NORTHWESTERN UNIVERSITY

Towards Understanding and Reducing Exploitability of Linux Kernel Bugs

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Zhenpeng Lin

EVANSTON, ILLINOIS

September 2023

© Copyright by Zhenpeng Lin 2023

All Rights Reserved

ABSTRACT

Towards Understanding and Reducing Exploitability of Linux Kernel Bugs

Zhenpeng Lin

The Operating System (OS) kernel is a key component of modern computing infrastructure, yet it is prone to numerous vulnerabilities, many of which cause memory corruptions that can be exploited by attackers to perform malicious activities. While various techniques have been introduced to secure the Linux kernel, it still constantly gets compromised.

CVE-2021-3715 is a kernel bug in the Linux system, which persisted for over six years and was initially fixed without significant attention due to its perceived low severity, leading to it remaining unaddressed in distro kernels for another year until further discoveries revealed its potential for use-after-free memory corruption and exploitation. In 2005, Microsoft introduced Data Execution Prevention (DEP) in Windows XP SP2 to prevent code execution on user memory by marking it as non-executable, but this approach was soon thwarted by Return Oriented Programming (ROP), a technique that allowed attackers to execute code by utilizing code gadgets within the execution region.

The above incidents underline the fact that a lack of understanding of kernel exploitability could compromise security. Therefore, I propose to conduct research on understanding kernel exploitability and then reduce exploitability. Kernel exploitation is a process of programming a ”weird machine”. Analyzing exploitability, which is finding a transition path from the entry point to the exploitation goal in the machine, is naturally challenging as the weird machine remains largely unknown. To address the challenge, I propose analyzing vulnerability capability and exploitation composability. In this dissertation, I develop techniques that combine static and dynamic analysis to explore kernel vulnerability capability, and introduce innovative exploitation techniques

capable of bypassing all existing defenses. With this understanding of exploitability, I then propose a mitigation strategy designed to prevent memory corruption, delivering effective protection with minimal overhead.

In the future, I aim to continuously investigate the composability of kernel exploitation. Due to its complexity, which precludes deduction through any general formal algorithm, I will progressively analyze this composability, piecing together its intricate portrait. Additionally, the current mitigation strategy in this dissertation operates within userspace. Its adaptation to the kernel space presents a unique set of challenges. My intention is to extend the solution into the kernel space, optimizing overhead through a combination of systematic methodologies and quantitative evaluations.

ACKNOWLEDGEMENTS

This journey of pursuing my Ph.D. has been an enriching and enlightening one, offering me not only academic insights but also life lessons. Honestly, pursuing a Ph.D. is not an easy task. I've felt struggled, confused, and frustrated. I even questioned myself – ”Is it right for me to do security research?”. I thought about quitting, but it was the help and encouragement from others that made me here. Writing this acknowledgment provides me with an opportunity to express my heartfelt gratitude to those who have been integral in making this journey possible.

First and all, I would like to thank my advisor, Dr. Xinyu Xing. Although we work remotely most of the time, his guidance, patience, and intellectual insights are never absent. His energy and passion inspired me, not only in research, but also in life. ”We should think about how to make the cake bigger, rather than how to get a bigger slice of it”, I still remember this advice he gave me at a discussion. Simple words, but very deep in its meaning. ”make the cake bigger” refers to increasing the total volume of the resource through increasing production, innovating, and optimizing resource allocation. ”cutting a larger piece of cake” relates more to obtaining a larger share of the existing resources. The words reveal the importance of inclusiveness and cooperation. His constant support and constructive criticism have pushed me to dig deeper and reach further in my quest for knowledge.

I would also like to express my gratitude to my advisor, Dr. Guojun Peng, and mentor Dr. Lei Zhao, who provided guidance during my time at Wuhan University. Their encouragement and support sparked the idea of pursuing a Ph.D. in the United States; and my mentors at ASU, including Dr. Adam Doupé, Dr. Yan Shoshitaishvili, Dr. Fish Wang, and Dr. Tiffany Bao, whose advice was instrumental in enlightening me on doing academic research; and mentor Dr. Kang Li at Certik, who is always supportive and helpful; and mentor Brad Spengler at grsecurity, who

enlightened me on designing secure protection for the Linux kernel.

To my collaborator at Pennsylvania State University, Northwestern University, Arizona State University, University of Minnesota, Certik, Baidu Research, and IBM, including Yueqi Chen, Dongliang Mu, Yuhang Wu, Dang K Le, Zheng Yu, Ziyi Guo, Yankai Jiang, Nick Wanninger, Simone Campanoni, Peter Dinda, Kyle Zeng, Kangjie Lu, Zhaofeng Chen, Dan Williams, Zhongshu Gu, and Hani Jamjoom, thank you for being a part of my academic journey. Our discussions, brainstorming sessions, and friendly debates have played a significant role in shaping my perspective and refining my ideas. Each one of you has contributed to making this journey a fulfilling experience, and for that, I am extremely grateful. Especially for Dr. Yueqi Chen, who worked closest with me at the lab. I personally learned a lot from his critical thinking. We are colleagues, roommates, and close friends, we spent the toughest time together during the pandemic, and we talked a lot about philosophy, which is not related to our research but benefits my career on how to be a good researcher.

Special thanks to Dr. Xiaoting Li, whose persistence and diligence are an inspiration to me. As someone who has wrestled with procrastination, I've learned a lot from you about the art of time management and the attitude of doing research, which led me toward the path of consistent diligence and rigorous attention to detail. Beyond this, your unwavering support and love have been my pillar of strength. Thank you for taking care of me and supporting me when times were tough. Your encouragement has been a constant motivation that propelled me toward this milestone.

To my mother, my father, and my sister, you have taught me the value of hard work and determination, and you have been my greatest supporters every step of the way. The lessons you have imparted and your constant encouragement have shaped me as a person and as a researcher.

To my friends, Wenbo Guo, Xian Wu, Qi Qin, Wenshu Mai, Jianpeng Li, Yuze Lu, He Song, and so many others, thank you for the countless conversations, the shared laughter, and the un-

wavering belief in my ability. You have reminded me that there is a life beyond the confines of this dissertation, and for that, I am profoundly grateful. Your friendship has indeed brightened my experience, and I am lucky to have each of you in my life.

Lastly, I would like to extend my gratitude to the members of the dissertation committee, Dr. Xinyu Xing, Dr. Peter Dinda, Dr. Xiao Wang, Dr. Adam Doupé, and Dr. Kang Li. I appreciate your invaluable insights, critical analyses, and thoughtful feedback that significantly improved the quality of my work. Your expertise and dedication to upholding the highest standards of academic scholarship are deeply appreciated.

To everyone mentioned and many others who have been part of this journey, this accomplishment is as much yours as it is mine. I am grateful beyond words. Thank you for your role in this academic endeavor and the moments we have shared together during this journey.

THESIS COMMITTEE

Xinyu Xing
Northwestern University
Committee Chair

Peter Dinda
Northwestern University
Committee Member

Xiao Wang
Northwestern University
Committee Member

Adam Doupé
Arizona State University
Committee Member

Kang Li
Certik
Committee Member

TABLE OF CONTENTS

Acknowledgments 4

List of Figures 14

List of Tables 16

Chapter 1: Introduction 19

Chapter 2: GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs 23

2.1 Introduction 23

2.2 Motivating Example 25

2.3 Design Rationale & Overview 28

2.4 Technical Details 30

2.4.1 Analysis of Reports and Taint Source Identification 31

2.4.2 Taint Propagation and Identification of Sink 34

2.4.3 Ranking of Kernel Structures 35

2.4.4 Object-Driven Kernel Fuzzing 39

2.5 Implementation 43

2.6 Evaluation 46

2.6.1 Experiment Setup & Design 46

2.6.2 Experiment Results 48

2.6.3 Security Implication 52

2.7 Related Work 55

2.8 Conclusion 58

Chapter 3: DirtyCred: Escalating Privilege in Linux Kernel 59

3.1 Introduction 59

3.2 Background & Threat Model 62

3.2.1 Credentials in Linux kernel 62

3.2.2 Linux Kernel Heap Memory Management 63

3.2.3 Threat Model 64

3.3 Technical Overview & Challenges 65

3.3.1 Overview 65

3.3.2 Technical Challenges 67

3.4 Pivoting Vulnerability Capability 69

3.4.1 Pivoting OOB & UAF Write 69

3.4.2 Pivoting Double Free 71

3.5 Time Window Expansion 73

3.5.1 Exploiting Userfaultfd & FUSE 73

3.5.2 Alternative Exploitation of Userfaultfd & FUSE in Later Kernel Versions . 76

3.5.3 Taking Advantage of Lock Mechanism in Filesystem 77

3.6 Allocating Privileged Credential 78

3.6.1 Initiating Allocation from Userspace 78

3.6.2 Initiating Allocation from Kernel Space 79

3.7 Evaluation 80

3.7.1 Experiment Design 81

3.7.2 Experimental Result 82

3.8 Countermeasure Against DirtyCred Attack 86

3.9 Discussion and Future Work 90

3.9.1 Escaping Containers 90

3.9.2 Android Rooting 91

3.9.3 Cross Version / Architecture Exploitation 92

3.9.4 Alternative Approaches for Capability Pivoting 92

3.9.5 Exploitation Stability 93

3.9.6 TOCTOU 93

3.10 Related Work 94

3.11 Conclusion 96

Chapter 4: CAMP: Compiler and Allocator-based Heap Memory Protection 98

4.1 Introduction 98

4.2 Background . . . 100

4.2.1 Corruption and Protection of Heap Memory . . . 100

4.2.2 Heap Memory Allocators . . . 102

4.3 Assumptions & Threat Model . . . 103

4.4 CAMP . . . 103

4.4.1 An Toy Vulnerable Program . . . 104

4.4.2 CAMP's Protection Mechanism . . . 104

4.4.3 Design Overview . . . 105

4.4.4 Instrumentation by the Compiler . . . 106

4.4.5 Runtime Support . . . 107

4.4.6 Compilation Optimization . . . 110

4.5 Implementation Details . . . 115

4.5.1 CAMP Compiler . . . 116

4.5.2 CAMP Memory Allocator . . . 117

4.6 Evaluation . . . 118

4.6.1 Security Evaluation . . . 120

4.6.2 Performance Evaluation . . . 123

4.7 Discussion . . . 130

4.8 Related Work . . . 131

4.9 Conclusion . . . 133

Chapter 5: Conclusion and Future Work 134

5.1 Conclusion 134

5.2 Future research 135

References 152

Appendix A: Appendix 153

A.1 Additional Details of GREBE's Evaluation 153

A.1.1 Detail of User Study 153

A.1.2 Procedure of Error Triaging 154

A.1.3 Detail of Distance Measurement & Hypothesis Validation 157

A.2 Identifying Credential Object 158

A.2.1 Design 160

A.2.2 Implementation 161

A.3 CAMP's Performance on SPEC CPU2006 162

LIST OF FIGURES

2.1 The workflow of GREBE is as follows. (a) Utilizing a kernel error trace extracted from a crash report, GREBE carries out a backward taint analysis and identifies all the kernel objects associated with the crash or panic. (b) GREBE then refines this list based on the rarity of the objects, pinpointing those that are critical to the kernel error. (c) Leveraging the objects filtered in the preceding step, GREBE modifies the kernel and designates the (de)allocation and dereference points of these critical objects as anchor sites. (d) GREBE then customizes Syzkaller to utilize the feedback from the reachability of these anchor sites to choose seeds. In addition, GREBE incorporates a specialized mechanism to alter seeds, thus allowing GREBE to vary the methods of triggering the same kernel bug. 29

2.2 An illustrative example showcasing a dominator tree, highlighting two distinct approaches to logging kernel errors. Line 7 represents a logging statement responsible for recording kernel errors, while line 15 acts as a wrapper for the logging statement at line 16. The variable ”conv” in line 1 is the identified taint source according to our proposed approach. It’s important to note that for the sake of simplicity, we have placed the two error logging functions in separate branches that share the same conditional jump block. However, it should be acknowledged that in real-world scenarios, error logging does not typically occur in this manner. . . . 32

3.1 The overview of exploiting CVE-2021-4154, the write operation to the opened file starts between step 1 and step 2 and finishes after step 3. 65

3.2 The memory layout before and after converting a heap overflow capability into the ability to deallocate a credential object. 68

3.3 The step-by-step example demonstrating converting a double-free capability into the ability to deallocate a credential object. 70

4.1 The design overview of CAMP. 106

4.2 Evaluation result of CAMP breakdown on SPEC CPU2017. From left to right, the bars show the normalized time of tcmalloc replacement, CAMP, CAMP with each optimization disabled, and CAMP without optimization. 124

A.1 Sampled questions from the exploitability survey form [191] 155

LIST OF TABLES

2.1 The example code snippets extracted from the PoC programs in two different kernel bug reports – 7022420 [18] and 692a8c2 [19]. 41

2.2 The performance of Syzkaller, Syzkaller variant, GREBE and GREBE without mutation optimization under some sampled kernel bugs. The "SYZ ID" column is the case ID. The "Critical Structures Identified" means the structures that are identified by the static analysis tools then are utilized by GREBE. The "Initial Error Behavior" column indicates the error behavior manifested in the corresponding bug report. The "Discovered New Error Behaviors" column is the error behaviors newly discovered. Note that, for each case, we sample only some of its newly identified error behaviors for illustration purposes. For more complete performance information across all 60 selected kernel bugs, the readers could find at [33]. In the "Time" column, T1 represents the number of hours Syzkaller took, T2 is for Syzkaller's variant, T3 is for GREBE without optimization, and T4 stands for GREBE. The dash "-" means the corresponding error behavior is not discovered by the corresponding tool. 49

2.3 The summary of exploitation potential improvement. In the column of "Exploitability Change", LL means the original error behavior is less likely to be exploitable. The letter L means the newly discovered error behaviors are likely to be exploitable. The number in the parenthesis represents the amount of newly identified error behaviors tied to probably exploitable. The star $\star$ denotes the bugs for which we have developed exploits based on the newly discovered error behaviors and their provided primitives. 52

2.4 The summary of the types of error behaviors in bug reports and their corresponding exploitation potential. 53

3.1 Exploitable objects identified in the Linux kernel. Note that the symbol ⋆ indicates an object tied to “file” credential whereas the symbol † represents an object associated with “cred” object. The column “Memory Cache” specifies the caches storing kernel objects. The column “Structure” represents the exploitable objects’ types. The column “Offset” describes where the credential object’s reference is located in the exploitable object. 83

3.2 Exploitability demonstrated on real-world vulnerabilities. Note that some CVEs provide both use-after-free and double-free capabilities. Here, we categorize such vulnerabilities into double-free and mark them with a ⋆ symbol. Note that the symbol † indicates the vulnerabilities that could corrupt only data in virtual memory area. 84

3.3 The performance evaluation results of the proposed defense on two different benchmarks – Phoronix and LMBench. 88

4.1 Security evaluation of CAMP on Juliet Test Suite. 118

4.2 The security evaluation results of CAMP and related tools on real-world vulnerabilities. ✔ represents that the corresponding tool successfully detected the memory corruption in the vulnerability. ✖ indicates the tool failed to detect the memory corruption that happened. ”/” represents the tool does not support protecting the corresponding type of vulnerability. ”Run Well” means the application runs well without causing any memory corruption with the PoC input. ”Run Fail” represents that the tool failed to run due to compatibility issues. ”Build Fail” means the tool failed to compile the targeted application to enforce protection. 119

4.3 The relative time and memory overhead of CAMP, ASAN --, ASAN, ESAN, and Memcheck on SPEC CPU2017. ”-” indicates the tool failed to run the corresponding benchmark. 123

4.4 The relative time and memory overhead of CAMP, LowFat, Delta Pointer, DangNull, FreeGuard, MarkUs, and FFmalloc on SPEC CPU2006 and SPEC CPU2017. 124

4.5 Time Overhead on mimalloc-bench. Native represents using the default allocator – ptmalloc, CAMP means using its customized seglist allocator based on tcmalloc. 126

4.6 CAMP and ASAN's output and latency evaluation results on Nginx. In the Latency column, the "Average" represents the average latency of the requested connection, the others show the latency distribution. 128

4.7 CAMP's performance evaluation results on the Chromium browser. In the Benchmark column, kraken, sunspider and Lite Brite are three browser benchmarks, whereas the following are websites used to measure the loading time of the browser. 129

A.1 This table shows false negative analysis results. The "SYZ ID" is the case ID, with the second column showing the basic block count between the crash and root cause site. A following $\star$ indicates the sites are from different syscalls; otherwise, the same. The third and fourth columns indicate new behaviors found manually and via GREBE, respectively. A "-" signifies no behavior found. 159

A.2 Time overhead of CAMP, ASAN--, ASAN, ESAN, Softbound+CETS, MemCheck on the SPEC CPU2006. "-" means the case where the tool failed to run the benchmark. 163

A.3 Memory overhead of CAMP, ASAN--, ASAN, ESAN, Softbound+CETS, MemCheck on the SPEC CPU2006. "-" means the case where the tool failed to run the benchmark. 164

CHAPTER 1
INTRODUCTION

The Operating System (OS) kernel forms the critical backbone of contemporary computing infrastructure. However, despite substantial efforts to secure it, OS kernels inevitably contain vulnerabilities, often leading to memory corruption and then exploitation. Exploiting these vulnerabilities allows attackers to engage in malicious activities. Given the high privilege level at which OS kernels operate, a successful breach can allow attackers access to sensitive data or even full control over the system. Thus, OS kernels have consistently remained attractive targets for attackers.

