UNDERSTANDING AND DEFENDING AGAINST USE-AFTER-FREE VULNERABILITIES

by

Zeyu Chen

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering

Spring 2025

© 2025 Zeyu Chen
All Rights Reserved

UNDERSTANDING AND DEFENDING AGAINST USE-AFTER-FREE VULNERABILITIES

by

Zeyu Chen

Approved: _____

Jamie Phillips, Ph.D.
Chair of the Department of Electrical & Computer Engineering

Approved: _____

Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____

Louis F. Rossi, Ph.D.
Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Haining Wang, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Hui Fang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Jidong Xiao, Ph.D.
Member of dissertation committee

TABLE OF CONTENTS

LIST OF TABLES . . . . . **viii**
LIST OF FIGURES . . . . . **x**
ABSTRACT . . . . . **xii**

Chapter

1 INTRODCTION **1**

1.1 Use-After-Free Vulnerabilities 1
1.2 Static UaF Vulnerability Detection 2
1.3 Dynamic Software Fuzzing 2
1.4 UaF Exploit Defense 3
1.5 Large Language Models in Static Analysis 4
1.6 Dissertation Roadmap 5

2 AN EMPIRICAL STUDY OF THE CHARACTERISTICS AND DETECTABILITY OF USE-AFTER-FREE VULNERABILITIES **6**

2.1 Motivation 6
2.2 Methodology 7

2.2.1 UaF Bug Collection 7
2.2.2 Threats to Validity 10

2.3 General Findings 11

2.3.1 Lifespan of UaF Bugs 11
2.3.2 Where Do Dangling Pointers Point To? 12
2.3.3 Nondeterminism of UaF 14
2.3.4 Line of Free to Line of Dereference 16

2.3.5 UaF Reproducibility 17

2.4 UaF Characterization 19

2.4.1 All UaF Bugs Are Not Created Equal 20

2.4.1.1 Immediate-use-after-free 20
2.4.1.2 Raise a red flag 21
2.4.1.3 Memory resize considered harmful 23
2.4.1.4 Misuse of APIs and features 26
2.4.1.5 Double free 27
2.4.1.6 Miscellaneous cases 28

2.4.2 Reference Counting, A Panacea for UaF? 28
2.4.3 UaF Benchmark 33

2.5 `Palfrey`: A Pattern-based UaF Detector 34

2.5.1 Design and Implementation of `Palfrey` 34
2.5.2 Evaluation of `Palfrey` 35

2.6 Summary 39

3 AN EXPLORATION OF FUZZING FOR DISCOVERING USE-AFTER-FREE VULNERABILITIES 40

3.1 Motiviting Example: A Challenging Case for UaF Fuzzing 40
3.2 Preliminaries 41

3.2.1 Fuzzing 41
3.2.2 Comparing Testing Efficiency 43

3.3 Goals and Objectives 45
3.4 Methodology 45

3.4.1 Fuzzer Selection 45
3.4.2 UaF Bug Collection and Analysis 47
3.4.3 Threats to Validity and Observer Errors 48

3.5 UaF Characteristics 49

3.5.1 Bug Code Scope and Its Implications for Fuzzing 50
3.5.2 UaF Bug Patterns 51

3.5.3 Nondeterminism and Its Influence on Fuzzing UaF 51

3.6 Performance of Existing Fuzzers 53

3.6.1 UaF Reproducibility 54
3.6.2 UaF patterns affect fuzzing detection 55
3.6.3 UaF code scope matters 56
3.6.4 Reference count bugs 57
3.6.5 Fuzzing randomness 58
3.6.6 Nondeterminism of UaF: A Fuzzing Challenge 59

3.7 Lesson Learned 61
3.8 Summary 63

4 BRIDGING THE SEMANTIC GAP: BENCHMARKING LLMS AND STATIC ANALYSIS IN USE-AFTER-FREE DETECTION 65

4.1 Methodology 66

4.1.1 Framework Structure 66
4.1.2 Workflow 67
4.1.3 Static vs. LLM Analysis Paths 68
4.1.4 Benchmark Description 68

4.1.4.1 UaF Bug Dataset 69
4.1.4.2 Linux Kernel Benchmark 70

4.2 Evaluation 71

4.2.1 Experimental Setup 71
4.2.2 Evaluation Objectives 72
4.2.3 RQ1: Evaluating the Effectiveness of LLMs for UaF Detection Under Varying Context Granularity 72
4.2.4 RQ2: Detection Accuracy and Error Characteristics of LLMs . 75
4.2.5 RQ3: How Does Prompt Design Influence Detection Accuracy? 78
4.2.6 RQ4: Do LLMs Require New Evaluation Metrics? 80
4.2.7 RQ5: Does the Level of Input Abstraction Affect Detection Accuracy? 83

4.3 Discussion 85
4.4 Summary 86

5 RELATED WORK **88**

5.1 Empirical Study on Bug Characteristics 88
5.2 Static Bug Detection 89
5.3 UaF Mitigation and Prevention 89
5.4 LLMs for Program Analysis 91

6 CONCLUSION AND FUTURE WORKS **92**

6.1 Future Work 93
6.2 Recent Work on Memory Allocator 93
6.3 Need for a Custom Allocator 94
6.4 Future Directions for LLM-Based Vulnerability Detection 95

BIBLIOGRAPHY **97**

LIST OF TABLES

2.1 Number of UaF bugs collected from three representative software suites and 38 GitHub open source repositories. 8

2.2 Github repositories for empirical study 10

2.3 Average lifespan of UaF and non-UaF bugs, measured in days between when a bug was introduced until a patch was committed. . 12

2.4 Memory segment of UaF bugs. 12

2.5 Distribution of basic block (BB) augmented LOFTLOD of the bugs in the collected C/C++ applications. 17

2.6 List of the reproduced 30 bugs. 18

2.7 The occurrence frequency of each UaF pattern. Note that the total number is more than 150 since there might be overlaps between the two patterns. For example, both flag error and misuse of reference count could lead to double free, and API misuse could result in reference count error. 20

2.8 Comparison of `UAFBenchPlus` and UaFBench [136]. †Application size is measured in lines of code. ‡Patch size is measured in lines of code added and deleted. ∗K for Thousand. ∗∗M for Million. 33

2.9 Detection results of `Palfrey`, Infer, and PC-lint on our collected UaF bugs. Infer and PC-lint failed to run on the other 75 bugs, due to various system environment and configuration problems. 36

2.10 Comparison of the time and memory overhead of three tools. † Time is measured in seconds. ‡ Mem is the memory overhead measured in gigabytes (GB). 36

2.11 Summary of new UaF bugs detected by `Palfrey`. †This column indicates if the bug is uniquely detected by `Palfrey`. 37

2.12 Detection accuracy of `Palfrey` in various bug patterns 38

3.1 Number of UaF bugs collected from three representative software suites. 48

3.2 Bug Info and Tools Comparison. † The crashed bugs are highlighted ‡ T/O means the fuzzer cannot discover vulnerabilities within 24 hours 52

4.1 Impact of Context Size on LLM UaF Detection in GPT-4 73

4.2 LLM vs. Clang Static Analyzer on UaF Detection 75

4.3 UaF Detection Performance Across LLMs and Static Analysis Tools 75

4.4 GPT-4 Performance Under Different Prompt Designs 79

4.5 GPT-4 UaF Detection: Source Code vs. LLVM IR 84

LIST OF FIGURES

2.1 A hypothetical example of stack UaF 13

2.2 Global UaF due to loss of memory flag update[41] 14

2.3 Static destruction order fiasco [42] 14

2.4 Dereference of an strup() object[43] 15

2.5 UaF as a concurrency bug [24] 15

2.6 Immediate use-after-free 17

2.7 A UaF bug of the save-before-free pattern[44] 21

2.8 UaF due to a falsely updated flag in `beiscsi_create_cqs()` defined in `drivers/scsi/be2iscsi/be_main.c` [30] 22

2.9 Missing check of a flag variable leads to UaF [33] 23

2.10 UaF due to realloc() with a special size parameter 0 in `/gfx/thebes/gfxAndroidPlatform.cpp` [31] 24

2.11 UaF when a function calls realloc [25] 25

2.12 auto and auto& [63] 26

2.13 A refcount UaF occurs because a borrowed reference is not properly converted to a non-borrowed one [38] 30

2.14 UaF due to a misused borrowed reference [26] 30

2.15 UaF due to reference counter over-decrement [27] 31

2.16 UaF due to the missing decrement for a reference counter [32] . . . 32

2.17 UaF due to a misused reference counting API [34] 32

3.1 Reference count overflow [39] 42

3.2 Runtime overhead of ASan 53

3.3 Total success runs by different fuzzing tools 55

3.4 UaF due to a misused borrowed reference [1] 58

3.5 Total success runs on the 3 concurrent bugs by different fuzzing tools 59

3.6 UaF when a function calls realloc [25] 61

4.1 Testing framework. 66

4.2 UaF bug involving pointer arithmetic and memory deallocation. . . 74

4.3 UaF Case with Interprocedural Aliasing. 81

ABSTRACT

Over the past decades, use-after-free (UaF) vulnerabilities have become a critical and widely exploited security concern. To address such increasing threats, this dissertation advances defense in multiple aspects, including UaF vulnerability detection, UaF exploit defense, and UaF bug fixes.

The first research direction delves into a comprehensive empirical study of UaF vulnerabilities. Utilizing a dataset of 150 real-world UaF cases randomly sampled from representative software suites such as the Linux kernel, Python, and Mozilla Firefox, this study seeks to unravel the commonalities, root causes, and recurring patterns in real-world UaF bugs. The findings highlight the diversity and non-uniform distribution of root causes among different software, emphasizing that a generic UaF detector or fuzzer might not be an optimal solution. By categorizing the root causes into 11 patterns, several of which can be translated into simple static detection rules, this work further introduces a static bug detector named Palfrey. Palfrey exhibits superior coverage and accuracy in UaF detection, while minimizing time and memory overhead, making it a promising tool to address UaF vulnerabilities.

The second research direction investigates the relationship between fuzzing techniques and UaF vulnerabilities. Fuzzing has proven effective in rapidly generating faulty inputs to discover software bugs, but its efficiency in identifying UaF instances with specific patterns and characteristics remains uncertain. The study explores whether the efficiency of fuzzing is dependent on the code scope of UaF, particularly for bugs with larger code scopes. Additionally, the research addresses the challenge of detecting non-deterministic UaF bugs and explores the necessity of auxiliary tools like address

sanitizers (ASan) to assist fuzzers in identifying UaF vulnerabilities. The study provides practical guidelines to enhance UaF detection using fuzzers, aiming to improve the effectiveness of dynamic testing techniques in identifying UaF vulnerabilities.

The third research direction explores how Large Language Models (LLMs) perceive and detect UaF vulnerabilities compared to traditional static analysis tools. Unlike deterministic analyzers, LLMs rely on probabilistic language reasoning and are influenced by prompt phrasing, input granularity, and abstraction level. To enable systematic evaluation, this work constructs a benchmark of 50 curated UaF cases specifically adapted for LLM-based analysis. These samples are distilled from the broader set of 150 real-world UaF cases and carefully reduced to single-file, self-contained programs that preserve critical semantic features while remaining within LLM token limits. The benchmark is used to evaluate LLM performance across prompt styles and input formats, including source code and LLVM IR. Novel evaluation metrics—such as Scalability Index, Context-Aware Precision, and Prompt Sensitivity Score—are proposed to capture the unique behaviors of LLMs in code analysis. The study reveals a semantic gap between LLM and static tool reasoning and highlights opportunities for hybrid approaches.

In conclusion, this dissertation seeks to advance the state of the art in UaF vulnerability mitigation by comprehensively understanding real-world UaF characteristics, introducing innovative static analysis techniques, and enhancing the efficiency of fuzzing for UaF detection. These research directions collectively contribute to the goal of addressing the critical issue of UaF vulnerabilities and enhancing the security and reliability of software systems.

Chapter 1

INTRODCTION

In C/C++ programs, a use-after-free bug[1] [23] is triggered when a pointer pointing to freed memory is dereferenced. This pointer is referred to as a *dangling pointer*. Normally, dereferencing a dangling pointer results in data corruption and/or program crashes (denial of services). However, when the dangling pointer and program state can be manipulated by attackers, UaF bugs may lead to much worse unexpected consequences. For example, if attackers can fill the freed memory with deliberately crafted contents, UaF bugs potentially allow attackers to run arbitrary code. UaF vulnerabilities also serve as a popular attack vector to leak the in-memory addresses of certain libraries like `libc` and thus defeat the address space layout randomization (ASLR) in modern systems. Actually, if a UaF bug is present in software, there is a high likelihood that an exploit could be developed to compromise the victim software system [152].

Unfortunately, UaF vulnerabilities are almost inevitable and difficult to detect and fix. Although significant progress [74, 77, 78, 82, 84, 88, 98, 102, 104, 105, 108, 138, 141, 148, 156, 157, 166, 167, 168] from multiple directions have been made over the past years to address this security challenge, there are still many open, unsolved issues.

1.1 Use-After-Free Vulnerabilities

UaF vulnerabilities occur when a program continues to use a pointer that points to memory that has been freed. This typically happens when the program deallocates or

[1] Throughout this dissertation, we consider that each UaF bug is a potential vulnerability. Thus, we use UaF bug and UaF vulnerability interchangeably.

frees a memory region but fails to update or invalidate the corresponding pointer. UaF vulnerabilities can lead to program crashes, data corruption, or even allow attackers to execute arbitrary code. The exploitation of UaF vulnerabilities can result in serious security breaches, including remote code execution and privilege escalation attacks.

UaF vulnerabilities often involve multiple stages:

- **Memory Allocation:** The program dynamically allocates memory for objects or data structures during its execution.
- **Memory Deallocation:** The program releases or deallocates the previously allocated memory when it is no longer needed.
- **Memory Dereference:** The program continues to access or dereference a pointer that points to the deallocated memory, resulting in undefined behaviors.

1.2 Static UaF Vulnerability Detection

Previous research [74, 168, 166, 157, 108, 156, 98, 167, 138, 88, 82] has used static analysis to detect UaF bugs from source code. In static analysis, detectors track memory allocation and pointer propagation. When a dereferenced pointer points to freed memory, a bug is detected. While some promising results have been achieved, it is still far from a complete solution to address UaF bugs. For example, some approaches only consider UaF bugs in specific software suites like Linux device drivers [74]. Also, static detectors often have many false positives [93]. More importantly, several essential questions remain unanswered: what portion of real world UaF bugs can be detected? How many patterns into which real-world UaF bugs fall? And, is there any UaF bug pattern that has not yet been discovered by existing detectors?

1.3 Dynamic Software Fuzzing

Software fuzzing is a popular dynamic technique to find bugs using automatically generated input data. The program used to generate these inputs is called a fuzzer.

Empirical studies show that fuzzing can effectively uncover many common bugs like buffer overflows [128].

Early fuzzers often generate inputs randomly. Therefore, it usually takes a long time, from hours to months, to provide sufficient coverage and find latent bugs. In recent years, the coverage-guided fuzzing has attracted much attention to improving the fuzzing efficacy [102, 148, 141, 104, 84, 77, 78, 105]. Coverage-guided fuzzers can effectively reveal many overlooked corner cases, and they have become quite appealing for use in production testing environments. Actually, by virtue of open-source fuzzing tools and frameworks like American fuzzy lop (AFL) [172] and oss-fuzz [45, 144], fuzzing has achieved great success in discovering many bugs from a variety of software. When coupled with memory corruption detection tools like Valgrind [134] and ASan [142], fuzzing can also uncover many UaF bugs in large software suites like the Chrome browser.

However, it is still unclear, both quantitatively and qualitatively, how effectively the state-of-the-art fuzzers can expose UaF bugs. The major challenge of software fuzzing is the exponential increase in space exploration. The exposure of UaF bugs can be even more challenging since it requires to trigger a premature memory free and explore all pointer propagation paths. Even worse, many UaF bugs are non-deterministic due to the uncertainty in memory allocators and the concurrency in multi-thread applications. Ultimately, the design of a practical and efficient UaF fuzzer requires a better understanding of the manifestation of real world UaF bugs, with which fuzzers can prioritize the program states and execution paths.

1.4 UaF Exploit Defense

As it is almost impossible to find and fix all UaF bugs during software development, another defense direction is to eliminate dangling pointers, the culprit of UaF vulnerabilities. Along this direction, many modern programming languages have adopted various mechanisms like tracing garbage collection (GC) in Java [46], reference-count GC in Python [47], automatic reference counting (ARC) in Swift [48],

smart pointers in C++ [49], and ownership in Rust [50]. There are also solutions to arm legacy C/C++ code with reference counting to address UaF bugs [76, 145]. Meanwhile, based on the insight that UaF vulnerabilities can be avoided when there is no memory reuse, some other methods attempt to avoid page sharing in virtual memory [94, 137, 123]. In essence, these methods follow the rule that a memory block cannot be freed as long as a pointer (may) still points to it.

Several recent works instead follow a different rule to harden C/C++ applications. The rule assumes that all explicit memory frees are what programmers expect, and all dereferences to dangling pointers can be considered as potential exploits. Guided by this rule, one defense mechanism is lock-and-key, in which a unique identifier is generated for each memory allocation (i.e., a lock) and is then associated to pointers (as the key) for accessing the memory [132, 91]. Another approach proactively nullifies the dangling pointers at the time of memory free [113, 169, 154, 119]. These methods can effectively defeat attackers' exploit attempts.

Although mechanisms like GC and dangling pointer nullification show great potential, there are still several open questions. (1) How effective are these mechanisms and what are the caveats to watch out in implementation? (2) How effective is the widely adopted reference count to thwart UaF in C/C++? In addition to programming language features like smart pointers and ownership, there are also ad-hoc implementations of reference count in multiple software suites like Linux kernel and Python. These solutions mostly depend on programmers to correctly understand and use the provided primitives. (3) Is there a better trade-off between performance overhead and protection coverage to facilitate these runtime defenses (e.g., dangling pointer nullification) to be widely adopted in a production environment?

1.5 Large Language Models in Static Analysis

The adoption of LLMs in static analysis represents a transformative shift in software engineering. Unlike traditional static analysis tools, which rely on predefined rules and patterns, LLMs are pre-trained on vast corpora of code and natural language.

This allows them to understand contextual information, reason about code semantics, and detect subtle vulnerabilities that might elude traditional tools. Recent research [86, 101, 149, 109] has explored LLMs in tasks such as vulnerability detection, automated code repair, and summarization, demonstrating their potential to augment traditional static analysis techniques.

Despite these advancements, LLM-based analysis faces several challenges. The models often operate as black boxes, with their internal decision-making processes remaining opaque. Their performance is also heavily influenced by the quality and scope of the training data, raising concerns about their generalizability to diverse or unseen codebases. Additionally, while LLMs excel at identifying certain bug patterns, they may lack the structural precision required for complex, memory-related vulnerabilities like UaF. These limitations suggest that a promising direction is to combine the contextual understanding of LLMs with the rigorous logic of static analysis tools, creating a hybrid approach that leverages the strengths of both paradigms.

1.6 Dissertation Roadmap

The remainder of this dissertation is structured as follows: Chapter 2 delves into a detailed empirical study of UaF vulnerabilities, including their patterns, manifestations, fix strategies, and other key attributes. Chapter 3 explores the effectiveness of fuzzing in discovering UaF vulnerabilities with varying characteristics. Chapter 4 investigates the emerging role of LLMs in static analysis, focusing on their capabilities and limitations in detecting use-after-free vulnerabilities. Chapter 5 scrutinizes existing work and its limitations, providing a comprehensive review of the current research landscape. Chapter 6 summarizes this dissertation and discusses the research direction for future work.

Chapter 2

AN EMPIRICAL STUDY OF THE CHARACTERISTICS AND DETECTABILITY OF USE-AFTER-FREE VULNERABILITIES

The first work in this dissertation is grounded in the intricate world of UaF vulnerability, a prevalent and critical security concern in software systems. When a program attempts to access memory that has been freed, it can result in data corruption, program crashes, and even potential code execution by malicious actors. These vulnerabilities have become a primary target for attackers seeking to compromise software applications. To address this increasing threat, we need to advance the defense in multiple directions, such as UaF vulnerability detection, UaF exploit defense, and UaF bug fix. Unfortunately, the intricacy rooted in the temporal nature of UaF vulnerabilities makes it quite challenging to develop effective and efficient defenses in these directions. This calls for an in-depth understanding of real-world UaF characteristics. This work presents the first comprehensive empirical study of UaF vulnerabilities.

2.1 Motivation

The motivation behind our empirical study is rooted in the significant challenges associated with detecting and mitigating UaF vulnerabilities. Static and dynamic analysis techniques have made progress, but open questions remain. This study aims to address these questions by examining real-world UaF bugs, their patterns, and their implications. By doing so, we can provide valuable insights into the landscape of UaF vulnerabilities, which will inform subsequent research and contribute to more effective detection and protection strategies.

The contributions of the first work in this dissertation are multifaceted. It provides an empirical study based on a selection of 150 real-world UaF bugs from 41

open-source projects, shedding light on UaF bug patterns, manifestations, fix strategies, and their root causes. The study yields valuable insights and guidelines for UaF bug detection and protection. Furthermore, it introduces an extensive UaF benchmark, a static detector named Palfrey, and an open dataset for future research endeavors.

This work forms the foundation for a deeper understanding of UaF vulnerabilities and lays the groundwork for the subsequent research directions explored in this dissertation. The transition from this foundational work to the subsequent chapters promises to unravel new insights and innovative solutions in the realm of UaF vulnerabilities and their detection.

2.2 Methodology

2.2.1 UaF Bug Collection

UaF bugs occur across a wide range of programs that have various features and development cycles. The programs can be commercial/free and closed source/open source. We use the following methodology and principles to compile the datasets for this study:

- **Open source.** In order to thoroughly analyze and understand UaF bugs, we need to access the source code. We leave the extension to closed source programs with only binaries available as our future work.

- **Public bug trackers.** We extracted the UaF bugs from the publicly accessible bug tracking systems of the open source projects. The bugs tracked in these systems are usually well-documented with high quality details. They are often confirmed or disputed by developers. We can also view the discussions on the bug trackers, including the causes and their potential effects, and track how the bugs get resolved like the code patches. Note that although some projects' bug trackers are open, the bugs' details are mostly hidden to the public, e.g., Chromium browser. We consider such cases as closed bug trackers.

