112], side-channel analysis [19, 20, 94], and exploit generation [9, 31, 99].

Despite the success of symbolic execution, existing symbolic execution engines still have fundamental limitations in modeling memory addresses related to memory allocation functions. Specifically, most existing symbolic execution engines, such as KLEE [28] and *symsize* [190], usually model the locations of dynamically allocated memory objects (e.g., `malloc` in C/C++) with concrete values, where each object is located at a fixed address. However, those address values should be *non-deterministic* during actual executions as they are dynamically allocated based on different run-time environments. The limitation may cause existing symbolic execution engines to miss detecting certain kinds of memory errors. First, since the engines do not encode memory addresses into path constraints, they may not be able to cover certain lines of code whose executions are dependent on the memory addresses and miss certain address-specific spatial memory errors such as *buffer overflow*. Second, without maintaining and tracking the constraints among the addresses of different memory objects, they may fail to reliably check for unsafe uses of memory objects and miss certain temporal memory errors such as *use-after-free* or *double-free*. Even though the engines allow symbolization of the memory addresses, without proper handling of read/write operations involving symbolic addresses, the engines can face memory state explosion and cannot explore the programs effectively. Therefore, existing symbolic execution engines need better ways to model the addresses of dynamically allocated memory objects for more effective detection of memory errors in programs.

Since the locations of dynamically allocated memory objects can be nearly arbitrary, a more complete (i.e., treat those locations as symbolic and make greater use of those symbols) modeling of those locations can help explore more execution

paths and detect more memory errors in the test program. Hence, we believe that a symbolic execution engine should satisfy at least the following three fundamental capabilities to comply with the more complete modeling requirements: 1) symbolization of addresses and modeling them into path constraints, 2) practical read/write operation from/to symbolic addresses, and 3) effectively tracking the uses of symbolic addresses. Unfortunately, no existing symbolic execution engine fulfills all the above requirements. For example, KLEE [28] or *symsize* [190] satisfies none of those requirements. They treat memory locations as concrete; even though one can force an address to be symbolic, the subsequent symbolic memory operations would fail as the engines usually concretize the symbolic addresses to invalid ones, thus producing false alarms of errors. A recent work, RAM [186], satisfies requirements #2 and partially #1 but not #3. It simply assumes that the memory locations do not affect the behavior or execution paths, so it does not encode them into path constraints. Unfortunately, such an assumption is not always valid in the real world (see more details in the example in Figure 3.1).

This chapter proposes SYMLOC to integrate three techniques, i.e., address symbolization, a symbolic-concrete memory map, and symbolic memory tracking, to meet all the above three fundamental requirements. With the new integration of the techniques, SYMLOC can more effectively detect memory errors: (1) by encoding symbolic addresses into path constraints and equipped with the symbolic-concrete memory map, SYMLOC is able to cover more code and detect more address-specific spatial memory errors; (2) by enabling automatic propagation of symbolic addresses in symbolic execution, SYMLOC is able to detect temporal memory errors reliably.

We have built SYMLOC on top of KLEE and evaluated its effectiveness. Our empirical evaluation results show that: (1) SYMLOC is able to detect 23 more unique address-specific spatial memory errors and cover 15% and 48% more unique lines of code than two symbolic execution engines (i.e., KLEE [28] and *symsize* [190]) over `GNU Coreutils`, `make`, and `m4` programs; (2) when compared against various state-of-the-art memory error detectors [93], including symbolic executors (KLEE

and *symsize*), static detectors (Frama-C [53] and Coccinelle [151]), and dynamic detectors (Vargrind [146] and AddressSanitizer [170]), over the C/C++ programs in Juliet Test Suite (JTS) datasets [79], SYMLOC can reliably detect 8%-64% more errors than comparative memory error detectors.

In summary, this chapter makes the following contributions.

- We propose a novel approach named SYMLOC that integrates three techniques to satisfy the three fundamental capability requirements for symbolic execution involving memory allocation functions, which facilitates more effective detection of memory errors in C/C++ programs.
- We empirically demonstrate the effectiveness of SYMLOC against many state-of-the-art memory error detectors over various real-world benchmark programs, and showcase two sample memory errors undetected by existing approaches and discuss their implications.
- We release a replication package for SYMLOC[1] to facilitate future work on more optimized symbolic execution engines and effective memory error detection.

3.2 Preliminaries and Motivating Examples

This section describes common memory errors briefly, and shows two examples to illustrate the limitations of existing approaches and highlight the advantages of our proposed approach.

3.2.1 Common Memory Errors

Memory errors can broadly be divided into two categories: *spatial* and *temporal* errors [41, 78, 141, 143].

Spatial Memory Errors. Such errors are usually in two forms: (1) *Out-Of-Bound (OOB)* or *buffer overflow* – assessing (i.e., read or write) to an address that is out of bound of a valid memory area (a.k.a. a *buffer*); (2) *NULL Pointer Error*

[1] https://github.com/haoxintu/SymLoc

— dereferencing *NULL* pointers or uninitialized wild pointers. Here, we refer to them as *address-specific* errors as they violate proper usage of the specific allocated address of a memory object.

Temporal Memory Errors. Such errors broadly fall into the following three types [78] involving a sequence of more than one memory operation: (1) *Use-After-Free* (UAF) – accessing a memory area via a pointer after the memory area pointed to by the pointer has been deallocated; (2) *Double-Free* (DoF) – deallocating a memory area again via a pointer after the memory area pointed to by the pointer has been deallocated, which can be considered a special case of UAF; (3) *Invalid-Free* (InF) – deallocating a memory area via a pointer while the pointer does not point to the beginning of a valid allocated memory area (i.e., freeing a pointer whose value was not returned by a heap allocator).

Keeping memory *spatial* and *temporal* safety is critical for assuring the quality of software systems [78, 93] because memory errors can have catastrophic security risks, e.g., being exploited by attackers [78, 93, 201]. Memory errors still rank among the most dangerous software errors in recent CVE announcements [183] and a recent analysis of the Chromium project shows that more than 50% of serious memory safety errors are temporal memory errors [166].

Approaches for Detecting Memory Errors. According to the way to analyze a program to detect certain memory errors, existing detection approaches can be broadly classified into three categories: static analysis-based, dynamic analysis-based, and symbolic execution-based approaches [93]. Static analysis-based approaches detect errors by analyzing the source or machine code of the program without executing it, while dynamic analysis-based approaches report errors by executing test programs. In contrast, symbolic execution-based approaches are in-between, and they search for memory errors by simulating the executions. In this study, we focus on improving the memory error detection capabilities of symbolic execution.

```
1  void *memmove((void *dest, const void *src, size_t n){
2      unsigned long int to = (long int) dest;
3      unsigned long int from = (long int) src;
4      if (from == to || n == 0) {
5              .. // Path-A
6      }
7      if (to > from) { /* copy in reverse */
8          if (to - from >= (int) n)     {
9              int i; // Path-B
10             for(i=n-1; i>=0; i--)
11                 to[i] = from[i]; // symbolic memory read/write
12             return dest;
13         } else {
14             ... // Path-C
15         }
16     }
17     if (from > to) { /* copy forwards */
18         if (from - to >= (int) n)) {
19             ... // Path-D
20         } else {
21             dest[n + 100] = 0; // address-specific memory error
22             ... // Path-E
23         }
24     }
25     return dest;
26 }
27 int main(){
28     char *buf1 = malloc (100 * sizeof (char));
29     char *buf2 = malloc (50 * sizeof (char));
30     if (buf1 == NULL) { abort(); }
31     if (buf2 == NULL) { abort(); }
32     memmove(buf1, buf2, 10);
33     free(buf1);free(buf2); return 0;
34 } /* Example 1 (E1) */
```

Figure 3.1: Existing symbolic executors (e.g., KLEE) are only able to cover `Path-B` while missing `Path-A,C,D`, and the memory error in `Path-E`

3.2.2 Illustrative Examples

The first example shown in Figure 3.1 originated from the widely used implementation of the function `memmove` [1, 59], and the second example shown in Figure 3.2 is adapted from the JTS benchmarks [79] used in our experimental evaluation. In the following, we first explain the execution flows of each program and then point out the limitations of existing approaches in detecting certain memory errors. Finally, we present the main advantages of our approach.

(1) E1: Missing Spatial Memory Errors

Execution Flow. The functionality of the `memmove` function shown in Figure 3.1 is to copy *n* bytes from memory area *src* to memory area *dest* and finally return the pointer *dest* to users. Note that the copying memory areas are allowed to be

overlapped. The typical implementation logic of this function is (1) casting two pointer values *dest* and *src* to integer values *to* and *from* to avoid undefined behavior[2] and (2) handling different scenarios according to the different locations of two input memory objects by comparing the cast values of pointers [1, 59]. There will be five scenarios in total: the values of the pointers *from* and *to* are the same or the copying size is zero, leading to `Path-A` in Line 5; the value of the pointer *to* is larger than *from* w/o overlap copying, leading to `Path-B` (Line 9)/`Path-C` (Line 11); the value of the pointer *to* is smaller than *from* w/o overlap copying, leading to `Path-D` (Line 16)/`Path-E` (Line 18), where an address-specific spatial memory error, i.e., *buffer overflow*, is hidden in `Path-E` in Line 21.

Limitations of Existing Approaches. As aforementioned in Section 3.1, static analysis-based approaches can detect the error but they are hard to produce a useful test case to reproduce the error, while dynamic analysis-based approaches also encounter difficulties as they usually produce redundant test cases for shadow area of the test program or suffer from limited analysis algorithm to catch certain errors. For symbolic execution-based approaches, although simple, existing symbolic execution engines miss the majority of (80%, 4 out of 5) paths due to the fundamental design flaw in those engines. Specifically, since existing engines linearly manage memory objects that are consecutively allocated, i.e., the value returned from the second `malloc` function (Line 29) is always larger than the value returned from the first one (Line 28), meaning the value of *to* is always larger than the value of *from*. Therefore, only `Path-B` (Line 9) will be covered, while missing the covering of the rest of the four paths, i.e., `Path-A, C, D`, and `E`, thus missing detecting the memory error in Line 21. Worse still, existing engines (e.g., KLEE) struggle to handle symbolic memory read/write operations as shown in Line 11.

It might be true for some variants of KLEE that do not always concretize the symbolic address, but for the main base version of KLEE, as far as we know, when

[2]Direct comparison of two pointers from different memory objects is undefined behavior according to C standard (ISO/IEC 9899:201x)

KLEE encounters a symbolic address, no matter whether the base address or the offset is symbolic value, KLEE concretizes it (with possibly many concrete values) before performing read/write to a symbolic address. KLEE sacrifices the accuracy of the memory modeling to continue the execution. KLEE generates false alarms of symbolic addresses mainly because KLEE concretizes a symbolic address with values, such as 0, that may not satisfy the program constraints. Say a symbolic address "p", when it was concretized to "0", since "0" is less likely to be a valid memory address, any read/write upon this address will cause a memory error. This would lead to false alarms due to invalid address concretization unless KLEE adds the same implementations as SYMLOC (i.e., to maintain a memory map to store the valid base address of each symbolic address).

Note that to assure the program quality, the problem of finding all faults in a program for any meaningful program is essentially unsolvable [90]. To improve the quality of the program, a common criterion is to use code path coverage to comprehensively test the program when we don't know beforehand where the faults are [3]. The idea is that if a large portion of code is covered with no faults, the program can be more reliable and contain fewer faults. Therefore, it is necessary to cover all the branches in the code example.

Advantages of SYMLOC. Instead of modeling memory addresses as concrete ones, SYMLOC first symbolizes those address values as symbolic ones and then encodes symbolic addresses into path constraints. Furthermore, a symbolic-concrete memory map is designed in SYMLOC for efficient symbolic memory operations, and a relaxed overlapping property is supported to have better modeling of dynamic memory allocation behavior. Thus, SYMLOC can smoothly cover all the branches in Figure 3.1 and detect the memory error in Line 21. Importantly, SYMLOC could provide a useful test case (i.e., in what conditions the error will be triggered) that helps developers quickly locate and further fix the error.

```
1  static  char * dothing (int  magic) {
2      if  (magic == 0x1234ABCD) {
3          wchar_t * data1 = NULL;
4          data1 = (wchar_t *) malloc (100 * sizeof (wchar_t));
5          if  (data1 == NULL) { abort(); }
6          wmemset(data1, L'A',100-1);
7          data1[100-1] = L'\0';
8          free(data1);
9          wprintf(L"%s\n", data1); // use-after-free error 1
10         return NULL;
11     } else  {
12         char * data2 = NULL;
13         data2 = (char *) malloc (100 * sizeof (char));
14         if  (data2 == NULL) { abort(); }
15         memset(data2, 'A', 100-1);
16         data2[100-1] = ''\0';
17         free(data2);
18         return data2; //  use-after-free error 2
19     }
20 }
21 int  main(int argc,  char** argv){
22     int  a;
23     read(0,  &a, sizeof (int));
24     char * ret  = dothing(a);
25     if  (ret  != NULL)
26         printf("%s\n", ret);
27     return 0;
28 }   /* Example 2 (E2) */
```

Figure 3.2: Missing the detection of UAF errors

(2) E2: Missing Temporal Memory Errors

Execution Flow. The program first reads a value from the user, passes it to the function `dothing`, and finally, prints out the content of the string pointed to by the return value `ret` if `ret` is not *null* (Lines 23-26). Inside the function `dothing`, it compares the argument `magic` number against a specific value (e.g., 0x1234ABCD). If the corresponding value is provided, the program will go through the subsequent branches and exercise potential *abort* failures or UAF errors. In the *if-branch* starting from Line 2, a 100-size buffer is allocated by invoking the `malloc` function and two *if-branch*es (Line 5 and 14) are used to check whether the allocated address (the value of `data1` or `data2`) equals *null*. Then, the whole allocated buffer is initialized by calling the `wmemset` function in Line 6 or the `memset` function in Line 15. Note that there are two UAF errors inside this function. The first UAF error lies in Line 9, where the object `data1` is freed (Line 8) but it is used as an argument in the `wprintf` function later in Line 9. The function raises the second UAF error since a value `data2` is freed in 17 but later returned to Line 18, where the object

`data2` is further used as an argument in the `printf` function in Line 26.

Limitations of Existing Approaches. Existing state-of-the-art memory error detectors are struggling to detect both two UAF errors. Static/dynamic memory error detectors have at least suffered from two major issues. Apart from the limitations of the test case (i.e., required values to satisfy the *if* conditions) generation as we mentioned, they are also difficult to detect the UAF errors due to their limited capability in inter-procedure analysis or handling certain C library functions. Symbolic executors are easy to get the concrete input to go through the two branches but still fail to catch both two UAF errors. For example, the well-known executor KLEE can not detect UAF error 2 due to its fundamental design issue in their memory management [63]. Specifically, there are some memory relocation operations inside of the `printf` function (code is not shown); during KLEE's execution, one of the addresses relocated inside of the `printf` function in Line 26 is the same as the address of the freed object in Line 17. Thus, KLEE will treat the freed address as "valid" which leads to the missing UAF error.

More importantly, both static/dynamic or symbolic execution-based approaches need to take extra effort to track, relate, and check the uses of the potentially enormous number of memory objects with concrete addresses.

Advantages of SYMLOC. Instead of extra analysis to track, relate, and check the uses of memory objects with concrete addresses, SYMLOC effectively validates the safety of using symbolic memories by checking the propagated symbolic expressions during symbolic execution. Therefore, SYMLOC is able to reliably detect the two UAF errors in Figure 3.2. Note that reliable detection in this study has two meanings: (1) effectively catches temporal memory errors and (2) precisely reports the root cause information of those errors such as "*memory error: a use after free is detected*" when a freed memory object is used rather than imprecise information (e.g., "*memory error: out of bound error*" reported by KLEE [28]).

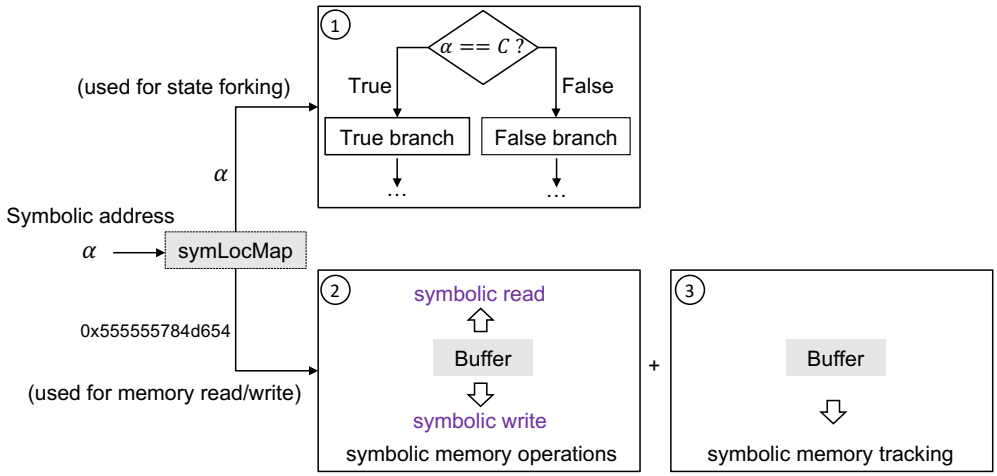


Figure 3.3: High-level design of SYMLOC

3.3 Design of SYMLOC

In this section, we describe the overview and detailed design of SYMLOC.

Overview. Figure 3.3 presents the high-level design of SYMLOC. The main insight is that, to satisfy the three fundamental requirements as aforementioned in Section 3.1, we symbolize the return address from dynamically allocated memory and maintain a concretely mapped map (i.e., `SymLocMap`) to enable the symbolic memory operation as well as symbolic memory tracking. To this end, as shown on the right side of Figure 3.3, we integrate three techniques in SYMLOC. In phase ①, the path constraint encoded address symbolization technique first treats dynamically allocated addresses as symbolic values and propagates them into path constraints to enable the comprehensive exploration of execution paths that depend on memory locations, thus enabling the detection of more spatial memory errors. In phase ②, the symbolic-concrete memory mapped memory operations technique utilizes the map, where each symbolic address holds its underlying concrete buffer, to support performing symbolic memory operations. In phase ③, to make greater use of newly defined symbols, a new error detection technique tracks the use of symbolic addresses to assist SYMLOC in performing reliable temporal error detection. Next, we detail these techniques separately.

3.3.1 Address Symbolization

(1) Definition of Symbolic Addressing Model

Existing Memory Addressing Models. Memory addressing models in most symbolic execution engines are designed to represent the allocated memory objects with as much concrete information as possible during executions [28]. For example, in KLEE, a memory object `mo` is presented as a tuple: $(addr, size, arry) \in N^+ \times N^+ \times A$, where *addr* is a *concrete* base address of the `mo`, *size* is a *concrete* size of the `mo`, and *arry* is a solver array that tracks the concrete or symbolic values written to the `mo`. N^+ means all the natural numbers corresponding to the indices for all memory objects. A is the set of all possible solver arrays.

Such memory addressing models often hold the *non-overlapping* property, i.e., every memory object in the address space is within its unique address range which does not intersect with other memory objects' address ranges. This *non-overlapping* property is useful for identifying a memory object via an address, i.e., a concrete address `a` is associated with at most one memory object that can be determined by checking if the following condition is true for a `mo`: $mo.addr \leq a \leq mo.addr + mo.size$.

Our Memory Addressing Model. A memory object in our model is presented as follows:

$$(symAddr, size, arry) \in S_{N^+} \times N^+ \times A$$

where `symAddr` is a symbolic rather than a concrete value and S_{N^+} represents the set of symbolic variables that maintain a concrete address in the range of N^+. Note that we focus on making those addresses that are dynamically allocated symbolic, i.e., the addresses determined at runtime by invoking the dynamic memory allocation function (e.g., `malloc` in C/C++). Such allocations can return different addresses based on the runtime environment or even some specific addresses desired by hackers when exploiting the errors [39, 98, 175, 198].

The major benefit of our memory model is that we enable path exploration in the paths that require a condition involving address arithmetic and comparison, which is different from existing symbolic execution engines such as KLEE [28] and *symsize* [190]. To exemplify how SYMLOC is capable of covering more lines of code, take again the code snippets shown in Figure 3.1 as an example. In SYMLOC, the returned address of the `malloc` function is represented as a symbolic variable, say α for the pointer `buf1` in Line 28 and β to the pointer `buf2` in Line 29. Then, during path exploration, every path where the condition relies on the comparison of pointers will be forked if the current path constraint is solvable. Later, a constraint solver (e.g., STP [178] or Z3 [214]) will be used to determine the feasibility of forked branches, and the satisfied path will be further explored as the normal symbolic execution does. For example, the `Path-C` in Line 11 in Figure 3.1 will be encoded as:

$$(\alpha > \beta) \ \ \& \ \ (!(\alpha - \beta >= n))$$

A capable solver can check the satisfiability of the constraint, and this constraint can be true and solved to a value that could help explore the corresponding path `Path-C`. Empowered by such a capability, SYMLOC is able to explore all five scenarios in the `memmove` function, enabling the covering of all five paths from `Path-A` to `Path-E` as well as the error handling code in Lines 30-31, importantly, the address-specific spatial memory error in Line 21.

It is worth noting that existing symbolic execution engines can easily make the addresses symbolic by invoking the supported API (e.g., `klee_make_symbolic` in KLEE). However, such a default solution brings an intractable problem since they do not handle symbolic memory operations (see more details in Section 3.3.2).

(2) Relaxed Non-overlapping Property

In SYMLOC, we no longer assume the *non-overlapping* property that is held in many other symbolic execution engines. We opt for such a design mainly because the

address can be dynamically allocated nearly anywhere for each memory object, and we aim to *over-approximate* those addresses to investigate how different memory locations could reflect program behaviors under test. Many system memory managers are capable of allocating the same memory for different objects for efficiency purposes, which is much different from the memory allocation supported in existing symbolic execution engines. For instance, considering the following code snippet, the addresses of `buff1` and `buff2` are often the same in a native-run (violating the *non-overlapping* assumption) because the memory manager reuses the memory allocated for `buf1` after the `free`. Existing symbolic execution engines always assume the address of `buff2` is higher than `buff1`, while SYMLOC does not have this restriction and provides more freedom to explore more execution paths.

```
1 int *buff1 = (int*) malloc(100 * sizeof (char));
2 free (buff1);
3 int *buff2 = (int*) malloc(100 * sizeof (char));
```

It is worth noting that there is no need to be free of the first allocated buffer between two allocation functions to make the memory chunks that `buff1` and `buff2` point to overlapping. Considering two sightly complicated code examples presented in Figure 3.4 and Figure 3.5, a programmer can still make it happen by manipulating the heap allocator. In the first example[3], a programmer may manipulate the heap allocator to make the chunk `p3` pointing to is overlapped at the beginning of `p2`, meaning the addresses of `p2` and `p3` are the same and the bug in Line 9 will be triggered. The mechanism behind it is that by using of the *tcache* (a thread local caching strategy in heap management) free lists, after allocating a chunk and mistakenly free it twice, the allocator can return a desired memory location by writing into the duplicated chunks. Similarly, in the second example[4], after manipulating the heap allocator, i.e., exploiting the overwritten of a freed chunk size in the unsorted bin (i.e., stores a small and large freed chunks, which acts as a cache layer to speed up allocation and deallocation requests) in order to make a new allocation overlap

[3]This allocation can be successful before the commit d081d of glibc.
[4]This allocation can be successful before the commit 88cd0 of glibc.

with an existing chunk, the chunk `p4` is overlapped in the middle of the chunk `p3`, making the condition in Line 13 true and the bug triggering. In this case, the offset "0x500" might be changed based on different manipulating strategies.

Applicable Scenarios. We recognize that the relaxed non-overlapping property is not good for all cases. However, we support it in SYMLOC mainly because we aim to have closer modeling of the heap allocator. As one of the best practices when managing memory, it is recommended that "*it is important to free up memory as soon as you are done using it*" [6]. Therefore, frequently allocating and deallocating memory can be widely used in programs (We also confirmed the benchmarks used in this study follow the above best practice). In such situations, the addresses from afterward allocations are more likely to be overlapped with previous freed ones. In addition, heap allocators may contain vulnerabilities that may be exploited to allocate overlapping memory ranges [213]. Therefore, the relaxed non-overlapping property should be a better choice for modeling the heap allocation behavior in practice. In contrast, existing symbolic execution engines (e.g., KLEE) apply strict non-overlapping property that all the memory objects cannot be overlapped, thus missing code coverage (e.g., `Paths A, C, D`, and `E` in Figure 3.1) and the detection of important categories of memory errors (e.g., the one shown in Figure 3.14). The relaxed non-overlapping property can have a limitation in the situation that the program frees memory objects only once before the execution terminates.

Note that we provide a post-processing option (see Section 3.3.3 for more details) to analyze the results by adding extra constraints for each symbolic address, although SYMLOC does not add more constraints at the first time mainly due to the performance issue. Therefore, when an error is reported by SYMLOC, we run SYMLOC again to perform a post-processing to filter out potential false positives (e.g., the ones that require the address to be allocated in the kernel space).

```
1  int main(){
2    int *p1 = malloc(8); // start manipulating heap allocator
3    free(p1);
4    ... // other implementation code
5    free(p1); // end of manipulating heap allocator
6
7    void *p2 = malloc(8);
8    void *p3 = malloc(8);
9    if ((long)p2 == (long)p3) BUG(); // bug is triggered
10   return 0;
11 }
```

Figure 3.4: Overlapping allocation example 1: overlapping at the beginning

```
1  int main(int argc , char* argv[]) {
2    long long *p1,*p2,*p3,*p4;
3    p1 = malloc(0x500 − 8);
4    p2 = malloc(0x500 − 8);
5    p3 = malloc(0x80 − 8);
6
7    free(p2); // start manipulating heap allocator
8    int designed_chunk_size = 0x581;
9    int designed_region_size = 0x580 − 8;
10   *(p2−1) = designed_chunk_size; // end of manipulating heap allocator
11
12   p4 = malloc(designed_region_size );
13   if ((long) p4 + 0x500 == (long) p3) BUG(); // bug is triggered
14   return 0;
15 }
```

Figure 3.5: Overlapping allocation example 2: overlapping in the middle

3.3.2 Symbolic Memory Operations and Tracking

(1) Symbolic Memory Operations

Existing Symbolic Memory Read/Write Operations. Even though the base addresses and sizes are concrete in this addressing model, symbolic addresses can still be introduced indirectly via symbolic offsets or other symbolic values stored in memory objects used for indirect addressing. Once a symbolic address is used during symbolic execution, a major concern is how to perform memory operations on the memory location(s) addressed by the symbolic address.

Modern symbolic execution engines (e.g., KLEE [28] or *symsize* [190]) often rely on constraint solving to resolve the symbolic addresses and then determine how to handle read/write. For example, if a symbolic pointer `p` is resolved to a single memory object `mo`, then a read via `p` can be represented as `select(mo.addr, e)`, where `e` indicates the offset of `p` in `mo` and can be calculated as (*p* - *mo.addr*);

a write can be represented as `store(mo.addr, e, v)` where `v` is the value to be written into the eth offset of `mo`. If the symbolic pointer `p` can be resolved to more than one value, the symbolic execution engine can choose to either fork the execution states during exploration so that `p` is resolved to a single value in each state for easier analysis [28, 155], or use disjunctive conditions to constrain `p` to all the resolved objects [86, 111, 173] and rely on capable constraint solvers to analyze different combinations of the `select` and `store` operations via `p`.

By equipping with the address symbolization technique, an intractable problem that needed to be resolved in SYMLOC is the symbolic memory read/write operations during execution. This problem is difficult to solve as the symbolic pointer (i.e., address) modeling and resolution is an unsolved and challenging problem in symbolic execution, although it has been actively studied in the literature [9, 12, 31, 42, 49, 77]. The difficulties mainly come from the possible state explosion due to the huge numbers of addresses for symbolic values and the heavy constraint resolution overheads. To bound the state search space and simplify constraint solving, existing symbolic execution engines (e.g., KLEE [28]) usually concretize symbolic addresses into possible concrete values or confine their possible values to certain memory segments based on current path constraints during state exploration. However, such strategies always lead to inaccurate/limited modeling of memory states and program behavior because only a few concrete values or approximated segments are taken into consideration, thus obstructing the effectiveness of analysis.

In this study, a symbolic-concrete memory map-based read/write mechanism is designed for accessing symbolic memory objects. Note that we do not claim to solve the challenging problem of symbolic read/write, as we only focus on enabling symbolic read/write operations by decoupling the operations from heavy constraint solving and avoiding incorrect concretization upon symbolic addresses.

Our key idea is that, for each symbolized address returned from a memory allocation function, via a memory map, we still associate the symbolic address with a concrete memory buffer that would be allocated by actually invoking the allocation

function. When encountering a symbolic memory read/write operation introduced via memory allocations, the associated concrete buffer address is provided to perform the read/write operations, while the address itself remains symbolic and can be encoded into path constraints when they are used in branch conditions. As such, our symbolic memory operations could decouple the symbolic memory read/write operations from heavy constraint resolution and guarantee the correctness of the execution. For example, considering the memory allocation code "char *p = malloc(100)", SYMLOC first declares a symbolic variable for the variable `p` and passes it to the path exploration (as shown in ① in Figure 3). At the meantime, SYMLOC maintains a concrete value (derived from KLEE) of the symbol. Such a concrete value will be used to construct a pair "`<sym_name, <mo, obj>>`" which is stored in the `SymLocMap`, where `sym_name` is a unique name for each memory object and `mo/obj` are memory object and object state maintained by KLEE. Then, the map will be used to further assist symbolic memory operation and tracking (as shown in phases ② and ③ in Figure 3.2.2).

```
1  int main() {
2      char temp;
3      char *buff = (char *) malloc(100 * sizeof (char));
4      klee_make_symbolic(&buff, sizeof(char*), "buff");
5      if ((long long) buff > (long long) some_address) {
6          buff[1] = '9'; // write to a symbolic address
7          ...;   // code to be explored further
8      } else {
9          temp = buff[1];  // read from a symbolic address
10         ...;    // code to be explored further
11     }
12     return 0;
13 }
```

It is also worth noting that the idea of using a memory map to support the symbolic memory operations is straightforward but practically useful for alleviating the path explosion problem in existing symbolic execution engines. In particular, it enables the engine to continue path exploration even when a symbolic address is unresolved. Considering the above example code, when KLEE is applied to explore

Algorithm 1: Symbolic memory operations and tracking

```
   Input: the map symLocMap, a symbolic expression symExpr
   Output: a concrete or symbolic expression, or an error
1  conExpr ⟵ ∅ // initialize a concrete expression
2  FreeList ⟵ ∅ // initialize a list to store freed
       objects
3  Function SymAddrRes(symLocMap, symExpr, func):
4      std::string fname ="free";
       // Situation 1:  handle read/write for normal
          functions
5      if (fname.compare(func->getName()) != 0) then
6          if symLocMap.find(symExpr) then
7              detectUAF(symExpr, FreeList)
8              conExpr = getAddr(symLocMap, symExpr)
9              return conExpr
10         else
11             return symExpr
       // Situation 2:  handle free function
12     if (fname.compare(func->getName()) == 0) then
13         if symLocMap.find(symExpr) then
14             detectDoFOrInF(symExpr, FreeList)
15             FreeList.add(symAddr)
16             conExpr = getAddr(symLocMap, symExp)
17             return conExpr
18         else
19             return symExpr
```

the *if-else* branches and using the KLEE's API `klee_make_symbolic` to make the pointer `buff` symbolic, KLEE can fork two execution states. However, its exploration of the *if* and *else* branches will fail (due to invalid concretized addresses) or take a long time (because it tries to explore all possible concrete values for `buff`) when it encounters the writing operation to the symbolic `buff[1]` (Line 6) or read from the symbolic address (Line 9). In contrast, SYMLOC enables the read/write supported by the underlying concrete buffer associated with the symbolic `buff` and thus can smoothly explore those branches.

(2) Symbolic Memory Tracking

As aforementioned in Section 3.1, existing symbolic execution engines may struggle to reliably detect certain temporal memory errors due to their fundamental design downside. One solution to mitigate the problem may be tracking the use of concrete values in symbolic execution engines and checking the possible errors based on tracking results. However, such a strategy is extremely hard and even practically impossible, for there will usually be millions of internal memory objects to maintain during execution. Therefore, maintaining the checking of those internal memory objects could be time-consuming, which aggravates the scalability and performance issues in symbolic execution.

In SYMLOC, the potential of symbolic memory locations is further activated to perform symbolic memory tracking for reliably detecting temporal memory errors. In general, SYMLOC reuses the capability in symbolic execution, i.e., automatically propagating and tracking the use of symbolic addresses during execution, and adds our designed checking strategy to reliably detect temporal memory errors. The designed checking strategy is straightforward but effective. Note that the reliability of error detection designed in SYMLOC has two important forms. First, SYMLOC is able to effectively catch those errors. Second, SYMLOC could precisely report the root cause information to end-users when a potential error is detected. For example, existing symbolic execution engines (e.g., KLEE [28]) usually emit bogus "*memory error: out of bound error*" to end-users, which may mislead the developers in the debugging process and significantly affect the development progress. In contrast, SYMLOC could precisely report "*memory error: a use after free is detected*" which could quickly help developers locate the root cause of the error. It is also worth noting that existing symbolic execution engines can detect the errors in the form of out-of-bound memory accesses, invalid pointers, etc.; however, it would be troublesome for them to reliably identify the exact *types* of errors as they would need to implement extra analysis to track, relate, and check the uses of the potentially enormous number

of memory objects with concrete addresses.