To improve kernel security, it is essential to have a comprehensive understanding of kernel exploitability. CVE-2021-3715 [1] is an exploitable kernel bug that had persisted in the Linux kernel for over six years. This bug was originally reported by Syzkaller [2] and fixed by the Linux community [3]. Due to the low severity of the error behavior shown (a warning message), the bug was inconspicuously fixed, without notification for downstream vendors and attention concerning exploitation. As a result, it remained unfixed in the distro kernels (such as Centos) for an additional year. Subsequent discovery [4] showed that this bug could indeed result in a use-after-free memory corruption that would immediately lead to exploitation. The bug was then fixed in the distro's kernel. This incident showcases the importance and complexity of identifying the capability of kernel vulnerability, which is one of the clues to understanding kernel exploitability.

In 2005, Microsoft implemented Data Execution Prevention (DEP) [5] in Windows XP SP 2, purporting that it could prevent code execution on user memory. DEP operates by marking memory as non-executable, thereby preventing shellcode execution. However, this technique was quickly undermined by Return Oriented Programming (ROP) **shacham2007geometr**, wherein attackers

could utilize code gadgets within the execution region to assemble desired shellcodes for execution. This instance illuminates the crucial aspect of analyzing the composability [6], [7] of the weird machine [6], [8]. DEP seemed to be a perfect solution to shellcode execution until people found ROP attacks. Without analyzing and knowing the composability of the weird machine, a mitigation measure may be swiftly circumvented using a novel exploitation pathway.

The above incidents suggest that analyzing vulnerability capability and exploitation composability is the key to understanding kernel exploitability. Kernel exploitation is a process of programming a weird machine. Analyzing exploitability, which is finding a transition path from the entry point to the exploitation goal, is naturally challenging as the weird machine remains largely unknown. To address this challenge, I proposed to **analyze the capability of kernel vulnerabilities** and **analyze the composability of kernel exploitation**. Analyzing vulnerability capability indeed is to find the entry point of the weird machine, whereas analyzing exploitation composability is finding the weird states that could be chained toward the exploitation goal in the machine. However, composability does not conform to a single, universally applicable definition. Its complexity and variability under diverse conditions and for different targets preclude a one-size-fits-all approach to its analysis and study. Therefore, I propose to examine the composability of kernel exploitation progressively to assemble the whole picture of it. With this understanding of exploitability, I then propose an effective countermeasure to **reduce the exploitability of kernel vulnerability**, thereby enhancing overall kernel security. The rest of this dissertation is organized as follows.

Chapter 1 presents `GREBE`, a novel kernel fuzzing methodology designed to unearth the potential range of error behaviors that a kernel bug may manifest. Our approach deviates from traditional kernel fuzzing methods, which primarily focus on kernel code coverage. Instead, we place greater emphasis on the erroneous code fragments. We pioneer an object-driven kernel fuzzing technique

that delves into various contexts and paths to provoke the reported bug, thereby causing it to exhibit a range of error behaviors. With the unveiling of these newfound errors, security researchers are equipped with the tools necessary to infer a bug's potential for exploitation more effectively. To quantify the performance, efficiency, and impact of the proposed methodology, we applied `GREBE` to 60 real-world Linux kernel bugs. On average, `GREBE` was able to elicit more than two additional error behaviors for each kernel bug. Furthermore, `GREBE` revealed increased exploitability potential in 26 of the kernel bugs.

Chapter 1 appeared in the *Procceedings of 43rd IEEE Symposium on Security and Privacy (S&P 2022)* [9]

Chapter 2 presents `DirtyCred`, a novel exploitation technique that takes a new exploitation transition path to the attackers' goal. More specifically, given a Linux kernel vulnerability, DirtyCred swaps unprivileged and privileged kernel credentials, thereby enhancing the vulnerability to a DirtyPipe-equivalent level of exploitability. This enhanced exploitability could empower a malicious actor with the ability to escalate privileges or even escape a container environment. The assessment of this exploitation technique on 24 real-world kernel vulnerabilities within a fully-protected Linux system reveals that DirtyCred successfully exhibited exploitability on 16 of these vulnerabilities. This outcome reveals the serious security implications of DirtyCred. Upon assessing exploitability, we also propose a new kernel defense mechanism. The proposed method isolates kernel credential objects into distinct, non-overlapping memory regions according to their individual privilege levels. experimental results indicate that this defense strategy imposes only a negligible overhead.

Chapter 2 appeared in *the Proceedings of 29th ACM SIGSAC Conference on Computer and Communications Security (CCS 2022)* [10]

Chapter 3 presents `CAMP`, a novel protection scheme designed for preventing heap memory

corruption by using a combination of a compiler and a customized memory allocator. The compiler embeds boundary-checking and escape-tracking instructions within the target program, while the allocator manages memory ranges, coordinates with the instrumentation, and nullifies dangling pointers. Thanks to its innovative error detection scheme, `CAMP` can employ various compiler optimization strategies. These strategies eliminate redundant instrumentation, thus reducing runtime overhead without compromising security guarantees. The performance evaluation of `CAMP`, compared to existing tools using both real-world applications and SPEC CPU benchmarks, demonstrates its superior ability to protect heap memory with lower runtime overhead.

Chapter 3 is conditionally accepted to *the Proceedings of 32nd USENIX Security Symposium (USENIX Security 2023)*

Chapter 4 concludes the dissertation and discusses future research directions.

CHAPTER 2
GREBE: UNVEILING EXPLOITATION POTENTIAL FOR LINUX KERNEL BUGS

2.1 Introduction

Linux is utilized today across a broad spectrum of computing systems. To bolster its security, kernel fuzzing techniques and assorted debugging/sanitization features have been implemented by researchers and analysts. These tools aid security researchers and kernel developers in locating bugs within the Linux kernel. However, it remains a difficult task to discern if the conditions of a bug equate to a security vulnerability. For instance, whether bugs displaying warning error behaviors would lead to kernel exploitation. Previous research thus suggests that the presented error behaviors of bugs are crucial in prioritizing exploit development efforts.

In a practical scenario, when current fuzzing tools detect a kernel bug, the displayed error behavior might just be one of its many possible error behaviors. Other potential error behaviors may significantly differ from the initially exposed one. For example, as we will detail in Section 2.2, triggering a kernel bug following various paths or execution contexts can cause it to demonstrate not only a less likely to exploit General Protection Fault (GPF) error behavior but also a highly likely to exploit Use-After-Free (UAF) error behavior. Therefore, relying solely on a single manifested error behavior to infer a bug's potential exploitability could be misleading.

To tackle this issue, a natural response might be to use a kernel bug report as an input, analyze the root cause of the bug, and deduce all potential consequences that the bug could lead to (e.g., out-of-bound access, null pointer dereference, memory leak, etc.). However, diagnosing the root cause is often time-consuming and labor-intensive. Therefore, we suggest a more practical strategy

is to reveal as many post-triggered error behaviors of a specific kernel bug as possible without conducting root cause analysis. From the exposed error behaviors, security analysts can more accurately infer its potential exploitability.

To implement the above idea, we can utilize kernel fuzzing techniques. But existing kernel fuzzing methods mainly aim to maximize code coverage (for instance, Syzkaller [11], KAFL [12] and Trinity [13]). Applying these methods to our task leads to inefficiency and ineffectiveness since they are not specifically designed or optimized to find various paths or contexts relevant to the same buggy code fragment. Hence, we propose a new kernel fuzzing method that focuses on the defective code areas and diversifies the kernel execution paths and contexts towards the target buggy code fragment.

From a technical standpoint, our suggested kernel fuzzing method could be seen as a directed fuzzing approach. It initially takes a kernel bug report as an input and extracts the kernel structures/objects linked with the reported kernel error. Following that, the fuzzing method performs fuzzing tests and uses the hits to the identified kernel structures/objects as feedback to the fuzzer. As these identified kernel structures/objects are crucial to trigger the reported bug successfully, guiding fuzzing using them can concentrate the scope of the kernel fuzzer on the paths and contexts related to the reported bug, thus identifying the other error behaviors of the bug. We have embodied this approach in a kernel-object-driven fuzzing tool called `GREBE`, an acronym for ”fuzzin**G** fo**R** multipl**E** **B**ehavior **E**xploration”.

Applying `GREBE` to explore error behaviors for 60 kernel bug reports demonstrated over 2 distinct error behaviors per bug report on average. In our experiment, we also noticed that the newly identified error behaviors for many kernel bugs (26 out of 60) typically exhibited a higher exploitation potential than those in the original bug report. Furthermore, we uncovered 6 kernel bugs that initially seemed unexploitable could be transformed into exploitable ones through the

newly identified paths and contexts. These bugs had previously shown no exploitability. We have communicated these findings to some kernel vendors who haven't yet applied the available patches in their products, prompting immediate adoption of the patches.

To the best of our understanding, this is the first work that uncovers multiple error behaviors of a bug for exploitability exploration. Demonstrating multiple error behaviors could potentially hasten the rectification and eradication of highly exploitable bugs from the kernel. Besides, it could assist security analysts in transforming an unexploitable primitive into an exploitable one. Lastly, exposing a bug with multiple error behaviors could also potentially facilitate the bug's root cause diagnosis [14].

To summarize, this paper makes the following contributions:

- We've developed a novel technical methodology that leverages specifically chosen kernel objects to guide kernel fuzzing, thereby unveiling a bug's multiple error behaviors.
- Based on our design, we've enhanced Syzkaller, implemented `GREBE`, and shown its effectiveness in identifying multiple error behaviors for 60 unique real-world Linux kernel bugs.
- We've demonstrated that for a kernel bug exhibiting a low probability of exploitation, our proposed technique can discover its other error behaviors that indicate a significantly higher exploitability.

2.2 Motivating Example

A kernel error report suggests the potential exploitability of the related bug. As previously stated, the error manifested depends on how the bug is activated. Therefore, using a single bug report (displaying one error behavior) might lead to an underestimation of the exploitability of that bug. In Listing 1, we provide a specific example to clarify this point.

```
1  static void tun_attach(struct tun_struct *tun, ...)
2  {
3      if (tun->flags & IFF_NAPI) {
4          // initialize a timer
5          hrtimer_init(&napi->timer, CLOCK_MONOTONIC,
6              HRTIMER_MODE_REL_PINNED);
7          // link current napi to the device's napi list
8          list_add(&napi->dev_list, &dev->napi_list);
9      }
10 }
11
12 static void tun_detach(struct tun_file *tfile, ...)
13 {
14     struct tun_struct *tun = rtnl_dereference(tfile->tun);
15     if (tun->flags & IFF_NAPI) {
16         // GPF happens if timer is uninitialized
17         hrtimer_cancel(&tfile->napi->timer);
18         // remove the current napi from the list
19         netif_napi_del(&tfile->napi);
20     }
21     destroy(tfile); // free napi
22 }
23
24 void free_netdev(struct net_device *dev) {
25     list_for_each_entry_safe(p, n,
26                 &dev->napi_list, dev_list)
27         netif_napi_del(p); // use-after-free
28 }
```

Listing 1: The Linux kernel code snippet contains a bug. When activated with varied sequences of system calls and arguments, the bug exhibits two distinct error behaviors - a general protection fault error and a use-after-free error.

In the list provided, the function `tun_attach` configures the network interface. The argument `tun` points to a global variable used by all open tun files. As shown in line 3, if `IFF_NAPI` is set in `tun->flags`, the kernel initiates a timer and attaches the associated `napi` to the network device's `napi_list`. Another function, `tun_detach`, seen in line 12, cleans up the data contained in `tun_file` and closes the file. If `IFF_NAPI` is set, the kernel will cancel the timer and remove the `napi` from the device's `napi_list`. The function `free_netdev` will, as shown in line 24, navigate through the `napi_list` to eliminate `napi` from the list.

The kernel bug emerges from a possible inconsistency between the flag `tun->flags` in `tun_attach` ↪ and `tun_detach`. For instance, the kernel bug report [15] created by Syzkaller exemplifies this. The proof-of-concept (PoC) program attached to the report demonstrates a system call invoking `tun_attach` with `IFF_NAPI` not set. Consequently, the kernel neither starts the timer nor adds the related `napi` to the list. After this setup, the PoC program uses the system call `ioctl` to set `IFF_NAPI` ↪ in `tun->flags` before calling `tun_detach`, leading to inconsistent flags between `tun_attach` and `tun_detach`. Then, in line 17, the kernel tries to stop the timer, dereferencing a pointer enclosed in the timer object in `tun_detach`. However, as stated earlier, the timer is not initiated in `tun_attach` ↪ , causing a general protection fault. This fault signifies an attempt to access storage that isn't allocated for use. Therefore, many security analysts may infer from this single observation that the bug is likely unexploitable.

On closer inspection of this bug, however, we realize that by adjusting the PoC program, thereby altering how the shared variable is assigned an inconsistent value, we can trigger a use-after-free error in the kernel. More specifically, we can set `tun->flags` with `IFF_NAPI` before invoking the function `tun_attach`. Following this, after `tun_attach` is executed, it can add the associated `tun_file` to the device list `napi_list`. After this setup, we can then call `ioctl` to clear `tun_flags` and then invoke `tun_detach`. As shown in Listing 1, the function `tun_detach` does not remove the corresponding `napi` from the list in lines 18 to 19, but frees it in line 21. Therefore, when the device list is traversed, the KASAN-instrumented kernel will throw a use-after-free error. Compared to the error in report [15], instead of accessing an invalid kernel memory address causing a general protection fault, this unpermitted access is tied to a valid kernel memory address and ultimately corrupts the kernel memory. Thus, based on this use-after-free error, many analysts might consider the bug likely exploitable.

2.3 Design Rationale & Overview

Considering a kernel bug report that showcases a specific error behavior, the natural inclination to discover other potential error behaviors is to apply the concept of directed fuzzing. This approach explores paths leading to an area of interest within the program. The expectation is that, via some of the newly detected paths to the problematic code segment, the bug detailed in the report could be reactivated, revealing new error behaviors. However, this method is often ineffective.

Firstly, to employ directed fuzzing in unveiling multiple error behaviors, it is necessary to identify the problematic code segment (i.e., the root cause of the error). This must then be treated as the point of interest and supplied to the directed fuzzer. The difficulty lies in correctly and automatically pinpointing the root cause of the kernel bug. Mistakenly identifying a non-root-cause site as the area of interest for the fuzzer could result in failure to trigger the bug, not to mention the inability to discover multiple error behaviors.

Secondly, being able to identify the root cause of the kernel bug and repeatedly reaching the buggy code with a directed fuzzer does not guarantee that the kernel will manifest multiple error behaviors. Aside from traversing different paths to the problematic code, the exhibition of error behaviors is also contingent upon the context post bug triggering. As an example, to trigger the bug and demonstrate different errors, not only does a particular path to the buggy code snippet need to be followed, but a separate kernel thread also needs to alter a global variable, thus diversifying the required contexts.