Source	Deterministic	Non-deterministic	
		Concurrent	Others
Firefox	25	5	0
CPython	27	2	1
Linux	16	10	4
GitHub	47	11	2
Overall	115	28	7

Table 2.1: Number of UaF bugs collected from three representative software suites and 38 GitHub open source repositories.

- **Mature and actively maintained.** The bugs are selected from mature projects that have stable releases and developers for active maintenance.

Based on these guidelines, we selected a final set of 150 real-world UaF bugs that were randomly sampled from the following sources, as summarized in Table 3.1.

Three representative software suites. We searched the bug tracking systems from three software suites: Linux Kernel [118], Mozilla Firefox [131], and CPython [140]. They represent three most widely used applications with large code base, more than 10 years development history, and million lines of code. These applications show variations in programming languages (`C` vs `C++`) and memory management mechanisms (manual memory collection vs reference-count based memory collection).

The search function of each bug tracking system was extensively used to find the reported UaF vulnerabilities. In particular, they provided many filters and keyword options to reduce the results to the desired bug reports. The keywords we used include `"UaF"`, `"use-after-free"`, `"dangling pointer"`, `"double free"`, and their variants. To make the bug search comprehensive, we also used the keywords `"data race"`, `"concurrency"`, `"weak reference"`, and `"reference count"`, etc. Then, we checked if such bugs are UaF. From there, we extracted the desired bugs in two steps. First, we cross-checked with the Common Vulnerabilities and Exposures (CVE) database [153]. If a bug has a CVE entry, it is more likely to be a security vulnerability, and thus we prefer such bugs in our study. Then, we randomly sampled a set of results and investigated to verify that the reported bugs were truly confirmed UaF vulnerabilities.

In some cases, our search terms were included in the bug reports, but they ended up being incorrect designations by developers. Such cases were ignored during our data collection.

There are three major challenges in the bug report collection. (1) Many bug reports are incomplete. In particular, it is unclear how to trigger the bugs and no PoC is provided. (2) UaF bugs are not easy to understand. Their patterns and manifestations usually involve complicated interactions among multiple software components. (3) Some bugs in different reports originate from the same root cause, e.g., issue 24099 [28] and issue 24101 [29] in Python.

We filtered out a significant number of UaF bugs during the selection process to ensure the quality and relevance of the dataset. Specifically, we excluded bugs that have incomplete reports, miss crucial details, or could not be fully understood or reproduced. In addition, we discarded bugs that were incorrectly classified as UaF or did not meet our defined criteria. As we encountered instances where multiple bugs originated from the same root cause, we made the decision to retain one representative bug to avoid duplications in our analysis. After this meticulous filtering process, we arrived at a final set of 150 real-world UaF bugs for our study, which provided a comprehensive and reliable base for our analysis. The excluded bugs, due to their limited or incomplete information, were not considered in our analysis to maintain the accuracy and reliability of our findings.

38 GitHub open source repositories. In order to complement our dataset with bugs from a more diverse set of projects, we leveraged GitHub [51], one of the largest code hosting services for open source software. The same keywords as above were used to search for UaF bugs on GitHub. Meanwhile, in order to find the bugs conforming to the principles described above, we adopted multiple searching criteria. Specifically, the bugs are limited to the code repositories with active commits within recent six months and more than 1,000 stars. Finally, we manually verified that the repositories contain production level code, instead of teaching or sample code. In total, we chose 60 bugs from 38 repositories. The full list of the repositories is presented in

Table 2.2.

kbd	nng	php	vlc
oboe	dcmtk.	Envoy	KeyDB
mongo	mySQL	swift	uacme
yosys	coturn	Jungle	spead2
SQlite	u-boot	f-stack	gnucash
osquery	electron	i2p.i2p	php-src
systemd	arangodb	goaccess	Open SSH
mosquitto	OMCompiler	oniguruma	profanity.
plan9port	verilator	librdkafka	timescaledb
foundationdb	qpid-dispatch		

Table 2.2: Github repositories for empirical study

Discussion. We would like to point out that the number of UaF bugs collected from each application does not reflect how secure or insecure an application is or how common UaF is in an application. Our dataset simply represents the UaF bugs that meet our selection criteria and get randomly selected.

2.2.2 Threats to Validity

As for all empirical studies, the conclusions drawn from our results are subject to some threats to the validity. In the following, we discuss the potential sources of bias and how we address them with our best-efforts.

One limitation of our study is the representativeness of the selected bugs, which may not be representative. There are millions of software packages in the wild, and it is impractical for our study to cover all of them. In particular, our dataset is exclusively collected from open source programs. As a result, our findings might not be generalized to a larger population of commercial software. However, modern open source programs have comparative quality as commercial ones, and many open source programs are widely used in production environments. We carefully select UaF bugs from *diverse* types of open-source programs. And, the bugs are *randomly* sampled from GitHub and bug trackers of popular software, which serve as a great population

of UaF bugs. Therefore, we believe that the findings derived from our dataset can provide valuable guidelines for future defense.

We also give higher priority to the latest bugs. Therefore, the findings might not represent the bug characteristics years ago. However, we argue that the latest bugs can provide more valuable and timely insights. Another limitation to the generalizability of our study is the size of our sample set. According to modern statistics, a random sample set with size greater than 30 is larger enough to represent the entire population [147]. Actually, our sample size is comparable to previous similar empirical studies [121, 110, 170, 115].

Finally, our study could suffer from potential observer errors. To minimize the possibility of observer errors, a group of researchers share the same detailed protocol to inspect and classify the bugs, and all bugs are cross-checked by two researchers. In addition, we are familiar with the three targeted programs and some of the randomly sampled applications.

2.3 General Findings

This section sheds light on some general findings from our study to provide a better understanding on the lifespan and manifestation of UaF bugs.

2.3.1 Lifespan of UaF Bugs

Table 2.3 shows the average UaF lifespan, which is defined as the duration from the time when the bug was introduced to when its patch was committed. As a comparison, we also randomly selected 40 non-UaF bugs (with 10 from each source) and analyzed their average lifespan.

Finding 1. *The lifespan of UaF bugs on average is 74.9% longer than non-UaF ones.*

Overall, our results show that UaF bugs have much longer lifespan, implying that UaF bugs are more sophisticated and more difficult to find and fix.

Source	UaF	Non-UaF
Firefox	641	391
Python	1,207	671
Linux	1,312	679
GitHub	1,222	765

Table 2.3: Average lifespan of UaF and non-UaF bugs, measured in days between when a bug was introduced until a patch was committed.

Memory Segment	# of bugs
Stack	0
Data	2
Heap	148

Table 2.4: Memory segment of UaF bugs.

2.3.2 Where Do Dangling Pointers Point To?

While UaF bugs often occur on heap memory (i.e., heap UaF) and are the focus of most UaF studies, a dangling pointer theoretically can point to prematurely freed memory located on any segments besides heap. Table 2.4 shows the memory segments of UaF bugs in our dataset.

Finding 2. *While heap UaF dominates (accounting for 98.7%), dangling pointers pointing to other memory segments do exist in the wild.*

First, we did not find a real stack UaF case. A stack UaF occurs when a pointer points to the address of a stack variable after its declaring function has returned. A hypothetical example of stack UaF is presented in Figure 2.1.

While no stack UaF was found, we did observe two data UaF. As Figure 2.2 shows, the static variable `module_initialized` is used to guard against multiple initializations. However, when the static module is freed and the corresponding resources get released, the static variable is not cleared. Consequently, if the static module is initialized again, the runtime system will incorrectly consider the module as initialized, and thus it will use the previously freed resources, resulting in UaF.

```
int *func_return_stack_addr() {
   char str[32];
   int value = 1024;
   return &value;
}
// stack_ptr is a dangling pointer to a freed stack frame
int *stack_ptr = func_return_stack_addr();
*stack_ptr = 2048;
```

Figure 2.1: A hypothetical example of stack UaF

Another interesting case is due to the static destruction order fiasco, as shown in Figure 2.3. In C++, the destruction order of static objects is undefined across different compilation units, and therefore, the destruction order of some static objects, e.g., `alias_table` and `parameter_set` in Figure 2.3, becomes non-deterministic at runtime. This could lead to UaF. This patch changes them to be static functions that return static member variables, avoiding the problem.

Finally, as expected, about 98.7% (148 out of 150) real-world UaF bugs occur on heap memory. This is because heap memory is usually dynamic and thus prone to be misused due to coding mistakes. In addition, the lifespan and content of objects on heap are easier to be manipulated by attackers. Consequently, heap UaF generally poses more serious threats.

One more issue to mention is that C/C++ supports arithmetic on pointers, and thus a pointer can point to any byte within a memory block at runtime. During our study, we found 1 heap memory case where the dangling pointer points to the internal (instead of the head) of a memory block. For instance in Figure 4.2, `*_haystack` and `*_needle` point to the memory assigned by `malloc()` since `strup()` calls `malloc()` to allocate memory. The memory will thus be finally released. `ret` points to somewhere within `_haystack` (or null if there is no match) when calling `strstr(_haystack, _needle)`. Since `_haystack` is released, the function then returns a freed memory, which results in UaF.

In summary, while non-heap UaF does exist in the wild, its pattern is relatively

```
static int module_initialized;
static void module_free(void *m) {
    ...
  + module_initialized = 0;
}
static int module_init(void) {
    if (module_initialized != 0) {
        return 0;
    }
    else {
        module_initialized = 1;
    }
}
```

Figure 2.2: Global UaF due to loss of memory flag update[41]

```
  - static std::unordered_map<...> alias_table;
  - static std::unordered_set<...> parameter_set;
  + static std::unordered_map<...> alias_table();
  + static std::unordered_set<...> parameter_set();
```

Figure 2.3: Static destruction order fiasco [42]

simple. By contrast, heap UaF is much more prevalent and complex. In addition, our last example indicates that it is imperative for some dynamic detection to keep track of not only where a pointer points to, but also the object size. Therefore, heap UaF deserves a deeper study and its characterization will be our focus.

2.3.3 Nondeterminism of UaF

Finding 3. *While the majority of UaF bugs are deterministic, a non-trivial portion (35 out of 150) are non-deterministic.*

Many UaF bugs (35 out of 150) are triggered non-deterministically. First, a large portion (28 out of 35) of non-deterministic UaF bugs are actually concurrency bugs. To highlight the distinctions between UaF and general concurrency bugs, we compare our data with the previous study by Lu et al. [121]. Specifically, in our study all collected UaF concurrency bugs are non-deadlock, while 66% of general concurrency bugs are

```
char *strcasestr(const char *haystack, const char *needle) {
    char *ret = NULL;
    char *_haystack = strdup(haystack);
    char *_needle = strdup(needle);
    // _haystack and _needle are assigned by malloc, and would be freed
    ret = strstr(_haystack, _needle);
+   if (ret)
+       ret = (char *)haystack + (ret - _haystack);
    free(_haystack);
    free(_needle);
    return ret; // points to freed memory
}
```

Figure 2.4: Dereference of an strup() object[43]

```
void js_DestroyContext(...){
  js_UnpinPinnedAtom(&atoms) // free
}
void js_DestroyContext(...){
  js_MarkAtom(&atoms,...); // read
}
//error: two functions run concurrently but js_UnpinPinnedAtom should
    happen after js_MarkAtom
```

Figure 2.5: UaF as a concurrency bug [24]

non-deadlock. In terms of bug patterns, 96% of UaF concurrency bugs are triggered by atomicity-violation and order-violation, which is consistent with the observation in [121] (97%). Among the atomicity-violation and order-violation bugs, four are caused by data race [69]. Besides, in our study, the manifestations of all UaF concurrency bugs involve concurrent accesses to a single variable (compared to 66% in [121]). Therefore, it is a good strategy for UaF concurrency bug detectors to focus on atomicity-violation, order-violation, and non-deadlock concurrent accesses to single variables.

Second, the manifestation of UaF could depend on the system state. For instance, when calling certain functions, insufficient memory allocation [52] will result in errors and then cause UaF (see §2.4.1.3). Another special case is static destruction order fiasco as explained above in Figure 2.3.

2.3.4 Line of Free to Line of Dereference

When analyzing and detecting UaF bugs, one important metric is the code range between the creation and dereference of a dangling pointer. This metric is especially valuable and useful for non-concurrent (122 out of 150) UaF analysis. Thus, we measured the lines of code between where the dangling pointer is created and where it is dereferenced for all sequential UaF bugs. We refer to this metric as `Line-of-free-to-Line-of-dereference (LOFTLOD)`. `LOFTLOD` does not represent the number of lines of code to execute at runtime. In particular, we do not count the lines of code in the body of a called function, which is counted as one line. And, in case of a conditional statement, each branch is counted in sequence until reaching the last branch or the statement of the dangling pointer dereference.

When computing the `LOFTLOD` metric, the valuable control-flow information is discarded. In order to provide a clearer picture of the UaF code ranges, we augment `LOFTLOD` with the basic block (BB) information. Specifically, we check if the dangling pointer creation and dereference instructions are in the same BB, adjacent BB, or non-adjacent BB. The BB-augmented `LOFTLOD` metric simplifies the code analysis while still providing valuable information.

BB relationship can be easily acquired in Clang if our checker successfully detects a UaF bug. When a specific bug pattern is identified, the `BugReporter()` function is invoked to provide a trace of the UaF bug. To determine whether the code resides within the same BB, we utilize the `CheckerContext::getBlockID()` function, which retrieves the block ID for a given position. If the block IDs of the memory-free and memory-dereference positions are the same, it indicates that they belong to the same basic block. Conversely, if they differ, we employ the `isAdjacent()` function to determine if they are in adjacent basic blocks. In cases where UaF bugs go undetected, we conduct an analysis of the root cause for each bug by identifying where the memory is freed and exploring potential positions for memory dereference. Based on our understanding of the bug and the analysis of the code structure, we compute the closest distance and obtain the corresponding basic block information.

LOFTLOD	Number of bugs			
	<10	<50	<200	>200
Same BB	27	6	0	0
Adjacent BB	8	3	0	5
Nonadjacent BB	0	21	8	44
Overall	35	30	8	49

Table 2.5: Distribution of basic block (BB) augmented LOFTLOD of the bugs in the collected C/C++ applications.

```
tstate_delete_common(tstate); // free
 if (autoInterpreterState && PyThread_get_key_value(autoTLSkey) == tstate)
     // invalid read
     PyThread_delete_key_value(autoTLSkey);
```

Figure 2.6: Immediate use-after-free

Our analysis results are shown in Table 2.5. Interestingly, there are 35 cases whose `LOFTLOD` is smaller than 10, and 49 cases whose dangling pointer creation and dereference happen in the same or adjacent basic block. For instance, Figure 2.6 presents a real-world case in CPython, where `tstate` is freed and then accessed almost immediately. This implies that a fair amount of sequential UaF bugs are relatively easy to detect. Specifically, it could be a high-rewarding strategy for a detector to give a higher weight to the code snippets after the instructions where dangling pointers are created.

2.3.5 UaF Reproducibility

We also reproduced some of the collected bugs and observed their behaviors by using AddressSanitizer (ASan) [142]. We aim to understand (1) how likely a UaF bug leads to software crashes and (2) how effective ASan is to detect UaF. To this end, we randomly selected 30 bugs from the data set. Table 2.6 lists all of the tested 30 bugs. Some bugs were reported along with the proof-of-concept (PoC) code that was directly used in our test. For the rest, we hand-wrote the PoC code to trigger the bugs. For

each bug, we ran the PoC code (1) without ASan to see if the bug crashes the software and (2) with ASan to see if ASan could detect the UaF bug.

Application crash serves as the main externally observable behavior that modern fuzzers rely on for bug detection. Our evaluation shows that only 3 out of the 30 tested bugs caused crashes. Table 2.6 shows the details of the 30 reproduced UaF bugs. This implies that most UaF bugs are actually "silent" and fuzzers require auxiliary tools for effective UaF detection.

Finding 4. *Without ASan, only 3 out of 30 UaF bugs caused crashes.*

Bug ID	Application	Crash	Pattern
issue24613	CPython	Yes	P5
issue18328	CPython	No	P1
issue25388	CPython	No	P4
issue24100	CPython	No	P8
issue24552	CPython	Yes	P1
issue29028	CPython	No	P5
issue29438	CPython	No	P2
issue24096	CPython	No	P8
issue28275	CPython	No	P4
Bug 1338772	Firefox	No	P1
Bug 750820	Firefox	No	P3
Bug 1388243	Firefox	No	P4
Bug 803853	Firefox	No	P7
Bug 634961	Firefox	No	P4
Bug 1033006	Firefox	No	P2
Bug 1417405	Firefox	No	P8
Bug 999274	Firefox	No	P7
Bug 557174	Firefox	No	P7
Bug 1338772	Firefox	No	P1
CVE-2019-8912	Linux	No	P3
Bug 199403	Linux	No	P4
Bug 200001	Linux	Yes	P4
Bug 199839	Linux	No	P4
#7614	frr	No	P2
#6469	Apache	No	P1
#601	coturn	No	P4
CVE-2018-10685	lrzip	No	P5
CVE-2018-11416	jpegoptim	No	P2
CVE-2018-20623	GNU Binutils	No	P5
#2072	scala-native	No	P2

Table 2.6: List of the reproduced 30 bugs.

When a dangling pointer points to a freed and then re-allocated memory block, ASan could have false negatives. To address this issue, ASan implements a quarantine mechanism. After a memory block is freed, it is marked as unaddressable and put into quarantine. The reuse of the freed memory in quarantine will be delayed and will not be allocated by `malloc` in a short time. The quarantine is implemented using a first-in-first-out (FIFO) queue, which holds a fixed amount of memory at any time. When the queue is full, some old memory will be moved out from quarantine and thus will become available for reuse. As a result, the size of the quarantine queue is critical for effective UaF detection, with a small one leading to many false negatives and a large one incurring high performance overhead. Clearly, it is meaningful to evaluate how reliable the quarantine mechanism is in practice, especially with the default quarantine size.

Finding 5. *Given the default quarantine size of 256MB, ASan successfully detected all of the tested 30 UaF bugs.*

Overall, our tests show that the default quarantine size of 256MB is sufficient in real world. Even when the quarantine size is reduced to 64MB, ASan can still reliably detect 27 out of 30 UaF bugs.

2.4 UaF Characterization

To better facilitate the manifestation and detection of UaF bugs, we conduct a deeper study for UaF characterization. In particular, we aim to answer the following questions:

- How are UaF bugs introduced? This can provide clues on the best practices to avoid UaF.
- Do UaF bugs have patterns that can provide insights on the design of bug detection tools?

We discover that concurrent UaF has no remarkable differences in root causes from general concurrency bugs studied in previous work [121]. Therefore, we mainly

Category		Freq.
Immediate UaF	move-free-before-use (P1)	12
	save-before-free (P2)	13
Raise a flag	not or falsely updated (P3)	12
	not checked (P4)	7
Memory resize	improper memory resize (P5)	9
API	API misuse (P6)	14
Double free	free inside a loop (P7)	20
Ref count	use of borrowed ref (P8)	26
	over-decremented ref (P9)	6
	non-decremented ref (P10)	3
	misused ref-count API (P11)	3
Others	other causes (P12)	35

Table 2.7: The occurrence frequency of each UaF pattern. Note that the total number is more than 150 since there might be overlaps between the two patterns. For example, both flag error and misuse of reference count could lead to double free, and API misuse could result in reference count error.

focus on sequential UaF bugs in this section. Meanwhile, as reference counting is a promising technique to address UaF and has been widely adopted, it is desirable to understand its efficacy in preventing UaF in practice. Table 2.7 shows the occurrence frequency of each pattern. The details are discussed in the rest of this section.

2.4.1 All UaF Bugs Are Not Created Equal

2.4.1.1 Immediate-use-after-free

The dangling pointers in 25 UaF bugs get dereferenced only after a few lines of code where the memory is freed. We divide the bugs into two patterns based on their fixes.

Move-free-behind-use (P1): For 12 bugs, the patch is simply to move the memory free after the pointer dereference statements. A typical example can be found in Figure 2.6. Such simple cases should have been caused by programmers' carelessness. And, because they often do not crash programs and even in some cases the expected data can be read through the dangling pointers, such bugs can be easily overlooked.

```
+ fstWriterContext* const parent = xc->xc_parent;
  free(xc);
- xc->xc_parent->in_pthread = 0;
- pthread_mutex_unlock(&(xc->xc_parent->mutex));
+ parent->in_pthread = 0;
+ pthread_mutex_unlock(&(parent->mutex));
```

Figure 2.7: A UaF bug of the save-before-free pattern[44]

Save-before-free (P2): We also observed 13 cases where a memory block is freed, while some internal pointer fields are still needed in the subsequent execution. Figure 2.7 shows a typical example. After `xc` is freed, the subsequent dereference of `xc->xc_parent` would lead to UaF. Actually, this pattern is more common during list manipulations, e.g., the node deletion in a linked list [53]. The fix to such bugs is usually straightforward. A new variable is used to hold the data needed in the subsequent executions, e.g., `parent=xc->xc_parent` in Figure 2.7.

Implications. Motivated by the fact that the UaF bugs of this "immediate-use-after-free" pattern often do not have complicated code logic and their fixes are usually straightforward, a static checker could be built to accurately detect such bugs with few false positives.

2.4.1.2 Raise a red flag

A flag variable is often used in software to track the runtime status. If falsely updated or inappropriately used, it could lead to memory corruptions including UaF. We found that 19 out of 150 UaF bugs are due to the inappropriate use of flags, and they fall into two specific flag related operations.