Algorithm 1 presents the details of symbolic memory operations and tracking designed in SYMLOC. The function `SymAddrRes` takes `symLocMap`, a symbolic address `SymExpr`, and a parameter `func` as inputs and returns a temporal memory error, a concrete address, or the original symbolic variable (when unresolved). Since a symbolic address can be used in different situations, e.g., normal read/write, parameters for calls to external functions that cannot be symbolically executed, and a memory object to be freed, we check for two situations when encountering a symbolic address based on their differences. First, they handle different usages of symbolic expressions that involve the symbols defined by SYMLOC. In terms of implementation, the first situation (with the function call name does not equal to "free") is implemented in functions `executeMemoryOperation` and `callExternalFunction` while the second situation (with the with the function call name does not equal to "free") is used for a special function handler in the function `handleFree`. Second, they indicate the locations for different temporal error detection: SymLoc implements two different functions `detectUAF` and `detectDoFOrInF` to detect UAF, DoF, or InF errors. Before handling those situations, the algorithm first initializes a concrete address `conAddr` to be returned (if any) and a list `FreeList` to record freed objects (Lines 1-2). Then, inside these situations, if the symbolic address `symAdddr` is in the map `symLocMap` (the *if* condition in Line 6 or 13 is true), each situation first checks the use of symbolic addresses aiming to detect potential temporal memory errors. In the following, if there are no such errors, the associated concrete address `conAddr` is returned (Lines 9 and 17) by the calling function `getAddr` (Lines 8 and 16); Otherwise, the `SymExpr` is simply returned (Lines 11 and 19) to be handled by existing resolution strategies in symbolic execution engines.

For more complex scenarios, we replace the symbol with a stored concrete address (i.e., base), and other portions (e.g., offset or scale) are kept the same in KLEE. For example, if the engine got a symbolic expression "α + 100" when

writing over an object, where α is the symbolic address symbolized by SYMLOC and 100 is the offset, SYMLOC will leverage Algorithm 1 to replace the symbolic α to a mapped concrete value in the `SymLocMap`, say, 0x555555784d654. Then, the symbolic expression "α + 100" will be written back to a constant expression "0x555555784d654 + 100" and the write operation will be conducted over the constant expression to avoid the potential state explosion due to forking on α and thus enabling the further execution. When ITE (If-Then-Else) involves different addresses, SYMLOC inherits the state-maintaining strategy when forking new states: if the forked ITEs are an `EqualExpression` and a `NotEqualExpression`, the first state will use the concrete value that complies with the constraints and the second will use the original symbolic value with the concretely mapped address when memory operations are involved; if the ITE is not an `Equal` or `NotEqualExpression` (e.g., `GreaterExpression`), SYMLOC uses the symbolic variable with the same concretely mapped address in both two forked states. Note that in the latter case, we assume the heap allocator could allocate a memory object under some conditions, and the concretely mapped addresses are only used for assisting further path exploration without trapping by the path explosion due to the forking from the symbolic address. Furthermore, If the writes happen on the same path, then the later write would overwrite the previous written values. This operation is inherited from KLEE's design for state forking.

During the symbolic execution process, SYMLOC records all the freed memory objects into a `FreeList` in Line 15, and such a recording will reliably guide the detection of temporal memory errors. Due to the different root causes of three main kinds of errors, SYMLOC performs the corresponding checking on them by invoking `detectUAF` (in Line 7) or `detectDoFOrInF` (in Line 14) function. Those two functions take the symbolic address `symAddr` and the list `FreeList` as inputs and report potential UAF, DoF, or InF errors. For detecting UAF and DoF errors, SYMLOC directly checks whether the symbolic variable under handling is in `FreeList`. For detecting InF errors, SYMLOC checks the type of symbolic

expression to decide whether the pointer pointers to the beginning of a heap object or does not point to any heap object.

With the above capabilities, SYMLOC is able to perform efficient symbolic memory operations and could reliably detect the UAF error in Figure 3.2 when the freed variable `buf` in Line 4 is used in the external function call `printf` in Line 8, where the error is reported in Line 11 in Algorithm 1. It is worth noting that there are static/dynamic detectors (e.g., Frama-C [53], Coccinelle [151], Valgrind [146], and AddressSanitizer [170]) were designed to detect such temporal memory errors, but they need extra analysis algorithms to identify and track the uses of memory objects to detect errors following certain patterns. SYMLOC essentially enables the tracking of memory locations by conveniently utilizing the capability of symbolic execution: it shares the same core idea as other program analysis-based checkers, enabling more error-detection capability in symbolic execution engines, but without the need for the troubles in building custom analysis tracking algorithms.

3.3.3 Implementation of SYMLOC

We implemented SYMLOC on top of KLEE (version 2.1) with LLVM 9.0.0 and STP 2.3.3. We modified KLEE's allocation API to return symbolic addresses instead of concrete ones and maintained a memory map to support practical symbolic memory operations. Our memory model allows the allocation of memory objects using either concrete or symbolic addresses for different usages (i.e., three options detailed in Section 3.3.1). We also modified the APIs involving memory read/write, external function calls, and `free` in KLEE to support the normal uses of symbolic addresses and the reliable detection of temporal memory errors (see Algorithm 1 in Section 3.3.2). Besides, for each test case generated by SYMLOC, a text file that records essential information (e.g., unique names of symbolic addresses, invoking locations, the forking point, and the free point) is provided to help facilitate error verification, debugging, or exploit generation.

SYMLOC does not add extra constraints for each symbolic address when detecting new errors mainly due to the performance issue: more complex constraints for each symbolic hardness the constraint solving and affect bug detection capabilities. However, to reduce potential false positives (e.g., the required address is not in user space) reported by SYMLOC, SYMLOC supports a post-processing option to filter out these cases, i.e., SYMLOC adds extra constraints to validate the validity of each allocated address. To be specific, , for each symbolized address, we augmented its constraints to be within the address range of user space (i.e., 0x0-0x7fffffffffff) and re-run the program to filter such error/false positive reports out. We believe such post-processing of reported errors can help users analyze the root causes of the error reports more effectively. For more complex modeling of relationships of different memory objects, we leave it as our future work.

Symbolization Strategies for Users. Although SYMLOC enables the exploration of more execution paths, introduced symbolic addresses may lead to additional execution states forking and more complex path constraints. To alleviate this problem, our implementation provides three options to end-users, i.e., the full, random, and selective symbolization of addresses. The first option is fully symbolization, which symbolizes every memory location (i.e., returned addresses of all the memory allocations). The second option is random symbolization, which randomly symbolizes memory locations, which can save some computing resources than the first option. The final option is selective symbolization. Such a selection allows users to specify the memory locations to be symbolized. We provide an API (i.e., `klee_make_malloc_symbolic` for this purpose, where the memory location after inserting the API will be symbolized. Users can have their own choices to select a more appropriate way to perform the address symbolization. For example, the first option takes more computing resources and may be used for programs with a few memory allocations; the second option can be used when the number of allocated buffers is large, and it is unknown yet which allocated addresses may affect the program behaviors during the initial testing of the subject programs; the last option

is preferred when users know what are the interesting allocation points in a program to be analyzed.

3.4 Evaluation

This section presents our experimental settings and results. We aim to answer the following research questions (**RQs**):

RQ1: How does SYMLOC perform in detecting spatial memory errors?

RQ2: How does SYMLOC perform in detecting temporal memory errors?

3.4.1 Spatial Memory Errors Detection Capability

We measure the performance in terms of the number of spatial memory errors detected and code coverage achieved.

(1) Experimental Settings

Baseline Approaches. We focus on a comparison with the widely used symbolic execution engine KLEE first since we implement SYMLOC on it. We also use a recent symbolic execution engine (*symsize* [190]) that models allocation sizes during symbolic execution, to see the effects of buffer sizes on error detection and code coverage. We did not compare SYMLOC with other static/dynamic memory detectors in RQ1 because SYMLOC is built on a symbolic executor KLEE, and we mainly aim to investigate the multi-path exploration capability (that is only applicable for symbolic executors) among comparative approaches, and it should be fairer to compare with them. To this end, we used a multi-path exploration mode of comparative symbolic executors and compared their performance in terms of code coverage and error detection capability benefiting from the code coverage improvements.

Benchmark Programs. We use 15 programs in GNU `Coreutils` (version 9.0) and two large programs (GNU `make` [132] and `m4` [128]) (cf. Table 3.1) for

the evaluation, as they are commonly used in evaluating various symbolic execution techniques [28, 97, 120, 186, 190]. We excluded some Coreutils programs that: (1) do not invoke dynamic memory allocation through the `malloc` function or (2) may cause non-deterministic behaviors (e.g., `kill`, `ptx`, and `yes`), following existing studies [97, 120].

Running Settings. We followed prior work [28, 120] to set symbolic inputs for GNU `Coreutils` programs; we configured the symbolic options based on their input formats and prior work [186] for the two large programs. We use *Breadth First Search (BFS)* to deterministically guide the path exploration of all the comparative approaches. For *symsize*, we run it under `Merging` mode with optimizations. We run the benchmarks with a timeout of one hour per test program, following the same setting as existing studies [28, 97, 120, 186, 190]. Then, we measure the code coverage achieved and the errors detected. Besides, we use the *full* symbolization option to run the small-size benchmarks and the *random* symbolization option to run the large-size benchmarks. We run our experiments on a Linux PC with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz x 12 processors and 64GB RAM running Ubuntu 18.04 operating system.

(2) Results

1) Spatial Memory Error Detection Capability. Table 3.2 presents the summarized results of detected errors by the three approaches. The first column represents the error types, and columns 2-4 record the number of detected errors by each approach. The row of *Spatial Memory Errors* represents the number of spatial memory errors detected, and the row *Others* indicates other errors such as unsupported modeling of certain program states (e.g., symbolic size), unsupported interpreting inline assembly code, and failed external function calls due to symbolic arguments. We show the unique errors detected by the three approaches as a Venn diagram in Figure 3.6.

We can observe that SYMLOC significantly outperforms the others: all the errors reported by KLEE and *symsize* can be detected by SYMLOC, and 17 unique errors

Table 3.1: The benchmarks used in the evaluation, with their version, size, and the source lines of code (SLOC)

Benchmark	Version	Size	SLOC
basename	9.0	small-size	1.0K
chroot	9.0	small-size	1.2K
date	9.0	small-size	2.8K
dd	9.0	small-size	2.1K
dircolors	9.0	small-size	1.1K
factor	9.0	small-size	2.2K
head	9.0	small-size	1.4K
ln	9.0	small-size	2.3K
od	9.0	small-size	1.8K
pr	9.0	small-size	2.6K
rm	9.0	small-size	2.6K
seq	9.0	small-size	1.2K
stat	9.0	small-size	2.8K
sum	9.0	small-size	1.6K
tee	9.0	small-size	1.0K
make	4.2	large-size	20K
m4	1.4.18	large-size	80K
Juliet Test Suite	1.3	-	various

are address-specific spatial memory errors that can only be detected by SYMLOC; SYMLOC improves the error-detection capability of KLEE and *symsize* by 169% and 218%, respectively.

To further inspect the causes of the memory errors, we manually analyze the 17 unique errors detected by SYMLOC. We categorize the errors mainly based on the concrete values solved by the constraint solver, and we can check whether a concrete address can happen in a user-space (0x0-0x7fffffffffff) or kernel-space (0xffff800000000000-0xffffffffffffffff) memory region:

Type 1: User space errors. A large portion (i.e., nine) of the errors are of this type, i.e., they can be potentially reproduced in user space. Seven of them are triggered when the location of the memory is *NULL*, which can help better check the error-handling code (see case studies 1 and 2 in Section 3.5). Two errors are caused by the locations that are not *NULL* and it can assist in boosting API usage checking

Table 3.2: Results of the overall number of detected errors

Error Types	KLEE	*symsize*	SYMLOC
Spatial Memory Errors	8	7	25
Others	5	4	10
Total	13	11	35

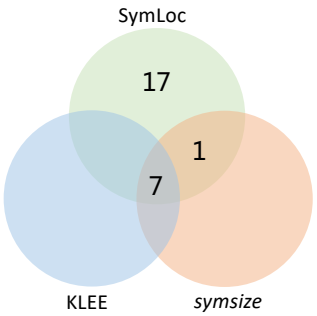


Figure 3.6: Distribution of address-specific spatial memory errors detected by comparative approaches

(see sample in Implication 2 below). The other eight errors are triggered mainly due to possible errors in program logic when a buffer is allocated at a specific address. Such errors require expert knowledge to troubleshoot the root causes.

Type 2: Kernel space errors. Five errors happen when the memory locations are in kernel space.

Type 3: Mixed-space errors. Three errors are under this type. The locations of their memory objects are a combination of user-space and kernel-space, which may involve complex interactions between the user-space program and the kernel's execution to reproduce them.

Since the types 2 and 3 are less likely to happen in user-space. Therefore, we use the post-processing option supported by SYMLOC to filter them.

It is worth noting that the errors uniquely found by SYMLOC can have important implications such as enhancing the null pointer dereference checking of the test programs that involve complex data/control flow and the potential side effects when comparing two pointers from stack and heap (see more details in Section 3.5).

2) Improved Code Coverage. Besides the errors detected, code coverage metrics are often used to measure the effectiveness of a software testing tool. We use

Table 3.3: Results of branch coverage (measured by `klee-stats`) and line coverage (measured by `gcov`)

Benchmarks	Branch Coverage (%)			Line Coverage (src+lib) (%)		
	KLEE	symsize	SymLoc	KLEE	symsize	SymLoc
basename	**29.17**	29.02	**29.17**	**42.20**	38.90	38.90
chroot	33.24	33.75	**33.93**	30.90	**32.40**	30.90
date	17.66	**26.41**	17.74	23.80	**35.80**	23.80
dd	35.96	29.42	**36.77**	**39.60**	32.90	39.40
dircolors	35.39	34.92	**35.60**	44.80	42.40	**46.40**
factor	26.84	22.76	**27.93**	24.00	21.30	**24.70**
head	34.58	31.44	**35.96**	37.70	17.60	**41.70**
ln	27.47	24.93	**27.80**	26.00	19.70	**31.40**
od	**28.43**	23.85	28.39	**42.10**	23.20	40.20
pr	17.99	17.39	**18.24**	34.50	33.70	**34.60**
rm	26.02	24.95	**28.05**	27.70	25.40	**31.80**
seq	34.71	**34.94**	34.71	**34.80**	**34.80**	**34.80**
stat	21.10	19.82	**21.56**	24.20	20.70	**24.80**
sum	36.20	34.72	**36.44**	**30.10**	17.80	30.00
tee	27.75	27.64	**28.19**	**44.10**	**44.10**	**44.10**
m4	4.58	**4.66**	4.47	**9.50**	**9.50**	8.50
make	20.12	15.90	**21.28**	22.10	22.10	**23.80**
Num. of Best	2	3	13	7	5	10

the tool `klee-stat` in KLEE to collect branch coverage and the tool `gcov` [83] to compute the line coverage information. `klee-stat` measures the "internal coverage" of the program under test, where the coverage is measured at the level at which the symbolic execution engine operates—LLVM *bitcode* instructions [105]. In contrast, `gcov` computes the "external coverage" of the test program. Note that internal coverage such as branch coverage reported by KLEE is more correlated with code coverage and even error-detection capability [26]. For line coverage, we measure the code covered in both the program itself and the libraries used by the program (src+lib). Further, we measure how SYMLOC covers code that is *not* covered by other approaches. Note that we re-run the test cases generated by SYMLOC and measure the code coverage of test programs under test. Therefore, all lines measured are feasible lines covered by the test cases produced by SYMLOC.

Unique Line Coverage. To calculate the unique line coverage of each approach (say A) with respect to another approach (say B), we first obtain the intersection of the lines covered by both approaches, i.e., $I(A, B) = A \cap B$. Then, the unique line

coverage achieved by A is measured as

$$A_u = \frac{N(A - I(A,B))}{N(A - I(A,B)) + N(B - I(A,B))}$$

where $N(A - I(A,B))$ represents the number of lines covered by A minus the number of intersected lines. Figure 3.7 and Figure 3.8 show the results of unique coverage achieved by three approaches on GNU Coreutils. The labels under the *x-axis* indicate the names of the programs, and the values on the *y-axis* represent the unique line coverage of each approach. We can see that SYMLOC achieves higher unique line coverage in most of the benchmark programs. On average, SYMLOC is able to cover 15% and 48% more unique lines than KLEE and *symsize*, respectively. The result is expected as SYMLOC has a unique capability to cover code blocks where the condition depends on the addresses allocated by the heap allocator. For example, existing symbolic execution engines (either KLEE [28] or *symsize* [190]) always assume that the address of "*buf2*" is larger than "*buf1*" in Figure 3.1, leading to limited code coverage. However, such an assumption is not held in SYMLOC. Specifically, SYMLOC could generate test cases that exercise unique lines of code, e.g., `Path-C` in Figure 3.1 where the address of "*buf2*" is smaller than "*buf1*".

From Figure 3.7 and Figure 3.8, we can see that some lines of code are uniquely covered by KLEE or *symsize*. The main reason for this is that SYMLOC takes relatively more time in constraint solving than the two comparative approaches and in exploring program branches that would not be explored otherwise, thus may have less time in exploring some branches that would be explored sooner by other tools. To support our claim, we use the tool `klee-stats` to get how much time is spent on constraint solving for each approach, i.e., `TSolver` reported by KLEE, where the solving time indicates the percentage of the relative time spent in the solver over the whole program execution time. As a result, for the four benchmarks (i.e., `dd`, `od`, `seq`, and `sum`) that cover more unique lines of code than SYMLOC, the results show KLEE spends 76.89%, 35.78%, 3.27%, and 49.83% time over the whole execution

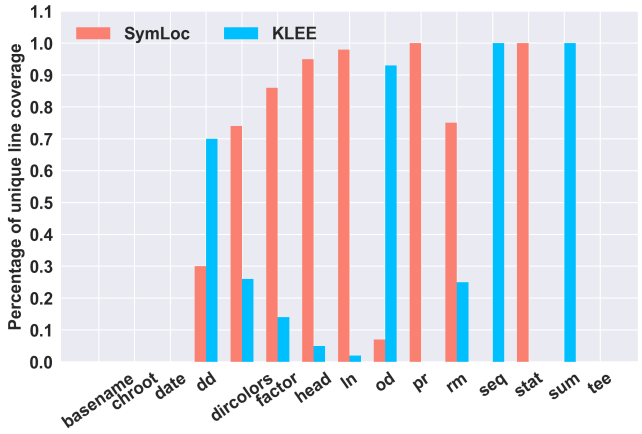


Figure 3.7: Unique line coverage (measured by `gcov`): SYMLOC vs KLEE

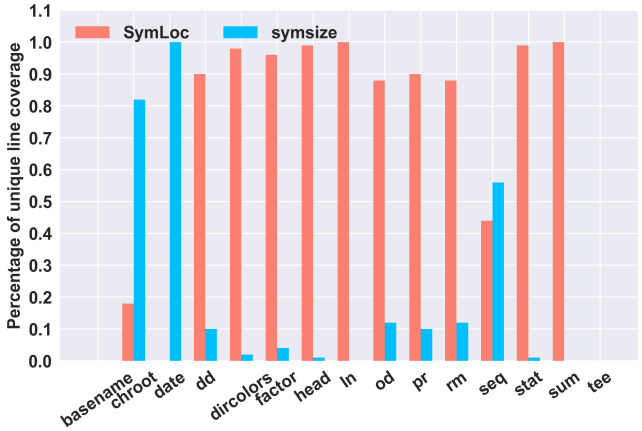
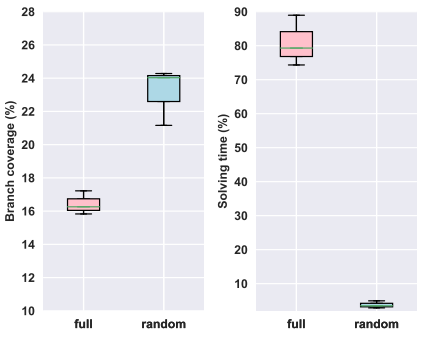


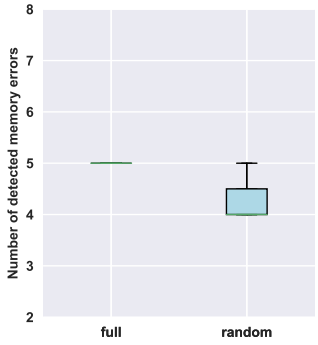
Figure 3.8: Unique line coverage (measured by `gcov`): SYMLOC vs *symsize*

time (i.e., 1 hour) on constraint solving, whereas SYMLOC takes 79.83%, 41.81%, 19.26%, and 65.41% time on solving constraints, respectively. The results show the same trends when comparing *symsize* with SYMLOC. *symsize* contributes 16.54%, 40.40%, and 45.71% constraint solving time, while SYMLOC uses 48.71%, 82.23%, and 48.45% constraint solving time over the three benchmarks basename, date, and seq, respectively.

Branch and Line Coverage. As presented in Table 3.3, the first column shows the names of benchmark programs, and the rest of the columns record the branch or line coverage over each program under comparative approaches. Columns 3-8 are divided into two groups; each group represents a different coverage metric, i.e., branch coverage or line coverage. Apart from the numbers in the last row, each coverage number is calculated as $\frac{N_{covered}}{N_{total}} \times 100$, where the $N_{covered}$ represents the



(a) Branch coverage and constraint solving time over `make-4.2`



(b) Number of detected errors over `make-4.2`

Figure 3.9: Comparison of different symbolization modes in SYMLOC (*Full* symbolization mode VS *Random* symbolization mode)

covered number of branches/lines and N_{total} corresponds to the total number of branches/lines. The coverage on m4 and make is the *median* value of five repeated runs. The last *row* counts the total number of the best coverage achieved by each approach for the programs. In terms of branch coverage, we observe that SYMLOC outperforms KLEE and *symsize* overall and dominates 72% (13 out of 18) over all the benchmarks. Specifically, SYMLOC improves at best by 18% (in `factor`) and 25% (in `dd`) than KLEE and *symsize*, respectively.

The line coverage results are shown in the last three columns in Table 3.3. We can observe that SYMLOC is able to cover more lines of code than KLEE and *symsize*; it improves the line coverage by up to 21% (in `ln`) and 137% (in `head`) than KLEE and *symsize*, respectively.

Impact of Different Symbolization Modes. We run random and full modes in one of the large-scale benchmarks (i.e., `make-4.2`) and compare branch coverage and the memory error detection capability of SYMLOC. Figure 3.9 shows the detailed results. As shown in the box plot of Figure 3.9a, where the *x-axis* represents two different symbolization modes and *y-axis* describes the branch coverage `Bcov` or solving time `TSolver` reported by KLEE. We can see the branch coverage in the *Full* mode is lower than the one under *Random* mode. This is reasonable because the *Full* symbolization means the more complex constraints during the symbolic

execution, which leads to more time on constraint solving. As shown on the right side in Figure 3.9b, we can observe that the constraint-solving time on *Full* mode is significantly larger than the one in *Random* mode, which supports our claim. For the error detection capability, the *Full* mode is better than *Random* mode. This is because the full symbolization mode has a better chance to explore more execution paths, thus making a larger number of errors detected. We did not report the *Selective* mode because this is designed for users who have a certain knowledge of the target program and know how to select a better object to be symbolized. We have manually tested this mode on selected benchmarks and the results show it works as expected.

Answer to RQ1: SYMLOC is able to detect 169% and 218% more spatial memory errors as well as cover 15% and 48% more unique lines of code than the two baseline approaches.

3.4.2 Temporal Memory Errors Detection Capability

We measure the performance in terms of the number of temporal memory errors detected and detection time cost.

(1) Experimental Settings

Baseline Approaches. We evaluate SYMLOC against different kinds of state-of-the-art memory error detectors. We use two static (Frama-C [53] and Coccinelle [151]) and two dynamics (Valgrind [146] and AddressSanitizer [170]) detectors studied in [93], as they are shown to be the top approaches in the categories of static and dynamic memory error detectors. For symbolic execution-based approaches, we opt for *symsize* [190] and several variants of KLEE for error detection, including, KLEE(DEF), KLEE(DET), KLEE(OPT), and KLEE(DET+OPT): KLEE(DEF) is the default setting of KLEE; KLEE(DET) enables "*–allocate-determ*" option for deterministic allocation on top of the default KLEE; KLEE(OPT) compiles the source code with higher compiler optimization (i.e., "`-O1`"); KLEE(DET+OPT)

turns on the "*–allocate-determ*" and "`-O1`" together. The choice of w/ or w/o the deterministic allocation or higher optimization is justified by the fact that different configurations of KLEE can have different effects [30] and the two selected options could affect the temporal memory error-detection capability of KLEE as confirmed by KLEE's authors [63].

Benchmark Programs. We use the C/C++ programs in Juliet Test Suite (JTS) [79] (cf. the last row in Table 3.1) that use the `malloc` function in this subsection. It includes 137 programs in CWE416 that have known UAF and 283 programs in CWE415 that have known DoF. Note that since the benchmarks used in RQ1 rarely have temporal memory errors (We run Coccinelle on the benchmarks used in RQ1 and the results show no single temporal memory error is detected), we then used the wide-used benchmark JTS for a fairer comparison. So, we compared SYMLOC with both symbolic executors running single-path mode and static/dynamic tools over the JTS benchmarks in terms of the number of memory errors detected.

Running Settings. We use the Frama-C (version Phosphorus-20170501) with "-val" and Coccinelle (version 1.0.4) with the UAF patterns specified with its official UAF (`osdi_kfree.cocci`) and DoF (`frees.cocci`) scripts to static analyses each test program. Valgrind (version 3.13) is used after compiling with "*gcc-7.5 -O2*". AddressSanitizer (the built-in version in LLVM-10) is run with "*clang-10 -fsanitize=address -O2*", following the existing setting [93]. We applied the same setting to symbolic execution-based approaches on the same machine as RQ1.

(2) Results

1) Comparison with Static/Dynamic Memory Detectors. Figure 3.10 and Figure 3.11 show the experimental results, where the labels under the *x-axis* correspond to the number of errors, and the values on the *y-axis* represent the total number of errors (the number on the right side of the bar) or the completeness of detecting all the errors (the percentage point inside the bar) detected by each detector. We can see that SYMLOC performs the best, detecting all (100%) the UAF and DoF errors in the

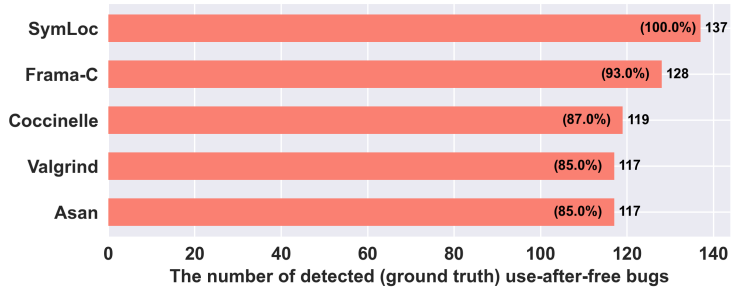


Figure 3.10: Completeness of UAF error detection among static/dynamic analysis-based approaches (137 in total).

JTS benchmark.

Reasons for Detection Failures. Static detectors (i.e., Frama-C [53] and Coccinelle [151]) can not detect certain errors due to two reasons. First, Frama-C is limited in its inter-procedural analysis and Coccinelle is limited in the comprehensiveness of its error detection patterns (which usually needs expert knowledge and is time-consuming to craft), which are insufficient to catch all possible UAF or DoF errors, especially inter-procedural ones [93]. Second, both the Frama-C and Coccinelle are limited in analyzing C++ language so they miss errors written in C++. Dynamic detectors (i.e., Valgrind [146] and AddressSanitizer [170]) miss errors mainly due to their capability of handling certain C library functions, e.g., `wmemset` and `wprintf` presented in Figure 3.2. Adding support for those functions is practically feasible, but it can amply much overhead. More specifically, they all use *shadow memory*, i.e., one-level or multi-level lookup tables, to store the state of all memory objects. When more functions are supported, the number of stored objects can be numerous which may significantly enlarge the search space. Furthermore, AddressSanitizer can only detect 44.5% DoF errors, as its memory error detection is significantly affected by aggressive compiler optimization levels due to the source-code level instrumentation [68].

For DoF errors, although SYMLOC and Valgrind are able to detect all the DoF errors in JTS datasets, SYMLOC can still have advantages in detecting more errors in general. For example, consider the example code in Figure 3.14 (adopted from a

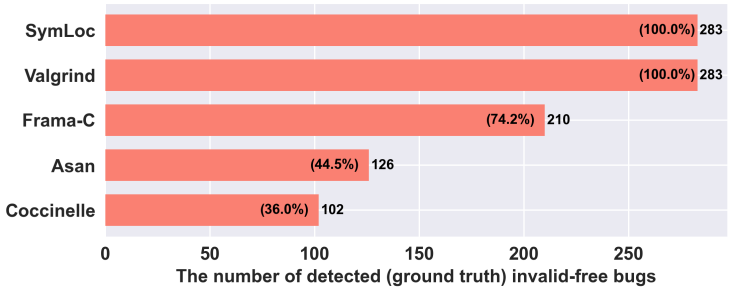


Figure 3.11: Completeness of DoF error detection among static/dynamic analysis-based approaches (283 in total.

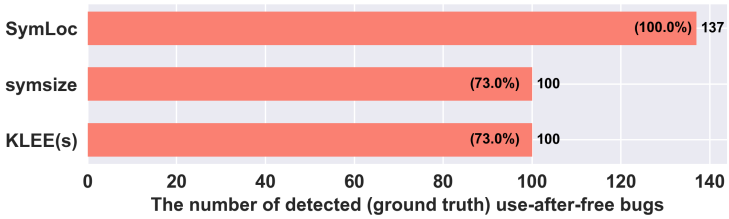


Figure 3.12: Completeness of UAF error detection among symbolic execution-based approaches (137 in total).

GitHub issue[5]) beyond the JTS dataset; Valgrind cannot detect this DoF sometimes due to the same reason why they miss UAF errors: the allocated buffer in Line 3 can have the same concrete address as the allocated buffer in Line 1 after `buff1` is freed. In contrast, SYMLOC has no such problem, and it attributes the freed object to the free list so that it will report errors if some of the objects in the list are used later. It is worth noting that some dynamic analysis tool fails to detect such error as well. For Valgrind, it cannot detect this bug mainly due to the limited modeling of dynamically allocated memory objects as well. Valgrind maintains a "shadow memory", which is a mirror of the actual memory used by the program. For every byte of memory in the application, there is a corresponding byte (or bytes) in the shadow memory that records whether the application's byte is defined, undefined, or addressable. However, such modeling of the heap still treats the address of x and y differently, causing the miss detection of the double-free error. Note that in a native execution of the program, we run this example 10000 times, and the executable always treats the address of x and y the same and terminates the execution with an

[5] https://github.com/staticafi/symbiotic/issues/89

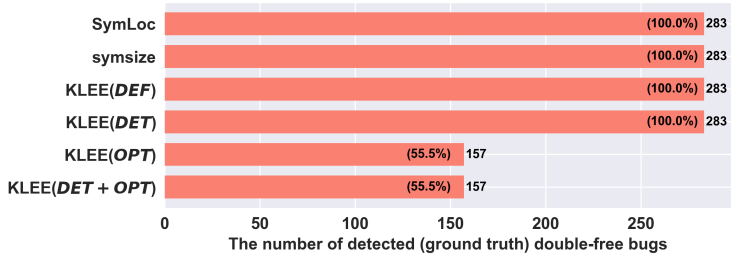


Figure 3.13: Completeness of DoF error detection among symbolic execution-based approaches (283 in total).

error "free(): double free detected in tcache 2, Aborted (core dumped)". Therefore, we believe this is an error that has a high possibility to occur in real-world execution.

2) Comparison with Variants of KLEE. For UAF error detection, Figure 3.12 shows the overall experimental results. Since KLEE and most of its variants detect the same number of errors, we use "KLEE(s)" to refer to each of those detectors (either KLEE(DEF), KLEE(DET), KLEE(OPT), or KLEE(DET+OPT)) and do not repeat showing them. From Figure 3.12, we can see that SYMLOC performs the best and detects all (100%) the UAF and DoF errors, while other detectors miss 27% of UAF memory errors and some variants (i.e., KLEE(OPT) and KLEE(DET+OPT)) miss nearly half (i.e., 54.4%) of DoF memory errors in the used benchmarks.

Reasons for Detection Failures. KLEE and its variants miss some certain UAF and DoF memory errors mainly due to the fundamental issue mentioned in Section 3.2.2, where the addresses of freed memory objects may overlap with addresses returned in subsequent allocations and thus the freed objects are mistakenly treated as "valid", where the objects should be marked as invalid instead. Besides, enabling "`-allocate-determ`" or with higher optimization "`-O1`" does not help alleviate the fundamental issue in KLEE's internal design. Worse still, the higher optimization may even aggregate the problem due to the fact that compiler optimization can be too aggressive [67, 206], so this is the reason why KLEE(OPT) and KLEE(DET+OPT) miss nearly half of Dof memory errors. Note that although the KLEE(DEF) and KLEE(DET) detect the same errors as SYMLOC, our approach can still have advantages in detecting more memory errors in general. For example,

```
1 int main() {
2     void *x = malloc(100);
3     free(x);
4     void *y = malloc(100);
5     if (x == y)
6         free(y);
7     free(y); // a double free error
8 }
```

Figure 3.14: A simple DoF error missed by KLEE [28] and Valgrind [146]

consider the example code in Figure 3.14 again, which is also confirmed by experts [69]; KLEE and its variants cannot detect this DoF due to the same reason of missing UAF errors.