Existing kernel fuzzing techniques are better equipped to address these two challenges compared to the limitations of directed fuzzing. Kernel fuzzers like Syzkaller do not require the input of a bug's root cause. They simply vary the sequences of system calls and their arguments, testing kernel code via different paths comprehensively. Furthermore, it introduces new system calls to

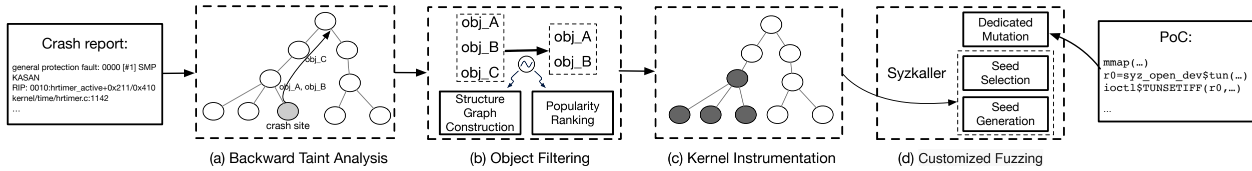

Figure 2.1: The workflow of GREBE is as follows. (a) Utilizing a kernel error trace extracted from a crash report, GREBE carries out a backward taint analysis and identifies all the kernel objects associated with the crash or panic. (b) GREBE then refines this list based on the rarity of the objects, pinpointing those that are critical to the kernel error. (c) Leveraging the objects filtered in the preceding step, GREBE modifies the kernel and designates the (de)allocation and dereference points of these critical objects as anchor sites. (d) GREBE then customizes Syzkaller to utilize the feedback from the reachability of these anchor sites to choose seeds. In addition, GREBE incorporates a specialized mechanism to alter seeds, thus allowing GREBE to vary the methods of triggering the same kernel bug.

vary execution contexts. These characteristics help fill the gaps left by directed fuzzing. However, as we will demonstrate in Section 2.6, this method faces substantial efficiency issues and poor effectiveness.

The design principle behind current kernel fuzzing techniques is to optimize kernel code coverage, avoiding the execution of already explored code paths. However, to activate the same bug and explore other potential error behaviors, the fuzzer must execute the same problematic code snippets repeatedly and expect the kernel to encounter the same buggy site in a different context. As we will reveal in Section 2.6, the code-coverage-based kernel fuzzing method (like Syzkaller) offers little assistance in identifying multiple error behaviors of a single kernel bug.

In this work, we address this issue by enhancing an existing kernel fuzzing approach with kernel-object guidance. Our observations of numerous kernel bugs reveal that the root cause of a kernel bug often arises from two practices: inappropriate usage of a kernel object, leading to a kernel error, or an incorrect value in computation with a kernel object, causing a critical kernel operation and forcing the kernel to demonstrate an error. Guided by the objects related to the error

specified in the bug report, we can divert the kernel fuzzer from paths and contexts irrelevant to the bug, substantially improving its efficiency.

To implement the concept outlined above, we design a multi-step procedure that merges static analysis and kernel fuzzing techniques. As is depicted in Figure 4.1, We begin by inputting a kernel bug report, running the enclosed PoC program, and identifying those kernel structures involved in the kernel errors. The objects within these types hint at potential objects under inappropriate usage or involved in computation with incorrect values. As such, we further examine the kernel source code and pinpoint the statements that operate the objects of these types, which we deem critical to the successful triggering of the kernel bug. Consequently, we instrument these statements to collect the feedback of object coverage during kernel fuzzing and adjust the corresponding PoC program. Our kernel fuzzing mechanism inputs the original PoC program attached to the bug report and uses a novel mutation and seed generation method to vary the PoC, improving the efficiency and effectiveness of exploring a bug's multiple error behaviors. In the following section, we will discuss these techniques in greater detail.

2.4 Technical Details

In this section, we delve into the specifics of our object-oriented kernel fuzzing strategy. Firstly, we explain the process of dissecting a kernel bug report and pinpointing vital structures (i.e., those involved in the corresponding kernel error). Secondly, we explore how to sift through kernel structures in order to further enhance kernel fuzzing efficiency while exploring error behavior. Lastly, we discuss the application of these identified structures in the design of our object-oriented kernel fuzzing mechanism.

In this work, we apply backward taint analysis to pinpoint critical kernel structures (i.e., those involved in the error detailed in the provided bug report). This section describes our method for

```
1 // in drivers/vhost/vhost.c
2 void vhost_dev_cleanup(struct vhost_dev *dev)
3 {
4     WARN_ON(!list_empty(&dev->work_list));
5     if (dev->worker) {
6         kthread_stop(dev->worker);
7         dev->worker = NULL;
8         dev->kcov_handle = 0;
9     }
10 }
11 // in include/asm-genric/bug.h
12 #define WARN_ON(condition)
   ↪  ({                                              \
13         int __ret_warn_on = !!(condition);                             \
14         if
   ↪  (unlikely(__ret_warn_on))                                     \
15                 __WARN();                                             \
16         unlikely(__ret_warn_on);                                     \
17 })
```

Listing 2: A piece of Linux kernel code that contains a bug. When the bug is activated through various system call sequences and parameters, it manifests in two distinct error behaviors - a general protection fault error and a use-after-free error.

identifying the source and sink and how we subsequently perform backward taint analysis.

2.4.1 Analysis of Reports and Taint Source Identification

The Linux kernel incorporates a range of debugging functionalities implemented in various ways (such as `BUG`, `WARN`, and `KASAN`). However, most follow a common pattern of enforcing checks during kernel execution and verifying whether pre-set conditions are met. If these conditions are not satisfied, the kernel enters an error state, logging essential information for debugging. Depending on the logged information, the kernel may either induce a panic or terminate the current process.

Consider the case presented in Listing 2 as an example. Here, the function `vhost_dev_cleanup()` cleans up the worker attached to the `vhost_dev` device. In line 4, the kernel checks the `work_list` ↪ . If the `WARN_ON` macro decides that the list is empty, the kernel proceeds with the cleanup task

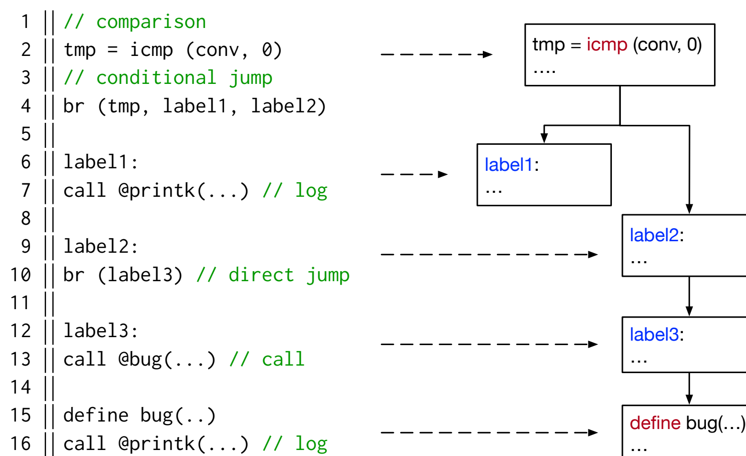

Figure 2.2: An illustrative example showcasing a dominator tree, highlighting two distinct approaches to logging kernel errors. Line 7 represents a logging statement responsible for recording kernel errors, while line 15 acts as a wrapper for the logging statement at line 16. The variable "conv" in line 1 is the identified taint source according to our proposed approach. It's important to note that for the sake of simplicity, we have placed the two error logging functions in separate branches that share the same conditional jump block. However, it should be acknowledged that in real-world scenarios, error logging does not typically occur in this manner.

at line 5. If not, the kernel will execute the code in `WARN_ON` macro and log the error. In this instance, an error is reported only if the pre-defined condition "`!list_empty(&dev->work_list)`" is true at runtime. Therefore, the variable `dev->work_list` in the condition implies a cause of the bug and should serve as the starting point of our analysis (i.e., the taint source of our backward analysis). In this example, the kernel developers have explicitly formulated the pre-defined condition as an expression and passed it to the macro `WARN_ON` for error handling. However, some other debugging features are instrumented by a compiler or completed by hardware, rather than being explicitly coded by kernel developers. For these features, the condition is implicitly formulated and cannot be identified from the kernel source code. We outline below how we handle different debugging features and their error logging mechanisms to identify the taint source.

Explicit Checking. As seen in the example above, kernel developers often formulate checks ex-

```
1 // source code
2 walk->offset = sg->offset;
3
4 // pseudo binary code after instrumentation
5 kasan_check_read(&sg->offset, sizeof(var));
6 tmp = LOAD(&sg->offset, sizeof(var)); // first access
7 kasan_check_write(&walk->offset, sizeof(var));
8 STORE(tmp, &walk->offset); // second access
```

Listing 3: The code snippet that performs implicit checking.

plicitly as expressions and pass them to standard debugging features like `WARN_ON` and `BUG_ON`. These macros contain a patterned code block with a condition statement and a logging statement, which gets executed if the condition is met. Apart from this standardized error logging approach, developers can also create their own macros encapsulating a logging statement in a helper function (see line 15 and 16 in Figure 2.2).

To pinpoint the condition triggering the execution of the logging statement, and hence identify the taint source, we first trace back along the dominator tree until we find a dominator basic block, the last statement of which is a conditional jump (for instance, given the wrapped logging statement in line 16 in Figure 2.2, line 4 links to the dominator basic block). Secondly, we treat the corresponding comparison as the condition that triggers the execution of the error logging (for example, line 2 in Figure 2.2). Lastly, we extract the relevant variable from the condition as our taint source (such as `conv` in Figure 2.2).

Implicit Checking. Implicit checking refers to checks that are not part of the kernel source code but instead are instrumented by a compiler or completed by hardware. The Kernel Address Sanitizer (KASAN) exemplifies compiler-instrumented implicit checking. In KASAN, the compiler instruments each memory access allowing the kernel to verify the legality of the access. KASAN uses shadow memory to track memory status, and if the instrumented kernel touches a freed mem-

ory region, it generates a bug report indicating the instruction that triggers a use-after-free error. In the case of implicit checking done by interrupts (like general protection faults detected by MMU), the interrupt handling routine is responsible for logging the relevant instruction.

From bug reports generated by these debugging mechanisms, we can readily identify the instruction that performs the invalid memory access. Our next step is to identify the variable associated with that invalid memory access. However, the binary instruction included in the report doesn't contain type information. To address this issue, we use debugging information to map binary instructions to their corresponding statements in the source code. If the mapped source code is a simple statement with only one load or store, we directly conclude that this statement causes the illegal memory access and treat the operand variable as a taint source. However, if the identified instruction links to a compound statement involving multiple memory loads and stores (e.g., `walk->offset = sg->offset` as shown in Listing 3), we perform additional analysis. Specifically, we first examine the bug report to pinpoint the exact instruction that captures the kernel error. Then, we treat the memory access associated with the error-catching instruction as our taint source. To illustrate, consider the case shown in Listing 3. The bug report shows that the error is captured by the statement `kasan_check_read(&sg->offset, sizeof(var))`, which associates with `sg->offset`. Hence, we treat `sg->offset` in line 2 as the taint source.

2.4.2 Taint Propagation and Identification of Sink

Remember that the objective of backward taint analysis is to unearth crucial structures, that is, the structures implicated in the error detailed in the provided bug report. To achieve this, we extract the call trace from the bug report once again. Using this trace, we then construct its control flow graph and retroactively propagate the taint source on this graph.

Throughout the backward propagation, we employ a particular strategy for variable tainting. If

the tainted variable is a component of a nested structure or a union variable, we extend the taint to its parent structure variable and deem this parent structure as critical. This is because the nested structure or union variable forms part of the parent structure variable in memory. If a field of the nested structure or union variable holds an invalid value, it's likely a consequence of misuse of its parent structure variable.

When backward taint propagation encounters a loop, we also propagate the taint to the loop counter if the taint source was updated within the loop. An example of this is seen in some out-of-bound access errors where the loop counter is compromised, unexpectedly increased, and eventually utilized as an offset to access an invalid memory region. By extending the taint to the loop variable, we can include the corrupted variable, which may help us identify additional structure variables related to the corruption.

In this work, we conclude our backward taint process when one of the following conditions is met. Firstly, we end our taint analysis if the backward propagation reaches the definition of a tainted variable. Secondly, we terminate our taint propagation if it extends to the entry of a system call, an interrupt handler, or the function that initiates the scheduler of a work queue. This is because these points signify where the kernel debugging features begin to track kernel execution for subsequent debugging. It's worth noting that, while carrying out backward taint propagation, we also extend propagation to aliases of the tainted variable. In this work, we consider the structural types of all taint variables as potential critical structures to guide our kernel fuzzing.

2.4.3 Ranking of Kernel Structures

By interpreting a kernel bug report and conducting the aforementioned backward taint analysis, we can recognize all the kernel structures associated with the error detailed in the report. Nevertheless, as we will discuss later, employing the identified structures to guide kernel fuzzing and uncover

```
1 // definition of struct sk_buff
2 struct sk_buff {
3     union {
4         struct rb_node rbnode;
5     };
6     ...
7     struct skb_ext *extensions;
8 };
```

Listing 4: The code snippet indicating structure definition.

```
1 static inline void *__skb_push(struct sk_buff *skb, ...)
2 {
3     return skb->data;
4 }
5
6 int ip6_fraglist_init(...)
7 {
8     struct frag_hdr *fh;
9     // type casting from void* to struct frag_hdr*
10    fh = __skb_push(skb, sizeof(struct frag_hdr));
11 }
```

Listing 5: The code snippet indicating type casting.

other error behaviors of the bug, we may still encounter reduced efficiency and potentially subpar effectiveness. As a result, prior to using these structures and their related objects to steer our kernel fuzzing, we need to further narrow down the kernel structures for guiding kernel fuzzing.

Selection of Kernel Structure. To uphold the code quality, Linux kernel developers utilize a variety of design patterns [16]. These patterns offer a recommended practice and structure for managing data in an universally recognized manner. For instance, consider the double-linked list. The `struct list_head` structure can be incorporated anywhere in a data structure, and the `list_head` ↪ from numerous instances of that structure can be connected together. As a consequence, the kernel objects can be controlled by standard interfaces, such as `container_of` which retrieves access to the parent for a given child structure, and `list_add/del` which carries out list operations. The

`struct list_head` is extensively used in the entire kernel codebase. If we include such prevalent structures and the corresponding objects for guiding kernel fuzzing, the kernel fuzzer would inevitably traverse a large code space, diverting the fuzzer from its focus on the error-causing code referred to in the report. Therefore, to maintain the kernel fuzzer's efficiency in investigating a bug's multiple behaviors, we need to exclude these structures from our kernel fuzzing guidance.

In addition to the structures previously mentioned, Linux kernel developers also construct numerous other structures related to abstract interfaces. These interfaces are paired with implementation layers to support a multitude of devices and features. For instance, the kernel creates a `struct socket` for all networking services requested from userspace, regardless of the specified protocol. Such structures are also widespread, appearing in many kernel code locations across various kernel modules. As a result, similar to `struct list_head`, they too should be omitted from the subsequent kernel fuzzing.

The aforementioned structures are simply examples of common structures. To identify and exclude them for exploring multiple error behaviors, we design a systematic approach to rank the kernel structures based on their ubiquity. Broadly speaking, this method constructs a graph depicting the reference relationship between kernel structures. Each node in the graph symbolizes a kernel structure, and the directed edges between nodes indicate reference relationships. On the graph, we apply PageRank [17], which assigns a weight to each structure. In this work, we regard a structure with a higher weight as more common than others and exclude them while performing kernel fuzzing to investigate other error behaviors.

Construction of Structure Graph. To build the structure graph mentioned earlier, we initially analyze all the structures defined in the kernel source code. For a given structure, we examine all its field members. If the field is a pointer to another structure, we connect the given structure to the referenced structure. If the field is a nested structure or union, we expand them repeatedly

until we detect a self-referenced structure, or there are no more nested structures/unions in the definition. We connect the given structure directly to the structure in the final layer of expansion, ignoring the union in the middle to minimize graph size. For example, in Listing 4, `extensions` ↪ is a pointer referencing `struct skb_ext`. We link `struct sk_buff` to `struct skb_ext` in our graph. However, it should be noted that `struct rb_node` is a self-referenced structure in an anonymous union. Following the above method, we skip the anonymous union and only link `struct sk_buff` directly to `struct rb_node` without additional expansion.