The flag is not or falsely updated (P3). Figure 2.8 presents an example of this flag related operation. `pci_alloc_consistent()` returns a `NULL` pointer if there is insufficient memory and its return value is checked. If the return value is `NULL`, the control flow will jump to label `create_cq_error`, free the memory and return the variable `ret`. Also, `beiscsi_cmd_cq_create()` returns `0` to `ret` if the execution succeeds.

```
static int beiscsi_create_cqs(...) {
   int ret = -ENOMEM;
   for (i = 0; i < phba->num_cpus; i++) {
        cq_vaddress = pci_alloc_consistent(...);
        if (!cq_vaddress) {
+           ret = -ENOMEM; // patch to fix
            goto create_cq_error;
        }
   }
create_cq_error:
    for (i = 0; i < phba->num_cpus; i++) {
        if (mem->va)
            pci_free_consistent(...); // free
    }
    return ret;
}
```

Figure 2.8: UaF due to a falsely updated flag in `beiscsi_create_cqs()` defined in `drivers/scsi/be2iscsi/be_main.c` [30]

Unfortunately, the flag variable `ret` is not correctly updated in the loop, and this could lead to UaF. For example, when all statements in the first few rounds of the loop succeed and thus `ret` has value `0`. After a few rounds, `pci_alloc_consistent()` fails and thus the control flow jumps to `create_cq_error`. Note that the value of `ret` is still `0` now and the caller of `beiscsi_create_cqs()` would consider the function succeeds to execute. However, the memory has actually been freed by `pci_free_consistent()`. A UaF bug is triggered when the freed memory is accessed in the caller of `beiscsi_create_cqs()`. Similar cases contribute to 12 out of 19 flag UaFs.

The flag is not checked (P4). A flag variable often tracks the status of an object at runtime. When the object is accessed, the program should carefully check the flag value. Otherwise, it could lead to UaF if the object is not initialized or has been freed. For instance, the code in Figure 2.9 does not check if the audio stream has been initialized. If not, further operations on `mAudioStream` would trigger UaF.

Implications. Previous work has proposed various methods to detect the bugs caused by the missing checks of certain variables (e.g., flags and permissions)

```
void AudioCallbackDriver::StateCallback(cubeb_state aState) {
-  if (aState == CUBEB_STATE_ERROR) {
+  if (aState == CUBEB_STATE_ERROR && mAudioStream){
    }
}
```

Figure 2.9: Missing check of a flag variable leads to UaF [33]

in source code [165, 129, 173]. Our empirical study finds that the root causes of flag related UaF are diverse and existing methods cannot cover all of them. This motivates further improvement of these work to provide better detection for UaF bugs. Meanwhile, fuzzing, one of the most popular bug detection approaches, might need to adopt special measures so that the flag related UaF bugs are more likely to get triggered.

2.4.1.3 Memory resize considered harmful

Memory resizing is another common root cause of UaF (P5). Specifically, 9 out of 150 UaF bugs are due to memory resizing, such as reallocation, manipulating resizable objects, and inappropriate memory change. There are a set of functions that can alter the size of memory blocks at runtime, e.g., `realloc()` and `resize()`. However, developers need to bear in mind multiple caveats when using these functions.

Figure 2.10 shows a typical example. `realloc(p, newsize)` returns a value to the pointer `pnew`. A non-`NULL` `pnew` indicates that a new chunk of memory has been successfully allocated. However, when `pnew` is `NULL`, it does not necessarily mean that the function has errors like insufficient memory assignment. Actually, it can also indicate that the memory is successfully freed when `new_size=0` [1]. The freed memory may be accessed later, resulting in UaF, when the special semantics of `realloc(new_size=0)` is ignored.

[1] More accurately, according to C18 standard (§7.22.3.5) [64], "If size is zero and memory for the new object is not allocated, it is implementation-defined whether the old object is deallocated."

```
void *pnew = realloc(p, new_size);
if (pnew) {
    sFreetypeMemoryUsed += FreetypeMallocSizeOfOnAlloc(pnew);
-  } else {
+  } else if (size != 0) {
    sFreetypeMemoryUsed += FreetypeMallocSizeOfOnAlloc(p);
}
```

Figure 2.10: UaF due to realloc() with a special size parameter 0 in `/gfx/thebes/gfxAndroidPlatform.cpp` [31]

Even when the `new_size` has a reasonable non-zero integer value, `realloc()` can exhibit three behaviors from the perspective of function callers: (1) simply expand the original memory block "in place" [2]; (2) the new memory is failed to allocate and the old memory is returned; or (3) allocate a new memory block, copy data, and free the original one. Actually, `realloc()` essentially implies a memory free [64] and can potentially cause dangling pointers. Figure 3.6 shows such an example. In Python, `array.fromstring()` appends items from the string, interpreting the string as an array of machine values [65]. This method expects the input string to be a different object from the array. However, the original buggy implementation failed to handle the case when the array itself is passed to `fromstring()`. In such a case, two pointers, `self->ob_item` and the pointer to the argument, point to the same memory block. Later in the method, `PyMem_RESIZE()` - which calls `realloc()` - is invoked to resize the memory pointed by `self->ob_item`. As described above, this operation can trigger the deallocation of the original memory, and thus the pointer to the argument becomes dangling. To fix this UaF bug, Python disallows the array to append to itself.

Moreover, many functions in practice implicitly assume that the input parameters are not resizable objects. For instance, the function `PySlice_GetIndicesEx()`

[2] C18 standard (§7.22.3.5) [64] only defines the behavior of deallocating the old object and returning a pointer to a new object, and has no notion of "in place" expansion. However, it is possible that the memory block is freed and then allocated at the same location. Therefore, to external observers, the behavior looks like "in place" expansion.

```
static PyObject *
array_fromstring(arrayobject *self, PyObject *args){
    char *str;
    Py_ssize_t n;
    int itemsize = self->ob_descr->itemsize;
    if (!PyArg_ParseTuple(args, "s#:fromstring", &str, &n)) //str is parsed
        from args. When an array is passed to itself (i.e., args=self), we
        get str == self->ob_item.
        return NULL;
+   if (str == self->ob_item) {
+       PyErr_SetString(PyExc_ValueError, ...);
+       return NULL;
+   }
    char *item = self->ob_item;
    // When str == self->ob_item, item == str.
    PyMem_RESIZE(item, char, (Py_SIZE(self) + n) * itemsize);
    // realloc() is invoked in PyMem_RESIZE(), frees the original memory
        pointed by item and allocates a new memory block which may or may
        not be the same as the original one. Therefore, if item == str, str
        may point to freed memory after PyMem_RESIZE().
    memcpy(item + (Py_SIZE(self) - n) * itemsize, str, itemsize*n);
    //If str is dangling, a UaF occurs here.
}
```

Figure 2.11: UaF when a function calls realloc [25]

in Python (`Objects/sliceobject.c`) expects to operate on a non-resizable sequence. Otherwise, the sequence can be resized, e.g., calling `PyNumber_AsSsize_t()`, after passing its length to `PySlice_GetIndicesEx()`. This can lead to returning indices out of the length of the sequence, and thus accessing that part of the sequence will trigger UaF [35]. To avoid such UaF, the parameters to such functions should be carefully checked.

Finally, memory resize has to be carefully handled in concurrent systems. Specifically, a memory block could get resized at any time during the execution of racy code, e.g., the resizable buffer in `SNDRV_RAWMIDI_IOCTL_PARAMS` ioctl [36]. Same as other concurrency bugs, synchronization such as locking is required to protect accesses to resizable memory.

Implications. Obviously, memory resize operations deserve special attention

```
- auto& subscriber = subscriber_it->second;
+ auto subscriber = subscriber_it->second;
  ef.event_subs_.erase(subscriber_it);
  subscriber->tearDown();
  subscriber->state(EventState::EVENT_NONE);
```

Figure 2.12: auto and auto& [63]

to detect and avoid UaF bugs. Specifically, a static checker can be built to detect the potential misuses of memory resize operations. In addition, static bug detection systems need to take the fact that parameters could be potentially resizable into consideration.

2.4.1.4 Misuse of APIs and features

Another common UaF root cause is **the misuse of specific APIs and features (P6)**. Developers often misunderstand the semantics of these APIs and features. The `realloc()` and `resize()` described above fall into this category. Our study also finds multiple other APIs and features whose misuses have resulted in UaF in practice.

First, the misuse of `list.erase(it)` and `list.remove(*it)` has caused multiple UaF bugs. In C++, `list.erase(it)` [54] only deletes one object from the list, while `list.remove(*it)` [55] can delete multiple objects with the same value as `*it` from the list. The misuse of `list.remove()` - where `list.erase()` should be used - could lead to UaF because more objects might get deleted unexpectedly [56]. A similar UaF bug occurred in [66], where the developers misused two user-defined functions and freed the whole chain of `rte_mbuf`, instead of one segment. `list.erase(it)` and `list.remove(*it)` are frequently misused because developers often get confused about their semantics. In particular, some applications have overridden the two methods using custom implementations with different semantics. For instance, `list.remove(*it)` in Python only removes the first matching element from the list.

Another case is the reference operator `&` in C++, which denotes an alias for an object. Developers sometimes get confused about a reference and a pointer. However,

a reference behaves quite differently from a pointer, and the misuse of a reference could lead to UaF bugs. Figure 2.12 shows an example of the UaF bug caused by the misuse of references. `auto& subscriber` is a reference to `subscriber_it->second`. Once `subscriber_it` is freed, `auto& subscriber` points to a freed object and thus causes UaF. Finally, the misuses of some APIs in various reference counting implementations could also cause UaF. We will describe these cases in §2.4.2.

Implications. As an immediate improvement for UaF detection, we can model the semantics for all of the APIs described above and build a static checker to detect the UaF bugs that are caused by the misuses of these APIs. Furthermore, we can explore to extend to more sensitive APIs. However, there are two main challenges to address. First, it requires a method that can systematically identify all potentially sensitive APIs. Second, the semantics of these APIs need to be modeled automatically. Actually, there has already been some work on sanitizing API usages [171]. Further research is needed to evaluate if and how such methods can be extended to detect UaF bugs caused by the API misuses.

2.4.1.5 Double free

Double free (P7) is a special type of UaF. It occurs when a memory block is freed more than once. Our study finds 20 double free UaF. For instance, the bug in [40] forgets to set the pointer `sock->sk` to `NULL` after the memory gets freed. As a result, the same memory could get repeatedly freed, resulting in double free. Moreover, we find that multiple double free UaF happens because `free()` is put inside a loop.

Implications. Loop is one major location for the occurrence of double free. Static analysis should focus on the memory tracking inside the body of the loop to detect double free. Basically, to avoid double free, a memory block should not have been freed when calling `free()`. For example, memory allocators can provide a new API `safe_free()`, which checks the memory status before calling `free()`. This helps to mitigate all nine double free bugs in our study. Therefore, it deserves further research

on how to achieve efficient checks in `safe_free()` without noticeably degrading the performance of memory allocators.

2.4.1.6 Miscellaneous cases

Finally, there are 35 bugs in our dataset that are mainly caused by the faulty code logic. Such logic errors are usually application-specific, and thus no meaningful patterns could be derived. For example, in plan9port [57], `thread0` is a special thread that is used for the context switches back to the scheduler, and it cannot be freed until the end of the scheduler function. If `thread0` gets freed unexpectedly, the UaF bug would be triggered during context switches. Another similar case occurs in Linux Kernel, where the unexpected free of a special memory block `block#0` could cause UaF [37]. Due to space limit, we will not enumerate all cases here.

2.4.2 Reference Counting, A Panacea for UaF?

In order to achieve memory safety, it is a common idea to use reference counters to manage the lifetime of objects. The core idea is to maintain a counter per object, which gets incremented whenever a new reference to the object is taken and decremented when a reference is released. An object can only be released when the counter reaches zero. Many software suites such as Linux kernel and Python have adopted reference counting. In the best case, reference counters can eliminate UaF bugs because no dangling pointer could be generated. Unfortunately, we find that a bunch of UaF bugs - 38 out of 150 - are caused by implementation mistakes or misuse of reference counters. In particular, around 66% of UaF bugs in Python are caused by errors related to reference counters. We refer to these bugs as *refcount UaF* and we present their common patterns in this section.

It is risky to use something borrowed (P8). Reference counters in most implementations such as Python [58] and Linux kernel [127] require explicit manipulations, e.g., using the API pair `Py_INCREF()` / `Py_DECREF()` in Python. A safe practice is to always increment the reference counter of an object whenever getting a reference

to the object and and decrement after the reference is released. But it is actually not necessary for every reference, especially when we are sure that the reference counter must be larger than zero during the whole lifetime of the reference. For instance, the reference counting implementation in Python guarantees to increment the reference counter for every object passed as a function argument at the entry point of the called function. In Python, the references that do not increment the reference counter of the referred object (*referent*) are called *borrowed references* [59]. In our study, we extend this definition to all reference counting implementations in C/C++ [3].

It is error-prone to use borrowed references in practice. Our study reveals that the majority of refcount UaF bugs - 29 out of 39 - are due to the misuse of borrowed references. For example, in Figure 3.4, `PyList_GET_ITEM(slotnames, i)` returns a borrowed reference `name` to the object at index `i` in list `slotnames`. However, `name` cannot prevent the referent from being freed. If the referent gets freed before or during the execution of `PyObject_GetAttr(obj, name)`, `name` becomes a dangling reference and the dereferencing operations will trigger UaF. Actually, it is a common pitfall to grab a borrowed reference to an object in `lists`, `arrays` or `tuples` and hold on to it for a while without incrementing the reference counter. In addition, developers sometimes are aware that a borrowed reference is unsafe to operate and that the reference counter should be incremented to avoid UaF, but they fail to do it properly. Figure 2.13 shows a typical example. `kvm_ioctl_create_device()` creates and initializes a device `dev` that holds a borrowed reference `kvm` to a virtual machine (VM) object. Then, `anon_inode_getfd()` transfers the ownership of the reference `dev` to the file descriptor table of the caller of `kvm_ioctl_create_device()`. After the ownership transfer, developers call `kvm_get_kvm()` to increment the reference counter. Unfortunately, the borrowed reference is still misused here and the reference counter should be incremented before `anon_inode_getfd()`. Specifically, an attacker can close the file

[3] Note that the borrowed reference in Rust has different semantics [60].

```
static int kvm_ioctl_create_device(...){
      dev->kvm = kvm;
      ops->init(dev);
   +  kvm_get_kvm(kvm);
      ret = anon_inode_getfd(...);
      if (ret < 0) {
   +      kvm_put_kvm(kvm);
     }
   -  kvm_get_kvm(kvm);
 }
```

Figure 2.13: A refcount UaF occurs because a borrowed reference is not properly converted to a non-borrowed one [38]

```
Py_LOCAL(PyObject *) _PyObject_GetState(...) {
     // name is a borrowed reference, and the referent might get
        unexpectedly freed.
     name = PyList_GET_ITEM(slotnames, i);
+    Py_INCREF(name);
     value = PyObject_GetAttr(obj, name);
     // error: misused borrowed reference
+    Py_DECREF(name);
 }
```

Figure 2.14: UaF due to a misused borrowed reference [26]

descriptor, which triggers the code logic that automatically drops the borrowed reference `kvm`. This can cause the reference counter of the VM object referred by `kvm` to drop to zero and the VM object will be freed prematurely.

Reference counters get-over decremented (P9). Another common error that developers make is that the API to decrement reference counters could be unexpectedly and redundantly invoked, causing reference counters to reach zero prematurely. In case there are still in-use references to the freed objects, UaF occurs. For instance, the function `sctp_assoc_update()` in Figure 2.15 calls `sctp_auth_asoc_init_active_key()` to generate a new key and delete the old one. For key deletion, the function `sctp_auth_key_put()` is called to decrement the reference counter of the old key by one. When the reference counter reaches zero, the memory associated with the key gets freed. In this buggy case,

```
void sctp_assoc_update(...){
-   sctp_auth_key_put(asoc->asoc_shared_key);
    sctp_auth_asoc_init_active_key(...);
}
int sctp_auth_asoc_init_active_key(...){
   sctp_auth_key_put(asoc->asoc_shared_key);
}
```

Figure 2.15: UaF due to reference counter over-decrement [27]

developers mistakenly call `sctp_auth_key_put()` twice, one in `sctp_assoc_update()` and the other in `sctp_auth_asoc_init_active_key()`.

Reference counters are not decremented (P10). While over decrementing can cause UaF, missed decrements can sometimes cause UaF too. Reference counters are usually represented using integers, e.g., `atomic_t` in Linux kernel. When developers forget to decrement a reference counter, the reference counter could overflow and become zero. Therefore, the referent would be freed prematurely. For example, the buggy code in Figure 2.16 gets a reference `keyring` and increments the reference counter by calling `find_keyring_by_name()`. Suppose that the reference counter should be decremented at the exit of `join_session_keyring()`; but under certain condition (`keyring == new->session_keyring`), the code jumps to label `error2` and fails to decrement the reference counter. Attackers can then repetitively trigger this buggy code and cause the reference counter to overflow.

Reference counting APIs are misused (P11). Some implementations of reference counting provide rich APIs, which can give developers more flexibility to trade off some runtime checks for better performance. Unfortunately, some APIs could be misused and thus lead to UaF. Figure 2.17 shows an example in Linux kernel. The API `maybe_get_net(struct net *net)` checks if the current reference counter for the struct `net` is zero. If so, it means that the struct has been freed and `NULL` is returned. Otherwise, a reference to `net` is returned and the reference counter is incremented. By contrast, the API `get_net(struct net *net)` simply increments

```
long join_session_keyring(const char *name){
   // Look for an existing keyring of this name.
   // If the keyring exists and the reference is returned, its reference
       counter is incremented by one.
   keyring = find_keyring_by_name(name, false);
   if (PTR_ERR(keyring) == -ENOKEY) {
   } else if (IS_ERR(keyring)) {
   } else if (keyring == new->session_keyring) {
+     key_put(keyring);
      ret = 0;
      goto error2;
   }
  key_put(keyring);
}
```

Figure 2.16: UaF due to the missing decrement for a reference counter [32]

```
   spin_lock_bh(&net->nsid_lock);
   peer = idr_find(&net->netns_ids, id);
   if (peer)
-     get_net(peer);
+     peer = maybe_get_net(peer);
   spin_unlock_bh(&net->nsid_lock);
```

Figure 2.17: UaF due to a misused reference counting API [34]

the reference counter and returns a reference to `net`. Obviously, a misused `get_net()` - where `maybe_get_net()` should be used - will cause UaF as shown in [34]

Summary. Our study shows that refcount UaF bugs are quite prevalent. However, there is only limited research on detecting reference counting bugs and all of these methods have certain shortcomings. The refcount tracing and balancing techniques in Firefox enable developers to track down the leak of referents at runtime [62]. The detection capability of such techniques highly depends on the quality of inputs. Unfortunately, it is often quite challenging to generate inputs with high quality. Meanwhile, researchers have also proposed multiple static analysis methods to detect reference counting bugs in source code [97, 117, 126, 151]. These methods are mostly limited in their scopes. Specifically, they fail to handle the unprotected borrowed references and

	UAFBenchPlus	UaFBench
# of Applications	41	11
Application Size†	90K∗ ∼ 2479M∗∗	9K ∼ 502M
Application Language	C, C++	C
# of Bugs	120	14
Patch Size ‡	1 ∼ 215	1 ∼ 13
# of Patterns	All	P2 ∼ P5, P7

Table 2.8: Comparison of UAFBenchPlus and UaFBench [136]. †Application size is measured in lines of code. ‡Patch size is measured in lines of code added and deleted. ∗K for Thousand. ∗∗M for Million.

misuses of APIs. More effort is needed to improve the coverage of refcount UaF bugs.

2.4.3 UaF Benchmark

Nguyen et al. [61] built a fuzzing benchmark UaFbench comprising 14 real bugs from 11 applications. However, UaFbench has several limitations. First, UaFbench only considers C applications, and no C++ UaF bug is included. Second, all of the included UaF bugs are sequential. Our empirical study shows that concurrent UaF bugs comprise a non-trivial portion and should not be overlooked. Third, there are limited patterns of the bugs and the buggy code is relatively simple in terms of the patch size. Motivated by the above limitations and our empirical findings, we construct a new benchmark UAFBenchPlus that contains 120 out of the 150 collected UaF bugs. UAFBenchPlus only includes bugs that support recent releases of Clang [70] which are commonly used to build bug detectors. The detailed comparison between UaFBench and UAFBenchPlus is shown in Table 2.8. It shows UAFBenchPlus's specification in terms of the number of applications, lines of code, and bug patterns. It is by far the most comprehensive benchmark for UaF.

Note that UAFBenchPlus only contains 120 out of the 150 collected bugs. We have to exclude 30 bugs because the bugs may involve different versions of code in a long time span and some bugs exist in the old versions of applications that cannot

be successfully configured in current environments. In addition, some software only supports GCC but not Clang.

2.5 `Palfrey`: A Pattern-based UaF Detector

Our results in §2.4 demonstrate that a large portion of UaF bugs follow specific patterns, some of which can be translated into simple static checkers. For instance, as described in §2.4.1.3, when calling the function `realloc()`, developers should carefully handle the case where the parameter size equals 0, to avoid potential UaF. To measure the applicability of our findings, we built a pattern-based static UaF detector, `Palfrey`. Next, we discuss how `Palfrey` is implemented and how effective it is for discovering UaF bugs.

2.5.1 Design and Implementation of `Palfrey`

Overall, we extracted seven rules from our empirical study and developed 11 checkers in `Palfrey`. These rules are quite straightforward. For example, to handle unprotected borrowed references, the checker conducts analysis on a list of manually crafted functions that return borrowed references, and then report a bug if a borrowed reference returned by these functions is not protected by primitives such as `Py_INCREF`.

Our tool includes a framework for Abstract Syntax Tree (AST) and Control Flow Graph (CFG) analysis, a module for pointer alias analysis, and a module for pattern analysis. Our checkers are mainly based on the AST, call graphs, and CFG of programs. First, the AST is generated from the source code of a program, and then the call graphs and CFGs are constructed based on the AST. Next, after the CFG, AST, and call graphs are constructed, a pattern matching engine is used to identify if UaF bugs exist. The inter-procedure analysis is performed to determine if a memory free operation happens, while the AST analysis is used to decide if specific functions are called. Subsequently, the data flow analysis is performed locally for the alias analysis, and the pattern matching engine is eventually used to report UaF.

The procedure to apply those rules is composed of the analysis on function call and return, the parameter analysis, and memory tracing. For example, to detect the use of unprotected weak reference in Figure 3.4, the AST analysis is first conducted to locate the functions of interest (in this case, `PyList_GET_ITEM(slotnames, i)`), and then the function return value `name` is further traced to see if it is protected before it is dereferenced.