It is worth noting that the error information reported by SYMLOC is more precise as aforementioned in Section 19. Specifically, SYMLOC could directly report the root cause of the error such as "*use after free*", or "*double free*", while KLEE only yields unclear ones such as "*out of bound error*" or "*invalid free*". There is no doubt that more precise information could assist developers in quickly debugging and fixing potential errors.

Speed Comparison. We also measured the time of the comparative approaches. For symbolic execution-based approaches, at first, we count the time spent on testing all the programs. The results show that SYMLOC, *symsize*, KLEE(DEF), KLEE(DET), KLEE(OPT), and KLEE(DET+OPT) take 150.5, 150.3, 152.8, 152.9, 152.9, and 155.1 seconds on running 137 programs that contain UAF errors, respectively. For DoF errors, those approaches spend 323.0, 321.2, 332.3, 333.9, 333.3, and 335.2 seconds to finish 283 programs that include DoF errors, respectively. Second, for the four approaches to detect the same number of DoF errors as presented in Figure 3.11, we compared the time spent in detecting each of the DoF errors. The results show that SYMLOC, *symsize*, KLEE(DEF), and KLEE(DET) spent 1.14, 1.12, 1.18, and 1.18 seconds to detect each error, respectively, indicating SYMLOC has a relatively lower overhead in detecting temporal memory errors. The results are reasonable as existing symbolic execution engines (e.g., KLEE [28]) usually look up the freed memory object from its self-maintained huge memory management pool,

which may take time. *symsize* takes less time than SYMLOC as it has a strategy to reduce the time in executing loops when the number of iterations depends on the buffer size; e.g., in the programs, KLEE and SYMLOC currently only use 100 as the default buffer size, while *symsize* tries small ones such as 1 first. In contrast, SYMLOC catches those errors by directly searching the memory objects in *FreeList* mentioned in Algorithm 1, resulting in a better performance.

For the other memory detectors, Coccinelle, Frama-C, Valgrind, and Address-Sanitizer spent 12.1, 60.5, 83.2, and 40.7 seconds to finish all 137 test programs that contain UAF errors, while 31.8, 150.4, 180.3, and 88.9 seconds to run over all 283 test programs that contain DoF errors. The results are reasonable as they are very different approaches from symbolic execution and are expected to be faster than symbolic executors as they do not need to interpret/simulate program executions.

Answer to RQ2: SYMLOC has an overall better temporal memory error detection capability for detecting UAF and DoF errors than other state-of-the-art static, dynamic, and symbolic execution-based approaches.

3.5 Case Studies

This subsection showcases two memory errors detected by SYMLOC but missed by other tools and discusses their implications.

3.5.1 Case 1: Single *NULL* Pointer Dereference in `rm`

NULL pointer dereferencing is a dangerous operation in memory assessing [183]. SYMLOC can detect certain *NULL* pointer dereferencing caused by improper checking on allocated pointers in complex programs.

Error Details. Figure 3.15 presents a code example showing an error that happens when programmers insufficiently check a possible *NULL* pointer that is passed across multiple functions in multiple source files. The functionality of the `rm`

```
1  // From the ./src/remove.c file :
2  enum RM_status rm (char *const *file, struct rm_options const *x) {
3    ...
4    FTS *fts = xfts_open (file , bit_flags , NULL);
5    while (true) {
6      FTSENT *ent = fts_read (fts); ...
7    }
8  }
9  // From the ./lib/fts.c file :
10 FTSENT * fts_read (register FTS *sp) {
11   ...
12   if (sp->fts_cur == NULL || ISSET(FTS_STOP)) return (NULL);
13   p = sp->fts_cur;
14   ...
15   if ((p = p->fts_link) != NULL) { ...
16     if (p->fts_level == FTS_ROOTLEVEL) {
17       if (restore_initial_cwd (sp)) { ... }
18       ...
19       setup_dir(sp) ;
20       goto check_for_dir;
21     }
22   }
23   ...
24 check_for_dir:
25   ...
26   if (p->fts_info == FTS_D) {
27     if (! enter_dir (sp, p)) { ... }
28   }
29   ...
30 }
31 // From the ./lib/fts-cycle.c file :
32 static bool setup_dir (FTS *fts) {
33   if (fts->fts_options & (FTS_TIGHT_CYCLE_CHECK | FTS_LOGICAL)) {
34     ...
35   } else {
36     fts->fts_cycle.state = malloc (sizeof *fts->fts_cycle.state);
37     if (! fts->fts_cycle.state)
38       return false;
39     cycle_check_init (fts->fts_cycle.state);
40   }
41   return true;
42 }
43 // From the ./lib/cycle-check.c file :
44 void cycle_check_init (struct cycle_check_state *state) {
45   state->chdir_counter = 0;
46   state->magic = CC_MAGIC;
47 }
48 // From the ./lib/fts-cycle.c file :
49 static bool enter_dir (FTS *fts, FTSENT *ent) {
50   ...
51   if (cycle_check (fts->fts_cycle.state, ent->fts_statp)) ...
52 }
53 // From the ./lib/cycle-check.c file :
54 bool cycle_check (struct cycle_check_state *state, struct stat const *) {
55   assure (state->magic == CC_MAGIC); // out-of-bound error
56   ...
57 }
```

Figure 3.15: Case 1: memory error at `cycle-check.c:60` in `rm`

function in Line 2 is to first find the target files/directories and then remove found targets if any. In the `rm` function, it opens files from one of the argument *file* in Line 4 and starts to read the file inside a *while-loop* in Line 6 through the function `fts_read`. Inside the `fts_read` function, several *if* checkings are performed

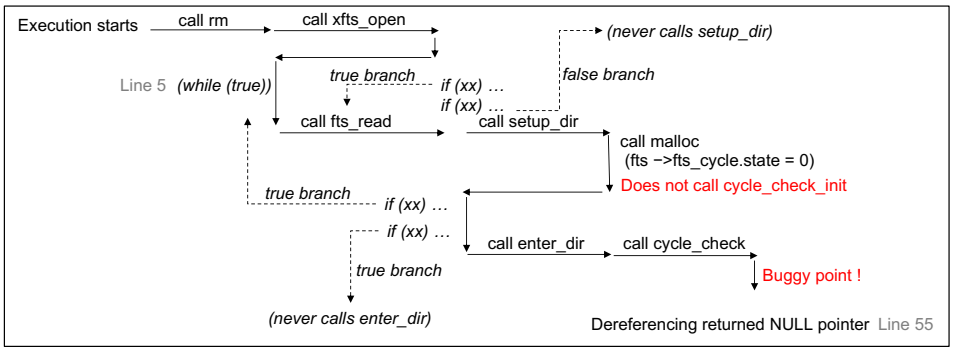


Figure 3.16: Execution flow of *NULL-pointer dereference* in Case 1

before it sets up the directory environment via function `setup_dir` in Line 19. Inside function `setup_dir`, the object *fts->fts_cycle.state* is allocated through function `malloc` in Line 36, and a *NULL* pointer checking is applied in Line 37. If the allocation returns *NULL*, the `setup_dir` function returns *false* in Line 38. Subsequently, the `fts_read` function continues to check for certain conditions and calls the `enter_dir` function.

The *NULL* pointer dereference error occurs when the allocation in function `setup_dir` returns *NULL* to the object *fts->fts_cycle.state* in Line 36 and the subsequent invocation of `enter_dir` in Line 27 in the function `fts_read` uses the *fts->fts_cycle.state* object. To be specific, the object *fts->fts_cycle.state* with *NULL* value is used via function `cycle_check` in Line 51 and the *NULL* pointer is finally dereferenced in Line 55 (reported as "*out-of-bound*" error). Figure 3.16 shows the execution flow of Case 1, where the arrows with the solid line represent the calling flow and arrows with dashed lines refer to the control flow for the error triggering. We can see that the error happens if the object *fts->fts_cycle.state* is assigned with *NULL* value but is used across many functions in multiple files. The developers have fixed the bug by adding an `if` to check if `setup_dir(sp)` returns true[6] in the latest versions of `rm`.

[6] https://git.savannah.gnu.org/cgit/gnulib.git/commit/?id=f17d397771164c1b0f77fea8fb0abdc99cf4a3e1

```
1  #define OUT_OF_MEM() O (fatal, NILF, _("info"))
2  #define O(_t, _a, _f)  _t ((_a), 0, (_f))
3
4  void * xrealloc (void *ptr, unsigned int size) {
5    void *result;
6    result = ptr ? realloc (ptr, size) : malloc (size);
7    if (result == 0)
8      OUT_OF_MEM();
9      return result;
10 }
11 void fatal (const floc *flocp, size_t len, ...) {
12   len += (strlen (fmt) + strlen (program) + (flocp && flocp->filenm ? strlen(flocp->filenm):0)+
           INTSTR_LENGTH+8+strlen(stop)+1);
13   char * p = get_buffer (len);
14   ...
15   die (MAKE_FAILURE);
16 }
17 static struct fmtstring {char *buffer; size_t size;} fmtbuf = {NULL, 0};
18 static char * get_buffer (size_t need) {
19   if (need > fmtbuf.size) {
20       fmtbuf.size += need * 2;
21       fmtbuf.buffer = xrealloc (fmtbuf.buffer, fmtbuf.size);
22   }
23   fmtbuf.buffer[need-1] = '\0'; // out-of-bound error
24   return fmtbuf.buffer;
25 }
```

Figure 3.17: Case 2 : memory error at `output.c:605` in `Make-4.2`

3.5.2 Case 2: Consecutive *NULL* Pointer Returns in `Make`

Error Details. Figure 3.17 shows a memory error that occurs when programmers lack the handling of consecutive *NULL* pointer returns. Normally, a *NULL* pointer checking will be conducted after the allocation of the memory. For example, in the function `xrealloc` in Line 4, the return value *results* of the allocation (either by invoking `realloc` or `malloc`) was checked in Line 7. If this value equals 0, the program will be terminated using the function `OUT_OF_MEM` in Line 8 as expected. The implementation of `OUT_OF_MEM` (Line 1) is a macro definition of the function `O` (Line 2) which finally invokes function `_t` (i.e., function `fatal` presented in Line 11). However, a memory error can happen when the `xrealloc` function is called *recursively* through the `OUT_OF_MEM` macro and its returned *NULL* pointer is dereferenced via *fmtbuf.buffer*.

Figure 3.18 shows the execution flow of Case 2. To be specific, the function `fatal` (Line 11) is invoked by the macro `OUT_OF_MEM`. The `fatal` function first allocates a buffer via function `get_buffer` (Line 13) based on the needed length *len* and terminates the execution via function `die`. The function `get_buffer`

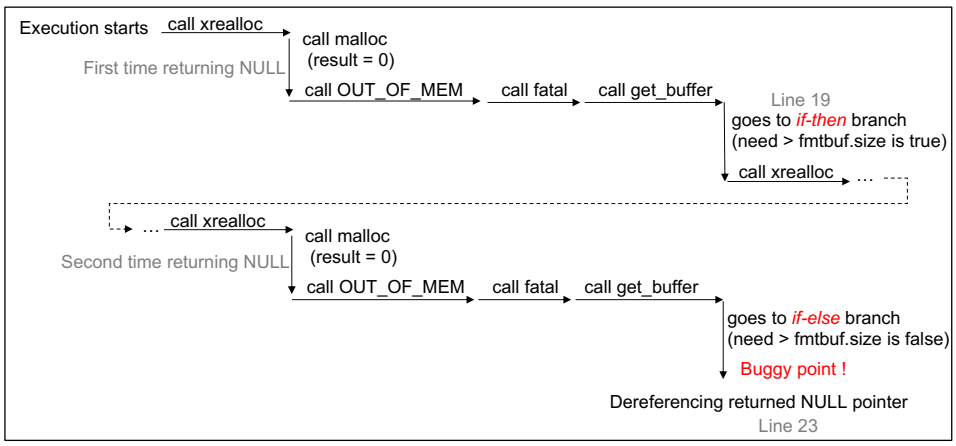


Figure 3.18: Execution flow of *NULL-pointer dereference* in Case 2

first checks whether the needed size *need* is larger than the buffer size *fmtbuf.size* in Line 19. The *need* argument is passed through function `fatal` and updated in Line 12 (holds a non-zero value) and *fmtbuf.size* is initially 0 (set in Line 17), so the if-then-branch in Lines 20-21 will be executed when the `get_buffer` is invoked for the first time, and the buffer *fmtbuf.buffer* will be updated through another call to `xrealloc`. The key point is that the function `OUT_OF_MEM` (invoked when the *result* gets a 0 when invoking `realloc` or `malloc` inside `xrealloc`) does not immediately terminate the program execution, and `OUT_OF_MEM` will continue to allocate another buffer via a second call to `xrealloc` for printing purposes. Then, the memory error will happen when the `malloc` buffer allocation function in the second call to `xrealloc` returns 0 and triggers `OUT_OF_MEM` again: since the *fmtbuf.size* was defined in the global scope and was set to double the needed size in Line 20, the condition of *if-statement* in Line 19 will be *false* when `get_buffer` is called again while the *fmtbuf.buffer* will still keep the original *NULL*. Then, the dereferencing of the pointer *fmtbuf.buffer[need-1]* will lead to the *NULL-pointer dereference* memory error (reported as "*out-of-bound*" error) in Line 23.

Note that we reported both of the errors in Case 1 and Case 2 to the developers and they have confirmed the issues as true bugs [58, 150]. The developer also acknowledged the quality of the report. For example, during the discussion of Case 2 with developers, one developer mentioned:

"There's a nice catch there - where, in that recursive failure, the writing of that terminator overflows a buffer that wasn't actually reallocated yet."

3.5.3 Can existing tools detect the errors?

Although many approaches are proposed to detect such errors, it is still challenging to capture cases involving complex control or data flows in the test program. Static analysis approaches should be able to detect them in theory. However, after we ran three well-known static analysis-based memory detectors, including `cppcheck` [50], clang static analyse [4], and `OCLint` [149], on the same benchmarks[7], the results showed that none of these tools could detect the two errors.

We note that conservative static analysis techniques should have the capability to find the errors in theory by over-approximating all possible execution flows through *loops* and *if* checks across many functions and multiple source files. However, to reduce false positives and make such techniques practical, they usually make certain trade-offs between soundness and completeness in their implementations and have limited supports for path-, flow-, context-, object-, and field-sensitive modeling of the code [10, 92, 205], leading to the missed errors in Case 1 and Case 2.

For example, to detect the *NULL* pointer deference in Case 1 in Line 55, many branching conditions need to be analyzed (e.g., if checks at Lines 12, 15, 16, 17, 26, 33, and 37). Also note that the code example in Figure 3.15 is a simplified version of the original code in order to illustrate the bug-triggering flow more clearly; the actual code involves many more if-checks and function calls across a few thousand lines of code in multiple files (e.g., `fts.c`, `fts-cycle.c`, and `cycle-check.c`); it can be computationally very expensive to track the inter-procedural flows of the FTS pointer across multiple source files in a path-/flow-/context-/object-/field-sensitive way, and the static analysis tools may have certain trade-offs in their implementations, limiting their accuracy in modeling the program semantics and thus missing the detection of such errors in practice.

[7] We tried to run Frama-C and Coccinelle but they failed to detect any of them due to the lack of supports for certain language features in Frama-C or desired error detection patterns in Coccinelle.

Dynamic analysis-based tools are less likely to catch the error as well due to the lack of concrete inputs to run the test programs and the low possibility that the function `malloc` returns *NULL* in a runtime environment.

For other symbolic execution-based approaches, as aforementioned in Section 3.1, they are limited by the modeling of dynamically allocated memory objects, thus missing the detection of two errors.

How Does SymLoc Detect the Errors? Since SymLoc supports path constraints encoded with symbolized addresses with concretely mapped symbolic memory operations, SymLoc can explore deeper execution paths and fork more states involving `malloc` at Line 36 in Figure 3.15 and at Line 6 in Figure 3.17. Afterward, two execution states (one holds the value of 0 and another keeps the symbol for the allocated objects for further path exploration) are maintained during symbolic execution. Thanks to the nature of symbolic execution, the specific states where the symbolic memory object contains the *NULL* value can easily run through all the code and reach Line 55 in Figure 3.15 or Line 23 in Figure 3.17, making the successful detection of two *NULL* pointer dereference issues.

In summary, SymLoc is an automatic approach specialized to detect memory errors related to allocation operations, complementary to existing approaches, and a step further to detect more complex memory errors caused by improper pointer operations involving allocated objects.

3.6 Discussion

3.6.1 Comparison with Other Existing Approaches

Many other static memory detectors can be facilitated to detect memory errors, but it may not be possible to evaluate them all. We compare and discuss a few more detectors that are most related to SymLoc.

Comparison with RAM [186]. RAM proposes a relocatable memory addressing

```
1 int main() {
2     int *buff = (int *) malloc (100000 * sizeof (int));
3     for (int i = 0; i < 100000; i++) {
4         buff[i] = i; // memory write via a symbolic address
5         printf(buff[i]); // read via a symbolic address
6     }
7     return 0;
8 }
```

Figure 3.19: Example compared with RAM [186]

model that supports symbolic addresses to facilitate more flexible memory merging and splitting and make constraint solving more efficient in symbolic execution. It is possible to extend RAM for memory error detection purposes. However, the extension will encounter certain difficulties to make RAM achieve the same goal as SYMLOC. First, RAM assumes that the actual address of a memory object should not affect the behavior of the program and does not encode address-related constraints into path constraints, meaning RAM is not able to explore more paths than standard KLEE. However, such an assumption is not always held in real-world programs. For example, the case studies (especially for case 2) show the locations of a memory object matters. Therefore, some important errors that rely on symbolic addresses might be missed due to the limited handling of symbolic memory addresses. Second, RAM relies on a constraint solver to resolve symbolic memory addresses, which can be time-consuming. Therefore, even though the errors do not involve a symbolic address, RAM will spend much time detecting certain errors. To demonstrate the advantages of SYMLOC in the second point in terms of performance, we ran SYMLOC and RAM on the following small piece of code shown in Figure 3.19 to confirm whether SYMLOC is prior in terms of performance or not.

The code example simply allocates a buffer *buff* and iteratively writes a value and reads the value from 1 to 100000. We ran this code to compare the efficiency of SYMLOC with RAM. RAM spent 55 seconds finishing the operations while SYMLOC only took 13 seconds. When the situation becomes complicated, e.g., more complex path constraints involving dynamically allocated addresses are involved, the performance downside may be amplified. Overall, SYMLOC could be a complementary

approach to RAM and other symbolic execution techniques, where RAM leverages more flexible symbolic addresses to support faster constraint solving, and SYMLOC aims to facilitate a more comprehensive exploration of pointer-related paths. Therefore, SYMLOC can complement RAM to facilitate a more comprehensive exploration of pointer-related paths.

Comparison with MEMSIGHT [49]. MEMSIGHT is an approach that models symbolic memory addresses that reduce the need for concretization. There are two major differences in the memory model design between MEMSIGHT and SYMLOC. First, MEMSIGHT and SYMLOC target to address different problems. MEMSIGHT addresses the problem of how to write or read a memory object that the pointer points to is symbolic. We recognize that this problem is challenging to resolve and MEMSIGHT provides a new memory modeling solution to this end. In contrast, SYMLOC focuses on memory error detection, and our concretely mapped symbolic memory model is for avoiding unnecessary state forking when a pointer is a symbol and continues the execution to facilitate path exploration and error detection. Second, from the implementation perspective, although the newly designed memory model in MEMSIGHT can have better capabilities in theory than ours, MEMSIGHT-KLEE fails to integrate its all features into KLEE due to the fundamental design of KLEE: KLEE utilizes Array Bit Vector (ABV) and MEMSIGHT leverages Bit Vector (BV) while transferring ABV to BV is practically impossible[8]. Limited by the above facts, MEMSIGHT-KLEE implements a variant of KLEE that does not alter the address modeling (i.e., it will not explore additional execution paths even though the symbolic addresses are used under conditions) and only optimizes the performance of KLEE's execution. In other words, MEMSIGHT-KLEE inherits one of the drawbacks of the memory model adopted by KLEE: whenever a memory accesses a symbolic object, MEMSIGHT-KLEE will fork a new execution state. To this end, since MEMSIGHT-KLEE can not explore additional paths when compared with the standard KLEE

[8]The authors of MEMSIGHT have explored several implementation solutions but finally failed mainly due to the difficulties in complex intervening inside several KLEE's internals.

[49], MEMSIGHT-KLEE uses the speedup to evaluate the effect on how MEMSIGHT-KLEE can improve KLEE in terms of performance. Therefore, SYMLOC and MEMSIGHT-KLEE are comparable only in terms of performance.

For a fair comparison, we use the reported speedup numbers in the paper MEMSIGHT-KLEE and run SYMLOC with the same setting. In particular, we used a fixed number of target instructions to be executed in both KLEE and SYMLOC counted the speedups based on the common benchmarks used in MEMSIGHT-KLEE and SYMLOC. Figure 3.20 presents the comparison results. We can see SYMLOC holds comparable speedups in the first three benchmarks. In the last two benchmarks, SYMLOC performs slower symbolic execution than MEMSIGHT-KLEE mainly because SYMLOC spent more time on solving complex path conditions involved.

Curious readers may still be concerned about the fundamental limitations that prevent us from employing this symbolic memory address in MEMSIGHT to reason about the memory errors and detect similar errors as can be identified by SYMLOC. Unfortunately, even though users choose to run the version of MEMSIGHT-Angr (i.e., the one combining MEMSIGHT with Angr [173]), MEMSIGHT-Angr still cannot detect the same errors reported by SYMLOC. This is because MEMSIGHT-Angr does not support symbolic modeling of constraints involving pointers from the heap due to the lack of implementation and thus loses the opportunities to explore more execution paths when the branch conditions involve heap pointers. In other words, users need to add the same kind of implementation code as SYMLOC into MEMSIGHT-Angr to support heap address modeling, such as the symbolization of heap addresses and circumvent the challenges of symbolic read/write, for more comprehensive path exploration. It is worth noting that such an implementation requires considerable engineering efforts as it involves a deep understanding of the architecture and complicated details (e.g., the heap and memory management) of Angr. Furthermore, Angr itself does not support bug detection capabilities, leaving

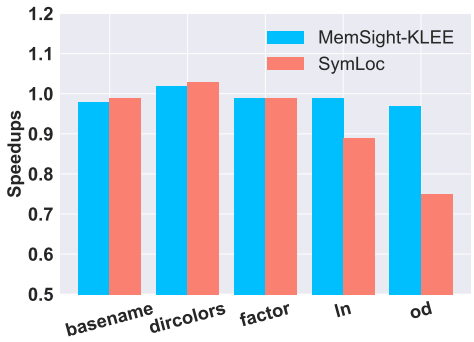


Figure 3.20: Speedups comparison between MEMSIGHT-KLEE and SYMLOC

it to the expert users to design and implement bug checking conditions[9],[10]. Thus, even if the support of constraints involving symbolic heap pointers were added, we still need to add extra implementation like SYMLOC to reason about and detect the same kinds of memory errors. We leave such an engineering-intensive extension of MEMSIGHT for future work.

We would like to emphasize the essential differences in the design goal/contributions between MEMSIGHT and SYMLOC. MEMSIGHT's principal contribution is a new memory model for symbolic execution engines such as KLEE and Angr to improve path exploration (involving stack symbolic pointers), but it has no optimizations to existing symbolic execution engines in terms of other important capabilities, e.g., read/write via symbolic heap pointers, or reasoning about memory errors or bug detection. In contrast, the new memory model is only a part of the contributions of SYMLOC, and SYMLOC is also a new memory error detection system that detects many memory errors, including the ones that can be detected by existing engines (e.g., buffer overflow errors) or others that are hard for existing engines to detect (e.g., use-after-free errors). Due to the different design goals and implementation details, neither MEMSIGHT-KLEE nor MEMSIGHT-Angr can detect the same errors reported by SYMLOC, unless users add the same implementation as SYMLOC.

Comparison with CRED [205]. CRED is a pointer analysis-based static UAF detector that aims to address the challenge of reasoning about the exponential number

[9] https://docs.angr.io/en/latest/faq.html#how-do-i-find-bugs-using-angr
[10] https://github.com/angr/angr/issues/1536#issuecomment-487047190

of program paths to find bugs at a low false positive rate. However, as a static analysis approach, as mentioned in Section 5.5 of that paper [205], CRED suffers from both false negatives and false positives. False negatives are due to limitations on handling loops, a linked list, and an array access alias. False positives are due to its imprecise path reduction and imprecise points-to information for out-of-budget points-to queries. Restricted by the above limitations, as reported by the evaluations of 10 real-world programs, CRED reports 85 bugs but 47 of them are false positives, with a false positive rate of 55.3%. However, existing studies [15, 136] show false positives do matter and a common industrial requirement of false positive rate is less than 30%. Beyond that number, true bugs are lost in false, and developers will discard such an approach for industrial uses. In contrast, SYMLOC inherits the advantages from symbolic execution techniques and can be more accurate. In addition, SYMLOC can also provide more precise inputs that can help trigger the detected errors, which can facilitate the debugging and fixing of such errors.

Since CRED is not open-sourced, we have contacted the authors of CRED multiple times and we haven't received any responses yet. Therefore, we cannot run CRED natively to conduct further fair comparisons. To understand whether SYMLOC detects any bugs that are not supported by previous static analyzers, we have conducted the experiments over JTS benchmarks, and the results presented in Figure 9 and 10 show that existing static analyzer (i.e., Frama-C or Coccinelle) miss the detection of certain UAF and DoF bugs due to limited inter-procedural analysis or error detection patterns. For CRED, the approach only targeted UAF bug detection, it cannot detect other types of bugs (e.g., buffer overflows) that can be detected by SYMLOC. Even for UAF bug detection, as mentioned by CRED authors, CRED suffers from both false positives (due to imprecise path reduction and imprecise points-to information for out-of-budget points-to queries) and false negatives (due to unsound modeling of loops during analysis, i.e., CRED chooses to handle only two iterations for a loop). Taking the case study one as an example, the UAF bug is triggered inside an infinite while loop (Line 5 in Figure15). Due to the unsound modeling of loop, CRED

has a high possibility of missing the detection of the bug inside the function call "`fts_read`" that is invoked in the while-loop (note that the number of times of the loop iterations to trigger the UAF bug depends on users' input). In contrast, SYMLOC, as a dynamic symbolic execution tool, excels at providing precise information (e.g., the concrete test inputs) to help debug the error once it was detected, with the result aligning the common industrial requirement of 30% false positives [15, 136]. For loop handling in case study one, SYMLOC follows the test program's semantics to naturally explore the while-loop without sacrificing the loop modeling, making SYMLOC capable of detecting this UAF bug.

We have also tried to fix the LLVM runtime errors, but the code implementation required for fixing the errors is intricate and demanding. For example, there is a list of unsupported LLVM instructions maintained by KLEE[11], and we have encountered the unsupported instruction "llvm.x86.sse2.packuswb.128" when running "ghostscript". KLEE's developers could not support it for more than three years due to implementation difficulties; supporting it requires a good amount of engineering work based on developers' feedback[12]; it is even harder for us to change the LLVM implementation in a few months. We also acknowledge that symbolic execution techniques have their limitations such as path exploration, which requires future improvements. We are actively working on combining normal and heap address symbolization to alleviate the problem in future work.

3.6.2 Integration with Other Techniques

It can be interesting to integrate various addressing models of RAM [186], *symsize* [190], and SYMLOC together into symbolic execution. RAM's relocatable addressing model improves the ability of symbolic pointer resolution and reduces the cost of solving array theory constraints with big arrays. *symsize* leverages a bounded symbolic-size model that symbolizes the size of allocated objects. SYM-

[11] https://github.com/klee/klee/issues/678#issue-235902374
[12] https://github.com/klee/klee/issues/1154#issuecomment-531295026

LOC considers program behaviors and execution paths that may be affected by memory addresses. Integrating the above three addressing models could have a more *complete* and *efficient* model for symbolic execution.

3.6.3 Threats to Validity

One threat lies in the address symbolization strategy designed in SYMLOC. Ideally, a test program may invoke multiple `malloc` functions. When a user selects the full symbolization option, it may produce complex constraints that could significantly slow down the execution. Although we did not design an extra constraint reduction solution to such a situation, we have designed a selective option with a new API (i.e., `klee_make_malloc_symbolic`) which allows users to identify interesting allocation (e.g., those addresses are extensively used in the comparison within a *if* condition) first. Such a strategy could potentially help release some pressure on the solver side. Another threat comes from the test programs. We used selected utilities in GNU `Coreutils`, two larger benchmarks, and JTS. Although they have been widely used for evaluating symbolic execution [28, 97, 120, 153, 186, 190] and temporal memory error detection [93], these programs may not be representative enough for various software systems. We are considering expanding the program sets in our future work.

3.6.4 Limitations of SYMLOC

SYMLOC has an implementation limitation. Different allocation functions (e.g., `malloc`, `calloc`, and `realloc`) are supported in C/C++ programming languages, and SYMLOC so far only supports the symbolization of addresses returned by `malloc`. However, the `malloc` is the basic function used for dynamic memory allocation in the program. We consider adding support for other allocation functions into the extended version of SYMLOC. Although SYMLOC provides more complete modeling of dynamically allocated memory objects, due to the sophisticated mech-

anism of heap management, SYMLOC still has some modeling limitations. First, SYMLOC does not support symbolic offset and symbolic size. Symbolic offset is designed in Mayhem [31] while the symbolic size is supported in *symsize* [190]. We plan to integrate them into SYMLOC. Second, when ITE involves two different addresses, SYMLOC is not able to provide the required addresses at run-time.

We also recognize that SYMLOC has a limitation on allocating heap buffers to the required concrete addresses that satisfy different path constraints during symbolic execution. That means, if SYMLOC forks two states that contain two different addresses (e.g., 0 and 0xffff543400) separately for a symbolic pointer, SYMLOC does not provide a new heap buffer that is located at the concrete address 0 or 0xffff543400 for the pointer in each execution state during symbolic execution. Instead,SYMLOC simply uses the buffer previously allocated for the pointer (if any) maintains the constraints for the symbolic address, and continues the exploration of the paths. We plan to add a heap simulator [119] in SYMLOC to simulate the run-time behavior of the heap allocator in future work to overcome such a limitation.

3.7 Summary

This chapter presents SYMLOC using concretely mapped symbolic memory locations to facilitate the detection of memory errors. A new integration of three techniques is designed in SYMLOC: (1) the symbolization of addresses and encoding of the symbolic addresses into path constraints, (2) the symbolic memory read/write operations using a symbolic-concrete memory map, and (3) the automatic tracking of the use of symbolic memory locations. Our evaluation results show that SYMLOC outperforms various state-of-the-art memory detectors in terms of detecting memory errors involving allocated memory addresses. We also discuss some interesting errors detected by SYMLOC but missed by other tools and their implications. We also provided a replication package for SYMLOC and reported some interesting findings to lead further research.

Chapter 4

Faster Path Exploration via Reducing Redundant Bound Checking of Type-Safe Pointers

4.1 Objective

Symbolic execution (SE) is a prominent technique that has been applied in many areas, such as software engineering [12, 28, 97], programming language [112, 117, 160], and security [9, 31, 42, 211]. The key idea of SE is to simulate program executions by using symbolic values for inputs and then each execution path will be encoded as path constraints during execution. A constraint solver (e.g., STP [178] or Z3 [214]) later is used to determine the feasibility of each path constraint, and the solved paths are further explored. Among all the existing SE engines, the IR-based ones (e.g, KLEE [28], S2E [42], or Angr [173]), whose execution is conducted by interpreting the Intermediate Representations (IR) of the target program under test, are prevalent and widely used. Typically, traditional IR-based SE first transforms the program (either source code or binary) under analysis into IR, and then it interprets the IR to execute the program, which follows the design in routes ① → ② → ③ in Figure 4.1. Such a design can have manifest benefits from the

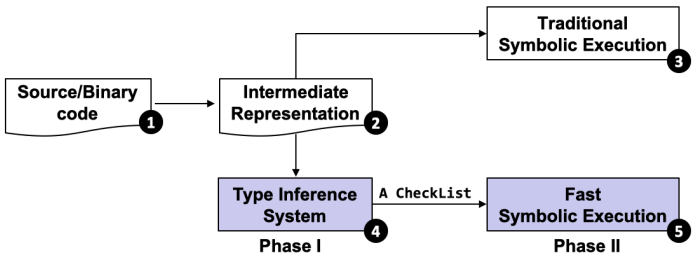


Figure 4.1: Overview of IR-based *Traditional* SE and *Fast* SE

implementation perspective. Instead of interpreting numerous instruction sets of popular CPU architectures, IRs typically represent program behavior at a high level with fewer instructions, thus making it easier to implement an instruction interpreter for architecture-independent instruction sets than manipulating complex instructions.

Although the design of IR-based SE is convenient in terms of implementation, existing studies [160, 161] point out that such a design carries out significant performance downsides and slows down the execution. To mitigate such a problem, two major flavors of studies are devoted to accelerating SE. Based on the fact that IR code can be transformed by aggressive optimizations, the first promote directions are dedicated to either selecting existing compiler optimizations [35, 65] that could help accelerate SE or designing a stand-alone customized optimization [196] for program verification (e.g., symbolic execution). Apart from the IR optimization side, other solutions such as reducing the number of paths to be explored [24, 188] or optimizing the constraint solving [108, 158] are considered for speeding up the execution process. However, most of the existing studies neglect the internal interpretation downside of IR-based SE in terms of performance. That is, IR code interpretation alone could slow down the performance of SE engines.