Besides analyzing the structure definition in kernel source code, we also construct the structure graph considering type casting. Since the kernel supports polymorphism, which employs a single interface to represent different devices and features, one abstract data type can be cast to a more specific type. For example, consider the function `ip6_fraglist_init` in Listing 5. In this function, `skb->data` is cast from `void*` to `struct frag_hdr*`, which is further used in the IPV6 networking stack. The `void*` is an abstract data type, whereas the destination structural type `struct frag_hdr*` is more specialized. As a result, we add one more edge to our structure graph, linking `struct skb_buff` to `struct frag_hdr`.

Intuition suggests that the structures with more references are likely to be more common. They are more likely to be abstract data types. In addition, the structures referenced by popular structures can also be common since they may also be used in numerous program sites in the kernel. For these structures, they are too prevalent to enhance our kernel fuzzer's efficiency in better examining the error behavior of a kernel bug. To identify these kernel structures, we use the PageRank algorithm on the graph to rank their ubiquity. In this work, we only use those kernel structures and objects with lower ranks to guide our fuzzing process. In Section 2.5, we discuss how we select the page-rank score threshold to differentiate popular structures from less popular ones.

Technical Discussion. The process of eliminating popular kernel structures narrows the scope of

our kernel fuzzer, but intuition suggests it may also inadvertently limit our kernel fuzzer's ability to uncover other error behaviors associated with a specific kernel bug. On the one hand, if the popular structures that have been removed are the primary cause of the kernel bug, our fuzzer may not be able to access them, thus missing the chance to trigger the relevant bug. On the other hand, if the root causes of most kernel bugs are indeed popular structures, our proposed methodology might offer only limited assistance in identifying multiple error behaviors of a bug.

However, in this work, we posit that the aforementioned concerns are unlikely to arise in practice. First, based on our review of hundreds of real-world kernel bugs, we observed that most kernel bugs are primarily linked to less popular structures. As such, the elimination of popular structures does not impede the fuzzer's capacity to trigger the targeted bug. Second, even if the discarded popular structures are associated with the root cause of our target kernel bug, directing the fuzzer's focus towards less popular structures can still enable us to interact with some objects in the popular structural types. This is because less popular structures often comprise popular ones (e.g., the uncommon structure `struct napi_struct` contains the widely used structure `struct hrtimer` as shown in Listing 1). Concentrating on these less popular structures still offers opportunities to interact with popular structures, albeit through fewer instances of these structures. In Section 2.6, we present several instances where the root causes of the relevant bugs are tied to those removed popular kernel structures. We illustrate that our fuzzing methodology can still trigger the bugs of interest and investigate their other error behaviors, even in these cases.

2.4.4 Object-Driven Kernel Fuzzing

Having identified the key structures, we now elaborate on how we employ these structures to enhance kernel fuzzing and thereby explore multiple error behaviors for a single kernel bug.

Instrumentation. Traditional kernel fuzzing techniques implement tracing functions to monitor

executed basic blocks. In our approach, we preserve this capability and introduce an additional instrumentation component. This component, designed as a compiler plugin, examines each statement in the basic blocks and pinpoints those that involve the allocation, de-allocation, and usage of crucial objects (i.e., objects of the key structures type). Specifically, the instrumentation component incorporates a new tracing function that substitutes the highest 16 bits of the recorded basic block address with a unique identifier, distinguishing these basic blocks from others. With this enhanced instrumentation, we can easily locate which basic block associated with the key objects is under operation by the corresponding fuzzing program by observing the top 16 bits of addresses in the code coverage feedback.

Seed Selection. Thanks to the enhanced instrumentation, we can readily ascertain if a fuzzing program interacts with a key object during its execution. When we identify new key object coverage, we can include the associated fuzzing program in the corpus of our seed fuzzing programs. In our methodology, we incorporate the mutated seed program or the newly created seed program into the seed corpus if either of the following two conditions is met. Firstly, the program reaches an unseen basic block that involves key object operations. Secondly, at least one system call in the program extends the code coverage and the same system call has demonstrated key object operation in previous fuzzing. We include the second condition because it enables kernel fuzzing to amass kernel states, thereby enhancing the likelihood for future mutations to interact with unseen basic blocks that involve key objects.

Seed Generation & Mutation. In our methodology, we initiate the seed corpus with the Proof of Concept (PoC) program included in the bug report we are examining. Each time we generate a new seed fuzzing program, we construct the program using only the system calls already present in the seed corpus. This approach contrasts significantly with seed fuzzing program generation methods utilized in contemporary fuzzing techniques (e.g., Syzkaller), which produce a seed fuzzing pro-

```
1 r0 = openat(..., '/dev/dsp1\x00');
2 ioctl(r0, ...);
3 write(r0, ...);
4 read(r0, ...);
```

(a) 7022420

```
1 // initial PoC: max = -1
2 bpf$MAP_CREATE(..., 0xffffffffffffffff);
3 // exit triggers GFP
4 exit(0);
```

(b) 692a8c2

Table 2.1: The example code snippets extracted from the PoC programs in two different kernel bug reports – `7022420` [18] and `692a8c2` [19].

gram by employing the system calls enclosed in the corpus and introducing new system calls. Our design change is informed by the need to trigger key object access under different contexts or along different execution paths to explore multiple error behaviors of a kernel bug. Although arbitrarily incorporating new system calls into the new seed fuzzing program could expand the code coverage that the program can explore, it inevitably diverts the fuzzing program away from the key objects.

Intuition suggests that using the aforementioned seed generation technique alone is unlikely to explore a sufficient number of contexts and paths related to the key objects. Consequently, we adopt the mutation mechanism used in existing kernel fuzzing techniques (i.e., Syzkaller). This mutation mechanism introduces new system calls relevant to the system calls already present in the seed corpus into the seed fuzzing program. We anticipate that the fuzzing program can continue to engage with the key object while diversifying the execution contexts or the paths leading to the object.

Mutation Enhancement. Syzkaller, when performing a fuzzing program mutation, employs preset templates to guide the synthesis of new fuzzing programs. Each template specifies the dependency and argument format of related system calls. For instance, Syzkaller's template dictates that the system call `read` requires a resource (i.e., a file descriptor) as one of its arguments, which the system calls `openat` or `socket` can generate. Following this template, Syzkaller can mutate a fuzzing program by appending the system call `read` or `socket` to the system call `openat`. Template-guided

mutation guarantees the legitimacy of the seed program, thus preventing early kernel rejection of the fuzzing program.

As stated earlier, our mutation method adapts the approach used in Syzkaller. As we demonstrate in Section 2.6, although this strategy is useful for avoiding the creation of invalid kernel fuzzing programs, it can still be inefficient and occasionally ineffective in guiding our kernel fuzzer to expose multiple behaviors for a single kernel bug. The reasons for this are twofold.

Firstly, Syzkaller, when conducting a fuzzing program mutation, tries to introduce various system calls relevant to the seed program and randomly alter system calls' arguments. However, we note that both the resource and the arguments that system calls manipulate are vital for successfully triggering a target kernel bug. Mutation without considering these factors would inevitably lead to low effectiveness in exploring multiple error behaviors.

For instance, take the case displayed in Table 2.1a. The table presents a code snippet illustrating a PoC program that triggers a kernel bug [18]. Using this PoC as a seed program and performing mutation, Syzkaller inserts the system call `socket`, which is irrelevant to the bug. This modification inevitably involves a resource that cannot trigger the bug and directs the fuzzer into a vast code space. Consider the case shown in Table 2.1b as another example. Similar to the previous example, the table presents a portion of a PoC program triggering a different kernel bug [19]. In the mutation stage, Syzkaller alters the variable `@max=0xffffffffffffffff` because the template indicates that the legitimate value range for this variable is `[INT_MIN, INT_MAX]`. However, for this specific kernel bug, which triggers an integer overflow in the kernel, the bug triggering condition is `@max=-1` or in other words `@max=0xffffffffffffffff`. As a result, this argument's random mutation is unproductive, significantly affecting the triggering of the bug in different contexts.

To address these issues, we enhance our fuzzing approach by optimizing its mutation mechanism. Specifically, we categorize system call specification templates based on the resource type

that the corresponding system calls rely on (e.g., classifying system calls related to the network socket and device file separately). Within each category, we further divide the enclosed system calls into two subgroups: one for resource creation and the other for their utilization. With this categorization, our fuzzing component can either replace system calls with ones in the same category or insert system calls that associate with the resource illustrated in the seed program when mutating a seed program.

In addition to categorizing templates based on resources, our mutation mechanism also preserves the values for the arguments seen in the original PoC program if the types of these arguments do not fall into four categories—constant, pointer referencing a memory region, checksum, and resource (e.g., a file descriptor for an opened file or an established socket). For arguments of the constant type, they usually indicate the protocol under fuzzing testing (e.g., `AF_INET` and `AF_INET6` in the system call `socket()` indicating the establishment of the IPv4 and IPv6 socket, respectively). In the fuzzing test, we need to modify these arguments to switch protocols and thus change the contexts under which the bug can be triggered again with different error behaviors. For arguments of pointer types and belonging to resources, when the kernel fuzzing alters the context or path towards the buggy kernel code, the original PoC program's addresses might become illegal. Maintaining these addresses could lead to early termination of the fuzzing program. Regarding the checksum, if the value of the calculation source changes in the mutation process, the checksum should be updated accordingly. Keeping the same checksum value could also result in the fuzzing program's early termination during the data validation phase.

2.5 Implementation

Leveraging the LLVM infrastructure and the kernel fuzzing tool Syzkaller, we have realized our concept as a tool – `GREBE`. In the following sections, we'll shed light on some key aspects of our

implementation. The source code for `GREBE` can be accessed at [20].

Identifying Critical Structures. Our tool receives LLVM IR and a single bug report as input. To generate the bitcode files, we apply the approach used in previous studies [21], [22]. Specifically, we modify the LLVM compiler to dump bitcodes before initiating any compiler optimization passes. This step helps us prevent compiler optimization from impacting the precision of our analysis. Remember that we extract the call trace from the bug report. This call trace indicates the functions that were invoked but not yet returned when the kernel encountered errors. In this extracted call trace, the function that was last called might be the one instrumented by the compiler. However, it does not indicate the buggy function contributing to the error. Therefore, we overlook these functions in the call trace and commence our analysis from the statement that initiates the debugging feature.

`GREBE` utilizes backward taint analysis to identify critical structures, extracting the type information of structures using three instructions. The first instruction is `BitCast`, where the types before and after casting are specified. `GREBE` records the types derived from this instruction as critical structures. The second instruction is `Getelementor`, which contains a pointer referencing a kernel object and the corresponding type information of the object. Analyzing this instruction allows us to swiftly identify critical structures. The third instruction is `CallInst`. We deduce the type information from the prototype of the callee and record the structural type as critical structures. As stated previously, we consider system calls' entries, interrupt handlers, and workqueue processings as our taint sink. For this work, we manually annotate all these sinks based on their naming patterns.

Ranking Critical Structures. As outlined in Section 2.4.3, when creating the structure graph for ranking critical structures, we factor in typecasting. In our implementation, if the cast variable is the return value of a callee function, we inspect the callee from the return statement and then link the destination type with the structure field. For instance, consider the case exhibited in Listing 5.

The cast variable `skb->data` is the return value of the callee function `__skb_push`. By analyzing the callee function, we connect `struct frag_hdr` with `struct sk_buff`.

Remember that we also rank structures based on their page-rank scores, using a page-rank score threshold to exclude the popular ones. In this work, we determine this threshold using a standard univariate outlier detection method [23]. This technique calculates the mean and standard deviation of the page rank scores and then computes the Z-score for each structure. We use 3.5 more standard deviations as the threshold according to the outlier detection method. Since most kernel structures are less popular, possessing a significantly low z-score, this threshold can effectively differentiate popular kernel structures from others.

Kernel Fuzzing. As detailed in Section 2.4.4, we equip the kernel to gather the usage of critical objects in real-time. Given that Clang support has been introduced recently and might not support all Linux kernel versions, we perform instrumentation using a GCC plugin instead of a Clang pass. While performing a fuzzing program mutation, we adhere to the design of MoonShine-enhanced [24] Syzkaller, randomly mutating 33% of system calls and replacing them with others we have manually grouped.

In implementing the optimization mechanism that reuses arguments from the original PoC, for each system call in the PoC program, we initially find its specification in Syzkaller and analyze the definition of its structural arguments (i.e., `StructType` and `UnionType`). Then, we recursively inspect the structural arguments until no more new definitions can be discovered. Within each structural definition, we disregard `ConstType`, `VmaType`, `ResourceType`, and `CsumType` as they represent constant, pointer, resource description, and checksum, respectively. As discussed in Section 2.4.4, they are not likely to aid in exploring new paths to the buggy code.

2.6 Evaluation

In this section, we first quantify GREBE's effectiveness and efficiency and compare it with a code-coverage-based fuzzing method. Then, we demonstrate and discuss how well GREBE could unveil exploitation potential for real-world Linux kernel bugs.

2.6.1 Experiment Setup & Design

Syzbot is a bug reporting platform that well archives the kernel bugs identified by Syzkaller. To evaluate our tool – GREBE, we select kernel bugs and their reports from the platform as our test cases. While selecting these bugs, we follow two different strategies.

Our first strategy is a purely random selection process that follows two criteria. First, the bug report has to attach a PoC program so that we can reproduce the error specified in the report. Second, the reported kernel error cannot associate with Kernel Memory Sanitizer (KMSAN) because KMSAN is still under development and has not yet been merged into the Linux kernel mainline. By following these two criteria, we construct a test corpus containing 50 Linux kernel bugs.

Our second bug selection strategy is a process dedicated to different kernel versions. To be specific, we select bugs from 5 different Linux kernel versions (5.6 5.10)[1]. From each kernel version, we choose two recently-reported reproducible kernel bugs as our test cases. In this way, we construct another test corpus with 10 Linux kernel bugs. Combining with the kernel bugs in the first corpus, we obtain a dataset with 60 unique kernel bugs. To the best of our knowledge, our dataset is the largest used in the exploitability research.

For each bug in our dataset, we built the corresponding kernel in four QEMU virtual machines (VMs) by following the description of their bug reports. For the first two VMs, we ran our tool – GREBE and Syzkaller. For the remaining two VMs, we ran GREBE without enabling its mutation

[1] At the time of our experiment, 5.10 is the latest long-term support Linux kernel version.

optimization and Syzkaller with our mutation optimization (i.e., Syzkaller's variant). With this setup, we can evaluate GREBE's effectiveness and efficiency in different settings. Besides, we can compare it with the code-coverage-based kernel fuzzing method and its variant (i.e., Syzkaller with our mutation optimization). It should be noted that we use Syzkaller as our baseline approach for evaluation because it is one of open-sourced, code-coverage-based kernel fuzzing tools but mostly because it can test nearly all kinds of kernel components[2]. It should also be noted that we extend Syzkaller with our proposed mutation optimization for the following reason. GREBE is an extension of Syzkaller. It combines both the object-driven component and mutation optimization. With our mutation optimization integrated into Syzkaller alone, we could examine whether mutation optimization could become a sole driving force to enable multiple error behavior exploration.

Given a kernel bug of our selection, its report, and a kernel fuzzing tool under our evaluation, we include the PoC program enclosed in the report into the initial seed set and deploy our VMs on bare-metal AWS servers. Each of the servers has two-socket Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz (48 cores in total) and 192 GB RAM, running Ubuntu 18.04 LTS. For each VM, we configured it with two virtual CPU cores and 2GB RAM. While performing kernel fuzzing, we set each of the fuzzers to run for 7 days. To utilize the computation resource of the AWS server efficiently, we assign only 30 VMs for each server. In total, it takes us two months to gather the experiment results shown in this paper.