`Palfrey` is built upon the `LLVM` (12.0.0) compiler infrastructure [71], which provides great support for both C and C++ code. Our checkers conduct path-insensitive analysis and they are mostly straightforward to implement. In particular, we leverage the data flow analysis provided by `LLVM` to detect the code snippets that match the bug patterns.

2.5.2 Evaluation of `Palfrey`

In order to demonstrate the effectiveness and efficacy of `Palfrey`, we first conducted evaluations on the 150 collected bugs. In particular, we compared `Palfrey` against four well-known static detectors, the open-source Infer version v1.1.0 [67], commercial PC-lint Plus [68], Clang [70], and GCC static analyzer [72]. All detectors support `C/C++` and claim to be able to find UaF bugs. Then, we applied `Palfrey` to the latest releases of a set of applications to evaluate its capability of finding new bugs. We ran the experiments on an x86-64 desktop with six-core Intel i7-9750 2.6G Hz processors and 16GB memory on Ubuntu 18.04.5. Our evaluation results are detailed below.

Effectiveness on 150 collected bugs. We successfully ran the five tools on only 75 bugs due to the system environment problems. For example, Infer relies on Clang to compile the target code, and it might fail if the target code cannot be compiled with Clang. Unfortunately, our dataset contains some bugs that cannot work with Clang. For instance, the Linux kernel cannot support Clang until 2016. Therefore, we only presented the results for the 75 successful cases.

Table 2.9 lists the results. All five tools successfully detected some of the bugs,

	C	C++	Overall
Infer	14/53	2/22	16/75
PC-lint	10/53	3/22	13/75
GCC	7/53	2/22	9/75
Clang	16/53	5/22	21/75
`Palfrey`	34/53	6/22	40/75
`Palfrey` ∩ Infer	9/53	2/22	11/75
`Palfrey` ∩ PC-lint	7/53	2/22	9/75
`Palfrey` ∩ GCC	7/53	1/22	8/75
`Palfrey` ∩ Clang	12/53	2/22	14/75

Table 2.9: Detection results of `Palfrey`, Infer, and PC-lint on our collected UaF bugs. Infer and PC-lint failed to run on the other 75 bugs, due to various system environment and configuration problems.

	`Palfrey`		Infer		PC-lint		GCC		Clang	
	Time†	Mem‡	Time	Mem	Time	Mem	Time	Mem	Time	Mem
swig	12	0.17	130	1.85	150	1.96	44	0.67	22	0.33
Python	68	0.44	570	4.03	482	3.95	211	2.02	101	0.79
Linux	1,682	1.2	4,867	3.12	5,016	3.91	2583	3.4	2699	2.4
u-boot	440	0.16	2,170	4.65	2,448	6.03	730	1.53	573	0.32
z-3	40	0.26	2,059	3.76	1,830	3.12	140	1.14	54	0.34

Table 2.10: Comparison of the time and memory overhead of three tools. † Time is measured in seconds. ‡ Mem is the memory overhead measured in gigabytes (GB).

especially those with simple patterns. However, `Palfrey` significantly outperforms Infer, PC-lint, Clang, and GCC static analyzer. A further investigation reveals that the superiority of `Palfrey` stems from the new bug patterns that the other four tools do not have. None of the other tools can reliably handle the UaF bugs caused by the misused borrowed references. While Infer can detect static initialization order fiasco, it fails to cover the static destruction order fiasco. Moreover, `Palfrey` is able to find some more complicated UaF bugs through the parameter analysis on certain functions.

We also evaluated the efficiency of `Palfrey` in terms of memory overhead and how long it takes to check a codebase. Due to space limit, we only present the results on five selected applications in Table 2.10 and the results on other applications are

roughly similar. Overall, `Palfrey` consumes 23.6% $\sim$ 97.2% less memory than Infer, PC-lint, Clang, and GCC static checker. `Palfrey` also takes much less time to check a codebase, with 28.6% $\sim$ 98.3% less time than Infer, PC-lint, Clang, and GCC static checker. `Palfrey` is much more efficient to detect UaF in terms of memory consumption and time overhead because pattern analysis is much more efficient to locate possible bugs than general syntactic analysis.

New bugs. To evaluate `Palfrey`'s capability to find new bugs, we applied it to the latest releases of Python (v3.8.12), Linux Kernel (5.14), Firefox (90.0), and GitHub `C/C++` projects with more than 1K stars as of 2021-10-01. `Palfrey` successfully reveals nine bugs in seven applications, three of which we reported and received confirmations from developers (details in Table 2.11). We were able to test six applications using Infer, seven using PC-lint, nine using Clang and GCC static analyzer. Infer can find four out of the nine bugs, PC-lint can find three, GCC static analyzer can find three, and Clang can find six.

Application	Buggy function	Pattern	Unique? †	Confirmation
u-boot	membuff_uninit [*li/membuff.c:389*]	P7	Yes	Reported
u-boot	setjmp [*lib/efi_loader efi_boottime.c:2930*]	P2	No	Reported
Z3	ziplistPush [*src/quicklist.c:511*]	P2	No	Reported
dcmtk	base_clear [*ofstd/include/dcmtk/ofstd/ofmem.h:261*]	P2	No	Reported
dcmtk	insertSequenceItem [*dcmdata/libsrc/dcitem.cc:4203*]	P2	No	Reported
dcmtk	destroy [*ofstd/include/dcmtk/ofstd/ofmem.h:261*]	P1	Yes	Reported
osquery	deregisterEventSubscriber [*osquery/events/eventfactory.cpp:190*]	P2	No	Confirmed
dandere2x	get_diamond_search_points [*block_plugins/block_matching/DiamondSearch.cpp:97*]	P2	Yes	Confirmed
yosys	module-¿remove [*src/database.c2022*]	P2	No	Confirmed

Table 2.11: Summary of new UaF bugs detected by `Palfrey`. †This column indicates if the bug is uniquely detected by `Palfrey`.

Detected bug patterns	Accuracy
Immediate-use-after-free	20/25
Unprotected borrowed references	20/26
Unsafe memory resize	7/7
Double free	4/20
Parameter analysis for sensitive functions	6/9
Functions return illegal stack memory	0/0

Table 2.12: Detection accuracy of `Palfrey` in various bug patterns

False positives and false negatives. There are eleven false positives by `Palfrey` in all of the tested applications. We found that the false positives are mainly caused by inaccurate pointer alias analysis. Compared to many other static bug detectors, `Palfrey` generates much fewer false positives. This is because our checkers target specific patterns and enforce strict constraints, which can reduce false positives.

Table 2.12 shows detection accuracy of `Palfrey` in various bug patterns. A single checker is specifically designed to detect unprotected borrowed references in CPython written in C, while two checkers are created for each of the other five bug patterns to analyze C and C++ programs. False negatives occur due to three reasons: (1) pointer alias analysis is inaccurate and the checker cannot support pointer arithmetic analysis, for example, the code in Figure 4.2; (2) the borrowed reference checker only applies to CPython, but some cases in Linux require deeper analysis to locate possible borrowed references; (3) our double free checker is able to detect repeated free in a loop but unable to detect all general cases.

Overall, our evaluation demonstrates that `Palfrey` is effective and efficient. In particular, leveraging the bug patterns identified in our study, `Palfrey` significantly outperforms existing static UaF detectors with much less overhead. Although `Palfrey` is still preliminary, we envision the promising results can motivate and inspire more future work.

2.6 Summary

This work provides a comprehensive study of real-world UaF bugs, examining their patterns, manifestations, fixes, and other characteristics. Our study is based on 150 UaF bugs, with 115 deterministic and 35 non-deterministic ones, collected from a diverse set of applications such as Linux Kernel, CPython, etc. Our study reveals a number of interesting findings that can provide valuable guidelines for UaF detection and protection. For instance, a main finding is that the root causes of UaF bugs can generally be categorised into 11 patterns. To demonstrate the applicability of these findings, we build a simple static UaF detector, `Palfrey`. Our evaluation shows that `Palfrey` outperforms existing tools in terms of both effectiveness and efficiency. In particular, `Palfrey` successfully detects nine new UaF bugs. Finally, to foster future research, we will make our dataset and tools publicly available.

Chapter 3

AN EXPLORATION OF FUZZING FOR DISCOVERING USE-AFTER-FREE VULNERABILITIES

In the previous chapter, we discussed the prevalence and critical nature of UaF vulnerabilities, emphasizing the importance of detecting and mitigating these vulnerabilities. We introduced the challenges associated with UaF bugs, especially when it comes to their detection through fuzzing techniques. This chapter delves into our second research work, which aims to explore the effectiveness of fuzzing in identifying UaF vulnerabilities, considering the inherently random nature of fuzzing and the complexities of UaF bug detection.

Fuzzing has gained recognition as a dynamic testing technique capable of uncovering software bugs by providing unexpected and often malformed inputs to target programs. Despite its remarkable success in discovering various types of vulnerabilities, the relationship between fuzzing and the detection of UaF vulnerabilities remains uncertain. UaF vulnerabilities present unique challenges because they involve multiple stages, including memory allocation, deallocation, and dereference, making them particularly difficult to detect.

3.1 Motiviting Example: A Challenging Case for UaF Fuzzing

Figure 3.1 shows an example of UaF and this bug is a reference counter overflow vulnerability in the FreeBSD kernel's `mqueuefs` subsystem. The vulnerability occurs in the `sys_kmq_timedsend()` function, where a failure in the `copyin()` function call can lead to an unreleased reference to a file object, causing a reference counter overflow. Specifically, if copyin() fails due to an invalid user space address provided by

`uap->abs_timeout`, the function returns without decreasing the reference count, potentially leading to a reference counter overflow. The patch fixes this vulnerability by adding a `goto out;` statement after the `copyin()` call, which ensures that if `copyin()` fails, the function will jump to the cleanup code (`out: label`) and properly release the reference to the file object using `fdrop()` before returning.

This case presents a unique challenge due to its inherent difficulty in triggering the vulnerability. To exploit the reference counter overflow in the FreeBSD kernel's mqueuefs subsystem, a specific sequence of events must occur, requiring the execution path to be traversed at least 65536 times before the counter overflow can potentially take place. This complexity implies that traditional parameters employed by fuzzing tools, such as code coverage, may not accurately represent the precision and depth of testing required to uncover such vulnerabilities. The intricacies of this example highlight the limitations of conventional fuzzing tools. The high threshold for triggering the UaF, coupled with the intricate conditions necessary for exploitation, underscores how these tools might struggle to effectively identify and exploit such intricate vulnerabilities. In light of this realization, we have extended our research efforts to better understand and address these limitations, aiming to enhance the capabilities of fuzzing tools and develop more accurate and effective techniques for uncovering complex UaF vulnerabilities. Through this continued exploration, we aim to refine fuzzing methodologies and contribute to the advancement of software security.

3.2 Preliminaries

3.2.1 Fuzzing

Fuzzing is a dynamic software testing technique that involves providing unexpected and often malformed inputs to a target program to identify software bugs. Fuzzing can be classified into different types based on the strategy, level of program analysis, and knowledge about the target program:

- **Black Box Fuzzing:** Black box fuzzing treats the target program as a black box and has no internal knowledge of its implementation. It randomly mutates

```
   int
   sys_kmq_timedsend(struct thread *td, struct kmq_timedsend_args *uap)
   {
           struct mqueue *mq;
           struct file *fp;
           struct timespec *abs_timeout, ets;
           int error, waitok;

           AUDIT_ARG_FD(uap->mqd);
           error = getmq_write(td, uap->mqd, &fp, NULL, &mq);
           if (error)
                   return (error);
                            if (uap->abs_timeout != NULL) {
                   error = copyin(uap->abs_timeout, &ets, sizeof(ets));
                   if (error != 0)
-                                  return (error);
+                                  goto out;
...
+out:
           fdrop(fp, td);
           return (error);
   }
```

Figure 3.1: Reference count overflow [39]

or generates inputs to explore different program states and paths. Black box fuzzers, such as American Fuzzy Lop (AFL) [172] and libFuzzer [143], rely solely on input mutations and code coverage feedback to guide the fuzzing process.

- **White Box Fuzzing:** White box fuzzing, also known as smart fuzzing or coverage-guided fuzzing, uses deep program analysis techniques to understand the internal structure, data formats, and control flow of the target program. It combines symbolic execution, taint analysis, and code coverage feedback to guide the generation of inputs that explore specific program paths likely to trigger bugs. White box fuzzers, such as AFLFast [79], AFLGo [77], and UaFuzz [135] focus on interesting paths or inputs and can improve the efficiency of bug discovery.

- **Gray Box Fuzzing:** Gray box fuzzing combines the black box approach with

some knowledge about the target program. It leverages static or dynamic analysis to gain partial knowledge about the program's structure or certain code paths, which helps generate inputs that target specific areas of the program. Gray box fuzzers aim to improve code coverage and detect vulnerabilities in specific program components.

In addition to the above classifications, there are also other fuzzing techniques that address specific challenges or have specialized objectives:

- **Continuous Fuzzing:** Continuous fuzzing [143] involves running fuzzing campaigns continuously or for an extended period, allowing the fuzzer to adapt and evolve over time. It leverages feedback from previous runs, such as identified vulnerabilities or code coverage improvements, to guide subsequent fuzzing iterations. Continuous fuzzing enables the fuzzer to explore deeper program paths and detect harder-to-find bugs. Projects like OSS-Fuzz [45, 144, 95] employ continuous fuzzing techniques.

- **Binary Level Directed Fuzzing:** Binary level directed fuzzing [135] targets specific parts of the binary executable by applying targeted mutations or generation strategies. It focuses on areas of the program that are known or suspected to be more vulnerable or critical. Binary level directed fuzzing can improve the efficiency of bug discovery by dedicating fuzzing efforts to specific components or functions. Notable instances of binary level directed fuzzing, including AFLFast and AFLGo, leverage binary-level instrumentation to achieve their objectives.

3.2.2 Comparing Testing Efficiency

When comparing the testing efficiency of different fuzzing techniques, it is common to use metrics such as code coverage or stack hash to assess their effectiveness. However, recent research has highlighted certain limitations of these metrics in accurately measuring the number of bugs found. This section discusses the drawbacks of code coverage and stack hash and the implications for evaluating fuzzing techniques.

Many studies have used code coverage [103, 111] as a measure of testing effectiveness, assuming that covering more code correlates with finding more bugs. While this assumption holds true to some extent, the correlation between code coverage and bug discovery may not always be strong. Research has shown that high code coverage does not necessarily guarantee the detection of all existing bugs, as some bugs may only manifest under specific inputs or conditions that are not covered during testing. Therefore, relying solely on code coverage as a metric may lead to an incomplete assessment of a fuzzing technique's bug-finding capabilities.

Similarly, stack hash [130] has been widely used as a de-duplication heuristic to identify unique crashes triggered by different inputs. However, studies have questioned the effectiveness of stack hash in accurately distinguishing between distinct bugs. It has been observed that multiple crashes with the same stack hash can originate from different underlying bugs. This limitation introduces the risk of over-counting or under-counting the number of unique bugs based on stack hash analysis.

To overcome these limitations, researchers have suggested alternative approaches for evaluating fuzzing techniques. One approach is to directly measure the number of unique bugs found rather than relying on de-duplication heuristics. By carefully analyzing the bug reports and considering the root causes and effects of each bug, a more accurate count of unique bugs can be obtained. Another approach is to use synthetic benchmark suites or known vulnerable programs with well-defined vulnerabilities like CGC [92] or LAVA [96]. These benchmarks provide a controlled environment for evaluating fuzzing techniques and enable comparisons based on the detection of specific types of bugs.

It is important to recognize the limitations of code coverage and stack hash when comparing the testing efficiency of fuzzing techniques. While these metrics provide useful insights, they should be supplemented with other measures, such as the number of unique bugs found or evaluations on synthetic benchmarks, to obtain a more comprehensive understanding of a fuzzing technique's effectiveness in bug discovery.

3.3 Goals and Objectives

Our primary goal in this research work is to conduct a series of experiments and analyze the results to understand the relationship between fuzzing and UaF vulnerability detection. We aim to provide insights into the performance of various fuzzing tools when dealing with different UaF bug patterns. Additionally, we investigate the role of auxiliary tools, such as AddressSanitizer (ASan), in enhancing fuzzers' abilities to detect UaF bugs. By addressing the challenges associated with non-deterministic UaF bugs and the use of auxiliary tools, our research contributes to the broader understanding of UaF bug detection and provides valuable guidance for the future.

3.4 Methodology

In this section, we describe the methodology used in our study, focusing on fuzzer selection, the process of collecting UaF bugs, and deriving bug patterns and code ranges.

3.4.1 Fuzzer Selection

Selecting an appropriate fuzzer is crucial for effective bug detection, including UaF vulnerabilities. Fuzzers employ different techniques and strategies to generate test inputs and explore program behaviors. When choosing a fuzzer, it is important to consider factors such as fuzzing techniques, popularity, and specialized fuzzer support for UaF analysis.

Here are our selected fuzzers for UaF detection:

- **American Fuzzy LOP (AFL):** AFL [172] is a popular and highly effective fuzzer that utilizes genetic algorithms to generate test inputs. AFL is capable of detecting UaF bugs through random input mutation and code coverage analysis.
- **AFLFast:** AFLFast [78] is an optimized variant of AFL. AFLFast aims to improve the performance and efficiency of AFL's fuzzing process.

- **LibFuzzer:** LibFuzzer [143] is a coverage-guided fuzzer integrated into the LLVM compiler infrastructure. It focuses on testing individual functions and libraries, making it suitable for targeted UaF analysis.

- **OSS-Fuzz:** OSS-Fuzz [45] is a project by Google that provides continuous fuzzing for open-source software. It utilizes various fuzzing techniques, including libFuzzer, to detect bugs in popular open-source projects. It can be a valuable resource for UaF detection in widely used software.

- **FairFuzz:** FairFuzz [116] is a fuzzing tool that enhances the efficiency of fuzz testing through sensitivity analysis, prioritizing test cases with higher potential for vulnerability discovery. It offers configurability and integration with existing tools for customizable and effective fuzzing workflows.

- **UaFuzz:** UaFuzz [135] is a specialized fuzzer designed specifically for UaF vulnerability detection. It combines symbolic execution with fuzzing techniques to identify UaF bugs by generating test cases that trigger UaF conditions.

- **UaFL:** UaFL [158] is another fuzzer developed for UaF detection. It focuses on detecting UaF bugs caused by double frees, dangling pointers, and other related issues. UaFL utilizes a dynamic taint analysis approach to identify UaF conditions. It's worth noting that this tool does not appear to be open-source.

- **AFLGo:** AFLGo [77] is an extension of AFL that combines genetic algorithms with directed fuzzing. It can be effective for detecting UaF bugs by guiding the fuzzer towards specific code paths related to memory management.

- **MOpt:** MOpt [125] employs a tailored Particle Swarm Optimization (PSO) algorithm to optimize the scheduling of mutation operations, resulting in enhanced efficiency when uncovering vulnerabilities.

3.4.2 UaF Bug Collection and Analysis

In order to study UaF vulnerabilities and derive bug patterns and associated code ranges, we conducted a comprehensive collection of UaF bugs from various sources. Our bug collection process involves the following steps:

Selection of Target Applications: We targeted three widely used open-source programs for our bug collection: the Linux kernel, Mozilla Firefox web browser, and CPython. These programs were chosen due to their substantial codebases and their use of memory-unsafe languages like C and C++. In addition, they have large user bases and well-maintained bug tracking systems, which facilitated the collection of UaF bugs.

Searching for UaF Vulnerabilities: We extensively searched the bug tracking systems of the selected programs using specific keywords related to UaF vulnerabilities. These keywords include terms like "UaF," "use-after-free," "dangling pointer," and "double free." To ensure a comprehensive search, we expanded our scope by incorporating additional keywords such as "data race," "concurrency," "weak reference," and "reference count." We also searched the Common Vulnerabilities and Exposures (CVE) database [153] to identify UaF vulnerabilities that had been confirmed as security issues. The bug reports and CVE entries provided valuable information about the causes, effects, and patches for the UaF vulnerabilities.

Bug Verification and Selection: After collecting a set of potential UaF bugs, we carefully verified each bug to ensure its validity and relevance to our study. We cross-checked the bug reports with external sources and developer discussions to confirm that they were indeed UaF vulnerabilities. We also assessed the quality of bug reports, including the level of detail provided, the clarity of the root cause descriptions, and the availability of source code and patch information. Finally, we randomly sampled bugs from the verified set for further analysis.

In total, we collected 150 UaF bugs from two main sources: the bug tracking systems of the targeted programs and GitHub repositories. The bugs were categorized

Table 3.1: Number of UaF bugs collected from three representative software suites.

Source	Deterministic	Non-deterministic	
		Concurrent	Others
Firefox	25	5	0
CPython	27	2	1
Linux	16	10	4
GitHub	47	11	2
Overall	115	28	7

based on their deterministic or non-deterministic nature, as well as whether they involved concurrent UaF. The distribution of bugs across different sources and categories is summarized in Table 3.1.

3.4.3 Threats to Validity and Observer Errors

As with any empirical study, our methodology is subject to certain threats to validity. We acknowledge and address the following potential sources of bias:

Representativeness of Selected Bugs: The bugs collected for our study may not be fully representative of all UaF vulnerabilities in the software ecosystem. Our focus was on open-source programs, and while they are widely used and considered of high quality, our findings may not generalize to the broader population of commercial software. Nonetheless, our careful selection process, including the use of diverse types of open-source programs and random sampling, provides valuable insights into UaF vulnerabilities.

Size of Sample Set: It is important to acknowledge that our sample set of UaF bugs may not cover the entire population of such bugs. However, the sample size of 150 bugs in our study is comparable to previous empirical studies [135], which constructed benchmarks containing 30 UaFs. While the sample size is not exhaustive, statistical principles suggest that it is large enough to provide representative results.

The determination of an appropriate sample size depends on various factors, such as the desired level of statistical significance, effect size, and variability within the population. In empirical studies, researchers strive to strike a balance between sample

size and available resources. While a larger sample size may provide more precise estimates and increase the confidence in the findings, it may not always be feasible due to constraints such as time, budget, and the rarity of certain bug types.