In this chapter, we propose FASTKLEE, a tool that aims to support faster SE by reducing the interpretation overheads of redundant bound checking of type-safe pointers. We design FASTKLEE based on the following two intuitions. First, the number of interpreted instructions tends to be stupendous (usually billions only in one hour's run), and reducing overhead for the most frequently interpreted ones (i.e., read/write) could potentially accelerate the execution. Second, type-safe is

an important property in the program for preventing certain errors such as memory out-of-bound assessing. As proven by prior studies [104, 140, 145], a large portion of pointers in C programs to be read/written can be statically verified to be type-safe. However, most of the existing IR-based SE engines treat all the pointers (memory addresses) equally and perform the bound checking for every pointer when interpreting read/write instruction. Thus, a plethora number of bound checking performed during the interpretation is unnecessary, which may induce performance downsides. Based on the above intuitions, we propose FASTKLEE to support faster SE. Specifically, in FASTKLEE, a type inference system is leveraged to perform type inference before executing the SE engine (which follows the design in routes ① → ② → ④ → ⑤ in Figure 4.1, where the latter two are designed for faster SE). The type inference system (i.e., ④) is first used to statically verify pointers to be *safe* or *unsafe* and produce a checking list `CheckList` for *unsafe* pointers. Then, during the execution in ⑤, a customized memory operation is designed to perform bound checking only for *unsafe* pointers (stored in `CheckList`), while *safe* pointers are no longer needed to perform any redundant checking. In this way, FASTKLEE could reduce the interpretation overheads of redundant bound checking of type-safe pointers for faster SE.

We implement FASTKLEE on top of the well-known SE engine KLEE [28] and the type inference system CCured [145]. To demonstrate the effectiveness of FASTKLEE, we compare it with the state-of-the-art approach KLEE over widely-adopted GNU `Coreutils` datasets. The evaluation results demonstrate that FASTKLEE is able to reduce by up to 9.1% (5.6% on average) as KLEE in terms of the time to explore the same number (i.e., 10k) of execution paths.

In summary, this chapter makes the following contributions:

- To our best knowledge, FASTKLEE is the first SE engine that aims to accelerate IR-based SE by reducing interpretation overheads.
- We leverage a type inference system to classify pointers and design a customized memory operation in the SE engine to avoid redundant checking of type-safe

pointers, thus facilitating the reduction of interpretation overheads.

- We open-source the tool FASTKLEE[1] and demonstrate its usability and effectiveness. We also discuss several important implications of FASTKLEE, such as facilitating valuable path exploration.

4.2 Usage Example

Users can execute "./setup.sh" in the code repository to set up FASTKLEE. To prepare the test programs under test, it is recommendable for users to follow the official instruction [61] to get the LLVM bitcode files of test programs to be analyzed (e.g., `cat.bc` utility in GNU `Coreutils`). After setting up the tool and the test program, the following two major phases are considered to use FASTKLEE through a command-line interface.

4.2.1 Phase I: Perform Type Inference

In the first phase, the target test program `cat-linked.bc` under testing will be first instrumented to be a new bitcode file named `cat-linked.bc` by invoking LLVM tool-chain `llvm-link`. Then, the `ccured` pass will be applied on the `cat-linked.bc` by invoking the tool `opt`. After the type inference of interpreted pointers, a text file `cat-checklist.txt` will be produced in the current folder, which will be used later in the next phase.

```
$llvm-link cat.bc neschecklib.bc -o cat-linked.bc
$opt -load libccured.so -nescheck -stats -time-passes < cat-linked.bc >& /dev/null
```

4.2.2 Phase II: Conduct Faster Symbolic Execution

In the second phase, users can utilize the same command line with the original KLEE to perform SE upon the test program. Specifically, users may follow the official document [62] to opt for the applicable `options`. During the running of

[1] https://github.com/haoxintu/FastKLEE

FASTKLEE, the file `cat-checklist.txt` will be loaded first and then will be used to guide the customized memory operation in FASTKLEE.

```
$fastklee [options] ./cat.bc --sym-args 0 1 10 --sym-args 0 2 2 --sym-files 1 8 --sym-stdout
```

The most related work to us can be broadly divided into two categories: compiler optimization-based and compiler optimization agnostic. In the former, Dong *et al.* [65] study the influence of standard compiler optimizations on SE, and Chen *et al.* [35] further leverage machine-learning-based compiler optimization tuning to select a set of optimizations to accelerate SE. Jonas *et al.* [196] later design a stand-alone optimization for optimizing programs for fast verification, which includes accelerating SE. In the latter, different approaches are proposed to reduce the number of paths to be explored [28, 117, 120, 188] or optimize the constraint solving [28, 158, 186] in SE, thus speeding up the execution process.

Unlike the existing approaches, our goal is to make IR-based SE more efficient by reducing the internal interpretation overheads, i.e., we aim to reduce redundant bound checking of type-safe pointers during IR code interpretation. It is worth noting that our approach can be complementary to existing approaches and further boost faster SE by combining them with our proposed approach.

4.3 Design of FASTKLEE

In this section, we present the detailed design of FASTKLEE. As shown in routes ① → ② → ④ → ⑤ in Figure 4.1, after obtaining the IR code, FASTKLEE first leverages a type inference system in ④ to classify different kinds of pointers and store the *unsafe* pointers in a checking list (i.e., `CheckList`). Then, the `CheckList` is passed to ⑤ and will be used in the customized memory operation in FASTKLEE. More specifically, if a pointer under read/write operation is in the checking list, a normal bound checking is conducted. Otherwise, FASTKLEE omits the bound checking and continues to the interpretation. In short, the omitted portion of bound checking is the key weapon inside FASTKLEE to make faster SE.

Algorithm 2: Type Inference System in FASTKLEE

Input: a IR code file `bc`
Output: a checking list of *unsafe* pointers `CheckList`

```
1  Function typeInferenceFunc(bc):
2      CheckList ⟵ ∅ // initialize a checking list
3      Instruction *i = processInst(bc)
4      switch i->getOpcode() do
5          ... // other instructions
6          case Instruction::Load do
7              ptr = classifyPointer(i)
8              if ptr.type != SAFE then
9                  key = generateKey(ptr, i)
10                 CheckList.append(key)
11         case Instruction::Store do
12             ptr = classifyPointer(i)
13             if ptr.type != SAFE then
14                 key = generateKey(ptr, i)
15                 CheckList.append(key)
16         ... // other instructions
17     return CheckList
```

4.3.1 Type Inference System

The type inference system in FASTKLEE is designed for classifying pointer types. Typically, pointer types in a type inference system are in the following three forms: (1) *SAFE* pointer can only be null and only needs a null-pointer check at runtime; (2) *SEQ* pointer can be null, be interpreted as an integer, or be manipulated via pointer arithmetic. At runtime, it needs a null-pointer and bounds check; (3) *WILD* pointer cannot be statically typed, and it needs null-pointer, bounds, and dynamic type checks at runtime to decide.

Since most IR-based SE engines lack runtime information during execution, i.e., programs are not executed like native runs, we categorize pointer types only into *safe* and *unsafe* (i.e., SEQ and WILD) in FASTKLEE. Specifically, the *safe* pointers can be statically verified to be type-safe so they do not need bound checking during interpretation, while only the *unsafe* pointers are needed to be bound-checked. Therefore, the functionality of the type inference system in FASTKLEE is, given an

IR code of the program under test, it records the *unsafe* pointers when the pointer is performed in the read/write operations. When all the instructions are inferred, a checking list `CheckList` that stores all the *unsafe* pointers are returned. Such a checking list will be the guidance for reducing redundant bound checking of type-safe pointers in FASTKLEE.

Algorithm 2 presents the detail of the type inference system introduced in FASTKLEE. The function `typeInferenceFunc` is responsible for classifying different types of pointers. It takes an IR code file `bc` as input and outputs a checking list `CheckList`. Inside the function, it first initializes the `CheckList` in Line 2 and accordingly processes instructions in the `bc` file in Line 2. Then, when encountering a read (Line 6) or write (Line 11) instruction, the pointer (i.e., the memory address under read/write) is classified by invoking the function `classifyPointer` (in Line 7 or 12). Later, an *if-branch* is performed to check whether the `ptr` under handling is an *unsafe* pointer (in Line 8 or 13). If the answer is yes, a `key` that represents a unique pointer is generated by calling the function `generateKey` and will be stored in the `CheckList` later (in Lines 9-10 or Lines 14-15). Finally, the checking list is returned in Line 17 and will be used in the customized memory operation in FASTKLEE.

4.3.2 Customized Memory Operation

The purpose of the customized memory operation in FASTKLEE is to take the output (i.e., `CheckList`) from the type inference system and use it to guide the customized memory operation during interpretation. Inside the operation, first, a Boolean variable `inBound` is initialized and the `key` is retrieved by calling the function `generateKey` in when a read/write instructions are interpreted. Then, a checking of whether the `key` is in the `CheckList` is performed to process either normal checking in traditional execution if the checking returns *true* or assignment of `inBound` to 1 if the checking returns *false*. Later, the execution same as the one

in symbolic execution continues.

By equipping the type inference system and the customized memory operation, FASTKLEE is capable of reducing the interpretation overheads of redundant bound checking of type-safe pointers.

4.3.3 Implementation

We implemented FASTKLEE on top of KLEE (version 2.1) and combined it with the well-known CCured type inference system [145]. Specifically, for the CCured system, we implement it on top of DataGuard [103]. In particular, due to the unmapped IR information between analysis and original IR (refers to an issue [55] for more details), we first use the debug information (i.e., file name, function name, the line number of instruction, and column number of instruction) inside the instruction to represent a unique `key` (implemented in Algorithm 2). Then, we reuse the APIs from DataGuard to record/store the *unsafe* pointers into a checking list `CheckList`. For the implementation of the customized memory operation in FASTKLEE, we modified KLEE's memory operation API to leverage the information (i.e., `CheckList`) from type inference analysis, and decide whether the bound checking is needed.

4.4 Evaluation

4.4.1 Experimental Setup

Benchmark. We evaluate FASTKLEE on the widely used GNU `Coreutils` (version 9.0) benchmarks. Specifically, we select 40 programs in it, and the excluded utilities can be categorized into the following types: (1) cause non-deterministic behaviors (e.g., `kill`, `ptx`, and `yes`), following existing studies [97, 120], (2) exit early due to the unsupported assembly code or external function call, and (3) can not successfully explore 10k execution paths in 2 hours (i.e., the timeout to run each test program we set in the experiment).

Approach Under Comparison. We adopted the notable SE engine KLEE as our baseline, as we built on top of it.

Running Settings. We followed prior work [28, 120] to set symbolic inputs for GNU `Coreutils` programs. Besides, we use *Breadth First Search* to deterministically guide the path exploration in KLEE and FASTKLEE. The experiments are conducted on a Linux PC with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz x 12 processors and 64GB RAM running Ubuntu 18.04 operating system.

4.4.2 Evaluation Results

To demonstrate the effectiveness of FASTKLEE, we run FASTKLEE against KLEE in terms of the time spent on exploring the same number of explored paths over GNU `Coreutils`. We set a timeout of 2 hours to run each program and count the time spent on exploring certain execution paths. Specifically, we follow the formula below to calculate the speedups:

$$\frac{T_{baseline} - T_A}{T_{baseline}} \times 100$$

where $T_{baseline}$ represents the time spent by the baseline KLEE on exploring 10k paths and T_A describes the time spent by our proposed FASTKLEE on exploring the same number of paths.

The scatter plot in Figure 4.2 shows the distribution of the speedups achieved by FASTKLEE. The labels under the *x-axis* correspond to the 40 utilities used in our evaluation, and the values on the *y-axis* represent the number of speedups by percentage. We can observe that most of the points in the scatter plot fall from 5% to 6%, and the highest point is up to 9.1%. Further, in Figure 4.3a, we present the box plot depicting the distribution of the percentage number of the speedups achieved by FASTKLEE in detail over 40 test programs. We could confirm that for the majority of the packages, the number of speedups ranges within [4.8%, 6.2%]. We also calculate the average of the speedups, which goes to 5.6% over those test

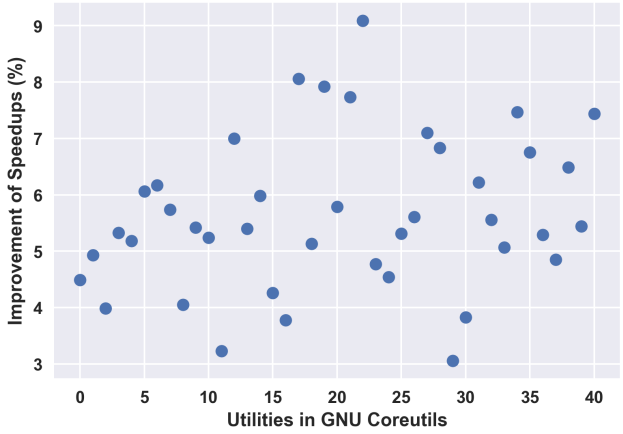


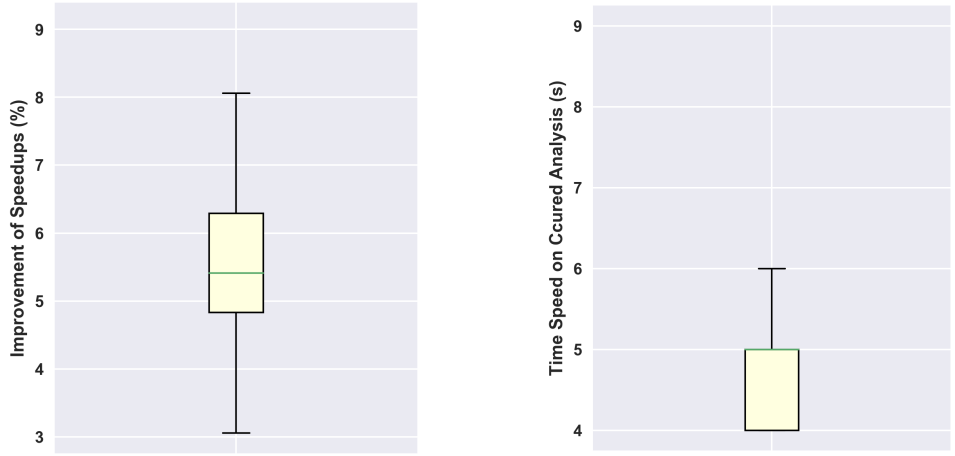
Figure 4.2: Scatter plot of the improvement in speedups

programs. Interestingly, from Figure 4.2, it seems there is no discernible trend for the destruction of the speedups. This is because the benefits gained under FastKLEE may differ based on different test programs. Specifically, if a test program extensively checks *safe* pointers during execution, FastKLEE would be able to significantly reduce its checking overhead and greatly speed up the execution.

To further understand the overhead reduction achieved by FastKLEE, we also evaluate the time used for the type inference in FastKLEE. Figure 4.3b describes the trend of the time spent on CCured analysis over 40 test programs. We can see that the time speed of the analysis is within the range of [4.0s, 5.0s], which can be neglected compared with the whole time (usually taking hundreds or thousands of seconds to explore the 10k execution paths).

4.5 Discussion

Potential Implications of FastKLEE. Although we only show the performance benefits of FastKLEE in this chapter, FastKLEE can have other important implications for path exploration in SE. For example, users can extend FastKLEE to assist SE to explore only valuable execution paths due to the time limit or path explosion challenge. The valuable paths can have at least two important forms. First, a path exploration strategy in FastKLEE can be guided by the results of type



(a) Box plot of the improvement in speedups (b) Box plot of the time spent on type inference

Figure 4.3: Improvement of speedups and time spent on type inference

inferences, meaning the paths that involve more *unsafe* pointers are more valuable (i.e., more likely to be buggy). That is, an *unsafe*-pointer-guided path exploration can be applied to explore valuable execution paths. Second, instead of targeting exploring multiple buggy execution paths, one may further leverage type inference results to explore the most valuable buggy path, i.e., paths that are more likely to be exploitable. Both the above implications can help improve the reliability and security of software systems. We leave this as our future work.

Threats to Validity. One threat lies in the implementation of FASTKLEE. We only implement FASTKLEE on a source code-based symbolic executor. Since it is feasible to introduce a type inference system into binary code [25], we consider extending the support for other SE engines (e.g., Angr [173]) into FASTKLEE in the future. Another threat comes from the test programs. We only used selected utilities in GNU `Coreutils`, and these programs may not be representative enough for various software systems. However, those test programs have been widely used for evaluating SE engines [28, 97, 120, 186, 190], and we consider expanding the program sets for more extensive evaluation in the future.

Limitation. FASTKLEE has a limitation in terms of implementation, i.e., we only implement FASTKLEE on top of a source code-based SE engine KLEE. Other binary

code-based SE executors such as Angr [173] are not supported yet for the time we submit the paper. We plan to add the above support into FASTKLEE in future work.

4.6 Summary

We present FASTKLEE, a tool that aims to reduce the interpretation overheads of redundant bound checking of type-safe pointers for faster symbolic execution. In FASTKLEE, a type inference system is first leveraged to classify pointer types before interpretation. Then, a customized memory operation is designed to perform bound checking only for *unsafe* pointers during interpretation. Evaluation results demonstrate that FASTKLEE outperforms the state-of-the-art KLEE in terms of the time spent on exploring the same number of execution paths. For future work, we are actively pursuing to (1) extend FASTKLEE to support more SE engines and (2) leverage the abilities in FASTKLEE to facilitate more effective path exploration, such as vulnerability-oriented path search.

Chapter 5

Vulnerability-Oriented Path Exploration via Unsafe-Pointer Guided Monte Carlo Tree Search

5.1 Objective

Memory unsafety is the leading cause of vulnerabilities in complex software systems. Based on a report from Microsoft, more than 70% of security bugs have been memory safety problems in the last 12 years [45]. To prevent memory errors, many advanced static/dynamic/symbolic analysis-based approaches are devoted to detecting such errors automatically. Among them, *symbolic execution* is considered one of the promising program analysis techniques [28, 173]. The key idea of symbolic execution is to simulate program executions with symbolic inputs and generate test cases by solving path constraints collected during execution. Taking advantage of the soundness of test case generation, symbolic execution has also been applied in many other areas, such as software engineering [12, 28, 97], programming language [112, 117, 160], and security [9, 31, 211].

One of the key challenges in symbolic execution is path explosion, where even a small program can produce a vast execution tree [12]. Two main alternatives

have been proposed to alleviate this challenge. The first alternative is to steer the exploration of the path to maximize the rate at which the new code is covered using various search heuristics. Notably, the symbolic execution KLEE [28] supports random, breadth-first (BFS), depth-first search (DFS), and other coverage-guided heuristics. A recent work CBC [209] proposes to use compatible branch coverage-driven path exploration for symbolic execution. However, existing studies show that achieving the best coverage does not necessarily mean that the largest number of vulnerabilities can be detected [17, 40].

Another solution is to skip the symbolic execution of certain code that is manually labeled as not related to the vulnerability using *chopped symbolic execution* [189]. However, setting up chopped execution requires prior expert knowledge of the program under test and intensive manual effort to decide which functions to skip. For example, to successfully detect a vulnerability (CVE-2015-2806) in the `libtasn1` library, users need to locate four specific functions and two lines[1] to skip. Since the library includes more than 20,000 lines of code, locating these functions/lines may require expert knowledge and involve intensive human efforts. Since we do not know where the vulnerabilities are until they are caught, it is difficult to find new vulnerabilities. Another issue that downgrades the chopped execution is the performance. It consumes significant memory when switching between skipped and normal execution (see more details in Section 5.4.2). Motivated by addressing the above limitations, we aim to investigate the following research question in this chapter: *How to automatically conduct vulnerability-oriented path exploration without relying on prior expert knowledge?*

Conducting vulnerability-oriented path exploration is non-trivial and presents challenges for at least the following two reasons. First, we should decide which indicators can effectively approximate the vulnerability of a path. For example, there are no unified metrics to indicate a path that contains *out-of-bounds* memory errors.

[1]Enabled options used in `Chopper` to detect the vulnerability: "*–skip-functions=__bb0,__bb1,_asn1_str_cat:403/404,asn1_delete_structure*"

Therefore, it is challenging to find a suitable indicator to represent vulnerable paths. Second, it is also difficult to effectively leverage indicators to guide path exploration. To make path exploration more effective, we argue that a promising path search strategy should maintain a good trade-off between exploiting the paths that have already been executed in the past and exploring the paths that will be executed in the future. However, existing search heuristics do not fully take advantage of past execution, which degrades the possibility of exploring the more promising paths, i.e., the more likely vulnerable paths.

In this chapter, we propose VITAL[2], a new **V**ulnerability-or**I**en**T**ed p**A**th exp**L**oration strategy for symbolic execution via type-unsafe pointer-guided Monte Carlo Tree Search (MCTS). Our core insight is that spatial memory safety errors (e.g., *out-of-bounds* memory accessing errors) can *only* happen when dereferencing *type-unsafe* pointers, i.e., pointers that cannot be statically proven to be memory safe [104, 145, 183]. As shown in Figure 5.1 and confirmed in our experiments, we find that an increase in the number of unsafe pointers exercised is directly related to an increase in the number of memory errors detected, with a strong positive Pearson coefficient 0.93. Hence, we suggest maximizing the number of unsafe pointers during path exploration to address the first challenge of approximating vulnerability. To address the second challenge of effective search within the symbolic execution tree, we drive a new pointer-guided MCTS which effectively balances the state exploration and exploitation to maximize the number of unsafe pointer of paths.

We implement VITAL on top of a well-known symbolic executor KLEE [28] and conduct extensive experiments to compare VITAL with existing search strategies and chopped symbolic execution. In the former, we run VITAL against six path exploration strategies in KLEE and a recently proposed strategy CBC [209] on the GNU `Coreutils`, and the results demonstrate that VITAL could cover up to 90.03% more unsafe pointers and detect up to 57.14% more unique memory errors. In the latter, we run VITAL against KLEE [28], `Chopper` [189], and CBC

[2]The name also reflects our aim for exploring *vital* program paths due to path explosion.

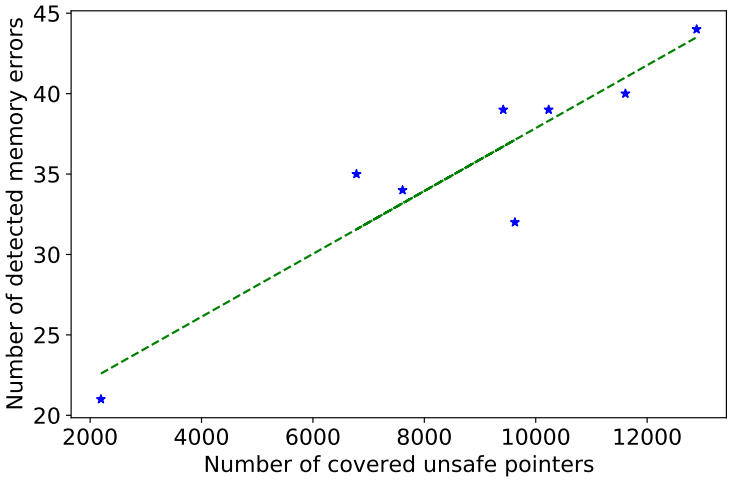


Figure 5.1: Correlation between unsafe pointers and memory errors (Pearson's coefficient [46]: 0.93).

[209] over six known CVE vulnerabilities, and the results show that VITAL could achieve a speedup of up to 30x execution time and a reduction of up to 20x memory consumption to detect all of them automatically without prior expert knowledge.

In summary, this chapter makes the following contributions.

- To our knowledge, VITAL is the first work performing vulnerability-oriented path exploration for symbolic execution towards effective vulnerability detection.
- We suggest a new indicator (i.e., the number of type-unsafe pointers of a path) to approximate the vulnerability (or vulnerability-proneness) of a path and utilize a new unsafe pointer-guided Monte Carlo Tree Search algorithm to navigate vulnerability-oriented path searches.
- Extensive experiments demonstrate the superior performance of VITAL in terms of unsafe pointer coverage and memory errors/vulnerability detection compared to state-of-the-art approaches. VITAL also detected a previously unknown vulnerability, indicating its practical vulnerability detection capability.
- We publish the replication package (including the source code and setup instructions for VITAL, benchmarks, and scripts to reproduce the experiments) of this study available[3] to foster research on advanced vulnerability detection.

[3] https://github.com/haoxintu/Vital

5.2 Preliminaries and Motivating Example

5.2.1 Preliminaries

(1) Symbolic Execution and Path Exploration

Symbolic execution is a program analysis technique that analyzes the test program by feeding the program with symbolic input. During symbolic execution, path constraints are collected, and corresponding test cases will be generated by off-the-shelf constraint solvers (e.g., STP [178] or Z3 [214]). The main activity in symbolic execution is to consistently select a path (or state) to explore and analyze an instruction one time until no state remains or a given timeout is reached. Notably, the widely used KLEE symbolic execution engine, the representative search strategies include BFS, DFS, Random, code coverage-guided (i.e., `nurs:covnew`), and instruction coverage-guided (i.e., `nurs:md2u` and `nurs:icnt`).

(2) Memory Safety and Type Inference

Existing memory safety vulnerabilities fall mainly into two main categories: *spatial* and *temporal* memory safety vulnerabilities [104]. The first ones occur when pointers reference addresses outside the legitimate bounds (e.g., *buffer overflow*), and *temporal* memory safety issues arise from the use of pointers outside of its live period (e.g., *use-after-free*). Previous studies [142, 218] show that serious vulnerabilities caused by violating spatial safety are well known in the community. For example, more than 50% of recent CERT warnings are due to breaches of spatial safety [195]. Furthermore, compared to temporal safety issues, spatial safety vulnerabilities can be exploited by many mature techniques, such as return-oriented programming [163] and code reuse attacks [16, 34]. The above fact indicates that designing advanced solutions to keep spatial memory safe is of critical importance.

To enforce complete spatial safety, the only way is to keep track of the pointer bounds (the lowest and highest valid address to which it can point) [183]. Integrating

a type inference system by static analysis into a type-unsafe programming language (e.g., C) is a well-established strategy for keeping track of pointer bounds [72, 104] to prove memory safety. For example, `CCured` [145] statically infers pointer types into three categories: *SAFE*, Sequence (*SEQ*), and Dynamic (*DYN*). The *SAFE* pointers are the dominant portion of the program that can be proved to be free of memory errors [145]. The *SEQ* pointers require extra bound checking and *DYN* pointers need to perform run-time checks to ensure memory safety. Since spatial memory error can only occur within *SEQ* or *DYN* pointer categories, we refer to *SEQ* or *DYN* pointers as *type-unsafe* or *unsafe* pointers in this chapter. Taking the code from Figure 5.2(a) for example, the pointers `p.y` (Line 9), `p.z` (Line 16), and `p.y` (Line 20) are used in pointer arithmetic, which, in turn, are classified as *unsafe* (*SEQ*) pointers.

(3) Monte Carlo Tree Search (MCTS)

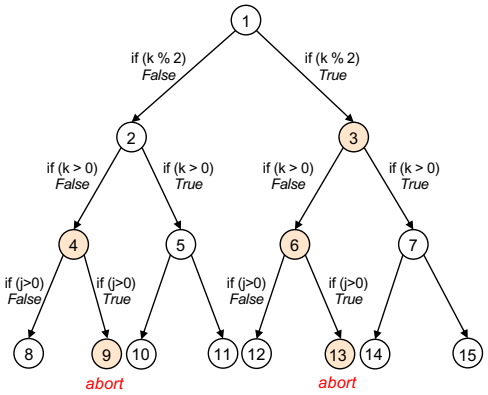
MCTS is a heuristic search algorithm to solve decision-making problems, especially those used in games (e.g., AlphaGo) [21, 126]. The fundamental idea behind MCTS is to use randomness to simulate decision sequences and then to use the results of these simulations to decide the most promising move. It excels at balancing between exploring new moves in the future and exploiting known good moves in the past.

Fundamentallly, the MCTS consists of four steps (see more details in [21]):

- *Selection*: starting from the root node, select successive child nodes that are already in the search tree according to the *tree policy* down to an expandable node that has unvisited children.
- *Expansion*: expand the unvisited node into the search tree.
- *Simulation*: perform a playout from the expanded node governed by a *simulation policy*. The termination of the simulation yields results (i.e., rewards) based on a reward function.
- *Backpropagation*: update the information stored in the nodes on the path from

```
1  struct point {int x, y, z;};
2
3  int main() {
4    struct point p = {0, 0, 0};
5    int j, k;  // j and k are symbols
6
7    f(&p, k);
8    if (j > 0){
9      if (p.y) abort();// unsafe pointer
10   } else
11       ;
12   return 0;
13 }
14 void f(struct point * p, int k){
15     if (k % 2)
16         p->z++; // unsafe pointer
17     if (k > 0)
18         p->x++;
19     else
20         p->y++; // unsafe pointer
21 }
```

(a) Code example



(b) Execution tree of the code example

Figure 5.2: Motivating example (adapted from `Chopper` [189])

the expanded node to the root node with the calculated reward. This also includes updating the visit counts.

Note that the *tree policy* (including node selection and expansion) and *simulation policy* are two key factors in conducting effective MCTS applications, and a large number of enhancements are proposed to facilitate the capabilities of MCTS from different persepctives [21, 182].

5.2.2 Motivating Example

To illustrate the motivation behind VITAL, we will use a simple example shown in Figure 5.2(a) (adapted from `Chopper` [189]) to show the limitations of the existing solution and what the advantage of VITAL. Figure 5.2(b) represents the corresponding process tree, where the arrow denotes the execution flow with branch conditions and each circle represents an execution state. The colored nodes mean that they include unsafe pointers, where the pointers are code blocks that are unique to two forked states. For example, when the executor forks at Line 17, the code in Lines 18 and 19 are two unique code blocks for two forked states. The eight leaf nodes are terminal states, while the others are internal states. Assuming that we are trying to answer the following simple question: *How can we efficiently trigger the problematic* `abort` *failure at Line 9?*

Direction 1: Path Search Heuristics. Two commonly used search strategies are `DFS` and `BFS`, both of which apply a fixed order to explore all program paths in the following ranked order, where the leaf nodes represent the termination states (the "→" represents the order of terminated states):

- **BFS**: (15) → (14) → (13) → (12) → (11) → (10) → (9) → (8)
- **DFS**: (8) → (9) → (10) → (11) → (12) → (13) → (14) → (15)

We can observe that both of them can not explore (or rank) the path containing the `abort` failure at the top-1 position, so they might not be efficient. A random search might first find the `abort` path, but the result is unreliable. Other search heuristics, such as coverage-guided (e.g., CBC [209]), are good at covering new code but are still likely to miss certain errors because achieving the best coverage does not mean that the largest number of bugs will be detected [17].

Direction 2: Chopped Symbolic Execution: An alternative solution to detect failure efficiently is to skip *uninteresting* functions (function `f` in the example) before execution. When the execution goes to the statement that depends on the skipped function (i.e., Line 9), it recovers the execution of function `f` and merges the remaining states to avoid unsound results. However, chopped execution relies heavily on the pre-defined skipped functions, which require prior expert knowledge. Furthermore, the recovery mechanism, which switches between recovering and normal execution, is memory consuming (see more details in Section 5.4.2).

Our Solution: VITAL. Unlike existing solutions, VITAL performs vulnerability-oriented path search. The basic idea behind VITAL is that, before execution, we obtain the type-unsafe pointer locations (e.g., Lines 9, 16, and 20), where the type-unsafe pointers approximate the existence of vulnerabilities (i.e., the `abort` failure in this example). Then, guided by the locations, VITAL will give a higher reward to those states with unsafe pointers, navigating the exploration towards the path where the number of unsafe pointers is maximized, i.e., (1) → (3) → (6) → (13). As a result, VITAL explores the state (13) that triggers the `abort` failure at the top-1 position.

It is worth noting that although the search strategies in **Direction 1 & 2** can

trigger the `abort` failure, due to the large spaces to explore in more realistic path exploration over complex software systems, they are very likely to miss important vulnerabilities due to inefficiency. In contrast, VITAL could effectively explore the most promising paths that are more likely to contain vulnerabilities.

5.3 Design of VITAL

Overview. The general procedure of VITAL is to continuously select a promising state that is more likely to contain vulnerabilities. Technically, VITAL first acquires a set of unsafe pointers by performing pointer type inference over the test program only once at compile-time. After that, guided by unsafe pointers, VITAL corporates a new search strategy (i.e., MCTS) to have an optimal balance between the exploration of future states and the exploitation of past executed states at execution-time. The main technical contribution of VITAL lies in a new symbolic execution engine that implements a new variant of MCTS equipped with the unsafe pointer-guided node expansion and the customized simulation policy.

5.3.1 Acquisition of Unsafe Pointers via Type Inference System

This subsection first justifies why the type-unsafe pointer is a reasonable indicator to approximate unknown memory safety vulnerability and then articulates how VITAL collects them.

Why Type-Unsafe Pointers?

The main root cause of memory safety problems in C/C++ is due to sacrificing type safety for flexibility and performance in the early design choice in the 1970s [145]. As mentioned in Section 5.2.1, to ensure spatial memory safety, it is essential to track specific properties (e.g., size and types) of the memory area to which the pointer refers. A type inference system is a well-established static analysis technique that was used to keep spatial memory safe in the literature [72, 184]. For example,

`CCured` [145] keeps spatial memory safe by classifying pointer types based on the usage of the pointers, while *SAFE* pointers that are free of memory errors can be soundly determined at compile time. For others (i.e., *SEQ* or *DYN*), memory safety must be ensured at run-time, which requires the insertion of safety checks at runtime.