After 7 days of fuzz testing against various versions of the Linux kernels by using four different fuzzers, we formed a 6-member team under the guidance of an IRB approval (STUDY00008566). Among the 6 members, 2 are experienced security analysts regularly developing kernel exploits in the security industry. The other 4 members are academic researchers actively contributing to

[2]The State-of-the-art kernel fuzzing tools – HFL [25], SemFuzz [26] have not yet been publicly released. DIFUZE [27], KRACE [28], and Razzer [29] etc. are designed for fuzzing specific bug types or kernel modules. Previous research [25] shows Syzkaller has better performance than KAFL [12] and Trinity [13] in terms of code coverage. Therefore, we choose MoonShine-enhanced Syzkaller as our baseline.

the Linux community and frequently invited to give talks at the Linux Security Submit or other Linux-related conferences. In our evaluation, we asked this professional team to collect the fuzzing results (i.e., reports) from all VMs, group the reports based on their title uniqueness, and eventually preserve only the kernel reports truly tied to the 60 bugs of our selection. Note that a kernel fuzzer might trigger other kernel bugs and thus demonstrate errors. Since there have not yet been highly accurate crash triaging tools, the professional team inspects each of the kernel errors manually and preserves only the errors associated with our selected bugs. The procedure of manually triaging the kernel errors is described in Appendix A.1.2.

In addition to the manual effort above, we also asked our kernel professional team to thoroughly and manually inspect whether there are any other missing paths or contexts that could trigger the kernel bugs and thus exhibit different error behaviors. In this way, we can evaluate `GREBE`'s false negatives or, in other words, understand how complete `GREBE` could expose a bug's multiple error behaviors. It should be noted that the Linux kernel's codebase is huge and sophisticated. Given a kernel bug, it usually requires extensive manual efforts and significant expertise, spending hundreds of hours to perform through manual analysis for exploring all the possible errors. As a result, we evaluate the false negatives of `GREBE` by sampling 30% of the selected kernel bugs (18 out of 60 selected bugs).

2.6.2 Experiment Results

Effectiveness. Table 2.2 shows the sampled experiment results[3]. First, we can observe that our tool – `GREBE`– could demonstrate a significant advantage in finding a bug's multiple error behaviors. In comparison with Syzkaller and Syzkaller variant, which discover a total of 9 additional error behaviors for only 6 and 7 test cases within 7 days, `GREBE` identifies 132 new error behaviors for

[3] It should be noted that, due to the space limit, we place the complete experiment results at [33].

SYZ ID	Critical Structures Identified	Initial Error Behavior	Discovered New Error Behaviors	Time (in hours)			
				T1	T2	T3	T4
bdeea91[30]	aead_instance, crypto_aead, , crypto_spawn, pcrypt_instance_ctx crypto_aead_spawn, crypto_type	WARNING: refcount bug in crypto_mod_get	WARNING: refcount bug in crypto_destroy_tfm	6.69	2.62	0.06	1.25
			KASAN: use-after-free Read in crypto_alg_extsize	-	-	-	83.69
5d3cce3[15]	napi_struct, tun_file	general protection fault in hrtimer_active	KASAN: use-after-free Read in free_netdev	-	-	155.76	30.30
			KASAN: use-after-free Read in netif_napi_add	-	-	77.41	9.08
521a764[31]	ax25_address, nr_sock	WARNING: refcount bug in nr_insert_socket	KASAN: use-after-free Read in release_sock	-	-	0.03	4.39
			KASAN: use-after-free Read in nr_release	-	-	-	20.00
			KASAN: use-after-free Read in nr_insert_socket	-	-	-	0.06
			KASAN: use-after-free Write in nr_insert_socket	-	-	-	126.82
			KASAN: use-after-free Read in lock_sock_nested	-	-	-	18.20
229e0b7[32]	delayed_uprobe	general protection fault in delayed_uprobe_remove	KASAN: use-after-free Read in delayed_uprobe_remove	-	-	3.83	6.66
			KASAN: use-after-free Read in uprobe_mmap	-	-	12.69	4.10
			general protection fault in uprobe_mmap	-	-	-	89.49
			KASAN: use-after-free Read in update_ref_ctr	-	-	-	157.46

Table 2.2: The performance of Syzkaller, Syzkaller variant, GREBE and GREBE without mutation optimization under some sampled kernel bugs. The "SYZ ID" column is the case ID. The "Critical Structures Identified" means the structures that are identified by the static analysis tools then are utilized by GREBE. The "Initial Error Behavior" column indicates the error behavior manifested in the corresponding bug report. The "Discovered New Error Behaviors" column is the error behaviors newly discovered. Note that, for each case, we sample only some of its newly identified error behaviors for illustration purposes. For more complete performance information across all 60 selected kernel bugs, the readers could find at [33]. In the "Time" column, T1 represents the number of hours Syzkaller took, T2 is for Syzkaller's variant, T3 is for GREBE without optimization, and T4 stands for GREBE. The dash "-" means the corresponding error behavior is not discovered by the corresponding tool.

38 out of 60 test cases. These kernel error behaviors have not been seen in the bug reports that we gathered from Syzbot. Second, we can observe the mutation optimization greatly improves GREBE's utility. In 7 days of our experiment, GREBE without mutation optimization pinpoints 58 new error behaviors for 27 cases. This result significantly outperforms that of Syzkaller. However, without mutation optimization, GREBE experiences more than 50% of a downgrade in terms of the newly identified error behaviors (132 vs. 58) and about 30% of decrease in terms of the cases it could handle (38 vs. 27). Third, we discover that, while generally performing worse than GREBE, GREBE without enabling mutation optimization sometimes demonstrates better performance. For the test cases – #8eceaff, #3b7409f, and #d5222b3, GREBE without mutation optimization tracks down

4 additional error behaviors. We argue this does not imply the ineffectiveness of our mutation optimization method. Our manual inspection indicates the missing error behaviors primarily result from the nature of these bugs. Even if our mutation mechanism successfully constructs correct inputs to trigger the bug, making the bug manifest a different error behavior also relies upon a specific thread interleaving that mutation-optimization-disabled solution luckily discovers.

False Negatives. As is mentioned above, we also randomly selected 30% of test cases, performed manual analysis, and examined how complete GREBE could identify the error behaviors of a given kernel bug. In our experiment, the test cases used for our false negative study are listed in Table A.1 in Appendix. Our manual inspection shows that GREBE misses one error behavior for the cases #d1baeb1, #85fd017 and #695527b, and two error behaviors for the case #d5222b3. To understand the reasons behind these missing error behaviors, we first measure the number of basic blocks between the root cause of a kernel bug and its error panic site. We hypothesize that the false negative might relate to the distance between the root cause and the error site. However, as is shown in Table A.1, we did not find clear correlation between the distance and the effectiveness of GREBE. For more detail about the measurement and hypothesis validation, readers could refer to Appendix A.1.3.

With the rejected hypothesis in hand, we further took a look at false-negative cases closely, exploring the conditions of triggering the missing error behaviors. We found that, in addition to finding different paths and contexts by using GREBE, the exhibition of the missing behaviors also requires the manipulation of thread interleaving. For case like #85fd017, the manifestation of error behaviors depends on the layout of memory. The undiscovered error behavior occurs only if the memory in the overflowed region is unmapped. We do not attribute this to the incompetency of GREBE. Rather, we will leave the manipulation of thread interleaving and memory layout as part of our future research.

Impact of popular kernel structure removal. Recall that in Section 2.4.3, we rank the identified

critical structures based on their popularity and avoid using popular structures to guide our kernel fuzzing. Intuition suggests this might influence the effectiveness of our kernel fuzzing on finding a bug's multiple error behaviors. However, from the 60 kernel bugs of our selection, we observe there are only 3 out of 60 test cases (5%) the root cause of which ties to popular structures (`sk_buff` ↪ for `#d1baeb1`, `nlattr` for `#b36d7e4` and `#27ae1ae`). Even for these cases, GREBE still demonstrates its utility in finding the bugs' multiple error behaviors. These observations well align with our aforementioned arguments – ❶ the kernel bug generally roots in the inappropriate usage of less popular kernel structures, and ❷ focusing on less popular structures can still allow our fuzzer to reach out to popular structures because of the strong dependence between them. In Table 2.2, we list some kernel object types that GREBE uses for fuzzing guidance. For more complete kernel object types identified for each kernel bug, readers could find them at [33].

Efficiency. Table 2.2 and the table at [33] show the time that each fuzzer spent on finding a new kernel error behavior. First, we observe that both Syzkaller and its variant have comparable efficiency (21546 hours vs 21528 hours). However, GREBE without mutation optimization spends less time than Syzkaller on identifying the new error behavior (15011 vs. 21546 hours)[4]. After applying the mutation optimization, GREBE further reduces the time spent on new error behavior identification (5445 vs. 15011 hours). This discovery indicates mutation optimization alone provides minimum benefits to the improvement of fuzzing efficiency whereas object-driven component alone or the combination of both brings significant improvement in fuzzing efficiency.

Second, we observe that GREBE succeeds in disclosing 79 new error behaviors for 32 test cases within 24 hours. Take the case `#5d3cce3` in Table 2.2 as an example. GREBE found the use-after-free read error in `netif_napi_add` in 9 hours. On the contrary, GREBE without mutation optimization

[4] Since the new error behaviors discovered by Syzkaller and its variant is too few compared with the other fuzzers, we conservatively use 7 days (7×24=168 hours) to represent the non-discovered error behaviors when computing the time.

SYZ ID	Exploitability Change	SYZ ID	Exploitability Change
d1baeb1 [34]	LL → L (2) ⋆	de28cb0 [35]	LL → L (5)
8eceaff [36]	LL → L (2) ⋆	f56bbe6 [37]	LL → L (1)
bb7fa48 [38]	LL → L (1)	f0ec9a3 [39]	LL → L (1)
d767177 [40]	LL → L (2)	5d3cce3 [15]	LL → L (2) ⋆
460cc94 [41]	LL → L (1)	692a8c2 [19]	LL → L (12) ⋆
0df4c1a [42]	LL → L (3)	4cf5ee7 [43]	LL → L (2)
229e0b7 [32]	LL → L (3)	502c872 [44]	LL → L (1)
163388d [45]	LL → L (1)	b36d7e4 [46]	LL → L (1)
bdeea91 [30]	LL → L (1)	1fd1d44 [47]	LL → L (1)
b9b37a7 [48]	LL → L (4)	695527b [49]	LL → L (1)
0d93140 [50]	LL → L (1)	85fd017 [51]	LL → L (4) ⋆
b0e30ab [52]	LL → L (1)	6a03985 [53]	LL → L (3) ⋆
d5222b3 [54]	LL → L (1)	575a090 [55]	LL → L (1)
3a6c997 [56]	L → L (10)	27ae1ae [57]	L → L (1)
cbb2898 [58]	L → L (1)	4bf11aa [59]	L → L (1)
e4be308 [60]	L → L (11)	7022420 [18]	L → L (1)
3b7409f [61]	L → L (1)	ddaf58b [62]	L → L (2)

Table 2.3: The summary of exploitation potential improvement. In the column of ”Exploitability Change”, LL means the original error behavior is less likely to be exploitable. The letter L means the newly discovered error behaviors are likely to be exploitable. The number in the parenthesis represents the amount of newly identified error behaviors tied to probably exploitable. The star ⋆ denotes the bugs for which we have developed exploits based on the newly discovered error behaviors and their provided primitives.

spent more than 3 days. The original Syzkaller and its variant performed even worse, failing to find this error behavior within the 7-day time window. This result empirically shows that the design of object-driven fuzzing and mutation optimization in `GREBE`, to a large extent, can save the time and resources for the discovery of new error behaviors.

2.6.3 Security Implication

Exploitation Potential Exploration. Recall that we design `GREBE` to explore a kernel bug's multiple error behaviors. With the multiple manifested behaviors in hand, we expect some newly

Exploitation Potential	Kernel Bug Errors
Likely to exploit	KASAN (e.g., use-after-free, out-of-bound access, double-free)
Less likely to exploit	BUG, GPF, NULL ptr dereference, panic, WARN, wrappers (e.g., pr_err)

Table 2.4: The summary of the types of error behaviors in bug reports and their corresponding exploitation potential.

exposed error behaviors to indicate a higher exploitation potential for a kernel bug (e.g., finding an out-of-bound write error behavior for a kernel bug that originally manifests less-likely-to-exploit error behavior – null pointer dereference). As a result, we further evaluate GREBE's capability in exploitation potential exploration. To do this, we first recruited 20+ security researchers and conducted a user study (detailed in Appendix A.1.1) under the approved IRB (STUDY00008566). From the user-study results, we obtain the relationship between a manifested error behavior and the exploitation potential. As is depicted in Table 2.4, each error behavior is categorized into either "likely to exploit" or "less likely to exploit". Using this error-behavior-to-exploitability mapping obtained from security researchers, we then compare our newly identified error behaviors with those specified in their original bug reports.

In our dataset, we have 60 Linux kernel bugs. For 44 bugs, their reports gathered from Syzbot demonstrate error behaviors associated with less-likely-to-exploit. For the other 16 kernel bugs, their reports expose errors tied to likely-to-exploit. As we can observe from Table 2.3, for 26 bugs (about 60% of 44 less-likely-to-exploit bugs), GREBE could find at least one likely-to-exploit error behavior. From that newly identified error behavior, one could imply a higher exploitation potential. This observation indicates that GREBE can help security researchers better infer kernel bugs' exploitation potential.

Among the rest 16 kernel bugs originally tied to likely-to-exploit, there are 8 bugs (50%). By using GREBE, one can identify their other likely-to-exploit error behaviors. We argue, this does

not mean that GREBE has no utility for these kernel bugs. Taking a closer look at the three cases #e4be308, #3b7409f, and #ddaf58b. Their original reports all indicate that the bug provides an ability to perform a write to an unauthorized memory region. However, the newly discovered error behaviors enable the adversaries to perform unauthorized read/write at different memory regions. Take the case #3619dec5 for example. Its new error behavior can write data to the kmalloc-64 from 56th to 60th bytes, whereas its error behavior shown in the report corrupts the first eight bytes of kmalloc ↪ -64. This enlarged memory access potentially diversifies the way to perform exploitation and bypass mitigation.

For the kernel bugs of our selection that do not show exploitation potential improvement (i.e., 26 bugs = 60-26-8), we argue that this does not dilute the contribution of GREBE. First, based on the aforementioned small-scale evaluation on the false negatives of GREBE, it is very likely that all the possible error behaviors of these bugs are exposed. In this situation, there are fewer chances for a security researcher to find unknown error behaviors indicating a higher exploitation potential. Second, although the exploitation potential remains unchanged, GREBE manages to find many other error behaviors (e.g., #1fd1d44 in the table at [33]). These additional error behaviors and the corresponding fuzzing programs can potentially facilitate the root cause diagnosis, as is demonstrated in [14].

Real-world Impact. For all the 44 kernel bugs (the original reports of which implies less-likely-to-exploit), we performed an exhaustive search and found no work demonstrating their exploitability in the past. As described above, using GREBE, we can turn 26 of them from less-likely-to-exploit to likely-to-exploit. For these 26 kernel bugs, we further explore their exploitability manually. We surprisingly discovered that 6 out of the 26 bugs (illustrated by a star sign in Table 2.3) could be turned into fully exploitable kernel vulnerabilities. Take the case #6a03985 as an example. Its error behavior initially reported by Syzkaller is a WARNING implying less-likely-to-exploit. Using GREBE,

we identified a use-after-free error behavior for this bug. Starting from this newly discovered error behavior and the primitive the error behavior provides, we successfully demonstrated the bug's exploitability, including leaking sensitive data (e.g., encryption key and hashed password), bypassing KASLR, and redirecting the kernel execution for privilege escalation. We have responsibly disclosed the bug details and our working exploits with the corresponding vendors, resulting their rapid fix adoption. RedHat assigned one of the exploitable bugs with CVE-2021-3715 for keeping track. To facilitate the future study, we release our exploits at [20].

2.7 Related Work

This section summarizes the works most relevant to ours.

Kernel Fuzzing. Syzkaller [11] and Trinity [13] are two popular code-coverage-based kernel fuzzers. While doing fuzzing, they use templates to specify the dependency between system calls and the expected value range of system calls' arguments. However, with only explicit dependencies between system calls, it is not enough to produce a high-quality fuzzing program because the OS kernel is a massive system with a complicated internal state transition. IMF [63] optimizes kernel fuzzing by tracking the system calls and analyzing them coordinately with type information to infer the kernel system's internal states. This approach, unfortunately, has the limitation of extracting internal dependencies inside the kernel. As such, taking a step ahead, Moonshine [24] leverages light-weight static analysis to detect internal dependencies across different system calls from system call traces of real-world programs. Recently, HFL [25] introduces hybrid fuzzing to the kernel, performing point-to analysis, and symbolic checking to figure out precise constraints between system state variables. To support closed-source kernel, instead of relying on the kernel interface to collect code coverage, kAFL [12] proposes a fuzzing framework that employs a hardware-assisted code coverage measurement. Although the kernel fuzzers above demonstrate effectiveness in find-

ing kernel bugs, like Syzkaller, their design inevitably fails multiple error behavior exploration simply because they rely on code coverage to guide kernel fuzzing tasks, making our task inefficient. In this work, `GREBE` introduces a new design that utilizes critical kernel objects to improve effectiveness and efficiency for multiple error behavior exploration.