In our study, we believe that the sample size of 150 UaF bugs provides valuable insights into common patterns and characteristics of such vulnerabilities. The bugs were carefully selected and analyzed to ensure a diverse representation of UaF instances across different software domains. While it is impossible to guarantee that every UaF pattern is captured in the sample, statistical principles suggest that a sufficiently large sample size can yield representative results, allowing for generalizations and observations about UaF bugs.

Observer Errors: There is a potential for observer errors during the bug collection and analysis process. To minimize this possibility, we followed a detailed protocol and involved multiple researchers who cross-checked the bugs for accuracy and consistency. Furthermore, we had expertise in the targeted programs and some of the randomly sampled applications, which helped in understanding and classifying the bugs correctly.

By addressing these threats and adhering to rigorous collection and analysis practices, we strive to provide reliable and meaningful results in our study.

3.5 UaF Characteristics

In this section, we delve into a comprehensive exploration of the distinct characteristics exhibited by UaF bugs. As we embark on this investigation, we uncover the code scopes, intricate patterns, and non-deterministic behaviors inherent to UaF vulnerabilities. These characteristics not only define the nature of UaF bugs but also play a pivotal role in shaping the effectiveness of fuzzing techniques aimed at their detection. By delving deep into the unique attributes of UaF bugs, we aim to illuminate the crucial insights that guide our subsequent analysis and shed light on the complex interplay between UaF characteristics and fuzzing efficacy.

3.5.1 Bug Code Scope and Its Implications for Fuzzing

Determining the code scope affected by a UaF bug is crucial for understanding its impact and developing effective mitigation strategies. To identify the bug code scope, we conducted a thorough analysis of bug reports and source code. We examined the patches applied to fix the UaF vulnerabilities and traced their effects to determine the specific files, functions, and lines of code involved. By carefully analyzing the code changes made in the patches, we identified the relevant code sections affected by the bug. We also considered explicit information provided in bug reports, such as function names or file names, to determine the code scope. Discussions and comments in the bug reports were valuable in understanding the developers' insights and discussions on the bug's impact.

The bug code scope may vary depending on the bug's root cause. Some bugs may have a narrow scope, affecting only a few lines of code within a single function, while others may have a broader impact, spanning multiple files and functions. By examining the bug reports, patches, and code changes, we derived insights into the bug code scope for each UaF vulnerability.

The depth of comprehension concerning bug code scope not only aids in prioritizing remediation efforts but also extends its significance to the realm of fuzzing. Fuzzing UaF vulnerabilities can indeed be influenced by the scope of the bug's impact. Bugs with broader code scope may pose greater challenges for fuzzing, as their effects might ripple through a larger portion of the codebase. Conversely, bugs with narrower code scope might offer a more focused target for fuzzing campaigns. Recognizing the balance between comprehending code scope and formulating fuzzing strategies becomes a critical factor in optimizing the bug detection and resolution processes. This insight empowers developers to channel their efforts into the precise areas influenced by the bug, amplifying the efficiency and effectiveness of bug detection and resolution endeavors.

3.5.2 UaF Bug Patterns

In [90], an empirical study was conducted to understand and classify UaF vulnerabilities into 7 distinct patterns. These 7 patterns include `Immediate-UaF` (P1), `Flag-related UaF Bugs` (P2), `Memory Resize Vulnerabilities` (P3), `Misuse of APIs and Features` (P4), `Double Free Vulnerabilities (P5), Reference Count Bug` (P6), and `Miscellaneous Cases` (P7).

The `Immediate-UaF` category encompasses `Move-Free-Behind-Use` and `Save-Before-Free patterns`. In the former, memory deallocation precedes pointer dereference, leading to a UaF bug. The latter involves premature memory release while internal pointers are still needed. `Flag-related UaF bugs` arise from improper flag handling. Memory resize vulnerabilities result from mishandling memory resize operations like `realloc()`. `Misuse of APIs and Feature` includes misusing functions like `list.erase()` and `list.remove()`. `Double Free Vulnerabilities` occur due to multiple memory block releases. `Reference Count` Bug involves incorrect reference counting. `Miscellaneous Cases` cover other UaF scenarios.

Understanding these patterns is crucial for effective UaF detection and mitigation strategies.

3.5.3 Nondeterminism and Its Influence on Fuzzing UaF

A significant proportion of UaF bugs exhibit non-deterministic triggering. Notably, a substantial subset of these non-deterministic UaF instances is rooted in concurrency-related issues. In comparison to the study by Lu et al. [121], a previous empirical study [90] delved into the analysis of concurrent UaF bugs and arrived at a distinct yet somewhat similar conclusion. Notably, the examination of UaF concurrency bugs in this prior work unveiled a specific pattern. It was observed that all the collected UaF concurrency bugs in that study turned out to be non-deadlock instances. This stands in contrast to Lu et al.'s findings where 66% of the general concurrency bugs exhibited non-deadlock behavior. Furthermore, an investigation into bug patterns demonstrated

Table 3.2: Bug Info and Tools Comparison. † The crashed bugs are highlighted ‡ T/O means the fuzzer cannot discover vulnerabilities within 24 hours

Bug Info				Time Usage to Expose the Vulnerabilities (hours)							
ID	Size	Pattern	Scope	AFL	FairFuzz	MOpt	AFLGo	OSSFuzz	UaFuzz	LibFuzzer	AFLFast
yosys-2498 [2]	23.3M	P1	small	0.04	0.03	0.02	0.01	0.03	0.01	0.05	0.01
issue-7760 [3]	26.1M	P1	small	0.10	0.10	0.10	0.20	0.10	0.11	0.12	0.15
kbd-50 [4]	9.4M	P1	small	0.02	0.01	0.01	0.02	0.01	0.01	0.01	0.01
numactl-114 [5]	1.6M	P2	medium	0.35	0.40	0.41	0.28	0.42	0.11	0.75	0.35
redhat-1484332 [6]	26Kb	P2	medium	0.16	1.06	0.79	1.98	0.42	0.21	0.96	0.30
coturn-601 [7]	4.8M	P2	small	0.02	0.03	0.02	0.04	0.03	0.01	0.06	0.01
dcmtk-942 [8]	47.5M	P3	medium	2.34	4.12	2.22	2.98	3.66	4.01	4.80	2.12
Python-24613 [9]	87.3M	P3	medium	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O
Python-27867 [10]	87.3M	P3	medium	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O
profanity-1428 [11]	22.1M	P4	medium	0.91	0.60	0.72	0.52	1.10	0.22	0.80	0.42
uacme-issue [12]	1.2M	P4	small	0.24	0.40	0.91	0.72	0.31	0.12	0.24	0.42
Bugzilla-3392414 [13]	1.8Mb	P4	large	9.95	T/O	T/O	8.44	T/O	5.54	T/O	3.20
lrzip-96 [14]	581Kb	P5	small	0.02	0.1	0.02	0.01	0.05	0.03	0.01	0.01
yasm-issue-91 [15]	1.4Mb	P5	medium	1.02	0.85	0.65	0.80	1.11	0.23	1.22	0.55
CVE-2019-6455 [16]	604Kb	P5	medium	2.14	2.01	2.10	2.02	3.25	0.54	4.32	2.46
Python-24094 [17]	87.3M	P6	small	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O
Python-24407 [18]	87.3M	P6	small	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O
Python-24097 [1]	87.3M	P6	small	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O
redhat-1484332 [19]	92.7M	P7	large	19.43	T/O	T/O	8.82	T/O	2.13	T/O	14.56
mjs-73 [20]	255Kb	P7	medium	6.52	3.68	4.90	2.21	5.52	0.59	4.45	2.57
mjs-78 [21]	255Kb	P7	medium	3.31	2.05	4.70	1.86	3.77	1.41	4.13	3.11

that 96% of the UaF concurrency bugs were triggered by atomicity-violation and order-violation, aligning with the observations made in Lu et al.'s study [121] (97%). An interesting observation was that among the bugs stemming from atomicity-violation and order-violation scenarios, four instances were attributed to data race conditions [69]. Additionally, the prior study indicated that all manifestations of UaF concurrency bugs involved concurrent accesses to a single variable, diverging from the 66% reported in Lu et al.'s research [121]. This implies that a potential strategy for UaF concurrency bug detection could center on atomicity-violation, order-violation, and non-deadlock concurrent accesses to single variables. This nuanced comparison highlights how previous empirical research [90] sheds light on the nature of UaF concurrency bugs, revealing both similarities and differences when compared to Lu et al.'s study.

Another aspect of UaF's non-determinism arises from its dependence on the system's state. For instance, certain function calls could result in insufficient memory allocation [52], leading to errors that subsequently trigger UaF. Another special case is static destruction order fiasco as explained above in Figure 2.3.

Detecting UaF vulnerabilities presents challenges due to their intricate nature

and non-deterministic behavior. Traditional static analysis techniques often struggle to accurately identify UaF bugs, as dynamic analysis is often necessary to capture essential program states and memory interactions. To address these challenges, specialized UaF fuzzers like UaFuzz and UaFL have been developed to specifically target UaF vulnerabilities, enhancing their detection capabilities. Understanding the concepts and characteristics of both fuzzing techniques and UaF vulnerabilities remains critical for effective exploration of fuzzing methods in detecting UaF bugs. In the subsequent sections, we delve into evaluating the performance of various fuzzers, encompassing both general and specialized ones, in the discovery of UaF vulnerabilities, while addressing the associated challenges tied to UaF detection through fuzzing.

3.6 Performance of Existing Fuzzers

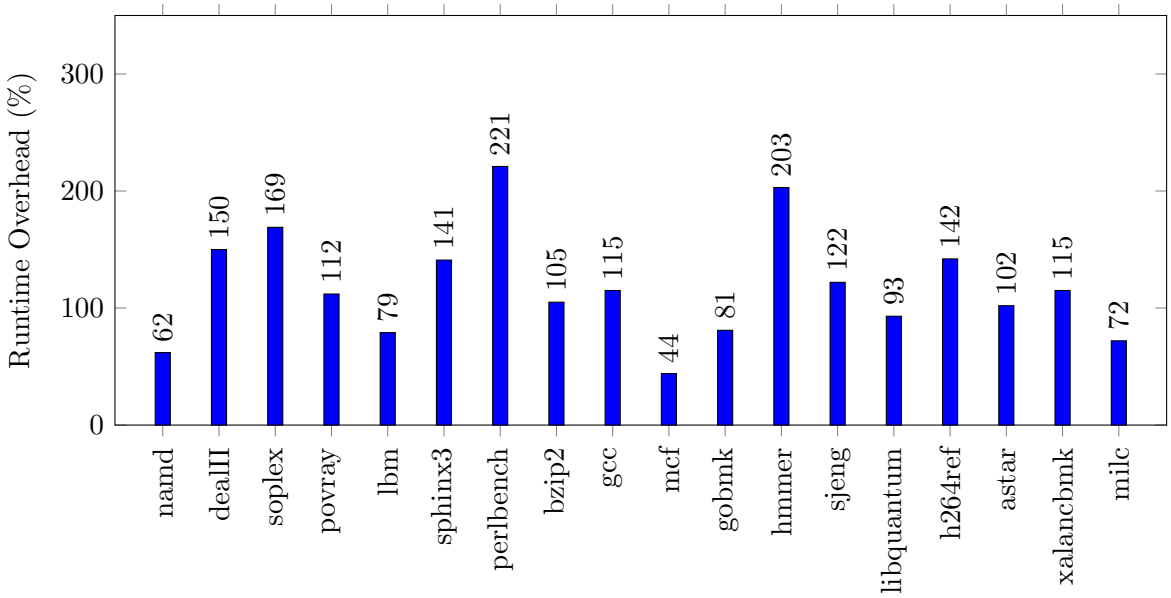

Figure 3.2: Runtime overhead of ASan

We employed a UaF fuzzing benchmark derived from real-world applications, as listed in Table 3.2. For each of the aforementioned 7 bug patterns, we selected three distinct bugs, and overall, the benchmark includes 21 carefully selected UaF bugs, each with detailed bug reports. To reduce the impact of randomness, we performed three separate tests for each bug. The experiments were conducted on an x86-64 desktop

system, comprising six-core Intel i7-9750 2.6 GHz processors and 32GB memory. The operating system used was Ubuntu 18.04.5. The main objectives of our evaluation were to answer the following questions:

- Would fuzzers exhibit different performances when applied to different UaF bug patterns?
- What is the necessity of ASan in fuzzing, and what effects does it have on the detection of UaF bugs?
- What is the main challenge for fuzzing UaF bugs?

The performance discrepancies between different fuzzing tools in relation to UaF bugs of various characteristics have been investigated in this study. We classified 21 bugs from the collected bugs into seven distinct categories based on their unique features. The fuzzing tools we utilized include AFL, LibFuzzer, OSS-Fuzz, UaFuzz, MOpt, FairFuzz, AFLGo, and AFLFast. The process was carried out three times for each bug, and the results were as follows, where we can draw some initial conclusions:

3.6.1 UaF Reproducibility

Finding 6. *Without ASan, only 2 out of 21 UaF bugs caused crashes, demonstrating the importance of ASan in fuzzing for effective UaF bug detection.*

Modern fuzzers heavily rely on application crashes as externally observable behaviors to detect bugs. In Table 3.2, 2 out of 21 bugs have been marked in bold, indicating that these UaF bugs would cause a crash. This implies that most UaF bugs are actually "silent". This finding indicates that the majority of UaF bugs remain "silent" without ASan, as swift re-allocation of freed heap memory by the operating system often disguises memory dereferencing as legitimate operations, thereby avoiding system crashes. The detection of UaF bugs by fuzzers requires the assistance of auxiliary tools like ASan for effective identification.

To assess the overhead introduced by ASan, we conducted experiments using SPEC CPU2006 [106], a widely accepted industry-standardized benchmark suite known for its CPU-intensive nature. We executed each of its programs three times. Through this assessment, we quantified the average percentage of overhead introduced by ASan. This analysis provides valuable insights into the performance implications of integrating ASan into the fuzzing process. Figure 3.2 shows ASan brings averagely 121.8% overhead.

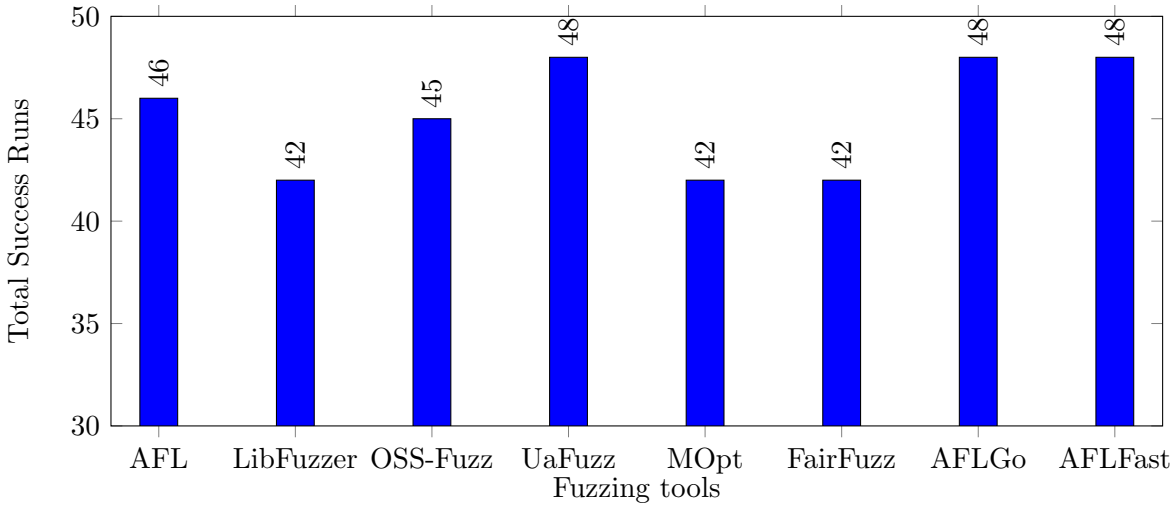

Figure 3.3: Total success runs by different fuzzing tools

3.6.2 UaF patterns affect fuzzing detection

Table 3.2 provides a comprehensive overview of the fuzzing results, showcasing the bug ID, scope, pattern, code size, crash status (highlighted in bold), and the fuzzing time for each bug tested with different fuzzing tools. The `ID` column corresponds to the unique identifier assigned to each UaF bug, and this identifier helps distinguish and reference specific bugs. The `Size` column represents the size of the software package or codebase associated with the UaF bug. The `Pattern` column specifies the particular UaF bug pattern associated with each bug, and the details can be found in Section 4.2. The `Scope` column refers to the code scope, which has been classified into three distinct levels: small, medium, and large. Small means both the creation of a dangling

pointer and its subsequent dereference instructions occur within the range of the same basic block or adjacent basic blocks. Medium mean dangling pointer creation and dereference instructions are not in the same basic blocks and their Line-of-free-to-Line-of-dereference (LOFTLOD) is within 50. Large means dangling pointer creation and dereference instructions are not in the same basic blocks and their LOFTLOD is larger than 50. The data highlights the significant impact of bug patterns and code scopes on fuzzing detection, shedding light on the varying performance of different fuzzers when exposed to different UaF vulnerabilities.

Finding 7. *There are significant differences in the performance of different fuzzing tools when dealing with various UaF bug patterns.*

Figure 3.3 exhibits total success runs by different fuzzing tools. During the experiments, AFL detected 46 instances of successful bug identification, while LibFuzzer, OSS-Fuzz, MOpt, and FairFuzz identified bugs in 42 cases. AFLFast, AFLGo, and UaFuzz exhibited the best performance by successfully detecting bugs in 48 instances each.

Table 3.2 highlights UaFuzz's superior performance in specific patterns, such as P2, P4, P5, and P7. It achieves a reduced time overhead ranging from 38.0% to 85.7%, 59.5% to 79.2%, 71.1% to 85.6%, and 64.8% to 79.7% for these patterns respectively. UaFuzz maintains similar performance levels across other patterns.

These results highlight the importance of selecting the appropriate fuzzing tool based on the specific UaF bug pattern being targeted. Different tools may exhibit varying levels of effectiveness and efficiency in detecting different types of UaF bugs.

3.6.3 UaF code scope matters

Finding 8. *When the UaF's code scope was small, the performance differences between fuzzers were relatively minor.*

Finding 9. *Certain fuzzers designed explicitly for detecting UaF bugs, such as UaFuzz, exhibited a significant performance improvement when the code scope was medium-sized.*

Analyzing the code scope affected by a UaF bug is key to understanding its consequences and planning proper mitigations. We pinpointed this scope by analyzing bug reports and source code, scrutinizing applied patches, and tracing their impacts on specific codes. The bug's impact could range from a narrow effect on few lines of code to broader ones involving multiple files and functions. This knowledge aids in prioritizing bug fixes and designing effective measures.

When the UaF bug affected a limited code scope, the performance of different fuzzing tools was similar. However, for medium-sized scopes, specific fuzzers like UaFuzz showed better results. Simple UaF patterns with moderate code scope didn't benefit from additional fuzzing optimizations; ASan usage was necessary as these bugs didn't always lead to crashes.

Fuzzing optimizations were particularly useful for UaF bugs related to flags and double-free issues. Nonetheless, some UaF patterns, such as reference count overflow, necessitated a blend of various analysis approaches, highlighting the limits of fuzzing alone.

3.6.4 Reference count bugs

Finding 10. *All of the fuzzing tools performed poorly when detecting Reference Count Bugs, failing to detect any instances of this bug type.*

Figure 3.4 shows the challenges of fuzzing reference counting UaF bugs. In this case, the triggering window for the bug is small, and the conditions for memory release are more difficult to encounter during a typical program execution. `PyList_GET_ITEM(slotnames, i)` returns a borrowed reference `name` to the object at index `i` in list `slotnames`. However, `name` cannot prevent the referent from being freed. If the referent gets freed before or during the execution of `PyObject_GetAttr(obj, name)`, `name` becomes a dangling reference and the dereferencing operations will trigger UaF.

The difficulty lies in encountering the precise timing and conditions for the referenced object to be freed within this narrow triggering window. Fuzzing techniques mainly work by exploring various execution paths, inputs, and code coverage, but it is

```
Py_LOCAL(PyObject *) _PyObject_GetState(...) {
     // name is a borrowed reference, and the referent might get
        unexpectedly freed.
     name = PyList_GET_ITEM(slotnames, i);
+    Py_INCREF(name);
     value = PyObject_GetAttr(obj, name);
     // error: misused borrowed reference
+    Py_DECREF(name);
 }
```

Figure 3.4: UaF due to a misused borrowed reference [1]

challenging to capture the specific sequence of events which lead to the UaF bug. It requires generating a test case that triggers the precise memory release conditions and capturing it within the limited execution window.

Additionally, the effectiveness of fuzzing for UaF bugs is hindered by the fact that this type of bug often requires complex program states or specific sequences of operations to manifest. Discovering such bugs through fuzzing may require substantial effort and potentially customized fuzzing strategies tailored to target the specific UaF vulnerability pattern.

These challenges highlight the need for specialized techniques, thorough test case generation, and careful code analysis to effectively identify and mitigate reference counting UaF bugs during fuzzing endeavors.

3.6.5 Fuzzing randomness

Finding 11. *The results varied between runs, suggesting that the detection of bugs isn't consistently guaranteed.*

Fuzzing is inherently a stochastic process, where the generation of inputs and exploration of code paths depend on random mutations and selections. This randomness is particularly influenced by the concurrent nature of the code. Figure 3.5 provides insight into the impact of concurrency on fuzzing results. The figure illustrates the testing of three concurrent bugs using various fuzzing tools. Each bug was tested three

times with each tool, and the number of times each tool successfully fuzzed out the bug is presented in Figure 3.5.

These results reveal that only five of the tested tools were able to identify one of the concurrent bugs, and even then, the success was not consistent across all runs. This variability in detecting bugs underscores the inherently stochastic nature of fuzzing, particularly driven by the non-determinism inherent in UaF vulnerabilities. The concurrency in the code can lead to different execution paths and interactions between threads, contributing to the uncertainty in bug detection during fuzzing. As a result, repeated and thorough testing remains crucial to cover the potential bug scenarios comprehensively.