Since there are no unified indicators to approximate unknown memory safety vulnerabilities, we suggest leveraging pointer types (i.e., *SEQ* and *DYN*) that can not be statically verified to be free of memory errors as *unsafe* pointers and use them to approximate vulnerable behaviors. We assume that if a program path contains more unsafe pointers, the path is more risky to contain vulnerabilities (the results shown in Figure 5.1 also support our claim).

How Does VITAL Acquire Type-Unsafe Pointers?

We follow an existing type inference algorithm `CCured` [104] to classify pointer types according to the following three rules:

1. All pointers are classified as *SAFE* upon their declaration.
2. *SAFE* pointers that are subsequently used in pointer arithmetic are re-classified as *SEQ*.
3. *SAFE* or *SEQ* pointers that are interpreted with different types are re-classified as *DYN* (e.g., casting from `int**` to `int*`).

The above design helps yield conservatively overestimated unsafe pointers, meaning that there are no pointers classified as *SAFE* but used in an *unsafe* way [140, 145]. In other words, VITAL may potentially identify *non-DYN* pointers as *DYN* pointers, but never misclassify *DYN* pointers as *non-DYN*.

5.3.2 Type-unsafe Pointer-guided Monte Carlo Tree Search

The goal of a search strategy in symbolic execution is to select a interesting state to execute next. Algorithm 3 shows the overall selection strategy designed in VITAL. It takes a set of unsafe pointers *unsafeSet* acquired from the previous step and the

Algorithm 3: Unsafe Pointer-guided MCTS State Selection

```
   Input: unsafe pointer set unsafeSet, a current state cur_state
   Output: an execution state to be executed next n_state
1  Function MCTSSearch::selectState():
2      ExecutionState n_state
3      while (! isTerminal(cur_state->node)) do
4          if (! hasEligibleChildren(cur_state->node)) then
5              n_state = doSelection(cur_state->node)
6              cur_state = n_state
7          else
8              n_state = doExpansion(cur_state->node)
9              if (! isWorthSimu(cur_state->node)) then
10                 reward=doSimulation(n_state->node,unsafeSet)
11                 doBackpropagation(reward, n_state->node)
12             break
13     return n_state
```

current execution state being executed *cur_state*, and outputs the expected state to be executed next. There are many tree search algorithms proposed in the literature, such as Greedy Best-First Search [100], Bidirectional Search [179], Uniform Cost Search [44], and MCTS [21]. We chose MCTS in this study because it excels at maintaining a good trade-off between exploring new states in the future and exploiting known states in the past. The overall workflow of Algorithm 3 customizes the standard MCTS algorithm with a few key steps guided by the unsafe pointer set. Before diving into the details of the algorithm, we want to clarify that we use the process tree (internal data structure supported in KLEE [28]) as the symbolic execution tree to be used for MCTS. Since every node represents an execution state in the process tree, we will use the term node and state interchangeably in the following sections.

The algorithm starts by performing a *while-loop* to check if the current state *cur_state* is a terminal state or not (Line 3). If not, it checks whether the current state has eligible (i.e., expandable tree node) children via function `hasEligibleChildren` (Line 4). The result will be either going back to the *while-loop* after performing node selection via `doSelection` (Algorithm 4) in the *if-true* branch (Line 5) or calling `doExpansion` (Algorithm 5) in the *if-false*

Algorithm 4: Procedure of Tree Node Selection in VITAL

```
   Input: a tree node node (with attributes such as isInTree)
   Output: a leaf node of node to be selected selected_node
1  Function doSelection(node):
2      if node->left->isInTree && node->right->isInTree then
3          selected_node = selectBestChild(node)
4      if node->left->isInTree && ! node->right->isInTree then
5          selected_node = node->left
6      if ! node->left->isInTree && node->right->isInTree then
7          selected_node = node->right
8      return selected_node
9  Function selectBestChild(node):
       /* return the node with the highest UCT        */
```

branch (starting at Line 8). After the node expansion, it simulates the expanded node. It gets a reward by invoking `doSimulation` (Algorithm 6 at Line 10) if the function `isWorthSimu` return *true* (Line 9). A node is worth simulating when it is never simulated or does not reach the simulation limit to avoid an infinite loop. Later, the reward is backpropagated through `doBackpropagation` function to all the parent nodes until the root. Finally, the expanded state is returned as normal (Line 13). It is worth noting that some key steps, such as node expansion and simulation, take actions based on the unsafe pointer set as one of the function parameters, where the record of unsafe pointers is essential to guide the smart search process in MCTS. We will explain each step in the following.

Tree Node Selection.

The goal of the tree node selection is to select a child node that is already in the search tree. Algorithm 4 shows the overall procedure of node selection. Given the input of a tree node, it checks whether both left and right nodes exist in the search tree. If both nodes are valid, it selects the best child by calling `selectBestChild` (Line 3). Otherwise, only the valid left or right node will be selected (Lines 4-7).

Inside function `selectBestChild`, the child node is selected based on the

Algorithm 5: Procedure of Tree Node Expansion in VITAL

```
   Input: a tree node node, the unsafe pointer set unsafeSet
   Output: a leaf node of node to be expanded expanded_node
1  Function doExpansion(node):
2      if ! node.left.isInTree && ! node.right.isInTree then
3          expand_node = expandBestChild(node, unsafeSet)
4      if node.left.isInTree && ! node.right.isInTree then
5          expanded_node = node->right
6      if ! node.left.isInTree && node.right.isInTree then
7          selected_node = node->left
8      return expanded_node
9  Function expandBestChild(node, unsafeSet):
       // return the node with the highest ExpScore
```

highest UCT (i.e., Upper Confidence bounds applied to Trees) value. UCT is a widely-recognized algorithm that addresses a significant limitation of MCTS [21, 116, 122], where the MCTS may incorrectly favor a suboptimal move that has a limited number of forced refutations. The UCT formula used to balance the exploitation and exploration is defined as $UCT(s, s') = \frac{R(s')}{V(s')} + C\sqrt{\frac{2 \ln V(s)}{V(s')}}$ (details can be found in Section 3.3 in [21]), where s is the current state, s' is the child state of state s being selected, $V(s)$ indicates how many times the state has been visited, and $R(s)$ is the cumulative reward of all the simulations that have passed through this state. C is a constant that controls the degree of exploration.

When a selected node is returned (Line 8), it either goes back to the *while-loop* again if the selected node is also in the search tree or jumps to the *if-else* branch and performs `doExpansion` if the selected node is not in the search tree.

Tree Node Expansion.

The expansion aims to expand a child that is *not* in the search tree yet.

Algorithm 5 presents the overall procedure of tree node expansion. Given an input node, different from node selection, it checks whether both left and right nodes do not exist in the search tree. If neither node is not in the search tree, it expands

the best child by calling `expandBestChild` (Line 3). Otherwise, only the left or right node will be augmented for expansion (Lines 4-7). Inside the function `expandBestChild`, the node is expanded based on the highest expansion score (*ExpScore*) among two branches.

To obtain the score of two branches after an execution state is forked, three steps are performed: (1) collecting unique program elements, i.e., basic blocks, that are *unique* to each state; (2) analyzing the collected basic blocks and finding a way to weigh which state is more vulnerable; and (3) scoring the two branches based on the weighting measurment.

To accomplish the first step, we use the dominance relationships of graph theory [124] on the Control Flow Graph (CFG) of a function to collect the unique basic blocks for each state. The idea is to find the *post-dominator* of the two new forked states and to collect the basic blocks in-between each of the states and their post-dominator; such basic blocks indicate how the two states may induce different executions. A node A on a graph is said to *dominate* another node B if every path from the graph's entry point (start node) to B must go through A. Conversely, a node A is a *post-dominator* of another node B if every path from B to the exit point (or end node) of the graph must pass through A. Taking the function `f` in the example shown in Figure 5.2 as an illustration, the CFG of the function is shown in Figure 5.3. Starting from the entry block (`BB1`), the execution will go to Line 15 and the engine will fork two states (`TrueState` and `FalseState`), where the *pc* (points to the next execution instruction) `TrueState` points to the basic block (`BB2`) starting at Line 16. The *pc* in `FalseState` points to the basic block (`BB3`) after the *if-branch* at Line 15. The unique basic block for `TrueState` is `BB2` as both states will go through the post-dominator block `BB3`. For the following forking at Line 17 in Figure 5.2(a), again, the unique basic block for two states is `BB4` and `BB5`, respectively.

Then, following the intuition that more unsafe pointers in a path implies more vulnerabilities, we use the number of unsafe pointers in the basic blocks to quantify

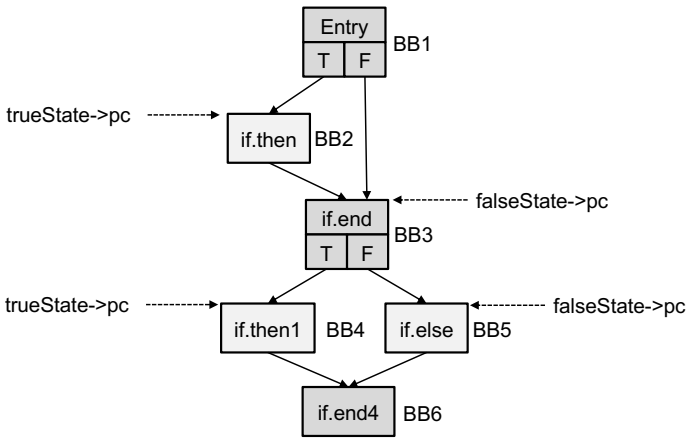


Figure 5.3: CFG of function `f` shown in Figure 5.2(a) illustrating how to get the score of true/false states

the expansion scores of the states: *ExpScore(s)* = $N_{unsafe}(s)$, where *s* is a state being scored and $N_{unsafe}(s)$ represents the number of unsafe pointers in the basic blocks collected as above for s.

Tree Node Simulation

Simulation is a crucial component of the MCTS algorithm as it enables the evaluation of non-terminal nodes by performing random playouts (i.e., simulation runs) to estimate the potential outcomes (i.e., rewards) of different actions. This process helps balance exploration and exploitation by providing statistical sampling to approximate the reward of execution states, which guides the overall algorithm in making promising decisions.

There can be many options to perform each simulation run. Intuitively for our symbolic execution, we could consider the following three options (OPs) to get the simulation reward of a node.

- *OP1*: statically analyze a single path following the state based on CFGs.
- *OP2*: symbolically run a single feasible path following the state with a fixed number of executed instructions.
- *OP3*: symbolically run a single feasible path following the state until the simulated state terminates.

Note that a high-quality simulation policy should be efficient, have few false positives, and have a certain degree of randomness [21, 122]. In particular, as shown in the literature [21], a certain degree of randomness is advantageous as it is simple and requires no domain knowledge, which will most likely cover diverse areas of a search space, helping on smart state search.

With respect to the above criteria, *OP1* might be good in terms of randomness but efficiency and false positives may be an issue: computing CFGs may be costly and paths in static CFGs may be infeasible. For *OP2*, it may comply to the criteria, but it might be difficult to decide what is the preferable number of instructions to use when evaluating the reward of a node across different runs. *OP3* could be better as it terminates based on the behavior of program execution. Technically, the simulation execution is essentially a lightweight depth-first execution of a single random feasible path. When a node is expanded, a new node is forked out for simulation. The new node's path constraints remain and the simulation is started by setting the node attribute "*inhibitForking=true*" to prevent further forking. Then, whenever a branch condition is encountered during simulation, a feasible random branch is chosen until the execution terminates. We empirically evaluate different fixed instruction numbers for *OP2*, and the results show that they are inferior compared with *OP3* (see more details in Section 5.4.3).

Algorithm 6 shows the simplified procedure of the simulation process designed in VITAL. It takes a tree node to be expanded and the unsafe pointer set as input, and yields a reward after simulation. Inside the algorithm, a vector of basic blocks *simulatedBB* is first initiated (Line 2). The vector stores all basic blocks executed during simulation via the function `executor.sim_run`. Then, two metrics (the number of unsafe pointers *nu_unsafe* and memory errors *nu_error*) are calculated (Line 4) or updated (Line 5), which will be used to get the reward of this simulation run (Line 6). In this study, VITAL is novel in utilizing the random and lightweight simulation runs to identify the likely vulnerable areas, and then utilizing MCTS to guide the symbolic execution capable of state forking to move toward those areas.

Algorithm 6: Procedure of Tree Node Simulation in VITAL

```
   Input: a tree node node, the unsafe pointer set unsafeSet
   Output: reward of the simulation reward
1  Function doSimulation(node, unsafeSet):
2      vector<BasicBlock*> simulatedBB
3      simulatedBB = executor.sim_run(node)
4      nu_unsafe = getNumUnsafePt(simulatedBB, unsafeSet)
5      nu_error = executor.numOfMemError
6      reward = getReward(nu_unsafe, nu_error)
7      return reward
8  Function isWorthSimu(node, unsafeSet):
9      if getNewRewardCount() < optimization_limit then
10         return true
11     else return false ;
```

When a simulation execution terminates, the reward function is used to calculate the reward for this node. We define the reward function as follows: $F_{reward}(s) = 0.5 * N_{unsafe}(s) + 0.5 * N_{error}(s)$, where the $N_{unsafe}(s)$ denotes the number of unsafe pointers covered during the simulated run and $N_{error}(s)$ is the memory errors detected throughout the run. We include the detected number of memory errors in the reward function based on a common observation that a root cause of one memory error may lead to various error symptoms in many code regions [210] and code regions related to a buggy code region likely contain bugs too, so we include this number in the reward expecting more memory errors can be found.

Simulation Optimization. When simulating a node in a for-loop statement, the default simulation process repeatedly simulates the same node. Such a process can be inefficient and waste a lot of time. (see more evaluation results in Section 5.4.3). Therefore, we design a new simulation optimization strategy in VITAL, to reduce the simulation of *unimportant* states that bring little new rewards due to the existence of loops. A straightforward way to avoid unlimited simulation on a loop is to set the loop bound when doing a simulation. However, this may still lead to unsound results [12]. Our insight is that instead of setting a fixed loop bound for the simulation, we consider a degree of increment based on the rewards of the previous simulation on

demand. Therefore, our solution is that when simulating a node in a loop, we record its reward after each simulation run of the loop. If it cannot get a higher reward after a limited number of iterations (we provide an extra option "*–optimization-degree*" to allow users a flexible control), it will not repeat the simulation of that node.

Backpropagation

Backpropagation is the final phase in MCTS-guided sampling, during which the total reward and visit count for each state are iteratively updated. For each state, the reward is adjusted based on the simulation outcome, and the visit count is incremented, with updates propagating from the leaf node to the root (at Line 11 in Algorithm 3).

5.3.3 Implementation of COTTONTAIL

We implemented VITAL on top of KLEE (v3.0). Following the instructions on the webpage, users can set up and run VITAL to find potential vulnerabilities in the test programs automatically.

Both the implementation of the type inference system and the MCTS algorithm are written in the C++ programming language. For the type inference system used in VITAL, we forked `CCured` (built on top of a static analysis tool SVF [180]) from a previous work [104]. We added a new search strategy `MCTSSearcher` in the `Search` class in KLEE's implementation to support the vulnerability-oriented path exploration. In general, we implemented our own `selectState` and `update` to maintain execution states. The functionality of the MCTS algorithm is implemented in the `selectState` function, where four major functions (i.e., `doSelection`, `doExpansion`, `doSimulation`, and `doBackpropagation`) mentioned in the Algorithm 3 are supported. We also modified the `executeInstruction` and `run` functions in `Executor.cpp` to support the checking of unsafe pointers during execution and the storing of the executed unsafe pointers into a file ("*unsafe-pt.txt*") to help users analyze the program further.

For setting the bias parameter C in Equation in Section 9 and the simulation optimization option "– *optimization-degree*", we set the value of $\sqrt{2}$ and 700 for them after evaluating their impacts, respectively (more detailed results and discussion in Section 5.5.2).

5.4 Evaluation

Extensive experiments are conducted to evaluate the effectiveness of the proposed VITAL from various perspectives. More specifically, we consider the following three research questions (RQs).

- **RQ1**: How does VITAL perform in terms of unsafe pointer coverage?
- **RQ2**: How does VITAL perform in terms of vulnerability detection?
- **RQ3**: Can each major component contribute to VITAL?
- **RQ4**: Can VITAL detect previously unknown vulnerability?

Among these RQs, RQ1 evaluates VITAL's capabilities in terms of unsafe pointer coverage and memory error detection compared with representative path exploration strategies in KLEE and CBC. RQ2 aims to investigate the vulnerability detection capabilities in terms of efficiency and memory usage compared with standard search strategies in KLEE and newly proposed search strategy in CBC as well as choppered symbolic execution. By comparing experimental results in RQ1 and RQ2, we could better understand the benefits of VITAL in vulnerability-oriented path exploration. RQ3 concentrates on the contribution of each component of VITAL. By comparing each component with its variant, we could gain more insights into the reason why VITAL works better. RQ4 demonstrates the practical vulnerability detection capability of VITAL.

All experiments conducted in this study run on a Linux PC with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz x 12 processors and 64GB RAM running Ubuntu 18.04 operating system.

5.4.1 Capability of Covering Unsafe Pointer

Benchmarks. We use the well-known GNU `Coreutils` (v9.5) in this RQ1, following many existing works [28, 97, 120, 189, 192]. The utilities include the basic file, shell, and text manipulation tools of the operating system. We selected 75 utilities in total for this study and excluded some utilities that: (1) cause nondeterministic behaviors (e.g., `kill`, `ptx`, and `yes`) and (2) exit early due to the unsupported assembly code or the call for external functions based on our experiments.

For the selected utilities, we measure the size of executable lines of code (ELOC) by counting the total number of executable lines in the final executable after global optimization. The distribution of ELOC ranges from 800-8,000, which could comprehensively evaluate the effectiveness of VITAL on test programs of various lengths.

Comparative Approaches. We select six representative path exploration strategies (i.e., three commonly evaluated in the literature [120, 187, 192, 193] and three coverage-guided heuristics) as follows:

- Breadth first search (`bfs`) and depth first search (`dfs`).
- Random (`random-state`) randomly selects a state to explore.
- Code coverage-guided (`nurs:covnew`) selects a state that has a better chance to cover new code.
- Instruction coverage-guided (`nurs:md2u`) prefers a state with minimum distance to an uncovered instruction, while (`nurs:icnt`) picks a state trying to maximize instruction count.

We also compared VITAL with a recent work CBC [209], a compatible branch coverage-driven path exploration for symbolic execution.

Evaluation Metrics. We use the following two metrics to assess the effectiveness of different path search strategies.

- **(1) The number of unsafe pointers covered** compares the unsafe pointer covering capabilities among different approaches.
- **(2) The number of memory errors detected** compares memory error detection

Table 5.1: Comparison results with existing search strategies

Search-Strategy	Unsafe Pointers		Memory Errors	
	number	*improvement*	*number*	*improvement*
`bfs`	11610	11.01%	40	10.00%
`dfs`	9626	33.89%	32	37.50%
`random`	7609	69.38%	34	29.41%
`nurs:covnew`	10232	25.96%	39	12.82%
`nurs:md2u`	9417	36.86%	39	12.82%
`nurs:icnt`	6782	90.03%	35	25.71%
CBC [209]	2194 (3890)	77.30%	21 (33)	57.14%
VITAL	12888	-	44	-

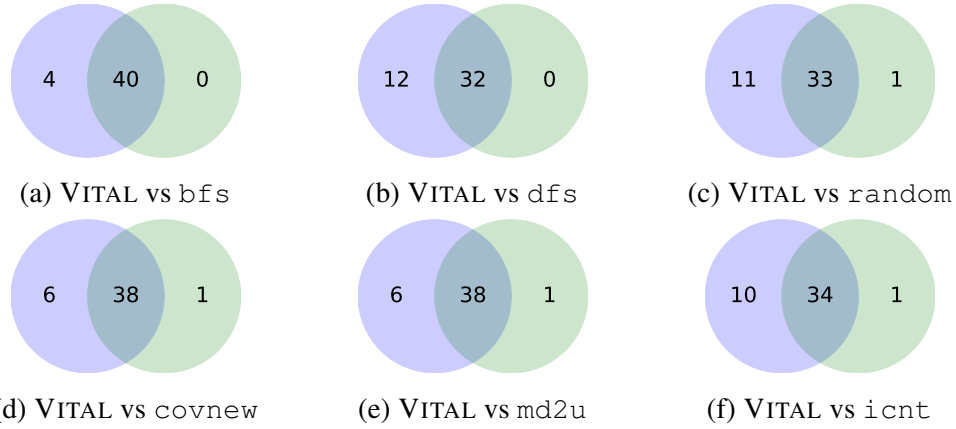
* CBC can only successfully run 50 out of 75 utilities, so we compare VITAL and CBC over the same 50 benchmarks. The numbers in the bracket represent the number of unsafe pointers/memory errors covered/detected by VITAL.

capabilities among comparative approaches based on the unsafe pointer coverage.

Running Setting. Following existing studies [28, 120, 192], we set a running time of one hour for each setting. For random searches, we ran them five times and reported the median results.

Results. Table 5.1 shows the overall results, where the numbers of two metrics and improvement (*improv.*) achieved by VITAL are recorded. From the table, it is evident that VITAL performs significantly better than existing search strategies in terms of both the number of covered unsafe pointers and detected memory errors. In particular, VITAL could cover 90.03% more unsafe points compared to `nurs:icnt` and detect 57.14% more memory errors compared to CBC, demonstrating the superior performance of VITAL.

For the covered unsafe pointers, we also dig in deeply into the individual improvement achieved by VITAL on each utility. The results show that VITAL overall outperforms all comparative search strategies (we omitted the results of CBC as VITAL outperforms it over all utilities). In a large portion of cases, VITAL produces significant (up to 3500% under `nurs:icnt`) improvements. This is mainly because the unsafe pointer-guided MCTS excels at navigating the best possible exploration-exploitation trade-offs, thus exploring the path where the number of unsafe pointers is maximized. For a small number of cases (especially for the utilities under BFS),



(a) VITAL vs `bfs` (b) VITAL vs `dfs` (c) VITAL vs `random`

(d) VITAL vs `covnew` (e) VITAL vs `md2u` (f) VITAL vs `icnt`

Figure 5.4: Numer of detected *unique* memory errors by VITAL compared to comparative search strategies

VITAL fails to cover more unsafe pointers. We investigated more on such cases and found that this follows one of the characteristics of the MCTS (Section 3.4.3 in [21]): MCTS favors more promising nodes and leads to an asymmetric tree over time. For test programs whose execution tree is symmetric, MCTS may miss a few nodes on the execution tree, thus missing certain unsafe pointers. However, such a characteristic is also advantageous in helping Vital to explore deeper paths, increasing the likelihood of vulnerability detection. Our experiments also showed that Vital is overall significantly better (e.g., up to 57.14% more memory errors).

We also investigate the *unique* memory error detection capability of VITAL and the Venn Figure 5.4 presents the overall results (we omitted the results of CBC as VITAL can cover all memory errors reported by CBC). From the figure, we can observe that VITAL could detect larger numbers (ranging from 4 to 12) of unique memory errors. VITAL only misses one unique memory error: this is due to a few missed coverage of unsafe pointers by VITAL as explained before.

Existing traditional search strategies in KLEE miss certain unsafe pointer coverage and memory errors mainly because they only target coverage improvement while concentrating less on vulnerability detection. Compared to the recent CBC search strategy, VITAL outperforms it for two reasons. First and most importantly, the assumption that many paths have no contribution to branch coverage and thus

bug finding held by CBC is not always valid for memory error detection, as most of the memory errors can only be triggered under specific contexts (e.g., function calls). One function calls the vulnerable function may trigger the bug but invokes another function call may not. Second, computing dependence information on demand in CBC takes time and memory consumption (as reported in the CBC paper [209], it introduces 30% more memory consumption and execution time compared to the baseline approach KLEE when performing path exploration).

TAKEAWAY: we analyze the correlation between the number of covered unsafe pointers and detected memory errors. Figure 5.1 presents the result using the data collected in Table 5.1, where the Pearson's coefficient [46] **0.93** suggests a strong positive correlation.

> **Answer to RQ1:** VITAL outperforms existing path search strategies by covering up to 90.03% unsafe pointers and detecting up to 57.14% more unique memory errors, indicating a better unsafe pointer covering capability.

5.4.2 Capability of Vulnerability Detection

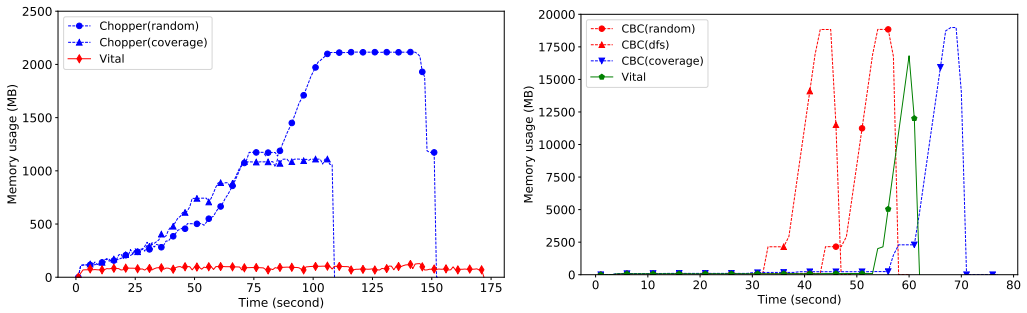
Benchmarks. We use four CVEs in GNU `libtasn1` library with different versions (followed `Chopper` [189]), namely CVE-2012-1569 (v3.21), CVE-2014-3467 (v3.5), CVE-2015-2806 (v4.3), and CVE-2015-3622 (v4.4). The `libtasn1` library facilitates the serialization and deserialization of data using Abstract Syntax Notation One (ASN.1). The vulnerabilities selected for analysis in this RQ predominantly involve memory out-of-bounds accesses. Note that each identified vulnerability is associated with detecting a singular failure, except for CVE-2014-3467, where this vulnerability manifests across three distinct locations, so the experiment seeks to detect six distinct vulnerabilities.

Comparative Approaches. We compared VITAL with the baseline approach (i.e., KLEE [28]) and two state-of-the-art approaches (i.e., `Chopper` [189] and CBC [209]) in this RQ. To be specific, we run KLEE, `Chopper`, and CBC under different

Table 5.2: Results of time on detecting CVE vulnerabilities.

Vulnerablities	KLEE[28]			Chopper [189]			CBC [209]			VITAL
	Rand.	*DFS*	*Cove.*	*Rand.*	*DFS*	*Cove.*	*Rand.*	*DFS*	*Cove.*	
CVE-2012-1569	11:36	02:40	11:03	01:50	01:03	01:57	01:01	0:48	01:18	01:03
CVE-2014-3467[1]	*TO*	03:07	*OOM*	08:24	0:12	04:22	N/A	N/A	N/A	01:06
CVE-2014-3467[2]	0:06	14:55:21	0:05	09:25	58:48	19:11	0:29	*TO*	0:23	0:19
CVE-2014-3467[3]	*TO*	*TO*	*TO*	26:48	0:29	11:22	0:54	0:18	*TO*	0:09
CVE-2015-2806	05:43	02:55:37	*OOM*	02:33	*TO*	01:48	N/A	21:48	N/A	02:53
CVE-2015-3622	*TO*	*TO*	07:49:39	0:58	13:56	0:57	02:40	01:40	01:19	0:26

* The time format is *hour:minute:second* (we omit *hour* if the time is less than 1 hour). "TO" refers to timeout and "OOM" represents out of memory. "N/A" represents the normal termination of execution without reproducing the vulnerability.



(a) Comparison with Chopper over CVE-2015-2806

(b) Comparison with CBC over CVE-2012-1569

Figure 5.5: Memory consumption comparison results of VITAL against Chopper [189] and CBC [209].

search strategies, i.e., *Random*, *DFS*, and *Coverage*.

Evaluation Metrics. We use the following two metrics to assess effectiveness.

- **(1) The execution time** records the time to detect a vulnerability.
- **(2) Memory consumption** measures memory usage in detecting a vulnerability.

Running Setting. Following the existing study [189], we set a timeout of 24 hours to detect each vulnerability. Again, for random searches, we ran them five times and reported the median results.

Results. Table 5.2 presents the overall results in terms of execution time. From the table, we can see that VITAL outperforms Chopper and CBC for almost all vulnerabilities. Upon detecting the vulnerability in CVE-2014-3467[2], VITAL achieves a speedup of 30x to Chopper. For the vulnerability CVE-2015-2806, VITAL takes slightly more time (less than 20 seconds) to detect it. This is because Chopper skips some large functions before execution, making it detect the vulnerability faster. How-

ever, as emphasized in Section 5.1, users of `Chopper` need prior expert knowledge to decide which functions/lines to skip, which are labor intensive. Also, `Chopper` consumes more memory when detecting the vulnerability (see more details below).

Compared to CBC[4], VITAL takes less time to detect 5 out of 6 vulnerabilities (except CVE-2012-1569 with *Random* and *DFS* search). For CVE-2014-3467[1], CBC does not detect it in any search strategies as this vulnerability happens in a while-loop condition, which requires multiple execution of the condition to trigger the vulnerability. This again emphasizes a fundamental design flaw in CBC, as it can only detect bugs about assertion failure that bugs are triggered when the code executes once. However, as shown in previous studies [9, 31, 95], many memory errors occur in loop iterations, and only executing the loop once is insufficient to detect certain errors.

In terms of memory consumption, we run VITAL against `Chopper` and CBC over two vulnerabilities in CVE-2015-2806 and CVE-2012-1569 respectively to gain more insights because VITAL takes comparable or slightly more time for the vulnerability detection. Figure 5.5 shows the results, where *x-axis* represents the time to detect the vulnerability and *y-axis* indicates the usage of memory. From the two figures, we can observe that VITAL consistently consumes less memory when detecting the vulnerability. In particular, as shown in Figure 5.5(a), we can observe that VITAL consumes significantly less memory when detecting the same vulnerabilities compared to `Chopper`. For example, `Chopper` with a random search consumes at most 2,115 MB of memory, while VITAL only takes around 100 MB of memory, producing a significant reduction (i.e., 20x) in memory consumption. This is reasonable as `Chopper` takes a state recovery mechanism to maintain the execution of skipped functions. Since the skipped functions can be very large, maintaining recovered states in `Chopper` tends to be memory-intensive. Compared

[4]Note that there are some discrepancies in the results between those reported in Table 5.2 and in the original CBC paper (Table 2 in [209]). We found that the discrepancies are caused by the various settings to find the vulnerability, while we selected a fixed setting that can produce the best overall results to present in this chapter (more details in the Appendix in original VITAL paper).

Table 5.3: Comparison results with variant approaches

Search	Unsafe Pointers		Memory Errors	
	number	*improvement*	*number*	*improvement*
VITAL(¬EXP)	11034	16.80%	41	7.32%
VITAL(¬SIM)	9010	43.04%	36	22.22%
VITAL(¬SOPT)	9863	30.67%	33	33.33%
VITAL(OPT2-500)	5449	137.52%	25	76.00 %
VITAL(OPT2-1000)	7805	65.12%	26	69.23 %
VITAL(OPT2-2000)	6477	98.98%	26	69.23 %
VITAL	12888	-	44	-

to CBC, it requires extra memory consumption to obtain the dependence information during the symbolic execution, which is memory intensive. In contrast, VITAL does not increase memory consumption compared to standard symbolic execution, providing a solution that is not only lightweight but also highly effective.

Answer to RQ2: VITAL outperforms chopped symbolic execution by achieving a speedup of up to 30x execution time and a reduction of up to 20x memory consumption, indicating a better vulnerability detection capability.

5.4.3 Ablation Study

Benchmarks and Evaluation Metrics. We use the same benchmarks (i.e., GNU `Coreutils`) and evaluation metrics (i.e., unsafe pointer coverage and number of detected memory errors) as in RQ1 to compare different approaches in this RQ.

Comparative Approaches. We design several variants of VITAL to gain a deeper understanding of the contribution of each component. We focus on evaluating the following variant approaches:

- VITAL(¬EXP) uses random expansion without guided expansion.
- VITAL(¬SIM) performs path search without simulation.
- VITAL(¬SOPT) adopts simulation but without optimizations.

Running Setting: We use the same setting as in RQ1.

Results. Table 5.3 presents the overall results between the VITAL and comparative approaches. We can conclude that VITAL performs better than all variant

approaches. Specifically, compared to VITAL(¬EXP), VITAL could cover 16.80% more unsafe points and detect 7.32% more memory errors, demonstrating the contribution of unsafe pointer-guided node expansion designed in Algorithm 3. The other two variant approaches share similar results, where VITAL achieves an improvement of 43.04% and 22.22% as well as 30.67% and 33.33% in terms of the number of covered unsafe pointers and detected memory errors, compared to VITAL(¬SIM) and VITAL(¬SOPT), respectively, indicating the postive contributions of simulation.

Evaluation of Different Simulation Strategies. We run *OP3* (the default simulation setting in VITAL), VITAL(OPT2-500), VITAL(OPT2-1000), and VITAL(OPT2-2000) (where x in VITAL(OPT2-X) means the fixed number of instructions to terminate the simulation) to evaluate different settings of *OP2*. We omit *OP1* as its static simulation of a node tends to be ineffective due to time-consuming CFG traversing and high false alarm rate, as discussed in Section 9. As shown in Table 5.3, all three settings of *OP2* perform inferior compared to *OP3* (normally terminated execution) in VITAL. This is reasonable as aborting too early would make VITAL prone to false negatives, thus missing interesting unsafe pointers coverage and downgrading the overall performance of MCTS.