Apart from kernel fuzzers aiming to find various types of bugs in the entire system, there are works focusing on specific kernel modules or bug types. DIFUZE [27] uses static analysis to effectively fuzz device drivers in the Android kernel. Periscope [64] fuzzes a device driver not via system call interfaces but mutating input space over I/O bus. Razzer [29] combines static and dynamic testing to reach program sites where race condition bugs may exist. KRACE [28] further customizes to find race condition bugs in the file system. While they demonstrate their utility in hunting bugs in specific kernel modules, it is difficult to generalize these techniques to explore kernel bugs' error behaviors.

SemFuzz [26] is the only work that aims to trigger a known kernel bug through kernel fuzzing to the best of our knowledge. However, this technique is not designed to diversify the paths and contexts for triggering the bug but simply to enable bug reproduction. Therefore, it is not suitable for the problem we address.

Exploitability Assessment. Automating exploit development can also facilitate exploitability assessment. For user space programs, Brumley et al. [65], [66] used preconditioned symbolic execution to generate exploits for stack overflow and format string vulnerabilities. Bao et al. [67] recently proposed shellcode layout remediation and path kneading approaches to transplant existing shellcode. The Shellphish team developed PovFuzzer and Rex to turn a crash to an exploit [68]–[70]. Heelan et al. focus on heap buffer overflow vulnerabilities in user space programs. In [71], they use regression test to learn how to automate heap layout manipulation so that one could corrupt the sensitive pointers. In [72], they further improve their proposed approach by using a genetic algorithm

to replace the random search algorithm for exploiting heap overflow vulnerabilities in language interpreters. Sharing the similar goals with the works [71], [72], Revery [73] also explores exploitable memory layouts for vulnerabilities in userspace programs. It utilizes fuzz testing along with a program synthesis method to guide the construction of a working exploit. Insu et al. [74] discovers new exploitation primitives in the heap allocator. They provide heap operations and attack capabilities as actions, driving the heap allocator to execute until primitives such as arbitrary write or overlapped chunks are identified. Unlike the works summarized above, `GREBE` focuses on a bug's exploitability assessment in the kernel space which is naturally more sophisticated than userland programs. Besides, our work is not designed for constructing exploitable memory layout or synthesizing working exploits. Rather, it focuses on exploring all the possible error behaviors for a single kernel bug.

Regarding the kernel space, existing exploitability assessment works are mainly in three directions. The first direction is to obtain exploitable primitives. Xu et al. [75] exploit use-after-free vulnerabilities using two memory collision mechanisms to perform heap spray in the kernel. SLAKE [76] facilitates the exploitation of slab-based vulnerabilities by first building a database of kernel objects and then systematically manipulating slab layout using the kernel objects in the database. Lu et al. [77] exploits use-before-initialization vulnerabilities using deterministic stack spraying and reliable exhaustive memory spraying. As a follow-up work, Cho et al. [78] further propose to use BPF functionality in the kernel for stack spraying. The second direction is to bypass mitigations in the kernel. For example, ret2dir [79] takes advantage of physical memory which is mapped to kernel space for payload injection. KEPLER [80] leverages communication channels between kernel space and user space (e.g., `copy_from/to_user`) to leak stack canary and inject ROP payload to kernel stack. ELOISE [22] bypasses KASLR and heap cookie protector using a special but pervasive type of structure. The third direction is to explore the capability of vulnerabilties,

which is most related to our work. In this direction, FUZE [81] explores new use sites for use-after-free vulnerabilities using under-context fuzzing and identifies exploitable primitives implied by the new use sites using symbolic execution. KOOBE [82] extracts capabilities of a slab-out-of-bound access vulnerability manifested in the PoC program and uncover hidden capabilities using capability-guided fuzzing. The techniques developed in both works are customized to the characteristics of a specific vulnerability type and are difficult to generalized to others. Besides, they require to manually diagnose root cause of the bug while GREBE does not. Moreover, they cannot explore possible error behaviors for a single bug, which is the main contribution of GREBE.

2.8 Conclusion

We design and develop an object-driven kernel fuzzing method. Using our proposed technique, security analysts could explore various contexts and paths toward a target kernel bug and exhibit the bug's many error behaviors. The newly identified error behaviors might have higher exploitation potential than the one shown in the original report. It indicates the bug's exploitability escalation. As such, we safely conclude, given a kernel bug, the object-driven kernel fuzzing method could help security analysts better understand and infer exploitability for a given kernel bug.

CHAPTER 3
DIRTYCRED: ESCALATING PRIVILEGE IN LINUX KERNEL

3.1 Introduction

Linux, owing to its wide application in mobile devices, cloud infrastructure, and web servers, has become a prime target for hackers. To safeguard Linux, kernel developers and security professionals implement a range of kernel protection and exploit mitigation measures (e.g., KASLR [83] and CFI [84]), making kernel exploitation more difficult than ever. However, to execute a successful exploit, the contemporary attacker needs to pinpoint potent kernel vulnerabilities capable of bypassing corresponding protection and mitigation mechanisms.

A recent vulnerability, CVE-2022-0847 [85], and its exploitation approach, have drawn substantial attention from the cybersecurity community. Due to its severity and potential impact, it's been dubbed DirtyPipe [86]. Unlike typical kernel vulnerabilities, DirtyPipe can achieve privilege escalation without disabling commonly used kernel protections and mitigations. This trait makes existing Linux defenses ineffective, posing a significant threat to Linux-kernel-driven systems, including Android devices.

While DirtyPipe is potent, its exploitability relies heavily on the vulnerability's ability to misuse the Linux kernel pipe mechanism for arbitrary file data injection. For other Linux kernel vulnerabilities, such a capacity is rare. Consequently, the Linux community and device manufacturers like Google have rapidly released patches to fix the kernel bug, thus eliminating the memory corruption. However, without such memory corruption, exploiting a fully-protected Linux kernel remains challenging. Other kernel vulnerabilities find it difficult to produce the same level of

security impact as DirtyPipe.

In this work, we introduce a novel, general exploitation method through which even typical kernel vulnerabilities could achieve the same exploitation goal as DirtyPipe. Our exploitation method, named DirtyCred, doesn't depend on the pipeline mechanism of Linux or the nature of the CVE-2022-0847 vulnerability. Instead, it utilizes a heap memory corruption vulnerability to substitute a low privileged kernel credential object with a high privileged one. This action leads the Linux kernel to mistakenly allow an unprivileged user to operate on high privileged files or processes.

Implementing DirtyCred presents three key technical challenges. Firstly, it needs to pivot a vulnerability's capability to facilitate a credential object swap, as different vulnerabilities offer varying abilities for memory corruption. Secondly, DirtyCred must strictly control the time window for object swap, as the viable time window is brief. Without a practical mechanism to prolong this window, the exploitation would be unstable. Lastly, DirtyCred needs an efficient method to enable an unprivileged user to actively allocate privileged credentials, as failing to achieve this would render the credential object swap ineffective.

To overcome these technical challenges, we introduce a series of vulnerability pivoting schemes, utilize three different kernel features to extend the necessary time window, and employ various kernel mechanisms to spawn high privileged threads, thus actively allocating privileged objects. We evaluate DirtyCred's exploitability using 24 real-world kernel vulnerabilities and surprisingly discovered that DirtyCred could demonstrate privilege escalation on 16 vulnerabilities. We shared our newly proposed exploitation method with Google Vulnerability Rewards Program (kCTF VRP [87]) and received their acknowledgment with $20,000 bounty reward. Compared to existing kernel exploitation techniques, DirtyCred offers unique characteristics. It is a general exploitation approach that allows privilege escalation for any heap-based vulnerabilities. It sig-

nificantly reduces the burden of exploit migration, can bypass many powerful kernel protections and exploit mitigation mechanisms, and can result in more severe security problems, like rooting Android and escaping a container.

With robust exploitability demonstration and lack of effective defenses, DirtyCred could soon become a significant threat to Linux. Consequently, we have proposed a new Linux kernel defense mechanism that hosts high and low privileged credentials in separate memory regions, introducing negligible overhead.

In summary, our research makes the following contributions:

- We introduce DirtyCred, a novel exploitation method that can bypass common kernel protections and execute privilege escalation in Linux systems.
- We showcase the strong exploitability of DirtyCred on a variety of real-world Linux kernel vulnerabilities.
- We analyze the limitations of existing kernel defenses and propose a new defense mechanism, demonstrating its minimal performance overhead.

The structure of this paper is as follows: Section 3.2 introduces the necessary background and discusses the threat model. Section 3.3 presents an overview of DirtyCred and the technical challenges it faces. Section 3.4, 3.5, and 3.6 detail various techniques to tackle these challenges. Section 3.7 evaluates the effectiveness of DirtyCred on real-world Linux kernel vulnerabilities. Section 3.8 presents a new defense mechanism and evaluates its performance. Section 3.10 discusses related work, followed by Section 3.9, which discusses related issues and future work. The paper concludes in Section 3.11.

3.2 Background & Threat Model

This section provides an overview of the critical technical background needed to comprehend our proposed exploitation methodology. Additionally, we discuss our threat model and underlying assumptions.

3.2.1 Credentials in Linux kernel

As described in [88], credentials are a set of properties within the kernel that house privilege information. These properties enable the Linux kernel to assess user access rights. The kernel represents credentials through specific objects laden with privilege data. As far as we understand, these objects comprise of ”cred”, ”file”, and ”inode”. However, for the exploitation methods in this paper, we exclusively rely on ”cred” and ”file” objects. We disregard the ”inode” object because it is only allocated upon the creation of a new file on the filesystem, offering inadequate flexibility for memory manipulation - a critical operation for successful exploitation. We provide essential background information for ”cred”, ”file”, and ”inode” objects in the succeeding subsections.

Every Linux task maintains a pointer referring to a ’cred’ object. The ’cred’ object houses the UID field, denoting the task privilege. For instance, `GLOBAL_ROOT_UID` signifies that the task possesses root privilege. When a task seeks to access a resource (for example, a file), the kernel reviews the UID in the task’s ’cred’ object to determine whether to grant access. Apart from UID, the ’cred’ object also carries capability information, outlining the task’s specific privileges. An example being, `CAP_NET_BIND_SERVICE` suggests that the task can bind a socket to an internet domain’s privileged port. For each task, their credentials are adjustable. In modifying the task credentials, the kernel adheres to the copy-and-replace rule - it copies the credentials first, then modifies the copy, and finally replaces the cred pointer in the task with the newly amended copy. In Linux, a

task may only modify its own credentials.

In the Linux kernel, every file possesses its owner's UID and GID, access permissions for other users, and capabilities. For executable files, they also maintain SUID/SGID flags indicating special permission that permits other users to execute with the owner's privileges. In the kernel's implementation, each file is tied to an 'inode' object linked to the credentials. When a task attempts to open a file, the kernel initiates the function `inode_permision`, inspecting the inode and the associated permission prior to granting file access. Once a file is opened, the kernel detaches the credentials from the 'inode' object and connects them to the 'file' object. Alongside holding the credentials, the 'file' object also contains the file's read/write permissions. Through the 'file' object, the kernel can index the cred object to evaluate the privilege. Furthermore, it can verify read/write permission to ensure a task does not write data to a file opened in read-only mode.

3.2.2 Linux Kernel Heap Memory Management

The Linux kernel has designed memory allocators specifically to handle small memory allocations. This design choice serves to enhance performance and mitigate fragmentation. Although the Linux kernel incorporates three distinct memory allocators, they all conform to a similar overarching design. Namely, they utilize caches to store memory of identical size. For each cache, the kernel assigns memory pages and partitions the memory into numerous same-sized segments, with each segment acting as a memory slot that hosts an object. When a memory page for a cache is exhausted, the kernel assigns new pages to the cache. If a cache ceases to use a memory page, that is, all the objects on the memory page have been freed, the kernel repurposes the memory page accordingly. The Linux kernel primarily consists of two kinds of caches, as outlined briefly below.

Generic Caches. The Linux kernel maintains several general-purpose caches to allocate memory of varying sizes. When memory is allocated from the general-purpose caches, the kernel initially

rounds up the requested size and locates the cache that fits the size request. Subsequently, it allocates a memory slot from the corresponding cache. In the Linux kernel, if an allocation request doesn't specify the cache type it should be allocated from, the allocation defaults to the general-purpose caches. Allocations that are categorized under the same general-purpose cache may share the same memory address as they may be housed on the same memory page.

Dedicated Caches. For performance and security reasons, the Linux kernel establishes specialized caches. As certain objects are frequently used in the kernel, providing caches specifically for these objects can cut down the time required for their allocation, thereby enhancing system performance. Allocations that fall into specialized caches do not share the same memory page with general allocations. Hence, objects allocated in the general-purpose cache are not positioned adjacent to objects in specialized caches. This can be perceived as cache-level isolation, which alleviates the overflow risk from objects in general-purpose caches.

3.2.3 Threat Model

Our threat model assumes an unprivileged user with local access to the Linux system, with an intent to exploit a heap memory corruption vulnerability in the kernel, thus elevating his/her privilege level. Furthermore, we operate on the assumption that Linux activates all available exploit mitigation and kernel protection mechanisms provided in the mainstream kernel (version 5.15). These mechanisms encompass KASLR, SMAP, SMEP, CFI [83], [89]–[91], KPTI [92], and others. These mitigations and protections ensure the randomization of kernel addresses, the inability of the kernel to directly access user-space memory during execution, and guaranteed control-flow integrity. Importantly, we do not assume the existence of a hardware side channel that could aid kernel exploitation.

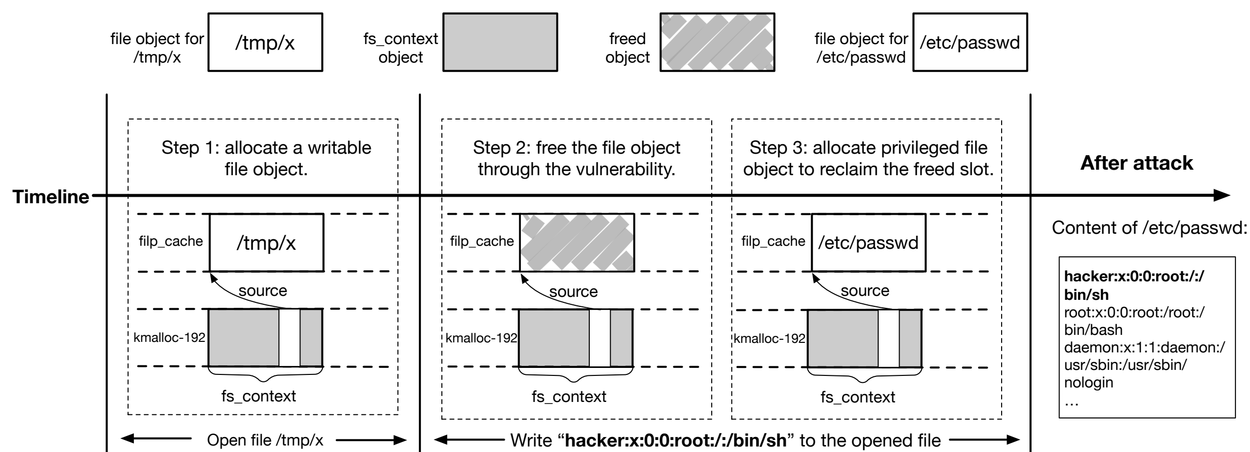

Figure 3.1: The overview of exploiting CVE-2021-4154, the write operation to the opened file starts between step 1 and step 2 and finishes after step 3.

3.3 Technical Overview & Challenges

This section initially offers a broad overview of DirtyCred by exemplifying its application in a real-world scenario. Following this, we delve into the technical obstacles that need to be addressed by DirtyCred.