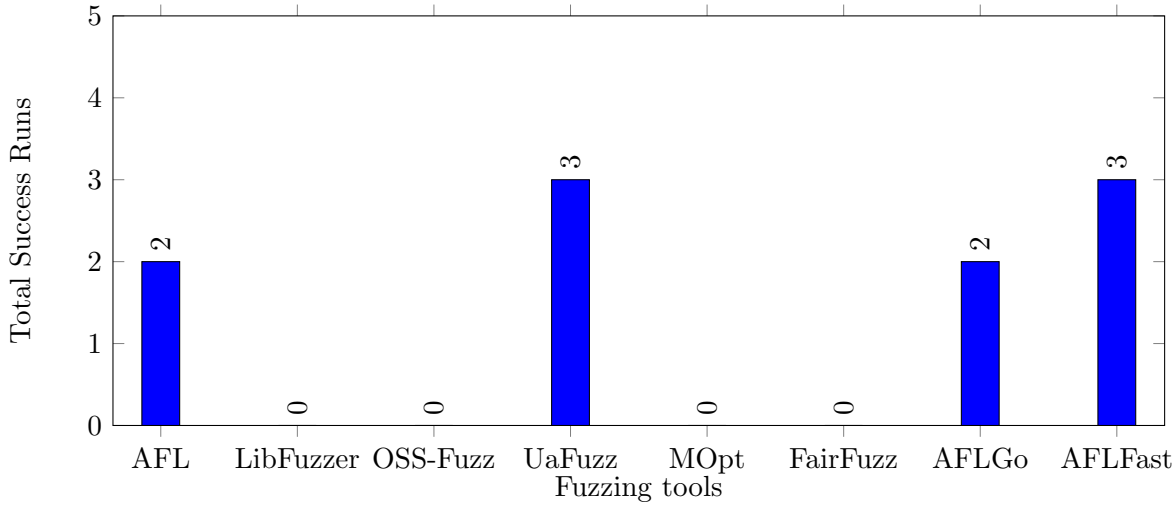

Figure 3.5: Total success runs on the 3 concurrent bugs by different fuzzing tools

3.6.6 Nondeterminism of UaF: A Fuzzing Challenge

UaF vulnerabilities' nondeterminism, stemming from code concurrency or unpredictable memory reuse or reallocation timing, poses a challenge to their detection and mitigation.

This nondeterminism arises from various factors, including concurrent execution, complex program control flow, and external influences like system configurations or input variations. In the context of fuzzing, the uncertainty in execution order and

the narrow time windows for UaF bug manifestation make detecting concurrent UaF bugs particularly challenging. The low success rate (approximately 13.9%) in the experiment in Figure 3.5 with three concurrent bugs and multiple fuzzers underscores the inherent difficulty in tackling this type of vulnerability.

The non-determinism of memory allocators adds another layer of complexity. An illustrative example of this case is shown in Figure 3.6. In CPython [25], the `array.fromstring()` method provides a case in point. This method appends items from a string to an array, interpreting the string as an array of machine values. In its original buggy implementation, passing the array itself as an argument to `fromstring()` resulted in both `self->ob_item` and the pointer to the argument pointing to the same memory block. Later in the method, the call to `PyMem_RESIZE()`, which invokes `realloc()`, aimed to resize the memory pointed to by `self->ob_item`. If the original memory block couldn't accommodate the desired expansion, `realloc()` could potentially allocate a new memory block, copy the data, and free the original block. This could lead to unintended deallocation of memory, creating a window for UaF bugs.

Language-specific rules, such as in C++, can also cause UaF bugs. Consider two static objects, `alias_table` and `parameter_set`, as shown in Figure 2.3. If `alias_table` is destroyed while `parameter_set` still holds a reference to it, any subsequent use of this reference will lead to a UaF bug. This was resolved by changing these objects to static functions returning static member variables, thereby ensuring a controlled and predictable order of initialization and destruction.

These examples highlight UaF bugs' non-deterministic nature due to concurrency, memory allocation behavior, or language-specific rules, complicating the fuzzing process and underlining the importance of advanced fuzzing techniques.

```
static PyObject *
array_fromstring(arrayobject *self, PyObject *args){
    char *str;
    Py_ssize_t n;
    int itemsize = self->ob_descr->itemsize;
    if (!PyArg_ParseTuple(args, "s#:fromstring", &str, &n)) //str is parsed
        from args. When an array is passed to itself (i.e., args=self), we
        get str == self->ob_item.
        return NULL;
+   if (str == self->ob_item) {
+       PyErr_SetString(PyExc_ValueError, ...);
+       return NULL;
+   }
    char *item = self->ob_item;
    // When str == self->ob_item, item == str.
    PyMem_RESIZE(item, char, (Py_SIZE(self) + n) * itemsize);
    // realloc() is invoked in PyMem_RESIZE(), frees the original memory
        pointed by item and allocates a new memory block which may or may
        not be the same as the original one. Therefore, if item == str, str
        may point to freed memory after PyMem_RESIZE().
    memcpy(item + (Py_SIZE(self) - n) * itemsize, str, itemsize*n);
    //If str is dangling, a UaF occurs here.
}
```

Figure 3.6: UaF when a function calls realloc [25]

3.7 Lesson Learned

The process of using fuzzing to test for UaF vulnerabilities has proven to be a valuable approach, but it also comes with certain limitations and challenges. Throughout this study, we have identified several key lessons that shed light on the effectiveness and shortcomings of fuzzing for UaF detection.

1. Tricky Bug Triggering Conditions: One of the major challenges in fuzzing for UaF vulnerabilities lies in the existence of tricky bug triggering conditions. These conditions include short time windows for bug manifestation, reference count overflow situations, and other non-obvious scenarios. For instance, UaF bugs with short time windows might only occur during specific program states or within a narrow window of execution, making them difficult to capture through random fuzzing inputs. Additionally, reference count bugs can be elusive, as they require a precise sequence of

events for the UaF to manifest, making them less likely to be detected during standard fuzzing runs.

2. Bug Non-determinism and Concurrency: The non-deterministic nature of UaF bugs, arising from factors like concurrent execution and memory allocation behaviors, adds further complexity to the fuzzing process. The inherent unpredictability of UaF bugs can lead to varying results between different fuzzing runs, even with the same initial setup and seed inputs. In particular, fuzzing concurrent UaF bugs is particularly challenging due to the uncertainty in execution order and the narrow time windows for bug manifestation. This non-determinism highlights the need for advanced fuzzing techniques and multiple rounds of testing to achieve comprehensive bug coverage.

3. Consideration of Memory Allocator State: Another important aspect to consider in fuzzing is the state of the memory allocator. In certain situations, bugs may not be triggered because of the specific memory allocation behaviors of a program. For example, as observed in Figure 5, realloc bugs may not lead to UaF if the memory block can be resized without needing a new allocation and copy. Additionally, in some cases, the program might run normally without triggering a crash, even if memory overwrites occur. Understanding and managing the memory allocator state is crucial to ensure the effectiveness of fuzzing for UaF detection.

4. Complementing Fuzzing with Static Analysis: While fuzzing is a powerful technique, it may not be sufficient to detect all UaF vulnerabilities, especially those with complex triggering conditions. In such cases, static analysis can serve as a valuable complementary approach. Static analysis techniques can identify potential UaF patterns by analyzing the source code and program behavior without executing the code. By combining static analysis with fuzzing, developers can potentially achieve better UaF detection coverage. Hybrid approaches, such as combining fuzzing and static analysis, may offer a promising avenue to effectively identify and mitigate UaF vulnerabilities.

Overall, fuzzing remains a valuable approach for UaF detection, but it has limitations when dealing with tricky bug triggering conditions, bug non-determinism, and memory allocator complexities. To enhance the UaF detection process, developers should consider employing a combination of fuzzing, static analysis, and potentially hybrid approaches. This holistic approach will provide a more robust and comprehensive defense against UaF vulnerabilities, ultimately leading to safer and more secure software applications.

Additionally, to obtain more accurate and reliable measurement results, conducting multiple runs with larger sample sizes is advised. Furthermore, there is potential for more in-depth research on concurrent UaF bugs and reference count bugs, which require specialized tools and techniques for effective detection. Comparing the performance of dedicated tools designed for these specific bug types with general-purpose fuzzers would provide valuable insights.

Despite the constraints of the current study, it lays the foundation for future research endeavors in UaF bug detection. The lessons learned and challenges discussed here contribute to a deeper understanding of UaF bug detection, while also opening avenues for further exploration and improvement in this field.

3.8 Summary

In this chapter, we systematically investigated the use of fuzzing techniques for discovering UaF bugs. We conducted a series of experiments followed by thorough analyses. Our findings provide valuable insights into the performance characteristics and limitations of different fuzzers when applied to detect various UaF bug patterns. This study not only highlights the challenges and limitations of current fuzzing techniques, but also emphasizes the need for further improvements. Specifically, fuzzers have limited effectiveness in detecting UaF bugs with certain features, such as reference count-related issues and non-deterministic bugs. Specialized techniques beyond traditional fuzzing are needed to reliably detect these types of bugs. Furthermore, this study demonstrates that specific fuzzers perform better for medium to large code

scopes, while performance tends to be similar for small code scopes. This insight can aid practitioners in making informed decisions regarding fuzzing tool selection based on the characteristics of the UaF bugs they aim to detect. Overall, this research contributes to a deeper understanding of UaF bug detection, uncovering current limitations, and paving the way for future advancements in fuzzing techniques and related tools.

Chapter 4

BRIDGING THE SEMANTIC GAP: BENCHMARKING LLMS AND STATIC ANALYSIS IN USE-AFTER-FREE DETECTION

The third work in this dissertation investigates a new paradigm in program analysis: the application of LLMs to software vulnerability detection. As LLMs have rapidly gained traction in software engineering tasks—from code completion to automated debugging—they have also been increasingly adopted for identifying security-critical issues such as UaF vulnerabilities. Unlike traditional static analyzers, which rely on deterministic reasoning over formal representations like abstract syntax trees and control-flow graphs, LLMs process code as unstructured text and operate based on statistical language understanding. This fundamental difference in analysis modality introduces a range of new questions: Do LLMs detect vulnerabilities in the same way as static tools or human analysts? Are current benchmarks and evaluation metrics suitable for LLMs? How does prompt formulation or input granularity affect detection performance? And most importantly, how should we fairly and rigorously evaluate LLM-based vulnerability reasoning?

To answer these questions, this work proposes a benchmark-driven framework for systematically comparing LLMs and static analyzers on UaF detection. We construct a curated dataset of real-world UaF cases, introduce new evaluation metrics tailored to LLM behavior, and analyze how variations in prompt design, input abstraction, and reasoning granularity influence performance. Through this study, we aim to bridge the semantic gap between pattern-based machine reasoning and structure-aware static analysis, offering insights into the unique strengths and limitations of LLMs in security-critical code analysis.

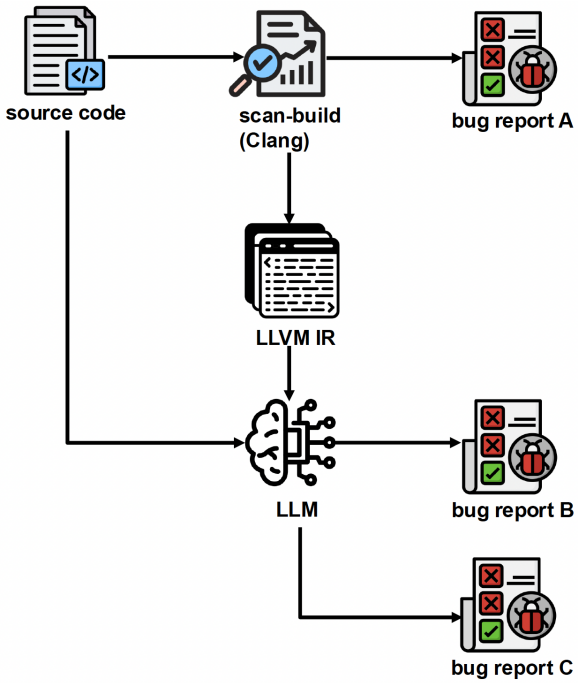

Figure 4.1: Testing framework.

4.1 Methodology

This section introduces a static analysis framework designed to evaluate and compare the capabilities of traditional static tools and LLMs in identifying undefined behavior, such as UaFs, in C and C++ programs. The framework emphasizes diagnostic performance, contrasting how different analysis modalities interpret and report potential vulnerabilities.

4.1.1 Framework Structure

As illustrated in Fig. 4.1, the framework incorporates three analysis pipelines. The first uses *scan-build* (Clang Static Analyzer) to inspect the original source code and generate a bug report, referred to as bug report A. In parallel, an LLM is prompted to analyze the same source code directly, simulating its reasoning process to detect potential vulnerabilities; this yields bug report B. Additionally, we explore whether exposing the LLM to a lower-level representation of the program affects detection capability: the source code is compiled into LLVM Intermediate Representation (LLVM IR), which is then fed to the LLM to generate bug report C. Each of the three methods

independently produces a bug report. These results are then compared to evaluate consistency, overlap, and coverage across analysis modes.

Key characteristics of the framework include a uniform interface for analyzing and collecting bug reports from both static tools and LLMs, a multi-view analysis setup that contrasts source-level and IR-level reasoning, and a structured comparison mechanism to assess detection differences, highlight strengths of each method, and identify complementary findings.

4.1.2 Workflow

The analysis process proceeds through three coordinated paths, each targeting the detection of undefined behavior from a different angle. Starting from the same source code, we first apply `scan-build` to conduct rule-based static analysis. This tool traverses the code's control and data flow to identify potential issues such as use-after-free, null dereference, or memory leaks, and emits bug report A.

Figure 4.1 shows an overview of our evaluation framework. The pipeline includes three analysis paths: (1) static analysis using *scan-build* (Clang Static Analyzer), (2) LLM analysis on source-level input, and (3) LLM analysis on LLVM Intermediate Representation (IR). Each path produces an independent bug report, which is then compared against a manually validated ground truth.

Simultaneously, we prompt a LLM with the same source code to simulate an expert review of the program, capturing its contextual and semantic reasoning. This yields bug report B. To evaluate the effect of abstraction level on LLM analysis, we also compile the source into LLVM IR, which serves as the input for a third LLM-based analysis. This results in bug report C, providing insights into how the same model behaves under lower-level code semantics. All three bug reports are then collected and compared. The comparison aims to surface cases where LLMs catch bugs missed by static tools, or vice versa, and to explore whether analyzing IR offers any systematic advantage or disadvantage relative to source-level reasoning.

The core design goal is to evaluate the impact of analysis modality (static vs. LLM), input representation (source code vs. IR), and context granularity (function-level vs. full-file) on the accuracy and reliability of UaF detection. The framework also supports prompt engineering strategies, enabling systematic experimentation with natural language instructions.

This setup enables a systematic evaluation of how analysis modality and input representation influence vulnerability detection, which we investigate in the following sections.

4.1.3 Static vs. LLM Analysis Paths

Path A (Static Analysis) uses *scan-build* to analyze the raw C/C++ code. The tool performs interprocedural static analysis to detect UaFs, using a combination of alias analysis and control/data-flow tracking. The output is a machine-generated report of flagged issues, which we filter to retain UaF-related findings.

Path B (LLM on Source Code) sends the full source code through a prompt to an LLM (e.g., GPT-4). The prompt requests the model to identify UaF vulnerabilities and explain its reasoning. We also evaluate variations in prompt specificity, length, and style.

Path C (LLM on LLVM IR) involves compiling the source into LLVM IR (with optimizations disabled to preserve structure), then feeding the IR to the LLM with adapted prompts. This tests whether lower-level representations help or hinder model reasoning. In each case, the LLM's output is parsed and manually annotated to determine whether it corresponds to a true positive (TP), false positive (FP), or false negative (FN).

4.1.4 Benchmark Description

To enable a comprehensive evaluation of detection performance, the framework adopts a dual-benchmark strategy that separately targets FN and FP. The first benchmark focuses on identifying real-world UaF vulnerabilities and serves as the primary

basis for assessing detection effectiveness. The second benchmark comprises clean, large-scale code from the Linux kernel and is used to evaluate analysis precision and quantify spurious detections. These complementary benchmarks are detailed in the subsections below.

4.1.4.1 UaF Bug Dataset

We focus on a deep and systematic analysis of UaF vulnerabilities, rather than attempting to broadly sample all categories of software bugs. Our decision is motivated by the observation that UaF bugs exhibit uniquely challenging characteristics—both temporally and spatially. These bugs arise from subtle memory lifetime violations, requiring precise reasoning over the program's allocation and deallocation behaviors, often across non-local scopes and asynchronous code paths. Compared to other bug types, UaF detection demands higher semantic precision and deeper memory reasoning, making it a representative and high-value target for evaluating the effectiveness of static analyzers and LLMs alike.

While some recent works aim to build large-scale vulnerability benchmarks that cover a wide spectrum of bug types, they often suffer from a lack of depth: individual bug categories are represented by only a handful of examples, many of which are shallow or synthetic. Moreover, these datasets frequently lack rigorous selection criteria to ensure the representativeness of each included bug. In contrast, we argue that evaluating a detection framework on a well-curated, deeply characterized class of bugs—such as UaF—yields more meaningful and interpretable insights.

Our starting point is the UaF benchmark proposed by Chen et al. [87], which includes 120 real-world UaF cases across ten distinct bug patterns. However, the original benchmark is designed for whole-program analysis and is ill-suited for LLM-based reasoning. Many of the included bugs span across multiple files and require analyzing entire software systems (e.g., Firefox or the Linux kernel) consisting of millions of lines of code. Due to LLMs' inherent context size limitations—currently capped at

around 64k tokens, which roughly translates to 10,000 lines of code—it is impractical to process such large programs directly.

To address this, we manually refactor and reduce each bug case into a minimal, single-file version that retains the full semantic structure necessary for static and LLM-based analysis. Specifically, for each bug, we preserve all relevant function definitions, pointer operations, and allocation/deallocation logic, ensuring the reduced program remains self-contained and compilable. In cases where the original bug required cross-file or cross-module analysis beyond our feasible reduction scope, we excluded the example.

The resulting dataset consists of 50 high-fidelity UaF bugs, each presented as a complete yet minimal unit of analysis. These bugs originate from diverse real-world software suites, including Firefox, CPython, the Linux kernel, and selected GitHub projects. This dataset serves as our primary benchmark for assessing the detection capabilities of both traditional static analysis tools and LLMs, particularly with respect to FN. Since this benchmark only includes buggy code, it is not suitable for evaluating FP. These 50 samples were manually reduced to minimal reproductions, each fitting within a single file and preserving the bug's core allocation and dereference logic. This reduction enabled controlled analysis while maintaining realism.

4.1.4.2 Linux Kernel Benchmark

To evaluate the framework's behavior on clean, non-buggy code and assess its FP rate, we introduce a secondary benchmark derived from the Linux kernel. This benchmark includes 364,122 compilable functions extracted from the AnghaBench project [89] and organized to preserve the original kernel directory hierarchy. The Linux kernel presents an ideal testbed for robustness analysis: it is large, complex, performance-critical, and written in a systems programming style that often challenges both static analyzers and LLMs.

By applying the framework to this high-quality corpus, we are able to quantify

spurious warnings and assess the precision of each analysis method in real-world, bug-free settings. This allows us to complement the UaF-focused evaluation with insights into over-approximation behavior and tool conservativeness.

4.2 Evaluation

4.2.1 Experimental Setup

To evaluate the effectiveness of LLMs in detecting UaF vulnerabilities, we conduct a structured comparison between LLM-based analysis and traditional static tools using two complementary benchmarks. The UaF Bug Dataset (Section 4.1.4.1) contains 50 real-world vulnerabilities, enabling the assessment of true positive and false negative rates. In contrast, the Linux Kernel Benchmark (Section 4.1.4.2) provides over 360,000 clean functions, allowing us to measure false positive rates and robustness on realistic, large-scale systems code.

We evaluate four representative LLMs—GPT-4, Claude, LLaMA-2, and GPT-3.5—under diverse input conditions that reflect realistic usage scenarios. These conditions vary along two primary axes: the level of abstraction and the size of contextual information. For abstraction, we provide either raw C and C++ source code (as seen by a developer) or LLVM IR, enabling a direct comparison of high-level versus compiler-lowered reasoning. For context granularity, we explore inputs ranging from isolated functions to complete single-file programs, allowing us to assess how much surrounding code is necessary for reliable detection.

Each LLM is prompted with handcrafted instructions designed to elicit static reasoning behavior. We also compare against traditional static analyzers such as *scan-build*, treating them as deterministic baselines.

Each model's output is manually reviewed and labeled by human evaluators. For buggy cases, we classify responses as TP, FN, or FP. For clean code from Linux kernel, any incorrectly flagged issue is considered an FP. We also analyze different prompt styles and content scopes to measure their influence on detection capability.

4.2.2 Evaluation Objectives

The main objectives of our evaluation are to answer the following research questions:

RQ1. How effective are LLMs in detecting UaF vulnerabilities in real-world code? We assess detection accuracy across a curated set of UaF bugs, comparing LLMs with traditional static analyzers. We also examine performance across varying context sizes (single function, multi-function, full file) to determine how much contextual information is necessary for reliable detection.

RQ2. What are the sources of false positives and false negatives in LLM-based analysis? We categorize LLM failure modes and contrast them with those observed in static analysis tools, identifying both overlapping and unique weaknesses.

RQ3. Are standard evaluation metrics sufficient to characterize LLM performance in bug detection? We discuss whether LLMs introduce new challenges—such as non-determinism or hallucinated reasoning—that necessitate rethinking evaluation criteria beyond traditional precision, recall, and coverage.

RQ4. How does prompt design influence LLM performance in static analysis tasks? We evaluate several prompting strategies—such as minimal instructions, pattern-primed prompts, and structured templates—to measure their impact on detection effectiveness.

RQ5. Does the level of input abstraction (source code vs. LLVM IR) affect detection outcomes? We investigate whether providing LLMs with lower-level IR representations improves or degrades their vulnerability detection ability.

4.2.3 RQ1: Evaluating the Effectiveness of LLMs for UaF Detection Under Varying Context Granularity

A key challenge in applying LLMs to static code analysis lies in their limited ability to reason over structural relationships and program-wide dependencies. Unlike

traditional tools, which construct control-flow and data-flow graphs for precise reasoning, LLMs operate solely on raw source code and may struggle with fragmented or localized inputs.