> **Answer to RQ3:** The newly designed components, including type-unsafe pointer-guided node expansion and node simulation as well as its optimization, all contribute to VITAL in terms of unsafe pointer coverage and memory error detection capability.

5.4.4 Practical Application of VITAL

To demonstrate the practical vulnerability detection capability of VITAL over large-scale software systems, we run VITAL over the intensively-tested and latest released *objdump* package (includes more than 20k lines of code) in `GNU binutils-2.44`. As a result, VITAL detected a previously unknown vulnerability[5] within one minute.

[5] https://sourceware.org/bugzilla/show_bug.cgi?id=32716.

The issue is a memory leak where an allocated object is not appropriately de-allocated, which may cause severe security risks such as critical information leakage or denial of service. This issue had been lurking for more than 7 years before we reported it[6]. Since the critical impact of the issue, developers confirmed and fixed it swiftly within 8 hours after we submitted the bug report. It has been assigned a new CVE ID (i.e., CVE-2025-3198).

It is worth noting that other approaches failed to detect this issue by giving a running timeout of 24 hours[7]. `Chopper` [189] failed to detect the issue mainly due to the lack of prior expert knowledge or technical support to set up the execution. KLEE's default search heuristics are limited by bypassing the input-dependent loops, which are prerequisites to reach the target vulnerable function. CBC [209] is restricted by the non-vulnerability-oriented path search heuristic, which ignores the fact that many memory errors can only happen after multiple loop executions [95].

VITAL discovered the vulnerable path because of its new way of simulating execution paths and its novel search based on MCTS. Compared with existing solutions, VITAL can skip *unimportant* loops that have no contribution to the accumulation of unsafe pointer coverage by using simulation optimization strategies (see Section 9). For loops that may lead to memory issues, our simulation and MCTS will evaluate the reward of each of their iterations and continue when it is worthy exploring, i.e., new unsafe pointers are covered.

Answer to RQ4: VITAL is able to detect previously unknown vulnerability in practice, working in a fully automatic manner without need of prior expert knowledge.

[6]The oldest version that can reproduce the issue is binutils-2.29 released on 25/09/2017.
[7]We provide detailed vulnerability analysis in the Appendix in the original VITAL paper.

5.5 Discussion

5.5.1 Overhead of Pointer Type Inference

To further understand the overhead of the type inference system (i.e., `CCured`) used in VITAL, we measure the time to infer pointer types. The results show that in most cases (64%, 48 out of 75), the time speed of the analysis is within 5 seconds, with an overall average time of 5.04 seconds. We believe that this can be neglected compared to the potentially large amount of time contributing to the entire testing period (e.g., 24 hours running timeout in RQ2).

5.5.2 Impact of Different Configurations

The selection of the value of bias parameter *C* defined in Equation in Section 9 and "–*optimization-degree*" in Algorithm 6 may affect the performance of VITAL. Thus, we conduct extra experiments to assess the impact of different running configurations.

For the configurations of bias parameter `C`, we run VITAL with values of $\sqrt{2}$, 5, 10, 20, 50, and 100. The results show that VITAL covered 12,888, 12,620, 13,144, 12,724, 13,188, and 12,578 unsafe pointers and detected 44, 41, 42, 43, 43, and 44 memory errors in each configuration, respectively. Since the goal of VITAL is to detect more memory errors/vulnerabilities, we select the value $\sqrt{2}$ as the default configuration of VITAL as this setting yields the best memory error detection and a comparable unsafe pointer coverage capability.

For the impact of "–*optimization-degree*", we run VITAL with values of 100, 300, 500, 700, 1,100, and 1,500. The results show VITAL covered 12,078, 12,349, 12,660, 12,888, 11,888, 12,720, 12,655, and 12,578 unsafe pointers and detected 35, 39, 42, 44, 41, 41, and 39 memory errors under each configuration, respectively. Since the value of 700 produces the best results in terms of both unsafe pointer covering and memory error detection capabilities, we set 700 as the default in VITAL.

5.5.3 Threats to Validity

The *internal* validity concerns stem from the implementation of VITAL. To mitigate this threat, we built VITAL on top of the well-maintained and recently released version (v3.0) of KLEE [28]. In addition, we have meticulously implemented VITAL reusing existing APIs in KLEE, as explained in Section 5.3.3, and have performed a detailed code check to mitigate the threat.

The *external* threat comes from the benchmarks used in this study. We used GNU `Coreutils` and a library `libstasn1` with four different versions. Although they have been widely used for evaluating symbolic execution [28, 97, 120, 187, 190], these programs may not be representative enough for various software systems. To further alleviate these potential threats, we are committed to expanding the program sets in our future work.

The *construct* validity threat is subject to configurations of parameters. We address this concern by thoroughly investigating their impact in Section 5.5.2, which enhances transparency and enables a deeper understanding of the influence of different configurations.

5.5.4 Limitations

One limitation lies in VITAL is that it can not detect all types of memory errors. This is mainly because the tool used for type inference `CCured` [145] can not classify all unsafe pointers (i.e., the ones that lead to *temporal* memory errors). This is reasonable as the identification of a *complete* set of unsafe pointers is challenging[72, 104]. It is worth noting that extending the support of other unsafe pointers as indicators to detect new types (e.g., *temporal*) of vulnerabilities in VITAL should only involve engineering effort (e.g., transporting the tool SAFECode [60] to work on compatible LLVM bitcode) to support the classification. We plan to leverage more advanced techniques (such as partial techniques proposed in DataGuard [104]) to address this limitation in future work.

VITAL suffers from certain inherent limitations in symbolic execution engines (e.g., KLEE [28]) due to limited memory modeling, environment modeling, and efficiency issues, which may restrict the memory error detection capability of VITAL. This is because some intractable vulnerabilities can only be triggered under a complex situation, which requires a more comprehensive modeling of program semantics of test programs. Efficiency is also an issue, as most symbolic execution engines analyze test programs by interpreting the intermediate representation code (e.g., LLVM Bitcode in KLEE), which is shown to be inefficient [159, 160]. Recent studies [154, 168, 192, 193] have been proposed to resolve these problems, and we plan to integrate them into VITAL in future work.

5.6 Summary

We presented VITAL, a new vulnerability-oriented symbolic execution via type-unsafe pointer-guided Monte Carlo Tree Search. VITAL guides the path search toward vulnerabilities by (1) acquiring *type-unsafe* pointers by a static pointer analysis (i.e., type inference), and (2) navigating the best possible exploration-exploitation trade-offs to prioritize program paths where the number of unsafe pointers is maximized leveraging unsafe pointer-guided Monte Carlo Tree Search. We compared VITAL with existing path search strategies and chopped symbolic execution, and the results demonstrate the superior performance of VITAL among both approaches in terms of unsafe pointer coverage and memory errors/vulnerabilities detection capability.

Chapter 6

Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation

6.1 Objective

Parsing software, including XML and SQL libraries, is widely used in modern systems. However, even after years of intensive testing efforts, residual vulnerabilities persist, reflecting the complexity and attack surface of such components. Highly structured (or syntactically valid) test inputs are demanded to comprehensively stress the parsing test programs; as only the parser-checking logic is passed, the deeper application logic can be examined. Considerable effort is paid to generate structured test inputs, including black, grey, and white-box fuzzing-based approaches. Among them, white-box fuzzing via concolic execution has shown considerable capabilities of test input generation for general test programs. Given a seed input, a concolic execution engine starts by concretely executing the program while symbolically tracking the same execution path to collect path constraints. The negation of path constraints is applied to explore alternative branches. An off-the-shelf constraint

solver is used to solve constraints and then generate new test cases[1] that satisfy the modified constraints, enabling the path exploration of uncovered paths systematically. Benefiting from the soundness of test case generation and a systematic way for path exploration, it has been promising and applied in many areas [36, 102, 160, 212].

Although promising, existing concolic executors (e.g., SYMCC [160] and MARCO [102]) remain significantly hindered by three fundamental limitations in their treatments for the problems of *which to solve*, *how to solve*, and *how to acquire new seed inputs* when handling parsing test programs.

- **#L1: The input structure-agnostic path constraints selection is either redundant or overly aggressive**. When path constraints are generated during concolic execution, we need to consider the problem of *which path constraints to solve*. A straightforward idea is to select all collected path constraints, hoping to explore all possible paths in the test programs. However, such a structure-agnostic option leads to many redundant path constraints, making the testing process impractical. An alternative is to select the path constraints based on some heuristics, e.g., bit-wise coverage map `Bitmap` from SYMCC [160] and Concolic State Transition Graph (CSTG) from MARCO [102]. Unfortunately, both heuristics overly eliminate interesting code coverage aggressively, either due to hash collisions or limited trace granularity in the `Bitmap` or incomplete modeling of program paths in CSTG (see more details in Section 6.3.1).
- **#L2: The solutions from constraint solving only comply with satisfiability while neglecting syntactic validity.** To obtain high-quality (especially for highly structured) test cases from constraint solving, an important question is *how to solve the constraints*. Traditional constraint solvers (e.g., `Z3`) equipped in concolic executors usually solve constraints for satisfiability, while ignoring the syntactic validity of newly generated test cases (see more explanation in Section 6.2.3). Such a limitation oftentimes causes the engine to produce syntactically invalid

[1] We may call a file a *test case* too when it is used to store output of a testing engine and a *test input* when it is used as seed input to set up testing.

test cases, rendering the testing effort largely in vain. We argue that an optimal solution for constraints should not only comply with satisfiability but also be aware of syntactic validity. Also, traditional solvers can not generate new test cases with flexible sizes by design, as the number of symbolized bytes is the same as the input seed, further decreasing the effectiveness of concolic testing.

- **#L3: The acquisition of highly structured seeds before testing starts or after testing is saturated is difficult.** Existing engines highly rely on manually collected *initial* seeds from bug repositories to start testing, where the manual work is often time-consuming. Randomly generating seeds could be an alternative, but it tends to be ineffective, as many random seeds do not boost coverages, and it would be wasteful to continue testing if the testing is saturated (i.e., code coverages have plateaued and no new code coverage after a certain time period). We thus need to acquire *meaningfully fresh* seeds to change the situation, but existing concolic executors are unable to generate such seed inputs during testing progress. Again, it is possible to naively feed random seeds into the testing process after saturation, but it rarely improves code coverage (as demonstrated in Section 6.4.2).

To overcome the above limitations, we propose COTTONTAIL[2], a new Large Language Model (LLM)-driven concolic execution engine to generate highly structured test inputs effectively. Our key insights are threefold. *First*, parsing programs typically implement their parsing/application logic structurally (e.g., by using *switch-case* statements) to process structured test inputs. We can thus build what we call *structural program paths*, i.e., paths that diverge depending on specific input values (e.g., case constants in *switch-case* statements) to help not only represent more meaningful program paths better but also reduce potentially redundant path constraints, thus addressing limitation **#L1**. *Second*, given the strong code completion capabilities of LLMs [125], it could be leveraged to *solve* the constraints for satisfiability and *complete* the solution to comply with the input syntax, thus alleviating limitation

[2]Cottontail rabbits are known for their structured running patterns (e.g., zigzagging) to evade predators using their cotton-ball tails. We use COTTONTAIL to reflect our aim for the generation of highly structured test input.

#L2 (see detailed motivations in Section 6.2.3). *Third*, benefiting from its knowledge and memorization [76, 130, 144, 181], LLMs could be prompted to produce new meaningful seed inputs, thus mitigating limitation **#L3**.

Based on the above insights, three new components are designed in COTTONTAIL to explore structural program paths for highly structured test input generation.

- **Structure-aware Constraint Selection**. To address the limitation of *which to solve*, a structural information collector is first introduced during the instrumentation phase to help construct a more complete representation of *structural program path*. Then, based on the representation, a new coverage map, named Expressive Structural Coverage Tree (ESCT), is constructed to keep track of program branch status (e.g., taken or untaken) in a global view. Finally, guided by ESCT, a path constraint selector is facilitated to select structure-aware path constraints.
- **LLM-driven Constraint Solving**. To mitigate the limitation of *how to solve*, an LLM-driven solver, which facilitates *Solve-Complete* paradigm, is leveraged to solve the path constraints. The paradigm first *solves* constraints for satisfiablity and then *completes* the solution for the syntax validity. Also, the LLM-generated test cases can have a flexible size, alleviating the restrictions that existing solvers can only generate test cases of fixed-size. Moreover, to increase the robustness of LLMs, a test case validator is designed to check and refine unsound results.
- **History-guided Seed Acquisition**. To address the limitation of *how to acquire new seeds*, history-guided seed acquisition strategies are designed for different timings. Before testing, we prompt LLMs to generate highly structured test inputs as *initial* seeds that may likely trigger new vulnerabilities based on their memories (e.g., from historical bug repositories). During testing, a history coverage recorder is utilized to record the mapping between test input and its code coverage. When the coverage gets plateaued, based on the mappings from the recorder and uncovered branches information from ECST, we prompt LLMs with Chain-of-Thought (CoT) to generate *fresh* test inputs that are likely to cover more uncovered branches or unexplored features.

In short, COTTONTAIL is a novel concolic execution engine that is capable of *structure-aware path constraint selection*, *smart path constraint solving*, *history-guided seed acquisition* for robust generation of highly structured test inputs.

We have prototyped COTTONTAIL on top of SYMCC [160] and demonstrated its test input generation capabilities over eight widely tested libraries across four different formats (`XML`, `SQL`, `JavaScript`, and `JSON`). Our experiments show promising results. Compared with state-of-the-art approaches, COTTONTAIL outperforms SYMCC [160] and MARCO [102] by achieving 14.15% and 14.31% higher line and 15.96% and 11.10% higher branch coverage, respectively. During the same period, COTTONTAIL significantly improves (more than 100x) the parser checking passing rate over the generated test cases. Our ablation studies also demonstrate that each component in COTTONTAIL has contributed to the better results. We have also found 6 previously unknown memory-related vulnerabilities and reported them to the developers (six new CVE IDs have been assigned, and 4 out of 6 have been fixed).

Contributions. We summarize our contributions as follows:

- To our best knowledge, COTTONTAIL is the first LLM-driven concolic execution engine for highly structured test input generation, automatically working in a white-box manner.
- Three new components, including structure-aware path constraint selection, smart LLM-driven constraint solving, and history-guided seed acquisition, are designed to make COTTONTAIL effective and practical.
- Extensive experiments are conducted to demonstrate the capabilities of COTTONTAIL. The results show that COTTONTAIL can not only outperform baseline approaches but also is able to find previously unknown security vulnerabilities.
- The prototype[3] of COTTONTAIL is open source to foster future research that combines program analysis and LLMs.

[3] https://github.com/Cottontail-Proj/cottontail

6.2 Background and Motivation

6.2.1 Concolic Execution

Concolic execution, also known as dynamic symbolic execution, integrates symbolic and concrete execution to explore program paths systematically. Concrete values guide the actual execution path, ensuring feasibility, while symbolic values enable exploration of alternative paths by generating new test cases. In recent years, compilation-based concolic execution (e.g., SYMCC [160] and MARCO [102]) has gained popularity due to its superior performance and practical applicability. Given an initial seed input, these engines embed symbolic reasoning/tracing logic directly into compiled binaries and collect path constraints at runtime. By negating selected path constraints, the engine produces new constraints representing unexplored branches, which are then solved using off-the-shelf constraint solvers (e.g., `Z3` [214]) to generate new test cases. Note that the new test cases usually hold the same size as the seed input because the size of symbolic bytes is fixed when the seed input is fed into the concolic engine in the initial phase. The concolic testing process is continued by iteratively feeding the new test cases back into the execution loop, which is ideally an endless process. However, concolic testing naturally encounters a saturation point when it can no longer cover new code due to all test inputs is lack of diversity (e.g., limited by the input size) [31, 191, 193] or the restricted covering capability of harnesses/test drivers [106, 107, 127]. When the saturation point is reached, the best practice is to acquire *fresh* new seed inputs that drive the exploration forward for continuous testing.

In short, *which constraints to solve* determine both effectiveness and efficiency of testing, *how to solve the constraints* to obtain test cases affects the effectiveness of testing, while *how to acquire new seeds* decides the continuity of testing.

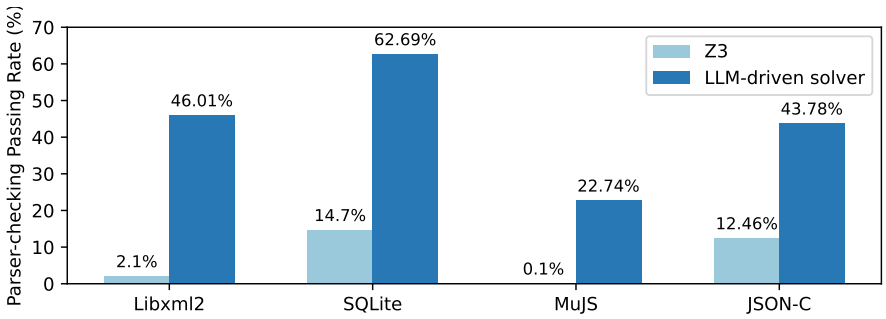


Figure 6.1: Parser checking passing rate comparison between traditional constraint solver (i.e., `Z3`) and LLM-driven solver (designed in COTTONTAIL).

6.2.2 LLMs for Test Input Generation

Large Language Models (LLMs) are popular AI systems designed to predict the next word or token in a sequence based on the context of preceding tokens. Recent emergence of LLMs has driven their application in numerous security-related domains [110, 156, 176]. However, most of the LLM-based systems lack analytical depth and robustness, which limits their effectiveness in more complex scenarios such as systematic program understandings [194]. Two promising directions to mitigate the problem are either to integrate advanced program analysis or design logical Chain-of-Thought (CoT) prompts to further improve the reliability and robustness of the LLM-based system. While LLMs have been combined with black-box [57, 203] or grey-box [127, 139] fuzzing techniques for structured test input generation, as far as we know, no study has yet attempted to integrate LLMs with more systematic techniques like concolic execution to enhance the security analysis of software systems. We believe it could be promising to further empower the potential of LLMs for more rigorous security guarantees by integrating LLMs with concolic testing.

6.2.3 Investigation Study

It is evident that there is a need for a new path constraint selection strategy to select optimal subsets of constraints and a new seed acquisition to generate highly structured inputs to improve both effectiveness and continuity of concolic testing. It may not be clear how the LLM-driven solver (which served as our core novelty) can

</> System Prompt (*Solve-Complete* Paradigm with Chain of Thought)

You are a smart Constraint Solver.

Given a constraint and a marked test input string, please generate new test case step by step:

1. Identify the location of ***Constraint Mask*** (`[k!n]`) and *solve* it with the solution of the constraint:
   - The solution is not arbitrarily chosen; instead, make the combined string align with input syntax, such as keywords, defined functions, or variable names to ensure the resulting string remains valid.
2. Identify the location of ***Flexible Mask*** (`[xxx]`) and *complete* it with a **flexible** size string:
   - The replacement is not randomly selected: instead, follow the syntax of JavaScript and make the whole string remaining a valid JavaScript code.
3. Ensure final output is enclosed with ```.

</> User Prompt

Given the path constraint ```bvsge #x00000039 (concat #x000000 k!95))``` and the marked seed input you need to change: ```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) { r[k!95][xxx]```, what is the new test input?

</> Response

Let's solve the constraint step by step:

1. The constraint mask is [k!95] and we now solve it based on the given constraint.
   - The constraint suggests the value should be large or equal to ASCII 0x39. Considering the previous string `r`, the solution `e` should satisfy the requirements, as `return` is a keyword in JavaScript.
2. The flexible mask is [xxx] and we now complete it with other flexible size strings:
   - Since previous string starting with `re` after solving, complete `[xxx]` with `turn y;} print(a[100]);`, as `return` is a keyword, and `y` and `a` are defined variables.
3. Final output:

```c = 30000; a = []; for (i = 0; i < 2 * c; i += 1) {a.push(i%c);} a.sort(function (x, y) {return y;} print(a[100]);```

Figure 6.2: The *Solve-Complete* paradigm and LLM's response. In the upper box, the underlined text represents different formats, and the colored text enclosed indicates either path constraints or marked test input string.

help smartly solve path constraints.

To justify our motivation, we conducted an investigation study to show the significant limitations of traditional constraint solvers and how an LLM-driven constraint solver can mitigate them. To do so, we first selected four libraries that process four different input formats (`XML`/`SQL`/`JavaScript`/`JSON`) as test programs, and ran them using the existing concolic executor SYMCC. Then, we collected the generated path constraints and used a traditional solver `Z3` to solve them to obtain new test cases. After running the new test cases with four target programs, respectively, we found that numerous test cases (85.3% to 99.9%, as shown in Figure 6.1) are syntactically invalid and cannot pass the parser checking logic. This is reasonable as existing solvers only solve constraints for satisfiability while ignoring the guarantee of syntactic validity of the solution by design. Also, the size of the resulting solution is fixed, restricting the diversity of generated test cases. For example, given a path constraint "*bvsge #x00000039 (concat #x000000 k!95)*" (where 'k!95' is a symbolic
```

variable, collected from parsing implementation in Figure 6.4) which requires the value of the symbol 'k!95' to be large or equal to ASCII 0x39 (i.e., char '*9*'). `Z3` simply solves it to '*9*' and keeps the remaining string unchanged with fixed size as seed input, producing an invalid JavaScript string '*r9turn ...*'. This fact indicates that the test cases generated by existing concolic executors can hardly examine the deeper code regions (e.g., the application logic), significantly hindering the effectiveness of concolic testing. In summary, such a limitation motivates us to investigate the following question: *How can we solve the constraints smartly, i.e., solving constraints for both satisfiability/syntactic validity and making the resulting solutions holding flexible input sizes*?

To answer the question, we design a new LLM-driven constraint solver based on *Solve-Complete* paradigm as shown in Figure 6.2. The idea behind it is simple but effective. Given a path constraint and the marked (two types of marks help *Solve-Complete* paradigm) seed input string as a user prompt, LLMs are prompted to smartly solve the given constraint within two consecutive steps. In step (1), LLMs are asked to *solve* the constraint and use the solution to replace the *Constraint Mask* '[k!n]', where the solution is supposed not only to meet the satisfiability but also possibly comply with the input syntax validity. In step (2), LLMs are required to *complete* the *Flexible Mask* '[xxx]' with a flexible size string which make the whole resulting strings remain valid. Taking the same path constraint includes 'k!95' again, our solver solves it to '*e*', which can be connecting with a remaining string (e.g., '*turn*') to form a syntactically valid JavaScript string (as shown in the LLM's response). To demonstrate the effectiveness of LLM-driven solver, we used it to solve the same path constraints solved by `Z3`, and the results presented in Figure 6.1 show significant improvements (100 times more) in the parser checking pass rate across various formats, helping achieve significant improvement in code coverage.

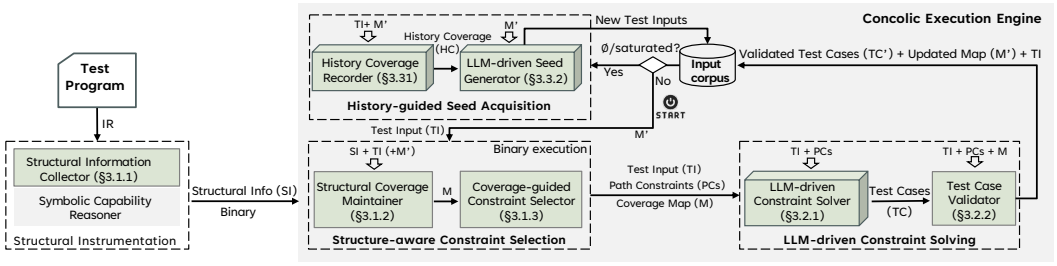


Figure 6.3: High-level design of COTTONTAIL concolic execution engine

6.3 Design of COTTONTAIL

Overview. Figure 6.3 presents the high-level design of COTTONTAIL. Given a test program, COTTONTAIL compiles and instruments it to a binary capable of symbolic reasoning. Despite the facilities for concolic execution, COTTONTAIL enhances instrumentation by collecting structural information and constructs a new program path representation called Expressive Structural Coverage Tree (ESCT) to assist in structure-aware path constraints selection (Section 6.3.1). Later, COTTONTAIL leverages LLM-driven constraint solving with *Solve-Complete* paradigm based on CoT prompts to solve constraints smartly and refine the non-robust results (Section 6.3.2). Finally, if there are no initial seed inputs to set up testing or whenever the testing process gets saturated, COTTONTAIL adopts a history-guided seed acquisition to obtain high-quality seed inputs continuously (see more details in Section 6.3.3).

6.3.1 Structure-aware Path Constraint Selection

This subsection first explains why a new selection strategy is needed and then details how we construct a new coverage map to guide the structure-aware selection.

Why structure-aware path constraint selection?

To perform a practical concolic execution over parsing test programs, we argue that an effective selection strategy should be structure-aware that meets the following *four* requirements:

#1 select a complete (e.g., include as much structural information) representation of structural path constraints;

#2 provide human-readable coverage information at runtime;

#3 exclude redundant structure-agnostic path constraints;

#4 has less chance of missing interesting structural coverage.

Satisfying requirement #1 helps users to have a better understanding of structural paths, #2 is essential to help craft useful inputs either by humans or LLMs at runtime when the testing gets saturated. Compared with branch information from binary code (which contains less semantic information), a human-readable coverage could provide clear and interpretable insights into which branches have been covered and what direction to create new test cases. #3 and #4 together guarantee a better trade-off between testing effectiveness and efficiency. There are three existing strategies to handle the path constraint selection for the general software systems, which are exhaustive search, the Bitmap-based approach [160], and the CSTG-based approach [102], yet they do not comply with all the requirements when handling parsing test programs (see more detailed comparison in the Appendix in the original COTTONTAIL paper). Justified by the above facts, it thus calls for a new path constraint selection that could satisfy all four requirements. In the following subsections, we detail the structural instrumentation and human-readable Expressive Structural Coverage Tree (ESCT) to meet requirements #1 and #2, and the ESCT-guided path constraint selector to comply with requirements #3 and #4.

Structural Instrumentation

Compilation-based concolic execution has shown promising performance in execution speed compared with IR (Intermediate Representation)-based execution, but it inevitably misses many interesting behaviors of the test programs due to the loss of semantics information after compilation [36, 160]. To alleviate this issue, we

```
1  // Parsing logic /* jslex.c */
2  static int jsY_isidentifierpart (int c) {
3      return isdigit (c) // " bvsge #x00000039 (concat #x000000 k!95)"
4      || isalpha(c) || c == '$' ||c == '_' || isalpharune(c);
5  }
6  static int jsY_lexx(js_State *J){
7      while (1) {
8          // ...
9          switch (J->lexchar) {
10             case '(': jsY_next(J); return '(';
11             case ')': jsY_next(J); return ')';
12             case ',': jsY_next(J); return ',';
13              // ...
14         }
15     // ...
16     }
17 }
```

Figure 6.4: Sample parsing implementation code from MuJS

propose to use extra instrumentation on the IR code to collect necessary information. It is worth noting that such instrumentation is crucial for capturing detailed structural information, particularly around complex conditional constructs like *switch-case* statements, because it allows developers and analysis tools to observe precisely which paths of the code are being exercised at runtime. Without instrumentation, it can be difficult to determine whether certain cases or branches within a switch statement are triggered, leading to potential blind spots in testing. To comprehensively cover structural program paths, we design a structural information collector during instrumentation to capture any possible structural paths in the test program. To be specific, our structural instrumentation systematically walks through every instruction in a given function, looking specifically for switch instructions. When it encounters a structure-aware branch (i.e., *switch-case* statements), it records the switch name and its associated case values. Such metadata is then added to a global map that associates each switch statement with its case values. Finally, the instrumentation phase saves all gathered information into a `JSON` file, enabling further analysis of the function's branching structures.

Structural Coverage Tree Maintainer

After collecting structural information, we introduce a new Expressive Structural Coverage Tree (ESCT) to help have a comprehensive representation of structural program paths.

We define ESCT as a hierarchical tree structure represented by a pair $T = (N, E)$, where:

- N is a set of nodes, with a special node called root node.
- $E \subseteq N \times N$ is a set of edges that define parent-child relationships between nodes, representing the calling context.

Each node $n \in N$ may have zero or multiple child nodes connected by edges. Nodes without children are called *leaf* nodes; nodes with children are referred to as *internal* nodes. Each node name is a unique identifier:

fileName_funcName_lineNum_colNum_brType_brId

which consists of several important attributes to represent a unique branch or differentiate between different branches. Those attributes include visiting status (*taken* or *untaken*), visit count (*visit_cnt*), branch type (*brType*), call stack size, and branch id (*brId*) — in *if* statement, it refers to 0 (*then* branch) or 1 (*else* branch); in *switch-case* statement, it represents the constant case value. With the help of such an informative coverage map, users can easily understand the testing process by checking the statistics recorded in the global map.

After defining the ESCT, we manage and update a global coverage tree to help guide the constraint selection. Thus, COTTONTAIL is able to reduce redundant coverage and avoid losing promising code coverage.

ESCT-guided Constraint Selector

The selector consists of two phases of selection: reducing redundant path constraints that do not reduce interesting coverage during single concolic execution, and reducing

```
(vsge #x00000039 (concat #x000000 k!0)
(vsge #x00000039 (concat #x000000 k!1))
....
(vsge #x00000039 (concat #x000000 k!95))
```

Figure 6.5: Duplicated path constraints in parsing logic (those path constraints aim to cover the same branch at Line 3 in Figure 6.4)

path constraints across different runs that have less chance to eliminate interesting code coverage.

First Phase Selection. As aforementioned in Figure 6.4, it is a common practice that the redundant path constraints are collected when every byte in the input string is repeatedly analyzed by a parsing function in a single concolic execution. Therefore, we need to remove we remove the deduplicated path constraints in a single concolic execution. To do so, we maintain a global branch recorder to record branch-constraint mappings during a single concolic execution. Since each element in the record is a unique branch identifier that records the context of the branch. When a branch is encountered and already exists in the recorder, we compare the current constraints and the constraints stored in the branch. If the constraints are only different in the symbolic index (i.e., each byte represented as 'k!n' in constraints, where 'n' indicates the index over the input bytes), we do not select constraints on those branches.

Note that such a constraint deduplication mechanism preserves the soundness of concolic execution by eliminating only redundant path constraints that arise from structurally identical branches applied repeatedly across input positions. As a result, our approach reduces constraint redundancy without omitting meaningful paths, thus preserving the soundness of the analysis (also demonstrated in Table 6.3).

Second Phase Reduction. Different from the first phase, in the second phase, we remove the redundant path constraints based on the newly built coverage tree across different runs. An important problem to handle is how we remove path constraints without affecting the overall performance (i.e., missing potential interesting code coverage). Simply excluding the branches that have been explored tends to miss much interesting coverage as such a strategy does not have a chance to examine the

remaining execution of the current test case or the remaining test cases to make a globally optimal decision [102]. Therefore, we need to be careful when making the selection decision. Inspired by prior work such as KLEE and Mayhem [28, 31, 177], we consider the factors such as untaken branches, frequency of visits, and depth of execution to balance the exploration potential and redundancy. Thus, we propose a new metric node weight to quantify selection priority, defined as follows:

$$Node_{weight} = \alpha \cdot untaken + \beta \cdot visit\_cnt + \gamma \cdot depth \tag{6.1}$$

The *untaken*, *visit_cnt*, and depth are node attributes and α, β, and γ are three parameters to optimize exploration for maximum program coverage. α prioritizes untaken branches, ensuring the discovery of new execution paths and highlighting untested code regions. β focuses on rarely visited nodes, balancing exploration by avoiding overemphasis on frequently traversed paths while paying attention to less common scenarios. γ rewards deeper nodes, encouraging exploration of complex execution paths and uncovering deeply nested bugs or vulnerabilities. Together, these parameters create a balanced strategy that drives efficient and thorough testing.

6.3.2 Smart LLM-driven Constraint Solving

This subsection introduces our *Solve-Compete* paradigm based on CoT prompts and test case validator to refine the unreliable results produced by LLMs.

LLM-driven Constraint Solver

It is important to give a precise and logical prompt if we intend to receive output feedback from LLMs (we show that normal prompts generate worse results in Section 6.4.2). We design a *Solve-Complete* paradigm that utilizes the CoT (Chain of Thought) prompt mechanism. Using CoT prompts instead of normal prompts is advantageous for tasks requiring complex reasoning or multi-step problem-solving. CoT prompts guide the model to think step-by-step, improving accuracy by reducing errors that arise from skipping intermediate steps. It also enhances transparency

by explicitly laying out the reasoning process, making it easier to verify the logic and correctness of the solution. In summary, the systematic reasoning makes CoT prompts ideal for tackling constraint-solving tasks.

The earlier Figure 6.2 illustrates a smart constraint-solving strategy grounded in a *Solve-Complete* paradigm based on CoT prompts, where the LLM is asked first to satisfy symbolic constraints and then complete the output to preserve syntactic correctness. This process is decomposed into two stages: (1) resolving the *Constraint Mask* ('[k!n]') (e.g., 'k!95') by synthesizing a character e that satisfies the path constraint under ASCII semantics, and (2) completing the surrounding code such that the entire string remains valid JavaScript to fill the *Flexible Mask* ('[xxx]'). This dual-stage approach mirrors classical symbolic execution techniques, but is uniquely enhanced by the LLM's ability to generate structurally and contextually coherent code fragments. In contrast to traditional program synthesis pipelines, which often treat constraint solving and code completion as decoupled steps, this strategy tightly integrates reasoning with generative synthesis. The mechanism also aligns with tasks like code infilling, notably benchmarked in CodeXGLUE [125], where models are expected to fill in masked code spans while preserving functional correctness. However, unlike pure statistical infilling, our approach exhibits explicit constraint awareness, solving constraints before code generation, highlighting the potential of LLMs to unify symbolic reasoning with syntax-preserving code completion. This capability unlocks new applications in symbolic execution based input generation that requires both formal constraint satisfaction and natural language-level synthesis.