3.3.1 Overview

We utilize an actual Linux kernel vulnerability, specifically CVE-2021-4154 [93], to demonstrate DirtyCred's operation on a high-level basis. This vulnerability, CVE-2021-4154, arises from a type confusion error where the `fs_context` object's source field inaccurately references a file object. In the Linux kernel, a reference count mechanism maintains a file object's lifecycle. When the reference count drops to zero, indicating the file object is no longer in use, it is automatically freed. However, activating the vulnerability leads to the file object being erroneously freed while it remains in use.

As illustrated in Figure 3.1, DirtyCred initiates by opening a writable file ”tmp/x”, which al-

locates a writable file object in the kernel. Upon triggering the vulnerability, the source pointer will reference the file object in the related cache. Then, DirtyCred attempts to write content to the open file ”tmp/x”. The Linux kernel, before performing the actual content write, checks for write permission and confirms the position’s writability. Once these checks pass, DirtyCred pauses the actual file writing process and moves to the second stage. Here, DirtyCred triggers the `fs_context` object’s free site to deallocate the file object, leaving the file object’s memory spot free.

Next, in the third stage, DirtyCred opens a read-only file ”/etc/passwd”, prompting the kernel to allocate the file object for ”/etc/passwd”. As shown in Figure 3.1, the freshly allocated file object claims the now available spot. After this setup, DirtyCred resumes its write operation, and the kernel proceeds with the actual content writing. Given that the file object has been swapped, the held content is redirected to the read-only file ”/etc/passwd”. Assuming the content written to ”/etc/password” is ”hacker:x:0:0:root:/:/bin/sh”, a malicious actor could use this method to inject a privileged account and achieve privilege escalation.

The example above merely serves as a demonstration of how DirtyCred employs file objects for exploitation. As discussed in Section 3.2, ”cred” objects are also classified as credential objects, in addition to ”file” objects. In a manner similar to the above-described file swap, a malicious actor could exploit a parallel strategy to swap cred objects, thus achieving privilege escalation. Owing to space constraints, we don’t expand on this topic. For those interested in cred object exploitation, we recommend our published exploitation demonstration at [94].

Based on the real-world example given, it’s clear that DirtyCred doesn’t change the control flow but leverages the innate kernel memory management to manipulate objects in memory. Consequently, many prevalent defense mechanisms that prevent control flow tampering do not interfere with DirtyCred’s exploitation. While certain recent research endeavors have enabled kernel defense by re-architecting memory management (e.g., AUTOSLAB [95]), these too fail to block

DirtyCred. As we will discuss in Section 3.8, the recently proposed memory management methodologies still operate at a coarse-granularity, making them inadequate to thwart our memory manipulation.

3.3.2 Technical Challenges

Despite the illustration of how DirtyCred performs exploitation leading to privilege escalation, several technical details require further clarification and numerous challenges await resolution.

- As previously discussed, DirtyCred requires the capability to perform an invalid-free operation to deallocate a low-privilege object (e.g., a writable file object) and reallocate a high-privilege object (e.g., a read-only file object). However, in reality, a kernel vulnerability may not always grant us this ability. For instance, a vulnerability may provide only the capacity for out-of-bound overwriting rather than an invalid-free directly against a credential object. Therefore, DirtyCred needs matching strategies to pivot the capabilities of vulnerabilities with varied capabilities. We delve into how to pivot capabilities for different types of kernel vulnerabilities in Section 3.4.
- As narrated in the example, DirtyCred needs to pause the actual file writing process after the permission check and before the file object swap. However, this proves challenging. In the Linux kernel, the permission check and actual content writing occur sequentially in rapid succession. Without a feasible strategy to precisely control the timing of the file object swap, the exploitation process would inevitably be unstable. In Section 3.5, we present a range of effective mechanisms that ensure the file object swap occurs within the intended timeframe.
- As discussed, one of the most critical steps in DirtyCred's process is to replace low-privilege credentials with high-privilege ones. This is done by allocating high-privilege objects that seize the freed memory spot. However, it's challenging for a low-privilege user to allocate high-

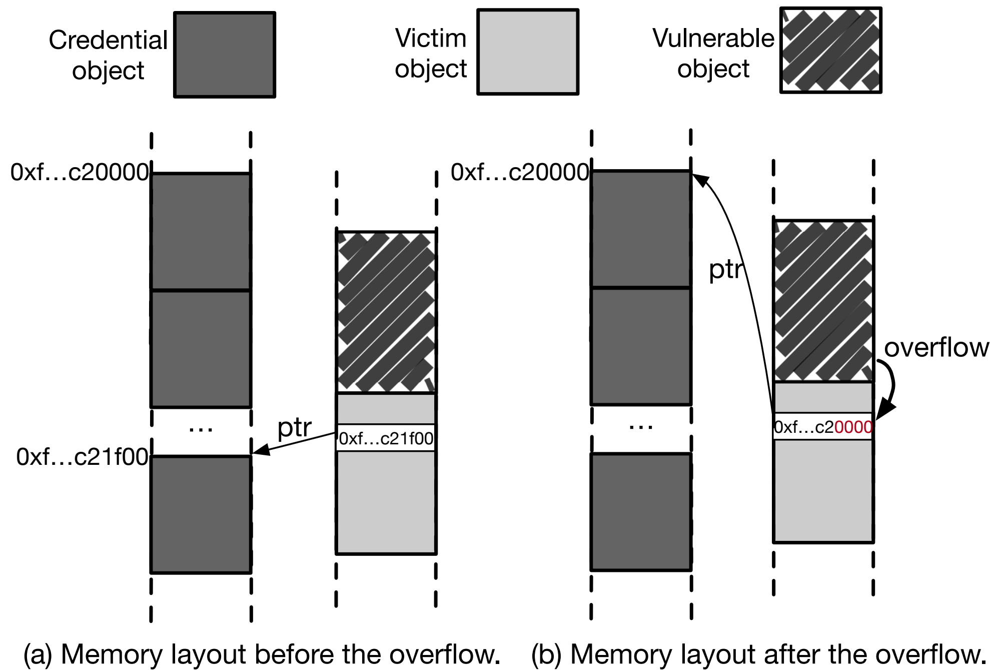

Figure 3.2: The memory layout before and after converting a heap overflow capability into the ability to deallocate a credential object.

privilege credentials. Although waiting for activities from privileged users might potentially solve this issue, such a passive strategy significantly impacts the stability of the exploitation. Firstly, DirtyCred lacks information on when the desired memory spot will be reclaimed, impeding continuous exploitation. Secondly, DirtyCred does not control the newly allocated objects, and therefore it's possible that the object taking over the desired memory slot does not possess the required privilege level. We introduce a userspace mechanism and a kernel space scheme to tackle this issue in Section 3.6.

3.4 Pivoting Vulnerability Capability

As the example in Figure 3.1 demonstrates, the kernel vulnerability denoted as CVE-2021-4154 endows DirtyCred with the means to improperly dispose of the file object. Nonetheless, in real-world scenarios, a vulnerability may not always offer such capacity. For instance, a double-free (DF) or use-after-free (UAF) capacity may not be directly related to a credential object. Some vulnerabilities, such as out-of-bound (OOB) access, lack the invalid free capacity. Therefore, DirtyCred requires the adaptation of a vulnerability's capacity. We elucidate below how DirtyCred is engineered to adapt capability.

3.4.1 Pivoting OOB & UAF Write

Given an OOB vulnerability or a UAF vulnerability with data overwriting capacity in a cache, DirtyCred first identifies an object (i.e., the victim object) that shares the same cache and contains a pointer referring to a credential object. It then leverages heap manipulation techniques [96], [97] to allocate the object in the memory region prone to overwriting. As depicted in Figure 3.2 (a), to pivot an OOB vulnerability, the victim object is strategically placed right after the vulnerable object. Using the overwriting capacity, DirtyCred modifies the enclosed pointer within the object. More specifically, DirtyCred applies the overwriting capacity to zero the last two bytes of the pointer referring to the credential object (see Figure 3.2 (b)).

Bear in mind that a cache is composed of contiguous pages. In the Linux kernel, the address of a memory page always conforms to a format where the last byte is zero. When allocating objects in a new cache, the object starts from the beginning of the memory page. As a consequence, the earlier zero-byte overwrite forces the pointer to reference the start of a memory page. For instance, as shown in Figure 3.2 (b), after nullifying the last two bytes of the pointer referring to

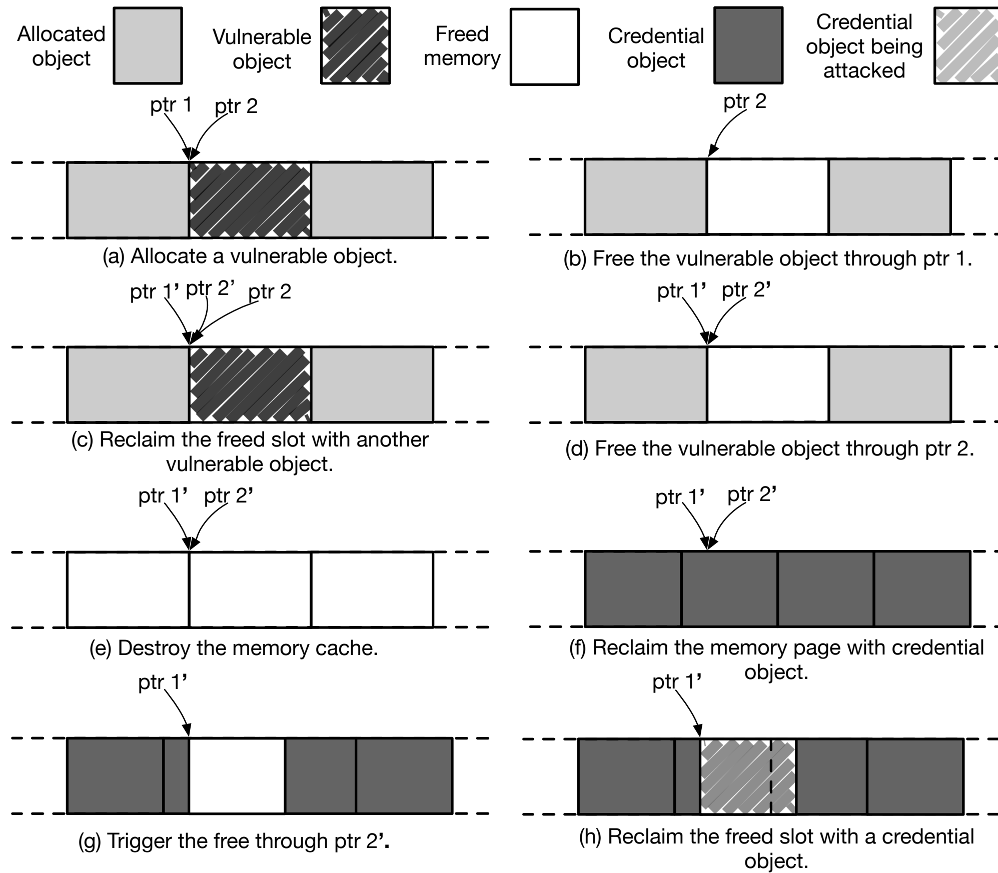

Figure 3.3: The step-by-step example demonstrating converting a double-free capability into the ability to deallocate a credential object.

a credential object, the pointer refers to the beginning of a memory page where another credential object resides.

As shown in Figure 3.2 (b), following the pointer manipulation, DirtyCred obtains an additional reference to the first object on the memory page. We assert that this additional object reference denotes a successful capability pivot. The rationale is that the kernel can free the object normally, leaving the pointer in the victim object as a dangling pointer. Then, following a procedure akin to the one described in Section 3.3, DirtyCred can execute a heap spray, occupy the freed spot with a high-privilege credential object, and thus accomplish privilege escalation.

```
1  struct iovec
2  {
3      void __user *iov_base; /* BSD uses caddr_t (1003.1g requires void *) */
4      __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
5  };
6
7  ssize_t vfs_writev(...)
8  {
9      // permission checks
10     if (!(file->f_mode & FMODE_WRITE))
11         return -EBADF;
12     if (!(file->f_mode & FMODE_CAN_WRITE))
13         return -EINVAL;
14
15     ...
16     // import iovec to kernel, where kernel would be paused
17     // using userfaultfd & FUSE
18     res = import_iovec(type, uvector, nr_segs,
19                        ARRAY_SIZE(iovstack), &iov, &iter);
20     ...
21     // do file writev
22 }
```

Listing 6: The code snippet of vfs_writev function in kernel before 4.13.

3.4.2 Pivoting Double Free

Within the Linux kernel, general caches (e.g., `kmalloc-96`) and specific caches (e.g., `cred_jar`) are separated. There's no overlap in the objects these caches contain. However, the Linux kernel includes a recycling mechanism, which when destroying a memory cache, recycles the associated unused memory pages and then allocates the recycled pages to the caches in need of more space. This feature allows cross-cache memory manipulation, offering DirtyCred the means to pivot the capability for double-free vulnerabilities.

Figure 3.3 outlines the process of how DirtyCred converts a double-free capability into the capability required for a privileged object swap. Initially, DirtyCred allocates a large number of objects in the cache where the vulnerability is found. Among these new objects, one is the

vulnerable object. DirtyCred can invalidate the vulnerable object by deallocating it twice using two distinct pointers. Due to the high volume of allocations, DirtyCred ensures that a cache is filled with newly allocated objects post the extensive object allocation (see Figure 3.3 (a)).

Following the mass allocation, DirtyCred uses the first pointer to improperly deallocate the vulnerable object, leaving the second pointer intact (see Figure 3.3 (b)). It then reallocates the vulnerable object, occupying the freed memory spot. As seen in Figure 3.3 (c), after reallocation, there are three pointers pointing to the vulnerable object. One is the pointer left by the first vulnerable object. The remaining two are linked to the double-free capability against the newly allocated vulnerable object.

DirtyCred further deallocates the newly allocated vulnerable object using one of the three pointers, leaving a free memory spot referenced by two dangling pointers (see Figure 3.3 (d)). As previously noted, the Linux kernel recycles the memory page and allocates it to another cache if a cache holds no allocated objects. Therefore, after deallocating the vulnerable object, DirtyCred further deallocates the other objects in the cache, thus freeing the cache (see Figure 3.3 (e)).

On the recycled memory page, the kernel establishes a new cache to store credential objects. The new cache divides the page memory into slots. As illustrated in Figure 3.3 (f), if the size of the vulnerable object differs from that of the credential object, the credential object's address won't align with that of the vulnerable object. This misalignment causes the remaining two pointers to reference the middle of the credential object. In this memory state, DirtyCred cannot follow the exploitation procedure described in Section 3.3, since successful exploitation requires the ability to deallocate a credential object.

To address this issue, DirtyCred uses one of the leftover pointers to improperly deallocate the central credential object. As shown in Figure 3.3 (g), following this deallocation, the kernel creates a free memory spot of the same size as a credential object. Therefore, when DirtyCred allocates a

new credential object, the kernel fills that free spot with the new object. As seen in Figure 3.3 (h), after the free spot is occupied, the final remaining pointer references the newly allocated credential object. This indicates a successful capability pivot. The reasoning is that DirtyCred could use the remaining pointer to improperly deallocate the credential object and then perform an object swap to escalate privileges.

3.5 Time Window Expansion

Recall that before executing a file write operation, the Linux kernel must verify file permission. For DirtyCred to perform a file object swap, it must occur between the permission check and the actual file writing. However, this window of opportunity is too brief to perform a successful exploit, as the swapping process involves triggering the vulnerability and manipulating the heap layout, which could take several seconds. DirtyCred addresses this issue by using various techniques to extend this window, ensuring it lasts longer than the swapping process. In this section, we explain these techniques and discuss how they aid in facilitating the exploit.

3.5.1 Exploiting Userfaultfd & FUSE

Userfaultfd [98] and FUSE [99] are two essential features of the Linux kernel. The Userfaultfd feature allows userspace to manage page faults. When a page fault is triggered on the memory registered with Userfaultfd, the registered page fault handler will be notified and take over the page fault handling. Unlike Userfaultfd, FUSE provides a userspace filesystem framework, allowing users to implement a userspace filesystem. Users can register their handler for this implemented userspace filesystem to dictate how to respond to file operation requests. Both Userfaultfd and FUSE can be leveraged to pause Linux kernel execution for a user-defined duration. With Userfaultfd, an attacker can register a page fault handler for a memory page. When the kernel tries to access that

memory and triggers a page fault, the registered handler is invoked, enabling the attacker to pause kernel execution. With FUSE, an attacker could allocate memory from the userspace filesystem. When the kernel accesses this memory, it invokes the predefined file access handler and thereby pauses kernel execution.