To investigate how much context an LLM needs to effectively UaF vulnerabilities, we focus on GPT-4 as a representative LLM due to its strong performance in prior evaluations. Additional models (e.g., Claude, GPT-3.5) are evaluated in later sections to provide a broader perspective. Rather than truncating the code input, we adopt a prompt engineering strategy to control the scope of analysis. The LLM is given the full source file in each case, but the prompt explicitly instructs the model to restrict its analysis to: (1) a single function, (2) two adjacent functions, or (3) the entire file. This approach simulates varying levels of local and interprocedural reasoning while holding the code content constant. Table 4.1 shows the TP and FP rates across these configurations.

Table 4.1: Impact of Context Size on LLM UaF Detection in GPT-4

Context Provided	**TP**	**FP**
Single Function	19/50 (38%)	2.1 per bug
Two Adjacent Functions	24/50 (48%)	3.5 per bug
Full File	32/50 (64%)	6 per bug

As context increases, LLMs show improved recall: full-file analysis recovers 64% of the ground-truth bugs, compared to only 38% with single-function input. However, this gain in recall comes at the cost of a significant rise in false positives—from 2.1 to 6.0 per bug—suggesting overgeneralization and hallucinated reasoning in larger contexts.

Figure 4.2 illustrates a UaF case where a pointer into a temporary buffer is returned after the buffer is freed. In this example, `strdup()` allocates memory for `_haystack`, which is later freed before the function returns. The returned pointer `ret` references a location within `_haystack`, leading to a classic use-after-free. When prompted to analyze only the local function, GPT-4 fails to detect the bug, as memory allocation and deallocation logic is obscured. With access to adjacent functions, the

```c
char *strcasestr(const char *haystack, const char *needle) {
    // strdup() allocates memory using malloc
    char *_haystack = strdup(haystack);
    char *_needle = strdup(needle);

    // strstr() returns pointer into _haystack
    ret = strstr(_haystack, _needle);

    if (ret)
        // convert to haystack offset
        ret = (char *)haystack + (ret - _haystack);

    free(_haystack);
    free(_needle);

    return ret; // UaF: points to freed memory
}
```

Figure 4.2: UaF bug involving pointer arithmetic and memory deallocation.

model identifies the vulnerability but incorrectly flags unrelated pointer operations as potential UaFs. Full-file analysis allows correct detection of the bug, but at the cost of six false positives due to hallucinated reasoning on unrelated memory operations.

These results indicate that larger context improves recall but reduces precision. For practical deployment, prompt constraints and FP filtering mechanisms may be necessary to contain spurious predictions in full-file mode.

Comparison with Static Analysis Tools

To assess the benefits and limitations of LLMs relative to traditional methods, we compare the LLMs with Clang Static Analyzer (CSA), using the scan-build interface to analyze the same source files. Table 4.2 shows that GPT-4 achieves higher recall (64% vs. 42%) but suffers from significantly more false positives (6.0 vs. 0.9 per bug). Static analyzers leverage AST/CFGs for precise control-flow reasoning, allowing them to avoid many hallucinated detections, but they often miss semantically complex cases that LLMs can recognize from learned patterns.

Table 4.2: LLM vs. Clang Static Analyzer on UaF Detection

Method	Requires AST/CFG	TP	FP
LLM (Full File)	No	32/50 (64%)	6.0 per bug
CSA (scan-build)	Yes	21/50 (42%)	0.9 per bug

Interestingly, we observe that LLMs are more robust than static analyzers when analyzing partial or incomplete code. Traditional tools often fail entirely without a complete AST or compilation context, whereas LLMs can still provide partial insights based on learned heuristics.

While this section focuses on the tradeoff between recall and false positives under varying context scopes, a more detailed investigation of false positive and false negative behaviors, along with their root causes is presented in RQ2.

4.2.4 RQ2: Detection Accuracy and Error Characteristics of LLMs

To assess how reliably LLMs detect UaF vulnerabilities in practice, we evaluate their TP and FP rates across the 50-bug UaF dataset. Table 4.3 compares the performance of four widely used LLMs against Clang Static Analyzer (CSA).

Table 4.3: UaF Detection Performance Across LLMs and Static Analysis Tools

Model	TP (Recall)	FP (per bug)	FN
GPT-4	32/50 (64%)	6.0	18
Claude	27/50 (54%)	4.2	23
GPT-3.5	20/50 (40%)	2.9	30
LLaMA-2	18/50 (36%)	1.5	32
CSA (scan-build)	21/50 (42%)	0.9	29

GPT-4 achieves the highest recall but also exhibits a substantial number of hallucinated reports. Static analysis tools, by contrast, demonstrate lower recall but maintain higher precision. These results confirm that LLMs are capable of reasoning about UaF semantics but still suffer from over-approximation and context confusion.

Failure Patterns: Sources of False Negatives

We perform a qualitative analysis of the 18 missed cases by GPT-4 to understand the underlying failure patterns. We identify five recurring challenges:

1. Memory Aliasing and Indirect Access. LLMs struggle to model memory aliasing relationships when two or more pointers refer to the same allocation. For example:

```
char *p1 = malloc(100);
char *p2 = p1; // alias
free(p1);
*p2 = 'X';     // UaF: missed by all models
```

The LLM fails to associate `p1` and `p2`, missing the fact that `*p2` dereferences a freed region. In contrast, static analyzers construct explicit aliasing models, enabling them to detect such bugs.

2. Multi-Stage Allocation and Free Across Functions. When memory is allocated and released across different functions, LLMs frequently fail to maintain the lifetime chain. Consider:

```
void cleanup(char *ptr) {
    free(ptr);
}

void foo() {
    char *data = malloc(100);
    cleanup(data);
    data[0] = 'a'; // UaF: not detected
}
```

Without call-graph understanding, the model cannot resolve that `data` was freed in `cleanup()`.

3. Loop-Based Deallocation and State Tracking. Loops that deallocate memory over multiple iterations confuse the LLM's state model. It often assumes one-time behavior:

```
void free_all(Node *node) {
    while (node) {
        free(node);
        node = node->next;
    }
}
```

If `node` is accessed after `free_all()` is called, the LLM may miss the UaF, especially when the access occurs several lines away or in another function.

4. Pointer Arithmetic and Offset Confusion. Complex pointer manipulations lead to reasoning breakdowns. For example:

```
char *buf = malloc(100);
char *mid = buf + 50;
free(buf);
*mid = 'x'; // UaF: missed by GPT-3.5 and Claude
```

Only GPT-4 detects this case; others fail to associate the offset `mid` with the freed base pointer `buf`.

5. Subtle API Misuse and Implicit Free. Certain APIs free memory implicitly (e.g., `strdup`, `fclose` on `FILE*` with internal buffers). LLMs struggle to recognize memory release when it is not an explicit `free()` call, resulting in silent FNs.

Overprediction Patterns: Sources of False Positives

In addition to missed vulnerabilities, LLMs also produce a substantial number of false positives. Across all models, we observe several recurring causes:

- *Alias Confusion (21%)*: Misinterpretation of pointer aliases or shared memory references.
- *Loop Oversight (18%)*: Inability to resolve deallocations occurring in loops or recursive calls.
- *Safe Copy Mislabeling (14%)*: Flagging memory reads on safe clones as UaF.
- *Implicit Free Unawareness (12%)*: Misses or wrongly assumes freeing through functions like `strdup()` or `free_list()`.
- *Order Dependency (10%)*: Fails to track execution order or misjudges dereference happening before deallocation.
- *Overgeneralization (25%)*: Flagging common pointer patterns seen in training data regardless of actual safety.

This taxonomy reveals that overgeneralization and alias confusion are the top contributors to LLM misjudgments. In a pilot experiment on 15 test cases, adding alias-aware prompting and loop summaries reduced FP rates by 22%, suggesting actionable paths for mitigation.

4.2.5 RQ3: How Does Prompt Design Influence Detection Accuracy?

LLMs are inherently prompt-driven: their behavior and reasoning style can shift dramatically based on how a task is framed. In contrast to traditional static analyzers—which are deterministic and rule-based—LLMs operate as probabilistic models that rely heavily on contextual cues. This section evaluates how different prompt strategies impact UaF detection, highlighting the balance between FPs and FNs.

Prompt Strategy Design

We compare three prompt formulations:

- **Default Prompt:** A general query such as *"Find vulnerabilities in this code."* This allows the model to interpret the task freely, typically resulting in broad pattern matching and higher FP rates.

- **FP-Aware Prompt:** A more conservative formulation such as *"Identify only high-confidence use-after-free vulnerabilities. Avoid speculative or uncertain reports."* This reduces FPs by instructing the model to filter marginal cases.

- **Contextual Prompt:** A detailed instruction including memory behavior, pointer lifetime, and UaF semantics, e.g., *"Identify use-after-free bugs where freed memory is later accessed. Focus on memory allocation, free operations, and pointer dereferencing."*

Each prompt is evaluated on the same 50 UaF cases using GPT-4. The model is given the same source input, with only the prompt text changed.

Experimental Results

Table 4.4: GPT-4 Performance Under Different Prompt Designs

Prompt Type	TP	FP	Precision
Default	31	130	0.19
FP-Aware	27	55	0.33
Contextual	30	40	0.43

The default prompt detects the most vulnerabilities but suffers from a 3× increase in false positives compared to contextual prompting. The FP-aware prompt significantly reduces hallucinations but also sacrifices recall, missing 4 more bugs than the default.

Prompt Tradeoff Analysis

Prompt design introduces a precision-recall tradeoff: broader prompts promote higher recall but also raise FP rates, while narrower prompts suppress overgeneralization but risk filtering real bugs.

Additionally, we observe that FPs under the default prompt often stem from benign pointer arithmetic or post-allocation memory usage—operations that resemble UaF in structure but not in behavior. The contextual prompt successfully avoids most of these cases by anchoring its judgment in semantic relationships.

Discussion: Prompt as a Control Interface

These results suggest that prompt design is more than a surface-level tweak: it acts as a semantic interface that modulates model behavior. Unlike deterministic tools, LLMs expose a degree of controllability via natural language, which can be leveraged for different use cases—broad scanning in exploratory auditing, or precise filtering in CI/CD pipelines. However, this flexibility introduces evaluation challenges. Metrics such as precision become prompt-sensitive, and benchmarking must account for prompt formulation as part of the experimental condition. We quantify this prompt volatility next.

Transition to RQ4

These findings reveal not only the behavioral flexibility of LLMs, but also their evaluation complexity. Conventional metrics such as precision and recall, while useful for static tools, fail to capture the dynamic, prompt-sensitive nature of LLM-based analysis. This motivates the need for new evaluation metrics, which we explore in the following section.

4.2.6 RQ4: Do LLMs Require New Evaluation Metrics?

Traditional evaluation metrics such as precision and recall offer a useful baseline for assessing static analysis tools. However, they fall short in capturing the unique

```c
char *alloc_buf() {
    char *p = malloc(100);
    return p;
}

void use_buf(char *q) {
    free(q);
    printf("%c\n", q[0]); // UaF
}

int main() {
    char *buf = alloc_buf();
    use_buf(buf);
}
```

Figure 4.3: UaF Case with Interprocedural Aliasing.

behavior patterns and failure modes exhibited by LLM-based code analysis. In this section, we examine these limitations and propose new metrics to better evaluate LLM-based static analysis systems.

Motivation: Human, Static Tools, and LLMs Reason Differently

To understand why existing metrics are insufficient, we compare how humans, static analyzers, and LLMs interpret the same vulnerable program. Consider the UaF example in Figure 4.3, which involves memory allocation in one function and deallocation in another.

Human analysts understand that `q` in `use_buf()` is an alias for the heap memory allocated in `alloc_buf()`. They deduce that printing `q[0]` after `free(q)` constitutes use-after-free.

Static analyzers resolve this through interprocedural control-flow and alias analysis, depending on the tool configuration. With full call-graph inlining, this bug is usually detected.

LLMs, however, often fail unless all relevant functions are placed adjacently in the prompt. Even then, they may not infer memory aliasing or ownership transfer

unless explicitly guided through the prompt. Their failure stems not from misunderstanding UaF as a concept, but from token-limited, local pattern-based reasoning. Thus, when an LLM misses such a bug, standard precision/recall metrics attribute it to a detection failure—when in fact, it is a context limitation. Similarly, if an LLM hallucinated a UaF in benign code due to stylistic similarity, this is fundamentally different from static tools' failure modes. This motivates the introduction of new evaluation metrics that expose these LLM-specific behaviors.

New Evaluation Metrics for LLM-Based Code Analysis

To address the limitations above, we introduce three supplementary metrics:

(1) Scalability Index: Can LLMs detect cross-file bugs?

LLMs operate within limited input contexts. To quantify their ability to detect vulnerabilities that span multiple files or non-adjacent functions, we define:

$$ScalabilityIndex = \frac{Cross - FileTruePositives}{TotalMulti - FileBugs} \tag{4.1}$$

In our dataset, 10 UaF bugs span across files. GPT-4 successfully detects 2:

$$ScalabilityIndex_{GPT-4} = \frac{2}{10} = 0.20$$

(2) Context-Aware Precision: Does more context improve accuracy?

More code context often improves recall, but may inflate hallucinated detections. We define:

$$CAPR = \frac{P_{FullFile}}{P_{SingleFunction}} \tag{4.2}$$

Precision decreases as more context is added. This suggests that context scaling introduces more surface-level matches mistaken for real vulnerabilities, underscoring the need for context-aware precision metrics.

(3) Prompt Sensitivity Score (PSS): Quantifies volatility due to prompt variation. We define PSS as:

$$PSS = \frac{\max(P) - \min(P)}{\max(P)} \tag{4.3}$$

A high PSS indicates significant fluctuation in performance depending on prompt phrasing, suggesting that benchmarking LLMs requires prompt standardization.

Why New Metrics Are Necessary

These new metrics offer a more faithful evaluation of LLM behavior and failure modes. We argue that any future benchmarking framework for LLM-assisted code analysis should incorporate such metrics to enable fairer comparisons and guide model development toward more reliable vulnerability detection.

4.2.7 RQ5: Does the Level of Input Abstraction Affect Detection Accuracy?

While most existing work on LLM-based vulnerability detection focuses on source-level reasoning, we explore whether exposing the model to a lower-level, compiler-normalized representation such as LLVM IR yields better results.

Motivation

LLVM IR abstracts away high-level syntactic noise while preserving control and data flow semantics. It is widely used in formal verification and static analysis pipelines. If LLMs can process IR effectively, it could serve as a language-independent input for future tools.

Experimental Setup

We evaluate GPT-4 under two input conditions:

- **Source-level**: Raw C/C++ code with standard prompting.
- **IR-level**: Clang-compiled LLVM IR (O0 level), converted to readable textual format.

For fairness, both inputs describe the same code logic. The prompts used for IR-level detection were adapted to include: "This is LLVM IR. Identify UaF vulnerabilities based on memory allocation, free, and use."

Results and Analysis

Table 4.5: GPT-4 UaF Detection: Source Code vs. LLVM IR

Input Format	TP	FP	Precision
Source Code	32/50	120	0.21
LLVM IR	28/50	60	0.32

LLM performance on LLVM IR showed slightly reduced recall (28 vs. 32 TP), but significantly fewer false positives. This suggests that normalization through IR reduces syntactic distractors, helping LLMs avoid overgeneralized pattern matching.

Error Analysis

False negatives on IR are mostly due to: - Pointer aliasing being harder to infer from flat memory loads/stores; - Lack of type annotations and variable names; - Opaque call structures (e.g., missing function names).

Meanwhile, source-level prompts fail mostly due to: - Local context isolation; - Overreaction to common function names (e.g., `strcpy`, `malloc`) without semantic grounding.

Evaluation Summary

Our evaluation reveals that LLMs exhibit promising capabilities in detecting use-after-free vulnerabilities, surpassing traditional static analyzers in recall but often at the cost of precision. Through a series of controlled experiments, we demonstrate that context size, input representation, and prompt design significantly impact model behavior. Despite their effectiveness, LLMs show high variability across prompts, frequent overgeneralization, and struggle with interprocedural reasoning. To better

evaluate these unique behaviors, we propose three new metrics—Scalability Index, Context-Aware Precision, and Prompt Sensitivity Score—that more faithfully capture the dynamics of LLM-based analysis. We also show that prompt engineering serves as a practical control lever for modulating model precision and recall. While our study focuses on single-file detection, it lays the groundwork for deeper integration with static analysis systems, multi-file scaling, and hybrid verification in future research.

4.3 Discussion

Our evaluation demonstrates that LLMs offer a new and complementary perspective on detecting UaF vulnerabilities, particularly in C/C++ programs. Through five structured research questions, we assess not only their raw detection capabilities but also their behavioral patterns, limitations, and evaluation challenges when compared to traditional static analysis tools.

LLMs outperform static analyzers in recall but not in precision. In RQ1 and RQ2, we find that LLMs, especially GPT-4, can identify more true UaF cases than rule-based analyzers such as scan-build (Clang Static Analyzer). However, this comes at the cost of higher false positive rates due to overgeneralization, imprecise lifetime tracking, and hallucinated reasoning chains. Our error taxonomy highlights that alias confusion and unsafe pattern overfitting are key contributors to these issues.

Evaluation must account for context sensitivity and prompt volatility. As shown in RQ3 and RQ4, LLM performance is highly sensitive to both input granularity and prompt formulation. Broader input context generally improves recall but introduces noisy matches. Prompt engineering can modulate this behavior, balancing precision and recall through explicit instructions. We show that prompts not only influence accuracy but also act as a semantic control interface, requiring new evaluation metrics to capture these dynamics.

New metrics are necessary for fair LLM evaluation. Traditional precision and recall are insufficient to reflect the unique behaviors of LLMs. RQ3 introduces three new metrics: (1) Scalability Index, to measure multi-file or interprocedural reasoning

ability under token limits; (2) Context-Aware Precision, to track false positive inflation as context scales; and (3) Prompt Sensitivity Score, to quantify volatility across prompt styles. These metrics provide more meaningful insights for researchers and practitioners deploying LLMs in real static analysis tasks.

Input abstraction affects interpretability. RQ5 reveals that analyzing LLVM IR can help reduce false positives by exposing lower-level program semantics. However, it may also suppress some true positives due to loss of high-level intent. Source code and IR provide complementary benefits, and the choice of abstraction should align with analysis goals.

Taken together, our evaluation framework not only benchmarks detection capability but also uncovers structural and behavioral properties unique to LLM-based tools. These findings lay the foundation for more robust LLM-guided security workflows, emphasizing the need for precision control, context management, and human-in-the-loop verification.

4.4 Summary

We present a comprehensive study of LLM-based static analysis for detecting use-after-free vulnerabilities. By constructing a carefully curated benchmark of 50 real-world UaF cases, we evaluate multiple LLMs under varying context sizes, input abstractions, and prompt strategies. Our results demonstrate that LLMs outperform static analysis tools in recall but introduce significantly higher false positives, particularly under default prompting and limited context control.

To better characterize these limitations, we introduce a taxonomy of LLM errors and propose three new metrics—Scalability Index, Context-Aware Precision, and Prompt Sensitivity Score—that highlight where traditional evaluation metrics fall short. These contributions not only reveal the strengths and weaknesses of current LLMs but also provide actionable insights for tool designers, prompting strategies, and future benchmark development.

Overall, our findings suggest that LLMs are a promising but currently imprecise static analysis tool. With careful prompt design, hybrid integration, and targeted evaluation, they hold strong potential for advancing practical software vulnerability detection.

Chapter 5

RELATED WORK

This section provides an overview of previous research in three interrelated domains: empirical investigations into bug characteristics, bug detection techniques, and software fuzzing. We also highlight existing gaps in the context of each domain, setting the stage for our study on UaF.

5.1 Empirical Study on Bug Characteristics

Several previous studies have focused on specific types of bugs other than UaF. Lu et al. [121] studied 105 concurrency bugs from four mature open source applications. They found that 97% of the studied concurrency bugs belong to two simple bug patterns. Jin et al. [110] conducted a comprehensive study of 109 real-world performance bugs from five software suites. Their findings led to the creation of 25 efficiency check rules that uncovered 332 previously unknown performance problems. Our work instead focused on the patterns and potential detection opportunities of UaF bugs. There are also a bunch of studies [99, 139, 170, 150] on generic bugs and failures in various applications. By contrast, our study found that UaF bugs have more diverse unique patterns. Different from the aforementioned work that focused on the buggy and vulnerable code itself, Holzinger et al. [107] studied the security vulnerabilities in Java platforms from the perspective of the exploits collected in the wild. The collection and analysis of UaF exploits in the wild will be our future work. Lastly, another line of empirical study that is marginally related to ours is the fault tolerance and recovery of software in face of triggered errors and bugs [85, 160]. In summary, while there have been a bunch of existing empirical studies on software bugs, none of them have focused on UaF.

5.2 Static Bug Detection

Static bug detectors have been popular among software developers. GUEB [100] is a static tool for detecting UaF vulnerabilities in binary code. It tracks heap operations and address transfers to identify program locations that allocate or free heap memory. By analyzing the resulting graph, GUEB identifies UaF vulnerabilities, offering better coverage than dynamic tools. However, GUEB's points-to analysis is imprecise, affecting its soundness and precision. Additionally, the path analysis and value set analysis in GUEB are resource-intensive and not easily scalable to large programs. DCUaF [74] applied the local-global analysis to detect concurrency UaF bugs in Linux device drivers. The tool has successfully confirmed 95 real cases that are previously unknown. `Palfrey` is instead built upon a set of patterns learnt from real bugs. In particular, `Palfrey` demonstrates that many UaF bugs can be detected without using sophisticated techniques. Also, our study provides guidelines for future work to build better static UaF detectors. CID [151] uses two dimension checking to detect INC-DEC inconsistency and DEC-DEC inconsistency, but it is unable to detect borrowed reference bugs since the cause of such bugs is due to the missing ownership not reference count inconsistency. There are many other detectors for various specific types of bugs and applications, e.g., API misuse [171], missing checks [165], buffer overflows [112], semantic bugs [129], violations of inferred programmer beliefs [99], configuration errors [163], and bugs in JavaScript bindings [80]. However, none of them target UaF bugs.