Test Case Validator

It is well-known that LLMs can not reliably generate expected output and can have hallucinations [33, 172]. Random output might be acceptable for black/grey box fuzzing, as they do not require the robust (i.e., new test inputs will cover new code coverage) results during each iteration. However, for a concolic execution, robustness is one of the essential features that should be guaranteed. Therefore, we need to

Algorithm 7: Test Case Validator

Input: a path constraint *pc*, a branch *br*, a test input from LLM *input*, the coverage tree *g_tree*

Output: original test input *intput* or refined test input *input'*, updated global tree *g_tree'*

```
1  Function TestCaseValidator(pc, br, input, g_tree):
2      res_eva = evalauteConstraint(PC, input)
3      if res_eva == True then
4          updateGlobalTree(g_tree, br)
5          return input, g_tree'
6      else
7          solution = getSolution (pc)
8          input' = refineTestCase (solution, input)
9          updateGlobalTree(g_tree, br)
10         return input', g_tree'
```

handle unreliable results produced from GPT to ensure it follows the soundness guarantee of traditional concolic execution and updates the global ESCT precisely.

Algorithm 7 presents the workflow of the test case validator, which validates or refines test inputs generated by LLMs. The algorithm takes input a path constraint *pc*, a branch *br*, the test input (*input*) produced by LLMs, and the global coverage tree (*g_tree*). It first evaluates whether the input satisfies the path constraint using the function `evaluateConstraint` (Line 2). If the constraint is satisfied, i.e., the returned boolean flag *res_eva* is *True*, the branch *br* is updated in the global tree using `updateGlobalTree` (Line 9), and the algorithm outputs the original test input (*input*) alongside the updated tree. If the constraint is not satisfied, a solution for the path constraint is computed using `getSolution` (Line 7), and a refined test input (*input'*) is generated through function `refineTestCase` (Line 8). Finally, the updated tree (*g_tree'*) is returned along with the refined input. In summary, this algorithm ensures the validity of test inputs while updating the global coverage structure to improve test coverage.

In particular, in `refineTestCase` function, we replace the unreliable solution generated by LLM with the correct solution generated by a traditional solver. By such, even though LLMs produce unreliable outputs, COTTONTAIL could fix them

and refine them to the same output as the ones generated by traditional solvers.

6.3.3 History-guided Seed Acquisition

In this subsection, we detail the strategy to generate initial seeds before testing or alleviate the saturation issue during testing, including the history coverage recorder and the history-driven seed generator. Since the key contributions lie in the generation during testing, we detail the history coverage recorder first in the following.

History Coverage Recorder

It is important to trace the testing history to know *which branches* can be covered by *what test inputs* and *which are uncovered branches* remaining uncovered. By investigating the connection between test input and its covered branches, we could not only understand the underlying processing logic of test programs but also highlight what are missing features within the test inputs. To practically collect the history information, we continue to leverage the benefits of the informative coverage map (i.e., ESCT) to get covered or uncovered branch information.

After collecting history coverage mappings and extracting branch information from the global coverage map, we then use this information to construct CoT prompts for *fresh* seed generation during testing.

LLM-driven Seed Generator

The generator is invoked based on two different timings: initial seed acquisition before testing and fresh seed acquisition during testing.

Initial Seed Acquisition. If there are no interesting seed inputs to set up testing, we define a prompt that help generate high-quality structured seed inputs. since LLMs were trained via tons of code and resources and inspired by many existing studies [57, 203], it is reasonable that LLMs have expert knowledge of what kinds of code have triggered vulnerabilities in bug repositories. Thus, we directly prompt LLMs to

</> System Prompt (with Chain of Thought)
You are a knowledgeable structural **Seed Generator** for JavaScript strings. You will be asked to generate new seed inputs **before** testing or **during** testing. During testing, given <test input, coverage history> map and uncovered branches, please generate a new seed step by step:

1. Compare the covered and uncovered branches.
   - Group by source location and identify where alternative branches exist.
2. Analyze input string and infer which byte(s) most likely influenced the decision at relevant locations.
   - Use the difference between branch IDs as clues to infer.
3. Randomly select one of the options to generate a new input based on above analysis.
   - Option 1: Generate new seed by modifying test inputs from history to explore uncovered branches.
   - Option 2: Generate a fresh input from bug repository to explore uncovered features.
4. Ensure final output is enclosed with ```.

</> User Prompt (before testing)
Please generate a high-quality JavaScript seed input (e.g., code samples from bug repositories) to start testing. The goal is to help cover as much as code coverage and detect new vulnerabilities.

</> User Prompt (during testing)
Given the historical coverage map ```[test1-cov1, test2-cov2 …]``` and untaken branches ```[branch1,branch2 …]```, please generate a new seed input.

Figure 6.6: CoT prompts for LLM-driven seed generation

generate high-quality structured test inputs from existing bug repositories. By such, no manual work will be required to collect historical buggy code examples.

Fresh Seed Acquisition. During testing, if there is no interesting coverage increase (i.e., saturated) after a timeout (i.e., three minutes), by checking the coverage achieved (collected from external tool `gcov`) at runtime. It is straightforward to apply the same strategy used in seed generation to get a fresh seed, but it is ineffective (demonstrated in Section 6.4.2). To make it more effective, we design a creative generation solution using CoT (Chain of Thought) prompts to effectively explore the unexplored branches/features during testing. Such a design is inspired by an interesting behavior investigated by prior studies that a better name can help LLMs better understand the program semantics [76, 199].

Figure 6.6 illustrates the prompts designed to guide a seed generator for `JavaScript` in creative generation of seed inputs for different timings. In particular, the CoT workflow guides an LLM to generate new `JavaScript` seed input through a structured multistep reasoning process during testing. First, the LLM compares covered and uncovered branches, groups them by source location, and identifies divergent execution points. Then, it analyzes input bytes likely responsible for branching decisions using its internal inferring capacities. Finally, it synthesizes

new inputs either by mutating existing test cases to explore specific branches or by drawing from existing bug repositories.

6.4 Evaluation

To evaluate the effectiveness of COTTONTAIL, we aim to investigate the following research questions (RQs):

- **RQ1**: How does COTTONTAIL perform compared with baseline approaches?
- **RQ2**: Can each component contribute to COTTONTAIL?
- **RQ3**: Can COTTONTAIL find new vulnerabilities?

Among these RQs, RQ1 focuses on demonstrating the effectiveness of COTTONTAIL compared with state-of-the-art approaches and investigating whether COTTONTAIL is superior to them. RQ2 conducts comprehensive ablation studies to analyze the significance of individual components or key features within COTTONTAIL. RQ3 assesses COTTONTAIL ability to discover previously unknown vulnerabilities, highlighting its practical value and potential for improving software security.

All experiments were run on a Linux PC with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz x 12 processors and 64GB RAM running Ubuntu 18.04 operating system.

Benchmarks. Table 6.1 presents eight widely tested open-source libraries used for evaluation, including libraries for `XML` (Libxml2/Libexpat), `SQL` (SQLite/UnQLite), `JavaScript` (MuJS/QuickJS), and `JSON` (JSON-C/Jansson), varying in size and popularity. This diverse set of libraries covers a broad range of formats, codebases, and community adoption levels, making it comprehensive for evaluation.

6.4.1 RQ1: Comparison with Baseline Approaches

Comparative Approaches. The following state-of-the-art concolic execution approaches are compared:

- SYMCC [160]: the tool COTTONTAIL built on top of (enable the `Bitmap`-guided

Table 6.1: Open source libraries cross four different formats used in the evaluation (LOC: lines of code; Stars: GitHub stars)

Libraries	Format	Version	LOC	Stars
Libxml2	XML	2.13.5	80.0k	0.6k
Libexpat	XML	2.6.4	14.6k	1.1k
SQLite	SQL	3.47.0	81.3k	7.0k
UnQLite	SQL	1.1.9	22.5k	2.1k
MuJS	JavaScript	1.3.5	10.0k	0.8k
QuickJS	JavaScript	0.7.0	46.4k	1.2k
JSON-C	JSON	0.18	4.7k	3.0k
Jansson	JSON	2.14	5.8k	3.1k

Table 6.2: Line and branch coverage comparison results against existing concolic execution engines SYMCC [160] and MARCO [102]

Fomat	Libraries	SYMCC		SYMCC(¬MAP)		MARCO		MARCO(MC)		MARCO(CFG)		COTTONTAIL	
		Line	Branch	Line	Branch	Line	Branch	Line	Branch	Line	Branch	Line	Branch
XML	Libxml2	5,390	3,820	5,554	3,894	5,411	3,918	5,395	3,907	5,364	3,805	**5,754**	**4,261**
	Libexpat	2,478	1,605	2,430	1,465	2,625	1,837	2,663	**1,870**	2,320	1,123	**3,360**	1,813
SQL	SQLite	17,431	11,138	17,682	11,365	17,624	11,395	18,143	11,752	18,168	11,671	**19,288**	**12,307**
	UnQLite	2,976	1,510	3,137	1,649	3,170	**1,764**	3,237	1,734	3,230	1,720	**3,257**	1,749
JavaScript	MuJS	3,914	2,082	3,968	2,021	4,053	2,233	3,777	2,031	3,800	1,976	**4,356**	**2,277**
	QuickJS	8,205	3,674	8,301	3,631	8,274	3,780	7,154	3,151	6,405	2,591	**11,267**	**5,307**
JSON	JSON-C	996	612	962	558	998	608	1,038	685	963	612	**1,054**	**688**
	Jansson	1,134	671	1,122	675	1,179	677	1,178	676	1,118	646	**1,199**	**719**
	Amount	42,524	25,112	43,156	25,258	43,334	26,212	42,585	25,806	41,368	24,144	**49,535**	**29,121**
	Improvement	14.15%	15.96%	14.78%	15.29%	14.31%	11.10%	16.32%	12.85%	19.74%	20.61%	-	-

path constraints selection by default).

- SYMCC(¬MAP): a variant approach of SYMCC that selects all newly generated path constraints without any guidance.
- MARCO [102]: a recent concolic execution engine that constructs CTSG to select path constraints from a global view.
- MARCO(MC): A variant of MARCO that adopts Markov Chain modeling in CSTG.
- MARCO(CFG): A variant of MARCO that applies the CFG-directed searching algorithm in CSTG.

We select SYMCC and SYMCC(¬MAP) as we built COTTONTAIL on SYMCC. MARCO is a recent approach proposed recently, and their experiments show that the two variant approaches (i.e., MARCO(CFG) and MARCO(MC)) could outperform MARCO in some cases, so we also include them.

Running Setting. We run each baseline approach for 12 hours, as suggested in

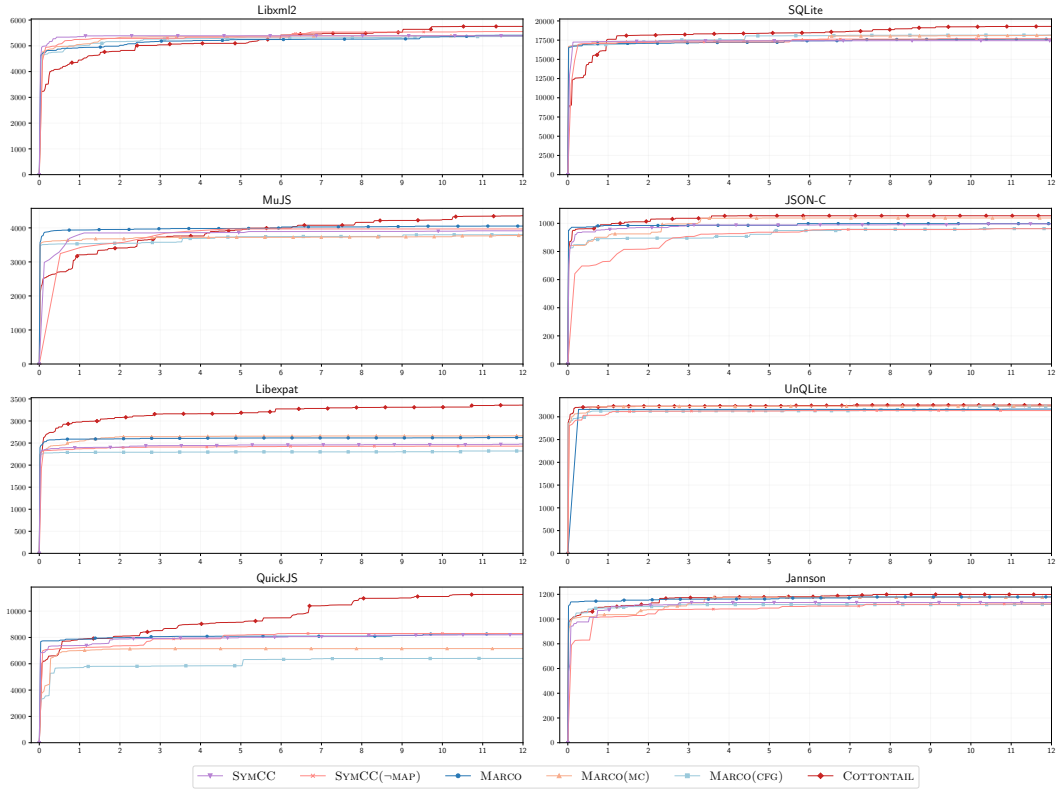


Figure 6.7: Line coverage comparison among COTTONTAIL and baseline approaches in 12 hours (*x-axis* indicates line coverage while *y-axis* represents the time)

the same settings [82, 115]. For the approaches that involved random choices, we repeated running them five times and reported the median results. To further conduct a fair comparison, we use the seed input used in MARCO and launch each tool with the same seeds. To help detect possible program issues, we compile the target program built with `AddressSan` [171] and use it as a test oracle to detect memory issues. In particular, although COTTONTAIL does not require pre-collected seed inputs to set up, for a fair comparison, we disable the seed acquisition contribution in COTTONTAIL. To clarify, the description of COTTONTAIL in this subsection refers to COTTONTAIL(INIT+¬SGEN), the variant version of COTTONTAIL where the same initial seeds as the baselines are used and without new seed generation (please check different versions of COTTONTAIL in Section 6.4.2).

Metrics. We use code coverage, including line and branch coverage measured by the external tool `gcovr` to compare the effectiveness of different approaches.

Results. Table 6.2 provides a comprehensive comparison results achieved by com-

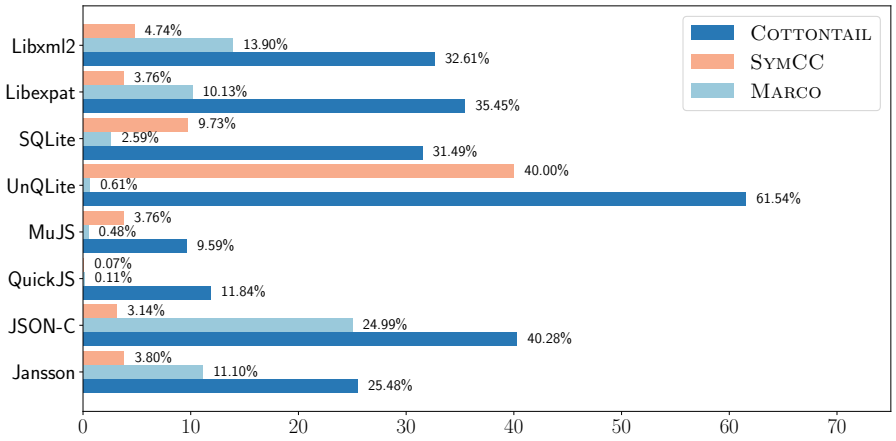


Figure 6.8: Comparison results of parser checking passing rate (% in *y-axis*) against SYMCC and MARCO

parative approaches. Notably, COTTONTAIL significantly achieves a superior code coverage, from 14.15% to 19.74% in terms of line coverage and 11.10% to 20.61% in terms of branch coverage over comparative approaches on average, demonstrating better code coverage capability. There are a few cases that COTTONTAIL covers less branch coverage than MARCO in Libexpat and MARCO(MC) in UnQLite. This is because MARCO designs the path selection based on random sampling, so a few more branch coverages may be expected. Figure 6.7 shows the code covering trend across running time, we can see that COTTONTAIL performs better than comparative approaches within 12 hours. Overall, the results shown in Table 6.2 and Figure 6.7 demonstrate superior code coverage capabilities compared with baseline approaches.

To have a better understanding of why COTTONTAIL is superior, we further analyze the validity of generated test cases among SYMCC, MARCO, and COTTONTAIL. Since SYMCC and MARCO produced millions of test cases in 12 hours, we ran them in another 1-hour setting for the same running time. Figure 6.8 presents a detailed comparative analysis of parser checking passing rates for three comparative tools. From the figure, we can observe that COTTONTAIL consistently performs better in several critical libraries: it achieves a significant 32.61% passing rate in Libxml2, notably higher than SYMCC's 4.74% and MARCO's 13.90%. The overall results suggest that the increased number of valid test inputs help yield better code coverage.

It is worth noting that on many benchmarks (e.g., Libexpat and SQLite), both SYMCC and MARCO exhibit early saturation in their coverage progress. For example, SYMCC quickly reaches a plateau within one hour and shows minimal improvement thereafter, indicating limited capability in uncovering additional program behaviors beyond its initial exploration. The final coverage achieved by both techniques remains substantially lower than that of other approaches, suggesting that their underlying strategies are less effective in sustaining exploration over time. Thus, fresh seeds are required to change the saturation and make the testing more effective.

Answer to RQ1: COTTONTAIL significantly improves the state-of-the-art approaches in terms of line and branch coverage, demonstrating the effectiveness of COTTONTAIL in generating highly structured test inputs.

6.4.2 RQ2: Ablation Studies

This subsection presents the carefully designed methodologies to evaluate the impact of the newly designed components.

RQ2.1: How effective is the ESCT-guided path constraint selection?

As mentioned in Section 6.3.1, we design two phases to remove redundant path constraints across single concolic execution and in-between runs. We here evaluate how many path constraints were filtered out in the two phases (e.g., single or in-between concolic execution), comparing with existing selection strategies (i.e., select all without map (`Nomap`) and guided by `Bitmap`).

To do so, we run the seed input and terminate it after the first iteration is done to evaluate the effectiveness in the first phase. Then, we run the seed input within two iterations to evaluate the effectiveness of the in-between runs. Finally, we count the number of total path constraints and code coverage achieved by different selection strategies in the two phases.

The results in Table 6.3 demonstrate the effectiveness of ESCT in guiding path

Table 6.3: Results of path constraint selector design in COTTONTAIL

Libraries	Comparison of Path Constraints Selection — First Phase						Comparison of Path Constraints Selection — Second Phase					
	NoMap		Bitmap		ESCT (ours)		NoMap		Bitmap		ESCT (ours)	
	No.pc	*Cover.*	*No.pc*	*Cover.*	*No.pc*	*Cover.*	*No.pc*	*Cover.*	*No.pc*	*Cover.*	*No.pc*	*Cover.*
Libexpat	461	1,441	187	1,441	159	1,441	6,149	1,684	615	1,681	685	1,687
SQLite	401	11,331	191	11,331	137	11,331	3,618	11,565	313	11,584	687	11,568
MuJS	486	1,404	185	1,404	273	1,404	7,688	2,758	653	2,666	2,550	2,743
JSON-C	223	566	106	566	67	566	7,024	899	355	861	926	887

* The number *A(B)* in the table represents the number of path constraints (*No.pc*) collected in the first iterative run (A) and the line coverage (*Cover.*) achieved (B). We omitted the comparison with CSTG as it does not follow the iteration working style.

constraint selection across both testing phases. In the first phase, ESCT significantly reduces the number of collected path constraints compared to both `NoMap` and `Bitmap` (e.g., 67 vs. 223 and 106 in JSON-C; 137 vs. 401 and 191 in SQLite), while maintaining identical line coverage, indicating that ESCT effectively filters redundant constraints without sacrificing exploration. In the second phase, ESCT continues to show a substantial reduction in the number of path constraints relative to `NoMap` (e.g., 926 vs. 7,024 in JSON-C; 687 vs. 3,618 in SQLite), yet it retains more path constraints than `Bitmap`, enabling it to achieve better coverage than `Bitmap` and comparable or even slightly improved coverage over `NoMap` in most benchmarks. These results highlight ESCT's superiority in balancing structure-aware constraint selection and exploration.

This is reasonable as AFL's `Bitmap` tends to miss interesting coverage due to hash collisions, limited granularity, and lack of path sensitivity, all of which cause distinct behaviors to appear identical, reducing fuzzing effectiveness [80, 133], while using `Nomap` will lead to the inefficient testing.

RQ2.2: How effective are the CoT prompts in *Solve-Complete* paradigm?

We have shown the superior performance of LLM-driven constraint solving in Figure 6.1. To better understand the benefits of the CoT prompt, we compare COTTONTAIL with COTTONTAIL(NORMPRO), a variant approach of COTTONTAIL that removes the CoT prompt.

The results in Figure 6.9 highlight the effectiveness of Chain-of-Thought (CoT)

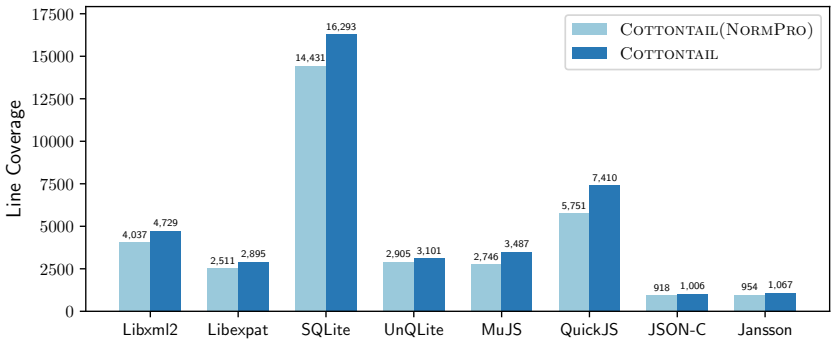


Figure 6.9: Comparison results of normal and CoT prompts for constraint solving

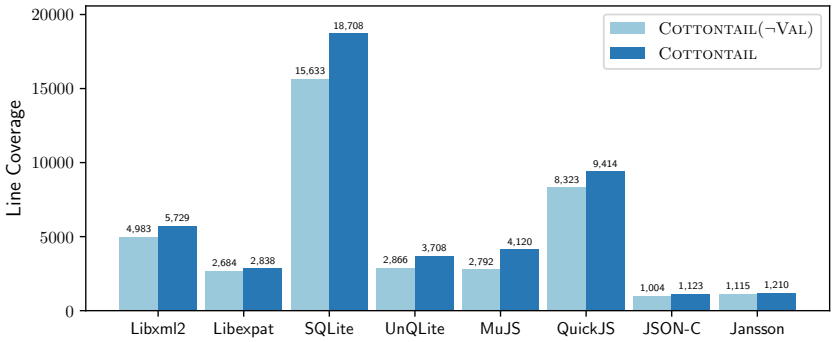


Figure 6.10: Comparison results of w/ or w/o test case validator

prompts in improving constraint solving for line coverage across a diverse set of libraries. In all cases, CottonTail with CoT prompts (dark bars) achieves higher coverage than the variant using normal prompts (light bars), with particularly notable improvements observed in SQLite, QuickJS, and MuJS. The substantial gain in SQLite, where coverage increases from approximately 13,000 to over 15,000 lines, underscores how step-by-step reasoning enables the solver to navigate complex constraint spaces better. These results suggest that CoT prompts provide a significant advantage in guiding the model's symbolic reasoning process, leading to more effective exploration and ultimately higher coverage.

RQ2.3: How effective is the test case validator?

Since it is critical to guarantee the soundness of test cases produced by concolic execution engines, we need to check if the validator designed in CottonTail works. To have a fair comparison, we measure the line coverage achieved by CottonTail

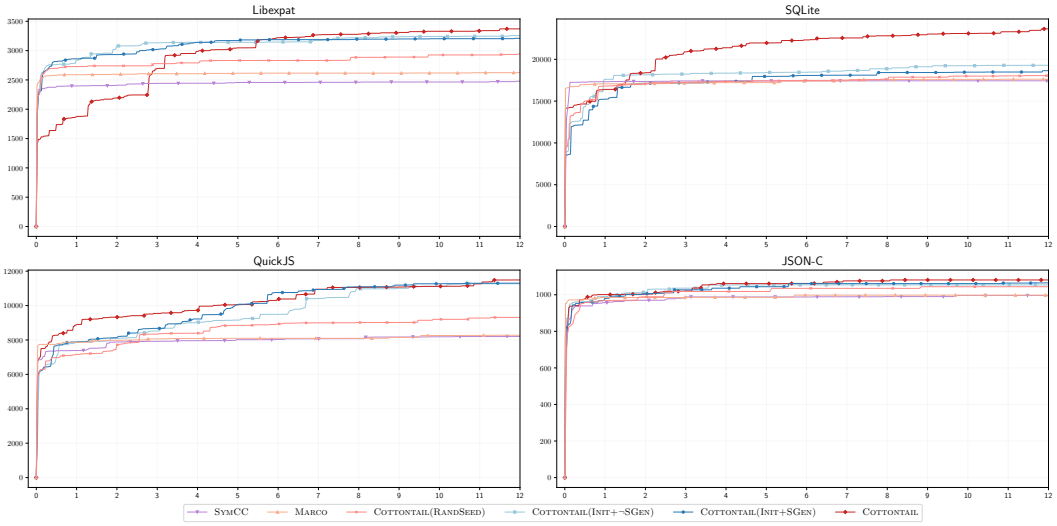


Figure 6.11: Line coverage comparison among COTTONTAIL and variant approaches in 12 hours (*x-axis* indicates line coverage while *y-axis* represents the time)

and COTTONTAIL(¬V) (a variant approach of COTTONTAIL that removes the test case validator) under the same setting.

Figure 6.10 presents the line coverage achieved by COTTONTAIL and its variant COTTONTAIL(¬V) across various libraries. COTTONTAIL consistently outperforms or matches its counterpart, with significant improvements in larger libraries like SQLite, QuickJS, and Libxml2, where it achieves substantially higher line coverage. Overall, the results presented in the figure demonstrate that COTTONTAIL configuration is more effective, especially in complex codebases, highlighting the importance of test case validation and refinement for comprehensive coverage. This is reasonable as our refinement, designed in Algorithm 7 guarantees the newly generated test cases are expected to explore different program paths. Without the validator, COTTONTAIL(¬V) can be treated as a special variant of smart grey-box fuzzing without the guarantee of systematic program analysis.

RQ2.4: How effective is the history-guided seed acquisition?

To have a better understanding of the contribution of seed acquisition, we design the following variant approaches.

- COTTONTAIL(RANDSEED): This variant performs random seed generation in-

stead of history-guided generation.

- COTTONTAIL(INIT+¬SGEN): This variant is ran with initial seed inputs and disables seed generation.
- COTTONTAIL(INIT+SGEN): This variant is ran with initial seed inputs and generates new seeds when the test process gets saturated (no new coverage increased in three minutes).
- COTTONTAIL: This is the *default* version of our approach, which is run without any initial seed inputs, enabling the history-guided seed acquisition component.

We also include the baseline approaches SYMCC and MARCO from RQ1 to provide a comprehensive comparison. We select each benchmark over the format and run it for 12 hours to compare its line coverage achieved. Figure 6.11 presents the details experimental results.

Contribution of Guided Seed Generation. By comparing the results of COTTONTAIL with COTTONTAIL(RANDSEED), we can observe that the history-guided seed acquisition is superior to random seed generation. In all selected benchmarks, COTTONTAIL consistently outperforms its random-seed variant, achieving noticeably higher line coverage throughout the 12-hour window. For instance, in SQLite, COTTONTAIL reaches over 23,000 lines covered, whereas COTTONTAIL(RANDSEED) stalls below 19,000. These results demonstrate that historical execution feedback could guide seed acquisition significantly by prioritizing seeds with higher potential for new code coverage.

Contribution for Changing Testing Saturation. We conduct two sets of comparative analyses to investigate it. First, by comparing COTTONTAIL(INIT+¬SGEN) with COTTONTAIL(INIT+SGEN), we can understand how this component boosts testing when the initial seeds are available. The results show that enabling seed generation significantly improves coverage when an initial seed is available, highlighting the importance of dynamic seed expansion. In particular, we can observe that baseline approaches usually get saturated within 2 hours, and it could be interesting to know

how many lines can be covered after the saturation point. As a result, COTTONTAIL(INIT+SGEN) covers 101, 376, 806, and 21 more lines over Libexpat, SQLite, QuickJS, and JSON-C than COTTONTAIL(INIT+¬SGEN) within the latter 10 hours, indicating that enabling the seed generation will continuously increase the coverage, unlike saturating the seed generation. Second, by comparing COTTONTAIL —including its variant configurations—with SYMCC and MARCO, we can find out how this component works when there are no seeds. The results show that COTTONTAIL maintains superior performance even as coverage begins to saturate, demonstrating its effectiveness in exploring deeper program states under constrained conditions.

Answer to RQ2: By conducting carefully designed ablation studies, our results demonstrate the positive contribution of the newly designed components, including structure-aware constraint selection, LLM-driven constraint solving, and history-guided seed acquisition.

6.4.3 RQ3: Vulnerability Detection Capability

Details of Vulnerabilities Detected

To evaluate the practical vulnerability detection capability of COTTONTAIL, we run it (using the setting of the variant approach COTTONTAIL(INIT+¬SGEN) for a fair comparison) and two baseline approaches in 12 hours and count the number of new vulnerabilities detected. During the experiments, COTTONTAIL found 6 previously unknown vulnerabilities across three testing subjects and reported them to developers. The vulnerabilities with their subject, version, short description, and report status are listed in Table 6.4. These bugs involved heap memory leaks, buffer overflows, and stack overflows, with potential risks such as resource exhaustion, arbitrary code execution, or denial of services. Among the detected issues, 4 out of 6 have been fixed when submitting the paper (six new CVE IDs have been assigned), highlighting the practical impact of COTTONTAIL in improving software security.

Comparison with Existing Approaches. Existing approaches failed or take too much

Table 6.4: New Vulnerabilities Detected by COTTONTAIL

ID	Subject	Description	Status	CVE-Assigned
#1	MuJS	Memory leak	Fixed	CVE-2024-55061
#2	MuJS	Heap overflow	Fixed	CVE-2025-26082
#3	QuickJS	Stack overflow	Fixed	CVE-2024-13903
#4	QuickJS	Stack overflow	Fixed	CVE-2025-26081
#5	UnQLite	Global overflow	Reported	CVE-2025-26083
#6	UnQLite	Heap overflow	Reported	CVE-2025-3791

time to detect it due to a structure-agnostic (or heavy) path constraints selection strategy or limited constraint-solving capabilities. To be specific, MARCO can only detect vulnerability #5. MARCO misses the other five vulnerabilities due to the limited path exploration and heavy scheduling on selecting nodes in CSTG. For example, when testing MuJS, we found that MARCO takes 3.2 out of 12 (26.67%) hours of computing time to schedule and select an optimal path constraint for solving. SYMCC can only detect four (#1, #3, #5, and #6) of them and misses others due to overly aggressive constraint selection and restricted constraint-solving capabilities.

Case Study

To have a better grasp of the new vulnerabilities, we present a case study to justify the difficulties faced by existing approaches and why COTTONTAIL is superior.

Figure 6.12 (a) shows the vulnerable function, and Figure 6.12 (b) shows the seed input and vulnerability triggering test input generated by COTTONTAIL. The issue occurs when `Ap_sort_cmp` (Line 2 in Figure 6.12 (a)) analyzes the ill-defined comparator ("*function(x,y){return y;}*" shown in Line 4 in Figure 6.12 (b)) in the vulnerability triggering input. The unexpected comparator causes `Ap_sort_cmp` to access invalid indices, i.e., *id_a* in the array during sorting. After invalid accessing, directly dereferencing the invalid pointer (*val_a*) leads to a heap overflow. In short, the unexpected return value from the `sort` function in the test input causes a heap overflow in MuJS's implementation. Given the seed input[4], to find a new test input

[4] https://github.com/unifuzz/seeds/blob/master/general_evaluation/mujs/sort.js

```
1  // Application logic (buggy function) /* jsarray.c */
2  static int Ap_sort_cmp(js_State *J, int idx_a, int idx_b){
3      js_Object *obj = js_tovalue (J, 0)->u.object;
4      if (obj->u.a.simple) {
5          js_Value *val_a = &obj->u.a.array[idx_a];
6          js_Value *val_b = &obj->u.a.array[idx_b];
7          int und_a = val_a->t.type == ...; // heap-overflow
8      // ...
9      }
10 }
```

(a) buggy function that triggers a new heap-overflow vulnerability[5] detected by COTTONTAIL.

```
1 // LLM generated test input
2 c = 30000; a = [];
3 for (i = 0; i < 2 * c; i += 1) {a.push(i%c);}
4 a.sort(function (x, y) { return y;}); print(a[100]);
```

(b) Seed input and vulnerability trigger generated by COTTONTAIL (the highlighted strings are from LLMs).

Figure 6.12: Vulnerable function and vulnerability triggering input in Case Study

to trigger the overflow, a testing engine should find a way to construct an ill-defined `sort` function that returns an unexpected value. The efficient way is to negate the program constraint that requires changing the bytes after the 94th byte '*r*' in `sort` function to a valid `return` statement that returns an unexpected value.

Due to structure-agnostic path constraints selection and limited constraint solving, MARCO [102] produced 26,575 test inputs (99.9% invalid) and failed to generate a trigger in 12 hours. SYMCC finds a trigger after 535th iterations while SYMCC($\neg$MAP) after $1{,}811^{th}$ iterations of constraint solving. In summary, while existing concolic execution techniques can negate that branch, the resulting input is likely syntactically invalid and requires extra work by the concolic engine to pass the parser checks and generate the syntactically valid input. It is worth noting that although they could finally generate a test case to trigger this vulnerability, they may miss important corner cases due to limited computing resources. In contrast, benefiting from advanced structure-aware path constraint selection and smart constraint solving, COTTONTAIL detects this issue faster within only a significant fewer iterations (i.e., 55th).

[5] https://github.com/ccxvii/mujs/issues/193.

Answer to RQ3: COTTONTAIL is able to detect many new vulnerabilities, showing a capable practical vulnerability detection capability.

6.5 Discussion

6.5.1 Implications

1) Potential in Detecting Other Bugs. We have shown that the highly structured test inputs could detect new memory-related vulnerabilities in RQ3. The test cases generated by COTTONTAIL could have a good potential to detect other types of bugs (such as parsing or semantic bugs), benefiting from the higher parsing checking passing rate. To detect more types of bugs, extra effort may be made to construct well-defined test oracles. To support our claim, we construct a simple test oracle by differential testing of JSON libraries to detect parsing issues, where we define that a potential bug is found if two parsers behave differently over the same test input. Since potential bugs can be false positives, as different parsers may be implemented in different standards (e.g., RFC 4627 for Jansson or RFC 7159 for JSON-C), new strategies must be applied to reduce such false positives caused by inconsistent standards. We manually analyzed a few of the potential issues and found a parsing bug[6] in JSON-C libraries. The bug is caused by an incomplete handling of control characters. Developers have confirmed and fixed this issue.

2) Potential in Practical Adoption. We believe COTTONTAIL can also have substantial potential to be applied in practical systematic white-box testing, such as SAGE [87] for the following two reasons. First, the path constraints that are worth solving are significantly reduced. As shown in Figure 6.3, the newly designed ESCT-based path constraints can eliminate many redundant path constraints, saving testing time in practice. Second, the cost of invocation of API is pretty low and can be applied within both closed-source and open-source LLMs. We use the `GPT-4o-mini` as our base

[6] https://github.com/json-c/json-c/issues/887

LLM, which is an affordable, cheap model ($0.150 / 1M tokens). Another LLMs such as `GPT-4o` (higher intelligence closure model), `GPT-4.1-nano` (fastest, most cost-effective GPT-4.1 model released on 14/04/2025), or `DeepSeek-V3` (cheap and open source model released on 20/01/2025) can also be easily integrated within COTTONTAIL[7]. Third, the potential of *Solve-Complete* paradigm for constraint solving can be further improved via advanced solutions. During our experiments, we found that when using the CoT prompts, it would be more beneficial to combine expert knowledge or more advanced CoT [221] into the completion phase. Our limited knowledge presented in Figure 6.2 has already shown significant enhancement for high-demand test input generation.

6.5.2 API Costs for Running Experiments

The average invocation of GPT at 816 calls per subject, with an average cost of 0.78 USD per hour while using `GPT-4o-mini` model, demonstrating that the cost of using GPT APIs for constraint solving is relatively low. Traditional methods of constraint solving are limited by the solving capabilities as the aforementioned in previous sections, they produce a large amount of invalid test cases that have limited contribution to the testing effectiveness for generating highly structured test inputs, although they are faster. We believe the response time and robustness of LLM could be improved to further facilitate the test input generation capabilities.

6.5.3 Threats to Validity

As with any empirical study, our findings and conclusions are subject to several potential threats to validity. T The first concerns external validity, which relates to the generalizability of our results. As the subject of our study, we only selected SYMCC and MARCO and their variants, the state-of-the-art approaches for concolic execution. As objects of our study, we selected eight widely tested open-source

[7]Detailed experimental results are presented in the Appendix in the original COTTONTAIL paper.

libraries covering diverse domains, including XML, SQL, JavaScript, and JSON, which vary in size and popularity. While we cannot claim that our findings apply to all software programs, we believe these subjects are representative of a broad spectrum of real-world applications. Another external validity concern driven by the use of LLMs is the risk of data leakage or memorization by LLMs. We believe this is unlikely, as the constraint-solving process in COTTONTAIL is unconventional and unique, making memorization improbable.

The second threat involves internal validity, referring to the extent to which the evidence supports the causal relationships claimed in our study. LLMs are known to exhibit the hallucination problem, generating outputs that may lack grounding in reality. However, COTTONTAIL addresses this issue by proposing a test case validator to validate and refine the generated test cases. To further reduce the influence of randomness, we also repeated each experiment five times and reported the median outcomes for approaches that involve random decisions. To assess the impact of various hyperparameters on our approach's effectiveness, we performed extensive ablation studies (in RQ2 and the Appendix of the original COTTONTAIL paper).

6.5.4 Limitations

COTTONTAIL has a few limitations. First, COTTONTAIL's effectiveness is limited by the completeness of the fuzzing driver. It is well-known that writing an effective fuzzing driver can be a challenging and time-consuming process. We plan to leverage the advanced technique [127] to mitigate the limitation in the future. Second, as a source-code-based concolic execution, the current version of COTTONTAIL can only work for the test program whose source code is available. If only the binary of the target program is available, our approach cannot be directly applied. We plan to further transfer the same idea to SYMQEMU [161], a binary concolic execution that shares the same idea of SYMCC, to alleviate the limitation.

6.6 Summary

We presented COTTONTAIL, a new LLM-driven concolic execution engine to generate highly structured test inputs for parsing testing. COTTONTAIL's novelties lies in the design of structure-aware path constraint selection to select interesting path constraints that are worth exploring, LLM-driven constraint solving to smartly produce test cases that not only stratify the path constraints but also align with syntax rules, and history-guided seed acquisition to generate new seed inputs whenever the engine starts testing or the testing process saturated. We compared COTTONTAIL with state-of-the-art concolic execution engines, and the results demonstrate the superior performance of COTTONTAIL in terms of code coverage and vulnerability detection capability. Our study has shown promising potential in combining traditional program analysis with LLMs, calling for more advanced proposals combining LLMs to improve software security.

Chapter 7

Conclusion and Future Work

7.1 Summary of Contributions

In this dissertation, we present one new memory model (SYMLOC), two new path exploration strategies (FASTKLEE and VITAL), and one new test input generation solution (COTTONTAIL) to boosting symbolic execution for vulnerability detection.

For the new memory modeling, we explore the problem of how can we provide a more complete modeling of dynamic memory allocations for facilitating memory error detection. To address the problem, we propose SYMLOC, a symbolic execution-based approach that integrates address symbolization, a symbolic-concrete memory map, and automatic tracking of symbolic memory locations.

For the efficient (i.e., faster) symbolic execution, we explore the problem of how can we reduce redundant bound checking of type-safe pointers during IR code interpretation to speed up vulnerability detection. To alleviate the problem, we propose FASTKLEE, a symbolic execution engine that leverages a type inference system to classify pointer types and applies customized memory operations to selectively perform bound checking only on unsafe pointers. For the effective (i,e., vulnerability-oriented) path exploration, we explore the problem of how can we effectively guide symbolic execution towards the paths that are likely to contain vulnerabilities. To mitigate the problem, we propose VITAL, a type-unsafe pointer-

guided Monte Carlo Tree Search (MCTS) strategy that prioritizes execution paths containing more unsafe pointers, leveraging the hypothesis that such paths are more likely to be vulnerable.

For the new test input generation, we explore the problem of how can we generate highly structured test inputs to effectively detect vulnerabilities in parsing program. To resolve the problem, we propose COTTONTAIL, a new concolic execution engine that leverages Large Language Model (LLM) along with precise program semantics to generate constraint-satisfiable and syntactically valid test cases.

7.2 Future Work

This dissertation has made some progress in addressing the challenges of memory modeling, path exploration, and test case generation in symbolic execution, yet certain limitations remain. For example, while SYMLOC can help detect more memory issues, it still struggles with subtle pointer aliasing and complex data structures, making the incomplete detection of important issues. Also, path exploration can be hindered by state explosion in large programs in VITAL when the vulnerable patterns are unknown. Given these opportunities and remaining challenges, we propose several future research directions to further enhance vulnerability detection, including refining memory modeling strategies, optimizing path exploration techniques, and improving test case generation capabilities through deeper integration with LLMs.

LLM-Guided Symbolic Path Prioritization. Symbolic execution suffers from path explosion, making it challenging to prioritize paths that lead to vulnerabilities. LLMs can be leveraged to predict high-risk execution paths by analyzing symbolic constraints and past vulnerability patterns as well as possible unknown patterns. By assigning vulnerability likelihood scores, an LLM-enhanced symbolic execution engine can prune unproductive paths and focus on those more likely to expose security flaws, improving both efficiency and effectiveness in vulnerability discovery.

LLM-Augmented Constraint Solving for Security Testing. Constraint solvers are

a major bottleneck in symbolic execution, especially for complex security analysis. LLMs can assist SMT solvers by generating approximate solutions, simplifying constraints, and identifying potential timeouts early. By learning from past symbolic execution traces, LLMs can help resolve constraints faster, making symbolic execution more scalable for real-world vulnerability detection.

Automated Vulnerability Signature Extraction. Manually crafting vulnerability signatures is tedious and error-prone. An LLM-enhanced symbolic execution system can automatically extract and generalize vulnerability patterns from execution traces leading to crashes, memory leaks, or privilege escalations. By comparing discovered vulnerabilities with known CVEs, the system can generate natural-language reports to aid security analysts in classification and triage, enabling faster and more automated vulnerability detection.

AI-Assisted Exploitability Analysis Using Symbolic Execution and LLMs. Not all detected crashes are exploitable, requiring manual analysis to determine real security risks. LLMs can automate crash triage, exploitability reasoning, and proof-of-concept exploit generation by combining symbolic execution insights with learned vulnerability patterns. This approach bridges the gap between bug discovery and exploitability assessment, making security testing more actionable and reducing the need for extensive manual analysis.

In short, many challenges remain unresolved and would be interesting to explore. We hope this discussion on future research directions will inspire further exploration of how symbolic execution and large language models can be effectively combined to enhance software security for large-scale software systems.

Bibliography

[1] Implementation of memmove in Apple Open Source. https://opensource.apple.com/source/network_cmds/network_cmds-481.20.1/unbound/compat/memmove.c.auto.html. (Assessed on 01/02/2025).

[2] M. T. Aga and T. Austin. Smokestack: Thwarting dop attacks with runtime stack layout randomization. In *The International Symposium on Code Generation and Optimization (CGO)*, pages 26–36, 2019.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[4] C. S. Analyzer. A source code analysis tool that finds bugs in c, c++, and objective-c program. http://clang-analyzer.llvm.org. (Assessed on 01/02/2025).

[5] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 367–381, 2008.

[6] Apple. Tips for Allocating Memory. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html#//apple_ref/doc/uid/20001881-CJBCFDGA. (Assessed on 01/02/2025).

[7] I. A. Astrakhantseva, R. G. Astrakhantsev, and A. V. Mitin. Randomized C/C++ dynamic memory allocator. *Journal of Physics: Conference Series*, 2001(1):1–6, 2021.

[8] A. Avgerinos. *Exploiting trade-offs in Symbolic Execution for Identifying Security Bugs*. PhD thesis, Carnegie Mellon University, 2014.

[9] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communication of ACM*, 57(2):74–84, 2014.

[10] G. J. Badros and D. Notkin. A framework for preprocessor-aware c source code analyses. *Software: Practice and Experience*, 30(8):907–924, 2000.

[11] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23, 2004.

[12] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Survey*, 51(3), 2018.

[13] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.

[14] S. Bauer, P. Cuoq, and J. Regehr. Deniable backdoors using compiler bugs. *International Journal of PoC—— GTFO, 0x08*, pages 7–9, 2015.

[15] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communication of ACM*, 53(2):66–75, 2010.

[16] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[17] M. Böhme, L. Szekeres, and J. Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 1621–1633, 2022.

[18] I. Bojanova and C. E. C. Galhardo. Bug, fault, error, or weakness: Demystifying software security vulnerabilities. *IT Professional*, 25(01):7–12, 2023.

[19] T. Brennan, S. Saha, T. Bultan, and C. S. Păsăreanu. Symbolic path cost analysis for side-channel detection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 27–37, 2018.

[20] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. T. Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 505–521, 2018.

[21] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[22] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *The International Symposium on Code Generation and Optimization (CGO)*, pages 213–223, 2011.

[23] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: a binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, page 463–469, 2011.

[24] F. Busse, M. Nowack, and C. Cadar. Running symbolic execution forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, page 63–74, 2020.

[25] J. Caballero and Z. Lin. Type inference on executables. *ACM Computing Survery*, 48 (4), 2016.

[26] C. Cadar and T. Kapus. Measuring the coverage achieved by symbolic execution. http://ccadar.blogspot.com/2020/07/measuring-coverage-achieved-by-symbolic.html. (Assessed on 01/02/2025).

[27] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communication of ACM*, 56(2):82–90, 2013.

[28] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *The USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[29] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2), 2008.

[30] S. Cha, M. Lee, S. Lee, and H. Oh. Symtuner: Maximizing the power of symbolic execution by adaptively tuning external parameters. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, page 2068–2079, 2022.

[31] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy(S&P)*, pages 380–394, 2012.

[32] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, pages 725–741, 2015.

[33] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3), Mar. 2024. ISSN 2157-6904. doi: 10.1145/3641289. URL https://doi.org/10.1145/3641289.

[34] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 559–572, 2010.

[35] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang. Learning to accelerate symbolic execution via code transformation. In *The European Conference on Object-Oriented Programming (ECOOP)*, volume 6, pages 1–27, 2018.

[36] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin. SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis. In *31st USENIX Security Symposium (USENIX Security)*, pages 2531–2548, 2022.

[37] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search . In *IEEE Symposium on Security and Privacy (S&P)*, pages 711–725, 2018.

[38] P. Chen, J. Liu, and H. Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 499–513, 2019.

[39] Y. Chen and X. Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1707–1722, 2019.

[40] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1580–1596, 2020.

[41] Z. Chen, C. Wang, J. Yan, Y. Sui, and J. Xue. Runtime detection of memory errors with smart status. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 296–308, 2021.

[42] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Notice*, 46(3):265–278, 2011.

[43] M. Cho, S. Kim, and T. Kwon. Intriguer: Field-level constraint solving for hybrid fuzzing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 515–530, 2019.

[44] F. Cicalese, B. Keszegh, B. Lidickỳ, D. Pálvölgyi, and T. Valla. On the tree search problem with non-uniform costs. *Theoretical Computer Science*, 647:22–32, 2016.

[45] C. Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues, 2024. URL https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[46] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen. Pearson correlation coefficient. *Noise Reduction in Speech Processing*, pages 1–4, 2009.

[47] J. Cohen. Contemporary automatic program analysis. https://www.blackhat.com/docs/us-14/materials/us-14-Cohen-Comtemporary-Automatic-Program-Analysis.pdf. (Assessed on 01/02/2025).

[48] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the Sixth Conference on Computer Systems*, pages 315–328, 2011.

[49] E. Coppa, D. C. D'Elia, and C. Demetrescu. Rethinking pointer reasoning in symbolic execution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 613–618, 2017.

[50] Cppcheck. A tool for static c/c++ code analysis. http://cppcheck.sourceforge.net/. (Assessed on 01/02/2025).

[51] W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[52] C. Csallner and Y. Smaragdakis. Check'n'crash: Combining static checking and testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 422–431, 2005.

[53] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, pages 233–247, 2012.

[54] CVE-2022-0667. Assertion failure on delayed ds lookup. https://kb.isc.org/docs/cve-2022-0667. (Assessed on 01/02/2025).

[55] DataGuard. Solution for getting the exact mapping between un-linked ir and linked ir. https://github.com/Lightninghkm/DataGuard/issues/2. (Assessed on 01/02/2025).

[56] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[57] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 423–435, 2023.

[58] G. Developer. Confirmation of the bug in Case 1, 2024. URL https://debbugs.gnu.org/cgi/bugreport.cgi?%20msg=10;bug=65269#10.

[59] G. Developers. Implementation of memmove in Glibc. https://github.com/lattera/glibc/blob/master/string/memmove.c. (Assessed on 01/02/2025).

[60] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 144–157, 2006.

[61] K. O. Document. Instruction for building gnu coreutils. http://klee.github.io/tutorials/testing-coreutils/, . (Assessed on 01/02/2025).

[62] K. O. Document. Instruction for selecting running options. http://klee.github.io/docs/coreutils-experiments/, . (Assessed on 01/02/2025).

[63] B. Dolan-Gavitt. KLEE may miss use-after-free in call to Libc function. https://github.com/klee/klee/issues/1434. (Assessed on 01/02/2025).

[64] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (S&P)*, pages 110–121, 2016.

[65] S. Dong, O. Olivo, L. Zhang, and S. Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 205–215, 2015.

[66] J. M. Dongge Liu and O. Chang. Fuzz target generation using llms. https://google.github.io/oss-fuzz/research/llms/target_generation/.

[67] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *IEEE S & P Workshops*, pages 73–87, 2015.

[68] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *IEEE Symposium on Security and Privacy (S&P)*, pages 73–87, 2015.

[69] K. Dudka. Missing a double free when heap pointers are compared. https://github.com/staticafi/symbiotic/issues/89. (Assessed on 01/02/2025).

[70] F. C. Eigler. Mudflap: Pointer use checking for c/c+. *Proceedings of the First Annual GCC Developers' Summit*, pages 57–70, 2003.

[71] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proceedings of the 8th International Symposium on Software Testing and Analysis (ISSTA)*, pages 129–140, 2009.

[72] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. Checked c: Making c safe by extension. In *IEEE Cybersecurity Development (SecDev)*, pages 53–60, 2018.

[73] D. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–4, 2007.

[74] J. Eom, S. Jeong, and T. Kwon. Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation. *arXiv preprint arXiv:2402.12222*, 2024.

[75] W. Ertel, J. M. P. Schumann, and C. B. Suttner. Learning heuristics for a theorem prover using back propagation. In *Proceedings of Österreichische Artificial-Intelligence-Tagung*, pages 87–95. 1989.

[76] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, R. Tsang, N. Nazari, H. Wang, H. Homayoun, et al. Large language models for code analysis: Do {LLMs} really do their job? In *33rd USENIX Security Symposium (USENIX Security)*, pages 829–846, 2024.

[77] B. Farinier, R. David, S. Bardin, and M. Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In *Proceedings of International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 363–380, 2018.

[78] R. M. farkhani, M. Ahmadi, and L. Lu. PTAuth: Temporal memory safety via robust points-to authentication. In *USENIX Security*, pages 1037–1054, 2021.

[79] N. C. for Assured Software. Juliet test suite 1.3. https://samate.nist.gov/SRD/testsuite.php/. (Assessed on 01/02/2025).

[80] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 679–696. IEEE, 2018.

[81] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 519–531, 2007.

[82] W. Gao, V. T. Pham, D. Liu, O. Chang, T. Murray, and B. I. Rubinstein. Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, pages 47–55, 2023.

[83] GCOV. A test coverage program in gnu gcc tool-chain. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. (Assessed on 01/02/2025).

[84] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 47–54, 2007.

[85] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 206–215, 2008.

[86] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network Distributed Systems Security Symposium (NDSS)*, pages 1–16, 2008.

[87] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–12, 2008.

[88] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[89] Google. oss-fuzz-gen. https://github.com/google/oss-fuzz-gen.

[90] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 72–82, 2014.

[91] R. Gopinath, B. Mathis, and A. Zeller. Mining input grammars from dynamic control flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 172–183, 2020.

[92] W. Griswold, D. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC '96. 4th Workshop on Program Comprehension*, pages 144–153, 1996.

[93] B. Gui, W. Song, H. Xiong, and J. Huang. Automated use-after-free detection and exploit mitigation: How far have we gone. *IEEE Transactions on Software Engineering*, 48(11):4569–4589, 2021.

[94] S. Guo, Y. Chen, J. Yu, M. Wu, Z. Zuo, P. Li, Y. Cheng, and H. Wang. Exposing cache timing side-channel leaks through out-of-order symbolic execution. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[95] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for {Overflows}: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, pages 49–64, 2013.

[96] R. Hastings. Purify: Fast detection of memory leaks and access errors. In *Proceedings of Winter USENIX Conference*, pages 125–136, 1992.

[97] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev. Learning to explore paths for symbolic execution. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2526–2540, 2021.

[98] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *USENIX Security*, pages 763–779, 2018.

[99] S. Heelan, T. Melham, and D. Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1689–1706, 2019.

[100] M. Heusner, T. Keller, and M. Helmert. Understanding the search behaviour of greedy best-first search. In *Proceedings of the International Symposium on Combinatorial Search*, pages 47–55, 2017.

[101] J. Hu, Q. Zhang, and H. Yin. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782*, 2023.

[102] J. Hu, Y. Duan, and H. Yin. Marco: A stochastic asynchronous concolic explorer. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1–12, 2024.

[103] K. Huang, Y. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger. Dataguard repo. https://github.com/Lightninghkm/DataGuard. (Assessed on 01/02/2025).

[104] K. Huang, Y. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2022.

[105] L. IR. A powerful intermediate representation for efficient compiler transformations and analysis. https://llvm.org/docs/LangRef.html#introduction. (Assessed on 01/02/2025).

[106] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security)*, pages 2271–2287. USENIX Association, Aug. 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[107] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang. Utopia: Automatic generation of fuzz driver using unit tests. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2676–2692, 2023. doi: 10.1109/SP46215.2023.10179394.

[108] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, page 177–187, 2015.

[109] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 474–484, 2012.

[110] S. Kang, J. Yoon, and S. Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *Proceedings of IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323, 2023.

[111] T. Kapus and C. Cadar. A segmented memory model for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 774–784, 2019.

[112] T. Kapus, O. Ish-Shalom, S. Itzhaky, N. Rinetzky, and C. Cadar. Computing summaries of string loops in c for better testing and refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 874–888, 2019.

[113] Y. Kim, S. Hong, and M. Kim. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 16–26, 2019.

[114] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[115] T. Klooster, F. Turkmen, G. Broenink, R. Ten Hove, and M. Böhme. Continuous fuzzing: a study of the effectiveness and scalability of fuzzing in ci/cd pipelines. In *IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 25–32, 2023.

[116] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

[117] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 193–204, 2012.

[118] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, 2023.

[119] R. Li, B. Zhang, J. Chen, W. Lin, C. Feng, and C. Tang. Towards automatic and precise heap layout manipulation for general-purpose programs. In *Network Distributed Systems Security Symposium (NDSS)*, pages 1–15, 2023.

[120] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. *SIGPLAN Notice*, 48(10):19–32, 2013.

[121] J. Lim and S. Yoo. Field report: Applying monte carlo tree search for program synthesis. In *Proceedings of 8th International Symposium Search Based Software Engineering (SSBSE)*, pages 304–310, 2016.

[122] D. Liu, G. Ernst, T. Murray, and B. I. Rubinstein. Legion: Best-first concolic testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 54–65, 2020.

[123] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–12, 2024.

[124] E. S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, 1969.

[125] S. Lu, Z. Feng, D. Guo, S. Wang, D. Tang, N. Duan, M. Zhou, et al. Codexglue: A benchmark dataset and open challenge for code intelligence. *arXiv preprint arXiv:2102.04664*, 2021.

[126] K. Luckow, C. S. Păsăreanu, and W. Visser. Monte carlo tree search for finding costly paths in programs. In *Proceedings of 16th International Conference on Software Engineering and Formal Methods (SEFM)*, pages 123–138, 2018.

[127] Y. Lyu, Y. Xie, P. Chen, and H. Chen. Prompt fuzzing for fuzz driver generation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3793–3807, 2024.

[128] G. m4. A traditional unix macro processor. https://www.gnu.org/software/m4, 2022. (Assessed on 01/02/2025).

[129] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proceedings of International Symposium on Static Analysis*, pages 95–111, 2011.

[130] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, and Y. Liu. Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138*, 2023.

[131] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 134–143, 2007.

[132] G. make. A building automation tool. https://www.gnu.org/software/make, 2022. (Assessed on 01/02/2025).

[133] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[134] D. Marjamäki. Cppcheck: a tool for static C/C++ code analysis. https://cppcheck.sourceforge.io/. (Assessed on 01/02/2025).

[135] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[136] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, page 554–564, 2013.

[137] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.

[138] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 129–139, 2018.

[139] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–15, 2024.

[140] D. Midi, M. Payer, and E. Bertino. Memory safety for embedded devices with nescheck. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, page 127–139, 2017.

[141] N. S. Z. J. M. M. M.K. and S. Zdancewic. CETS: compiler enforced temporal safety for c. In *Proceedings of International Symposium on Memory Management (ISMM)*, page 31–40, 2010.

[142] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258, 2009.

[143] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258, 2009.

[144] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers. Using an LLM to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[145] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 128–139, 2002.

[146] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[147] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.

[148] M. Nowack. Fine-grain memory object representation in symbolic execution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 912–923, 2019.

[149] Oclint. A static code analysis tool for improving quality and reducing defects. https://oclint.org. (Assessed on 01/02/2025).

[150] D. of make package. Confirmation of the bug in Case 2, 2024. URL https://github.com/haoxintu/SymLoc/blob/main/experiments/confirmation-case2.pdf.

[151] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix. Coccinelle: tool support for automated cert c secure coding standard certification. *Science of Computer Programming*, 91:141–160, 2014.

[152] W. Pan, Z. Chen, G. Zhang, Y. Luo, Y. Zhang, and J. Wang. Grammar-agnostic symbolic execution by token symbolization. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 374–387, 2021.

[153] A. Pandey, P. R. G. Kotcharlakota, and S. Roy. *Deferred Concretization in Symbolic Execution via Fuzzing*, pages 228–238. 2019.

[154] A. Pandey, P. R. G. Kotcharlakota, and S. Roy. Deferred concretization in symbolic execution via fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 228–238, 2019.

[155] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20:391–425, 2013.

[156] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2339–2356, 2023.

[157] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 697–710, 2018.

[158] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar. Accelerating array constraints in symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, page 68–78, 2017.

[159] P. Pitigalaarachchi, X. Ding, H. Qiu, H. Tu, J. Hong, and L. Jiang. Krover: A symbolic execution engine for dynamic kernel analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2009–2023, 2023.

[160] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *USENIX Security*, pages 181–198, 2020.

[161] S. Poeplau and A. Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *Network Distributed Systems Security Symposium (NDSS)*, pages 1–18, 2021.

[162] S. Poulding and R. Feldt. Heuristic model checking using a monte-carlo tree search algorithm. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1359–1366, 2015.

[163] M. Prandini and M. Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[164] N. Ruaro, K. Zeng, L. Dresel, M. Polino, T. Bao, A. Continella, S. Zanero, C. Kruegel, and G. Vigna. Syml: Guiding symbolic execution toward vulnerable states through pattern learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 456–468, 2021.

[165] R. Rutledge and A. Orso. Pg-klee: Trading soundness for coverage. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 65–68, 2020.

[166] M. safety in the Chromium project. https://www.chromium.org/Home/chromium-security/memory-safety/. (Assessed on 01/02/2025).

[167] J. F. Santos, P. Maksimović, S.-É. Ayoun, and P. Gardner. Gillian: Compositional symbolic execution for all. *arXiv preprint arXiv:2001.05059*, 2020.

[168] D. Schemmel, J. Büning, F. Busse, M. Nowack, and C. Cadar. Kdalloc: The klee deterministic allocator: Deterministic memory allocation during symbolic execution and test case replay. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1491–1494, 2023.

[169] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*, pages 317–331, 2010.

[170] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC*, pages 1–28, 2012.

[171] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. {AddressSanitizer}: A fast address sanity checker. In *USENIX annual technical conference (USENIX ATC)*, pages 309–318, 2012.

[172] S. Shankar, J. Zamfirescu-Pereira, B. Hartmann, A. Parameswaran, and I. Arawjo. Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 1–14, 2024.

[173] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, pages 138–157, 2016.

[174] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of International Conference on Information Systems Security (ICISS)*, pages 1–25, 2008.

[175] A. Sotirov. Heap feng shui in javascript. https://llvm.org/docs/LangRef.html#introduction. (Assessed on 01/02/2025).

[176] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, and W. Le. A comprehensive study of the capabilities of large language models for vulnerability detection, 2024. URL https://arxiv.org/abs/2403.17218.

[177] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network Distributed Systems Security Symposium (NDSS)*, pages 1–16, 2016.

[178] STP. Simple theorem prover, an efficient smt solver for bitvectors. https://github.com/stp/stp. (Assessed on 01/02/2025).

[179] N. Sturtevant and A. Felner. A brief history and recent achievements in bidirectional search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1–7, 2018.

[180] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, pages 265–266, 2016.

[181] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959*, 2024.

[182] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023.

[183] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy(S&P)*, pages 48–62, 2013.

[184] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, pages 48–62, 2013.

[185] Tencent. A fast and accurate static analysis solution for C/C++, C#, Lua codes. https://github.com/Tencent/TscanCode. (Assessed on 01/02/2025).

[186] D. Trabish and N. Rinetzky. Relocatable addressing model for symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 51–62, 2020.

[187] D. Trabish and N. Rinetzky. Relocatable addressing model for symbolic execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 51–62, 2020.

[188] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 350–360, 2018.

[189] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 350–360, 2018.

[190] D. Trabish, S. Itzhaky, and N. Rinetzky. A bounded symbolic-size model for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1190–1201, 2021.

[191] D. Trabish, S. Itzhaky, and N. Rinetzky. A bounded symbolic-size model for symbolic execution. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2021, page 1190–1201, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468596. URL https://doi.org/10.1145/3468264.3468596.

[192] H. Tu, L. Jiang, X. Ding, and H. Jiang. FastKLEE: faster symbolic execution via reducing redundant bound checking of type-safe pointers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1741–1745, 2022.

[193] H. Tu, L. Jiang, J. Hong, X. Ding, and H. Jiang. Concretely mapped symbolic memory locations for memory error detection. *IEEE Transactions on Software Engineering*, 50(07):1747–1767, 2024.

[194] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 862–880, 2024.

[195] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network Distributed Systems Security Symposium (NDSS)*, pages 1–15, 2000.

[196] J. Wagner, V. Kuznetsov, and G. Candea. Overify: Optimizing programs for fast verification. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, pages 1–8, 2013.

[197] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, pages 579–594, 2017.

[198] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou. MAZE: Towards automated heap feng shui. In *USENIX Security*, pages 1647–1664, 2021.

[199] Z. Wang, L. Zhang, C. Cao, N. Luo, X. Luo, and P. Liu. How does naming affect llms on code analysis tasks?, 2024. URL https://arxiv.org/abs/2307.12488.

[200] D. R. White, S. Yoo, and J. Singer. The programming game: evaluating mcts as an alternative to gp for symbolic regression. In *Proceedings of the Companion Publication of the Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1521–1522, 2015.

[201] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities. In *USENIX Security*, pages 781–797, 2018.

[202] Z. Wu, E. Johnson, W. Yang, O. Bastani, D. Song, J. Peng, and T. Xie. Reinam: reinforcement learning for input-grammar inference. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 488–498, 2019.

[203] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639121. URL https://doi.org/10.1145/3597503.3639121.

[204] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368, 2009.

[205] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 327–337, 2018.

[206] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283—294, 2011.

[207] F. Yao, Y. Li, Y. Chen, H. Xue, T. Lan, and G. Venkataramani. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *Proceedings of 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–120, 2017.

[208] C.-C. Yeh, H.-L. Lu, J.-J. Yeh, and S.-K. Huang. Path exploration based on monte carlo tree search for symbolic execution. In *Proceedings of International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 33–37, 2017.

[209] Q. Yi, Y. Yu, and G. Yang. Compatible branch coverage driven symbolic execution for efficient bug finding. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1633–1655, 2024.

[210] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (FSE)*, page 26–36, 2011. ISBN 9781450304436.

[211] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security*, pages 745–761, 2018.

[212] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security)*, pages 745–761, 2018.

[213] I. Yun, D. Kapil, and T. Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *USENIX Security*, pages 1111–1128, 2020.

[214] Z3. A theorem prover from microsoft research. https://github.com/z3prover/z3. (Assessed on 01/02/2025).

[215] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems (Euro CCS)*, pages 321–334, 2010.

[216] C. Zhang, Y. Zheng, M. Bai, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu. How effective are they? exploring large language model based fuzz driver generation. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1223–1235, 2024.

[217] H. Zhang, Y. Rong, Y. He, and H. Chen. Llamafuzz: Large language model enhanced greybox fuzzing. *arXiv preprint arXiv:2406.07714*, 2024.

[218] T. Zhang, D. Lee, and C. Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, 2019.

[219] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu. Regular property guided dynamic symbolic execution. In *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, pages 643–653, 2015.

[220] Y. Zhao, X. Wang, L. Zhao, Y. Cheng, and H. Yin. Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*, pages 534–547, 2022.

[221] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations (ICLR))*, pages 1–9, 2022.