5.3 UaF Mitigation and Prevention

Mitigation and prevention strategies for UaF vulnerabilities encompass a diverse range of approaches aimed at minimizing the risks posed by these vulnerabilities. These strategies address the potential exploitation of UaF by identifying dangling pointers, preventing reuse of freed memory, and enhancing memory management practices. Some of these strategies include:

Page Protection: Certain techniques leverage the virtual memory system to prevent UaF attacks. Examples like Electric Fence [22], Dhurjati and Adve [124], and Oscar [91] utilize this approach. They allocate each object on a separate virtual page, which helps prevent the reuse of freed memory. However, this technique can lead to high Translation Lookaside Buffer (TLB) pressure, causing performance degradation under pointer-intensive workloads.

Pointer Nullification: Another approach involves zeroing out pointers upon deallocation, effectively removing dangling pointers. Techniques like DangNull [114], FreeSentry [169], and DangSan [155] follow this strategy. However, this method can result in performance loss and memory overhead, especially in scenarios with numerous pointer copies.

Automatic Memory Management (Garbage Collection): Garbage collection [162] prevents UaF by deallocating memory only when the last pointer to an object is deleted. While this approach can prevent UaF, it can introduce high overheads and may not be suitable for C and C++ applications that use manual memory management.

Pointer Labeling: Some systems label pointers and their allocations uniquely to prevent the reuse of dangling pointers. Techniques like CETS [133] utilize this method. However, these techniques can introduce false positives and high overhead due to the need for checks on every pointer access.

Hardening Techniques: Techniques like DieHard and FreeGuard [75, 146] introduce probabilistic reuse delays to reduce the likelihood of UaF attacks. While not foolproof, these techniques usually result in relatively low performance loss compared to their security benefits.

These mitigation and prevention strategies provide various trade-offs between security and performance. While some methods focus on nullifying pointers or adding runtime checks to prevent UaF, others rely on memory protection mechanisms or enhanced memory management techniques. Each strategy aims to minimize the risk of UaF vulnerabilities and their potential exploitation by attackers, contributing to the

overall security of software systems.

5.4 LLMs for Program Analysis

Recent work has explored using LLMs such as Codex, GPT, and LLaMA for software engineering tasks including code summarization, generation, and bug detection. Studies like CodeT5+ [159] show that pretrained models can generalize to vulnerability prediction with limited supervision. However, most efforts evaluate performance on synthetic benchmarks (e.g., CodeXGLUE [122]) or focus on general bug types without targeting security-critical patterns like UaF. Our work complements this by offering a task-specific framework, tailored prompt strategies, and more nuanced error categorization.

Chapter 6

CONCLUSION AND FUTURE WORKS

In this dissertation, we have delved into the significant reliability challenges posed by UaF vulnerabilities in software systems. These vulnerabilities, which occur when a program accesses memory that has been previously freed, present a serious threat to system security and reliability. This dissertation aims to address these challenges by exploring various aspects of UaF detection and mitigation.

The first part of our proposed research focuses on understanding the characteristics of UaF vulnerabilities and aims to improve static and dynamic analysis techniques for bug detection. We have conducted an empirical study to investigate the patterns, manifestations, fix strategies, and other features of real-world UaF bugs. This study is based on the analysis of 150 randomly selected UaF vulnerabilities from 41 large and mature open-source projects, providing valuable insights into UaF bug detection and protection. Our work has identified common patterns and characteristics, which can guide the development of more effective bug detection and protection methods.

The second part of our research centers on the exploration of the effectiveness of dynamic software fuzzing in identifying UaF vulnerabilities. Fuzzing is a popular dynamic testing technique that generates a large amount of input data to stress-test target programs. However, the relationship between the inherent randomness of fuzzing and the detection of UaF vulnerabilities remains unclear. This research investigates the performance of fuzzers when testing UaF vulnerabilities with different characteristics and patterns. Additionally, we evaluate the impact of fuzzing configuration parameters and auxiliary tools like ASan [142] on bug discovery. Our findings provide practical guidelines for enhancing UaF detection through fuzzing techniques and contribute to the advancement of software security practices.

The third part of this dissertation investigates how LLMs interpret UaF vulnerabilities compared to traditional static analyzers. We constructed a light-weight benchmark of 50 real-world UaF cases specifically adapted for LLM testing by distilling and simplifying samples from a broader 150-case dataset. These minimal, single-file programs enabled consistent evaluation across different LLM configurations. Our analysis explored the impact of prompt design, code abstraction level (source code vs. LLVM IR), and context granularity on detection outcomes. We introduced three LLM-specific metrics, Scalability Index, Context-Aware Precision, and Prompt Sensitivity Score, to better understand the models' reasoning limitations. The results uncovered key weaknesses, including misinterpretation of aliasing and failure to track interprocedural memory state. These findings emphasize that while LLMs are promising, they benefit most when integrated with traditional static analysis in a hybrid pipeline.

6.1 Future Work

Looking ahead, our future work aims to address the challenge of silent UaF vulnerabilities, which hinder dynamic analysis tools, such as fuzzers, from effectively detecting bugs. Silent UaF vulnerabilities do not lead to program crashes, making them difficult to identify using conventional dynamic analysis tools. To overcome this limitation, we propose to design a memory allocator that increases the likelihood of UaF vulnerabilities triggering crashes, thus making them more detectable. This memory allocator will aim to minimize memory overhead while providing reliable UaF detection. By ensuring that UaF vulnerabilities do not remain silent, we can significantly enhance the effectiveness of dynamic analysis tools in identifying these critical bugs.

6.2 Recent Work on Memory Allocator

The field of secure allocators has made significant strides in addressing memory safety violations, with various designs aimed at preventing UaF vulnerabilities. Early work in this area demonstrated that controlling object placement could effectively mitigate UaF risks, particularly by randomizing freed page reuse. However, these early

solutions suffered from impractical levels of memory overhead, making them unsuitable for real-world deployment.

Subsequent designs, such as DieHard [75] and its successor, DieHarder [81], improved the efficiency of spatial memory safety simulation, showing promise in securing memory operations. FreeGuard [146] further advanced this work by integrating optimizations from performance-focused allocators. Although FreeGuard achieved better security compared to the default Linux allocator, it fell short of the security provided by DieHarder.

Guarder followed, proposing ways to bridge the performance-security gap between FreeGuard and DieHarder. While this body of work has contributed significantly to memory safety, the approach of PUMM [164] takes a distinct angle. Instead of relying on bounded reuse probabilities, PUMM proposes the concept of Execution Unit Partitioning (EUP) as a fundamental basis for preventing UaF.

There is also prior work that aims to prevent UaF vulnerabilities by restricting reuse to within objects of the same type, offering partial UaF protection. In this work, a direct comparison is made against FFmalloc [161] and MarkUs [73], two allocator designs that provide UaF protection with better performance than the FreeGuard line of work.

FFmalloc is the latest design based on Object Type Attributes (OTA), surpassing the performance of previous solutions like Oscar [91]. Similarly, MarkUs retrofits garbage collection into legacy software and achieves improved performance compared to prior work in the same domain. PUMM shares similarities with MarkUs in temporarily quarantining addresses to prevent reallocation. However, it distinguishes itself through its use of EUP, a concept that is not limited by pointer hiding in target programs.

6.3 Need for a Custom Allocator

Limitations and the Need for a Custom Allocator Despite the progress in allocator-related work, certain limitations persist. While these designs show promise in enhancing security and performance, no single allocator fully addresses the complex problem

of UaF prevention. As a result, there remains a need to design a custom allocator that optimally balances performance, security, and the prevention of silent UaF vulnerabilities.

Addressing these limitations and creating a custom allocator can contribute to further advancements in the field of memory safety, making dynamic analysis tools more effective in detecting and mitigating UaF vulnerabilities.

6.4 Future Directions for LLM-Based Vulnerability Detection

This dissertation explores UaF vulnerabilities from multiple perspectives, including empirical characterization, fuzzing-based discovery, and LLM-assisted detection. However, the exploration of LLMs represents only an initial step toward understanding and leveraging large language models for vulnerability analysis. Several research directions remain open for expanding the role of LLMs in secure software development.

LLM + Formal Verification. Our current study evaluates LLMs in static, single-file scenarios and focuses on detection. A natural extension is to integrate LLMs with formal methods and symbolic execution tools to support patch generation and validation. For instance, combining LLM-suggested fixes with tools like Alive2 [120] or KLEE [83] could enable post-hoc correctness checking, rule out invalid edits, and close the loop between generation and verification.

Beyond Use-After-Free. Although UaF vulnerabilities offer a rich testbed, LLM behavior may differ significantly when applied to other vulnerability types. Future work should explore model performance on bugs such as double free, null dereference, memory leaks, and race conditions. Each category may present different reasoning barriers and require specialized prompts or architectural adaptations.

Cross-File and Project-Scale Reasoning. One of the primary limitations of current LLMs is their bounded input window. As many real-world vulnerabilities span multiple files or modules, evaluating LLMs in a project-wide setting remains an open challenge. Promising directions include code segmentation techniques based on call

graphs or symbol dependencies, as well as retrieval-augmented generation (RAG) using code-aware embeddings to reconstruct broader execution contexts dynamically.

Interactive and Hybrid Tool-chains. A future vision is the development of interactive analysis systems where LLMs and static tools collaborate in a feedback loop. For example, static tools could flag suspicious control-flow regions for deeper LLM review, while LLMs could propose areas overlooked by deterministic rules. Such a pipeline would combine the strengths of formal soundness with LLM-guided flexibility, potentially bringing LLM-assisted analysis closer to production use in secure development environments.

BIBLIOGRAPHY

[1] https://bugs.python.org/issue24097.

[2] https://github.com/YosysHQ/yosys/pull/2498.

[3] https://github.com/OpenModelica/OpenModelica/pull/7760.

[4] https://github.com/legionus/kbd/.

[5] https://github.com/numactl/numactl/pull/114.

[6] https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2018-6555.

[7] https://github.com/coturn/coturn/issues/601.

[8] https://github.com/DCMTK/dcmtk/.

[9] https://bugs.python.org/issue24613.

[10] https://bugs.python.org/issue27867.

[11] https://github.com/profanity-im/profanity/pull/1428.

[12] https://github.com/ndilieto/uacme/commit/de76f19.

[13] https://bugzilla.nasm.us/show_bug.cgi?id=3392414.

[14] https://github.com/ckolivas/lrzip/issues/96.

[15] https://github.com/yasm/yasm/issues/91.

[16] https://nvd.nist.gov/vuln/detail/CVE-2019-6455.

[17] https://bugs.python.org/issue24094.

[18] https://bugs.python.org/issue24407.

[19] https://bugzilla.redhat.com/show_bug.cgi?id=1484332.

[20] https://github.com/cesanta/mjs/issues/73.

[21] https://github.com/cesanta/mjs/issues/78.

[22] https://linux.softpedia.com/get/Programming/Debuggers/Electric-Fence-3305.shtml.

[23] Cwe-416: Use after free. https://cwe.mitre.org/data/definitions/416.html.

[24] https://bugzilla.mozilla.org/show_bug.cgi?id=133773, 2001.

[25] https://bugs.python.org/issue24613, 2015.

[26] https://bugs.python.org/issue24097, 2015.

[27] https://bugzilla.redhat.com/show_bug.cgi?id=1196581, 2015.

[28] https://bugs.python.org/issue24099, 2015. Accessed: 2020-11-01.

[29] https://bugs.python.org/issue24101, 2015. Accessed: 2020-11-01.

[30] https://bugzilla.kernel.org/show_bug.cgi?id=188941, 2016.

[31] https://bugs.python.org/issue27867, 2016.

[32] https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0728, 2016. Accessed: 2020-07-12.

[33] https://hg.mozilla.org/releases/mozilla-beta/rev/fb00d84ec825, 2017.

[34] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15129, 2017.

[35] https://github.com/python/cpython/commit/4d3f084c035ad3dfd9f8479886c41b1b1823ace2, 2017. Accessed: 2020-07-12.

[36] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=39675f7a7c7e7702f7d5341f1e0d01db746543a0, 2018. Accessed: 2020-07-12.

[37] 2018.

[38] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6974, 2019.

[39] https://nvd.nist.gov/vuln/detail/CVE-2019-5603, 2019.

[40] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9060cb719e61b685ec0102574e10337fa5f445ea, 2019.

[41] https://bugs.python.org/issue40294, 2020. Accessed: 2020-11-01.

[42] https://github.com/microsoft/LightGBM/pull/2743, 2020. Accessed: 2020-11-01.

[43] https://github.com/ndilieto/uacme/commit/de76f1926f405a6d884dc0f5bb5001ee34a1e5ad, 2020.

[44] https://github.com/verilator/verilator/commit/f98782c061e1f1718677090e4adc1c7576377b68, 2020.

[45] Available online: https://google.github.io/oss-fuzz/, 2020. Accessed: 2020-7-19.

[46] https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html, 2020. Accessed: 2020-07-12.

[47] https://devguide.python.org/garbage_collector/, 2020. Accessed: 2020-07-12.

[48] https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html, 2020. Accessed: 2020-07-12.

[49] https://en.cppreference.com/book/intro/smart_pointers, 2020. Accessed: 2020-07-12.

[50] https://doc.rust-lang.org/1.30.0/book/2018-edition/ch04-00-understanding-ownership.html, 2020. Accessed: 2020-07-12.

[51] https://github.com/, 2020. Accessed: 2020-11-01.

[52] https://github.com/coturn/coturn/issues/601, 2020.

[53] https://github.com/scala-native/scala-native/pull/2072, 2020.

[54] https://www.cplusplus.com/reference/list/list/erase/, 2020. Accessed: 2020-07-12.

[55] https://www.cplusplus.com/reference/list/list/remove/, 2020. Accessed: 2020-07-12.

[56] https://github.com/DCMTK/dcmtk/commit/7c3ca88c6197af3d03b8376aeb46e6cc7d7c3724, 2020.

[57] https://github.com/9fans/plan9port/commit/2991442aef1cf020ffde43673433ee97ef322a53, 2020.

[58] https://docs.python.org/3/c-api/intro.html, 2020. Accessed: 2020-07-12.

[59] https://pythonextensionpatterns.readthedocs.io/en/latest/refcount.html, 2020. Accessed: 2020-07-12.

[60] https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html, 2020. Accessed: 2020-07-12.

[61] https://github.com/strongcourage/uafbench, 2020. Accessed: 2022-01-12.

[62] Refcount tracing and balancing. https://firefox-source-docs.mozilla.org/performance/memory/refcount_tracing_and_balancing.html, 2020.

[63] https://github.com/osquery/osquery/pull/6880/files, 2021.

[64] https://www.iso.org/standard/74528.html, 2021. Accessed: 2021-09-09.

[65] https://docs.python.org/2/library/array.html#array.array.fromstring, 2021. Accessed: 2021-09-09.

[66] https://github.com/F-Stack/f-stack/pull/565, 2021.

[67] https://fbinfer.com/, 2021. Accessed: 2021-01-12.

[68] https://www.gimpel.com/, 2021. Accessed: 2021-01-12.

[69] https://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html, 2022. Accessed: 2022-01-17.

[70] https://clang.llvm.org/, 2023.

[71] https://llvm.org/, 2023.

[72] https://gcc.gnu.org/, 2023.

[73] Sam Ainsworth and Timothy M Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (S&P '20)*, pages 578–591.

[74] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 255–268, 2019.

[75] Emery D Berger and Benjamin G Zorn. Diehard: Probabilistic memory safety for unsafe languages. *ACM Sigplan Notices*, pages 158–168, 2006.

[76] H. Boehm, A. Demers, and M. Weiser. A garbage collector for c and c++. 2002.

[77] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 2329–2344.

[78] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering (ICSE '17)*, pages 489–506.

[79] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pages 1032–1043.

[80] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (S&P '17)*, pages 559–578, 2017.

[81] Robert G Brown, Dirk Eddelbuettel, and David Bauer. Dieharder. *Duke University Physics Department Durham, NC*, pages 27708–0305, 2018.

[82] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143, 2012.

[83] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[84] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy (S&P '15)*, pages 725–741, 2015.

[85] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *International Conference on Dependable Systems and Networks (DSN '00)*, 2000.

[86] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[87] Zeyu Chen, Daiping Liu, Jidong Xiao, and Haining Wang. All use-after-free vulnerabilities are not created equal: an empirical study on their characteristics and detectability. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 623–638, 2023.

[88] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[89] Anderson Faustino Da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390. IEEE, 2021.

[90] Zeyu Chen Daiping Liu, Jidong Xiao and Haining Wang. All use-after-free vulnerabilities are not created equal: An empirical study on their characteristics and detectability. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23)*.

[91] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX Security Symposium (USENIX Security '17)*, pages 815–832.

[92] DARPA. Darpa cyber grand challenge binaries, 2018.

[93] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, pages 166–177, 2015.

[94] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN '06)*, pages 269–280.

[95] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR '21)*, pages 131–142.

[96] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE symposium on security and privacy (S&P '16)*, pages 110–121.

[97] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[98] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, pages 237–252, 2003.

[99] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88.

[100] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, pages 211–217, 2014.

[101] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[102] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *2008 Network and Distributed System Security Symposium (NDSS '08)*.

[103] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pages 72–82.

[104] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *22nd USENIX Security Symposium (USENIX Security '13)*, pages 49–64, 2013.

[105] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In *2018 Network and Distributed System Security Symposium (NDSS '18)*.

[106] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, pages 1–17, 2006.

[107] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pages 779–790.

[108] Shin Hong and Moonzoo Kim. Effective pattern-driven concurrency bug detection for operating systems. *Journal of Systems and Software*, pages 377–388, 2013.

[109] Ziniu Hu, Wenbo Yin, Shreya Jain, Yihan Chen, Zhaocheng He, Jialu Xu, and Jimmy Lin. Self-debugging large language models via chain-of-bug correction. *arXiv preprint arXiv:2305.17409*, 2023.

[110] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 77–88.

[111] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER '15)*, pages 560–564.

[112] D. Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium (USENIX Security '01)*, pages 177–190.

[113] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *2015 Network and Distributed System Security Symposium (NDSS '15)*.

[114] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *2015 Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[115] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 517–530.

[116] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering (ICSE '18)*, pages 475–485.

[117] Siliang Li and Gang Tan. Finding reference-counting errors in python/c programs with affine analysis. In *European Conference on Object-Oriented Programming*, pages 80–104, 2014.

[118] Linux Kernel Organization, Inc. https://bugzilla.kernel.org/, 2020.

[119] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, pages 1635–1648.

[120] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

[121] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339.

[122] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[123] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 115–124.

[124] Vitaliy B Lvin, Gene Novark, Emery D Berger, and Benjamin G Zorn. Archipelago: trading address space for reliability and security. *ACM SIGARCH Computer Architecture News*, pages 115–124, 2008.

[125] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security '19)*, pages 1949–1966.

[126] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. Rid: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 531–544, 2016.

[127] Paul E. McKenney. Overview of linux-kernel reference counting. 2007.

[128] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, pages 32–44, 1990.

[129] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, pages 361–377.

[130] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *18th USENIX Security Symposium (USENIX Security '09)*, pages 67–82.

[131] Mozilla Foundation. https://bugzilla.mozilla.org, 2020.

[132] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety.

In *2012 39th Annual International Symposium on Computer Architecture (ISCA '12)*, pages 189–200, 2012.

[133] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40, 2010.

[134] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, pages 89–100, 2007.

[135] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20)*, pages 47–62.

[136] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for Use-After-Free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20)*, pages 47–62, 2020.

[137] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 1–11.

[138] Mads Chr Olesen, René Rydhof Hansen, Julia L Lawall, and Nicolas Palix. Coccinelle: tool support for automated cert c secure coding standard certification. *Science of Computer Programming*, 91:141–160, 2014.

[139] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 305–318, 2011.

[140] Python Software Foundation. https://bugs.python.org/, 2020.

[141] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *2017 Network and Distributed System Security Symposium (NDSS '17)*, pages 1–14, 2017.

[142] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 309–318, 2012.

[143] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157, 2016.

[144] Kostya Serebryany. Oss-fuzz - google's continuous fuzzing service for open source software. August 2017.

[145] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Y. Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. In *Network and Distributed System Security Symposium (NDSS '19)*, 2019.

[146] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 2389–2403.

[147] C. Spatz. Basic statistics: Tales of distributions. 1981.

[148] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *2016 Network and Distributed System Security Symposium (NDSS '16)*, pages 1–16.

[149] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

[150] Gang Tan and Jason Croft. An empirical security study of the native code in the jdk. In *Proceedings of the 17th Conference on Security Symposium (USENIX Security '08)*, pages 365–378, 2008.

[151] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting kernel refcount bugs with two-dimensional consistency checking. In *30th USENIX Security Symposium (USENIX Security '21)*, pages 2471–2488, 2021.

[152] The MITRE Corporation. Cwe-416: Use after free. https://cwe.mitre.org/data/definitions/416.html, June 2019. Accessed: 2019-7-1.

[153] The MITRE Corporation. Search cve list. https://cve.mitre.org/cve/search_cve_list.html, January 2019. Accessed: 2019-7-24.

[154] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*, pages 405–419.

[155] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419, 2017.

[156] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the goblint approach. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*, pages 391–402.

[157] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '07)*, pages 205–214.

[158] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*, pages 999–1010.

[159] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[160] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. Characterization of linux kernel behavior under errors. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*

[161] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing use-after-free attacks with fast forward allocation. In *30th USENIX Security Symposium (USENIX Security '21)*, pages 2453–2470.

[162] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42, 1992.

[163] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 619–634, 2016.

[164] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. Pumm: Preventing use-after-free using execution unit partitioning. In *32nd USENIX Security Symposium (USENIX Security '23)*, pages 823–840.

[165] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 499–510, 2013.

[166] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*, pages 42–54.

[167] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE '18)*, pages 327–337, 2018.

[168] Jiayi Ye, Chao Zhang, and Xinhui Han. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, pages 1529–1531.

[169] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *2015 Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[170] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 249–265.

[171] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security '16)*, pages 363–378.

[172] Michal Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, November 2017. Accessed: 2019-7-31.

[173] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security '19)*, pages 1205–1220.