DirtyCred uses these features to pause kernel execution after completing the file permission check. In the following, we use Userfaultfd as an example to explain how DirtyCred achieves this kernel pause and extends the exploitation window. The procedure for pausing the kernel with FUSE is similar. Readers can refer to the exploit sample we developed [94] for more details.

When executing a file write, DirtyCred uses the `writev` system call, which is the implementation of vectored I/O. Unlike the `write` system call, `writev` uses the `iovec` structure to transfer data from userspace to kernel space. List 6 from lines 1 to 5 defines the `iovec` structure. It includes a userspace address and a size field that specifies the quantity of data to be transferred. In the Linux kernel space, to copy the data enclosed in `iovec`, the kernel must first import the `iovec` to the kernel space. Therefore, before Linux kernel version v4.13, as depicted in List 6, the implementation of `writev` ↪ first verifies the file object, ensuring the current file is open and writable. Once the check is successful, it imports the `iovec` from userspace and writes user data to the appropriate file. In this implementation, the import of `iovec` falls between the permission check and data write. DirtyCred can utilize the aforementioned Userfaultfd feature to pause kernel execution immediately after the permission check, securing enough time to swap the file object. To our knowledge, this technique was first employed by Jann Horn's exploit for CVE-2016-4557 [100], but it is no longer applicable after kernel v4.13.

```
1 ssize_t vfs_writev(...)
2 {
3     ...
4     // import iovec to kernel, where kernel would be paused
5     // using userfaultfd
6     res = import_iovec(type, uvector, nr_segs,
7                                  ARRAY_SIZE(iovstack), &iov, &iter);
8     ...
9     // permission checks
10    if (!(file->f_mode & FMODE_WRITE))
11        return -EBADF;
12    if (!(file->f_mode & FMODE_CAN_WRITE))
13        return -EINVAL;
14    ...
15    // do file writev
16 }
```

Listing 7: The code snippet of vfs_writev function in kernel after 4.13.

```
1 ssize_t generic_perform_write(struct file *file,
2                 struct iov_iter *i, loff_t pos)
3 {
4     /*
5      * Bring in the user page that we will copy from _first_.
6      * Otherwise there's a nasty deadlock on copying from the
7      * same page as we're writing to, without it being marked
8      * up-to-date.
9      */
10    if (unlikely(iov_iter_fault_in_readable(i, bytes))) {
11        status = -EFAULT;
12        break;
13    }
14    ...
15    // call the write operation of the file system
16    status = a_ops->write_begin(file, mapping, pos, bytes, flags,
17                            &page, &fsdata);
18    ...
19 }
```

Listing 8: The code snippet of generic_perform_write function in the Linux kernel.

3.5.2 Alternative Exploitation of Userfaultfd & FUSE in Later Kernel Versions

From Linux kernel version v4.13 onwards, the kernel implementation was modified. The import of the `iovec` structure was moved to occur before the permission check (see List 7). In this updated implementation, DirtyCred could still utilize the Userfaultfd feature to pause kernel execution at the point of `iovec` import. However, it no longer allows DirtyCred to extend the time window between the permission check and the actual file write. To overcome this challenge, DirtyCred takes advantage of the design of the Linux filesystem.

In Linux, the filesystem design adheres to a strict hierarchy where the high-level interface is universally used for file write operations, while the low-level interface differs between filesystems. During a file write, the kernel first invokes the high-level interface. As shown in List 8, `generic_perform_write` is a high-level interface for file write operations. As seen in lines 15 to 17, `generic_perform_write` triggers the write operation of the filesystem and writes data to the file. To ensure performance and compatibility, the kernel triggers a page fault for the userspace data enclosed in the `iovec` just before the write operation. Consequently, by using the Userfaultfd feature in line 10, DirtyCred can pause kernel execution before actual file writing, thus securing a sufficient time window for privileged file object swap.

We argue that exploiting the filesystem's design to pause kernel execution at the page fault is harder to mitigate than pausing at the `iovec` import site. First, as mentioned in the Linux code comment, eliminating page fault in `iovec` could potentially lead to deadlock issues (see List 8). Some filesystems will unavoidably face issues if the page isn't pre-faulted. Second, while moving the page fault to occur prior to the permission check could theoretically resolve the issue, this straightforward defensive measure sacrifices kernel performance and is also susceptible to potential evasion. For instance, DirtyCred could remove the page immediately after triggering the initial page fault. This action would inevitably cause the kernel to trigger the page fault once more,

```c
static ssize_t ext4_buffered_write_iter(struct kiocb *iocb,
                        struct iov_iter *from)
{
    ssize_t ret;
    struct inode *inode = file_inode(iocb->ki_filp);
    inode_lock(inode);
    ...
    ret = generic_perform_write(iocb->ki_filp, from, iocb->ki_pos);
    ...
    inode_unlock(inode);
    return ret;
}
```

Listing 9: The code snippet of ext4 filesystem in the Linux kernel.

thereby pausing kernel execution immediately after the permission check.

3.5.3 Taking Advantage of Lock Mechanism in Filesystem

In order to prevent the corruption of a file's content, a filesystem doesn't permit two processes to write to a file concurrently. Linux's filesystem enforces this rule via a locking mechanism. To illustrate, List 9 demonstrates a simplified code segment that performs a write operation in the ext4 filesystem. As can be seen, the filesystem first tries to acquire the inode lock in Line 6. If another file operation is currently interacting with the inode (i.e., the lock is held by others), the filesystem will wait until the lock is released. After gaining the lock, the filesystem invokes `generic_perform_write` to write the data to the file. Upon completing the write operation, the filesystem releases the lock and exits the function.

While the aforementioned lock mechanism can ensure the accuracy of the write operation, it inadvertently provides DirtyCred with an opportunity to extend the time window, thus making an object swap feasible. Specifically, DirtyCred could initiate two processes – process A and process B – to write data to the same file concurrently. Assuming process A holds the lock and is writing a large amount of data, process B would be obliged to wait for an extended period until the lock

is released in Line 10. Since process B has already passed the file permission check prior to the invocation of `generic_perform_write`, the time spent waiting for the lock gives DirtyCred an ample time window to execute the file object swap without interference from the permission check. From our observations, the wait time could stretch up to several tens of seconds when writing a 4GB file to a hard disk drive. Within this time window, triggering the vulnerability and performing memory manipulation can be carried out without causing any stability issues in the exploitation.

3.6 Allocating Privileged Credential

As outlined in Section 3.3.2, DirtyCred cannot rely on passive observance of privileged users' actions in hopes of their activities leading to the allocation of a privileged object in the desired freed spot, which would thereby enable privilege escalation. Consequently, DirtyCred must take proactive measures to instigate the allocation of a privileged object within the kernel space. This section delves into how DirtyCred, operating as a minimally privileged user, initiates privileged object allocation.

3.6.1 Initiating Allocation from Userspace

In the Linux kernel, ”cred” objects signify the privilege levels of their associated kernel tasks. The root user possesses a privileged cred object, symbolizing the apex of privilege. Therefore, if DirtyCred can actively stimulate a root user's activities, the kernel may allocate the privileged cred objects as required. In Linux, when a binary holds SUID permission, its execution occurs as though it's executed by the owner, irrespective of who actually initiates the execution. Leveraging this attribute, a low-privileged user can instigate a root process when they execute a binary owned by a root user that also carries the SUID permission.

Historically, attackers have primarily focused on exploiting a vulnerability within a privileged

binary, thereby achieving privilege escalation. In this work, however, DirtyCred does not depend on vulnerabilities within privileged binaries. Instead, it misuses the above-mentioned feature to spawn SUID-set binaries owned by root users, triggering the allocation of a privileged cred object to fill the free memory slot. In Linux, there are numerous binaries that fit this profile, including executables such as su, ping, sudo, mount, pkexec, and others.

As noted earlier, aside from cred objects, DirtyCred can also swap file objects to escalate privileges. Contrary to cred objects, the allocation of file objects is relatively straightforward. Remember, DirtyCred substitutes a write-permitted file's object with a read-only file's object during file object swapping. To allocate file objects that specify read-only permissions, DirtyCred can open multiple target files with read-only permissions. This will result in the kernel allocating numerous corresponding file objects in the related kernel memory.

3.6.2 Initiating Allocation from Kernel Space

The previously described method highlights a strategy for allocating privileged objects from userspace. However, DirtyCred can also instigate the allocation of privileged objects from within the kernel space. When the Linux kernel initializes a new kernel thread, it clones its currently running process. Simultaneously, it assigns a duplicated cred object to the kernel heap. In the Linux kernel, the majority of kernel threads possess a privileged cred object, meaning the copied cred object also holds high privilege. Leveraging this ability to generate privileged kernel threads, DirtyCred can actively allocate privileged cred objects.

As far as we are aware, there exist two primary methods for allocating highly privileged credential objects. The first approach involves interacting with the kernel code snippets, provoking the kernel to internally generate a privileged thread. For instance, creating workers for a kernel workqueue can be employed to spawn kernel threads. The Linux kernel's work queue is designed

to manage deferred functions. A work queue is paired with several work pools, each of which contains workers. A worker is the fundamental execution unit that runs the tasks committed to the workqueue. The number of workers within each work pool is, at most, equivalent to the number of CPUs. Initially, the kernel creates only a single worker for each work pool. However, when more workers are needed, or in other words, when more tasks are committed to the work queue, the kernel dynamically generates additional workers. Each worker is a kernel thread, and by modulating the tasks committed to the kernel work queue, one can manipulate the spawning of kernel threads as required.

Apart from the method described above, the second approach to spawning kernel threads involves invoking the usermode helper. The usermode helper is a mechanism enabling the kernel to generate a user-mode process. One of the most straightforward applications of the usermode helper is the loading of kernel modules into the kernel space. During this process, the kernel invokes the usermode-helper API, which subsequently executes the userspace program – modprobe, in a high privilege mode, thereby creating high privileged credential objects in the kernel. A function of modprobe is to search through the standard installed module directories to find the required drivers. While conducting this search, the kernel must maintain its execution. To prevent modprobe from blocking kernel execution, when invoking a usermode-helper API, the kernel simultaneously spawns a new kernel thread.

3.7 Evaluation

In this section, we design two experiments to evaluate the exploitability of DirtyCred on real-world kernel vulnerabilities.

3.7.1 Experiment Design

As previously highlighted, DirtyCred leverages vulnerable objects (i.e., those housing credential objects) to perform memory manipulation, which is primarily used for vulnerabilities like out-of-bound access and use-after-free. This manipulation is a crucial step in enabling DirtyCred to achieve privilege escalation. During the memory manipulation process, DirtyCred assigns the vulnerable object to the cache where the vulnerability is situated. Given that different vulnerabilities exhibit memory corruption capability in varying caches, DirtyCred's success relies heavily on its ability to identify vulnerable objects that can be successfully accommodated within the respective cache. Bearing this in mind, our initial step involves identifying the distinct vulnerable objects applicable to each cache.

One might naturally assume that pinpointing these objects involves manually combing through the Linux kernel code, identifying vulnerable objects, and determining the input that could trigger the corresponding allocation. However, the size and complexity of the Linux kernel code space render manual code examination unfeasible. To address this challenge, we propose an automated method to detect vulnerable objects and determine the corresponding input necessary to instigate their allocation. For our assessment, we applied this automated method to the most recent stable kernel (i.e., version 5.16.15 at the time this paper was written). We only consider an object as a vulnerable object if the automated method can find an object containing the credential object and can demonstrate an input to allocate that object on the kernel heap. We provide a detailed explanation of the design and implementation of our automated method in Appendix A.2.

Alongside the identification of exploitable kernel objects, our experiment also investigates DirtyCred's exploitability against real-world vulnerabilities. Remember that DirtyCred requires pivoting a vulnerability's capability if the vulnerability doesn't offer DirtyCred the facility to directly swap credential objects. As discussed in Section 3.4.1, when performing vulnerability piv-

oting, DirtyCred might need to overwrite certain critical data within the vulnerable object. For varying vulnerabilities, their overwriting capabilities could differ substantially, subsequently influencing the success of privilege escalation. Consequently, we assess DirtyCred's effectiveness by exploiting numerous real-world vulnerabilities and examining how well it can execute exploitation against these vulnerabilities.

We conduct the exploitation assuming a Linux kernel equipped with cutting-edge exploit mitigation techniques. Hence, we need to select vulnerabilities identified in recently developed kernels. In our evaluation, we only considered Linux kernel CVEs reported after 2019. During our CVE selection process, we eliminated those vulnerabilities that do not corrupt data on the kernel heap, as well as those vulnerabilities for which we couldn't reproduce the corresponding kernel panic. Additionally, we discounted those vulnerabilities requiring the installation of specific hardware to trigger. Following these CVE selection criteria, we compiled a dataset comprising 24 unique CVEs. These CVEs' IDs and the corresponding vulnerability types are listed in Table 3.2. As can be seen, our selected test cases encapsulate nearly all types of vulnerabilities on the kernel heap.

3.7.2 Experimental Result

Exploitable Objects. Table 3.1 presents the vulnerable objects we identified in each kernel cache. It is evident from the data that the vulnerable objects encompass almost all the general caches, except for kmalloc-8, which is infrequently utilized in the Linux kernel. For most memory caches, there are several vulnerable objects that could potentially be advantageous for DirtyCred's privilege escalation. The offset of the field referencing a credential object is also shown in each vulnerable object in Table 3.1. Notably, the offsets differ across various vulnerable objects. This suggests that DirtyCred has a better probability of identifying an appropriate object that matches a vulnerability's capability, thus ensuring successful exploitation. For instance, if a vulnerability shows the ability

Memory Cache	Structure	Offset
kmalloc-16	vdpa_map_file	0 ⋆
kmalloc-32	binder_task_work_cb	16 ⋆
	binder_txn_fd_fixup	16 ⋆
	coda_file_info	8 ⋆
	shm_file_data	16 ⋆
kmalloc-64	fuse_fs_context	8 ⋆
	ovl_dir_file	24⋆ 32⋆
	bpf_event_entry	8 ⋆
kmalloc-96	gntdev_dmabuf_priv	80 ⋆
	nfs_access_entry	40 †
	request_key_auth	32 †
	watch	64 †
	bpf_perf_link	64 ⋆
kmalloc-128	async	32 †
	nfs_delegation	16 †
kmalloc-192	fs_context	88 †
	sync_file	0 ⋆
	vmci_ctx	144 †
	coda_vm_ops	8 ⋆
	nfs_open_context	80 †
	nfs_unlinkdata	144 †
	nfs_renamedata	152 †
	nfs4_layoutreturn	144 †
	ovl_fs	112 †
kmalloc-256	usb_dev_state	152 †
	autofs_sb_info	8 ⋆
	shmid_kernel	128 ⋆
	bsd_acct_struct	144 ⋆
kmalloc-512	linux_binprm	48 ⋆, 64 ⋆
	loop_device	96 ⋆
	dma_buf	8 ⋆
	nvmet_ns	24 ⋆
	ksmbd_file	0 ⋆
	rpc_clnt	440 ⋆
	nfs4_state_owner	56 †
	nfs4_ff_layout_mirror	96 †, 104 †
	p9_trans_fd	0 ⋆, 8 ⋆
kmalloc-1k	sock	600 †
	binder_proc	80 †
	kfd_process_device	256 ⋆
	send_ctx	0 ⋆
	nlm_host	520 †
	nfs4_layoutcommit_data	472 †
kmalloc-2k	vsock_sock	864 †
	io_ring_ctx	408 †
kmalloc-4k	vduse_iova_domain	3824 ⋆
vm_area_cachep	vm_area_struct	160 ⋆
ashmem_area_cache	ashmem_area	288 ⋆
client_slab	nfs4_client	736 †
nfsd_file_slab	nfsd_file	48 ⋆, 56†
kioctx_cachep	kioctx	512 ⋆

Table 3.1: Exploitable objects identified in the Linux kernel. Note that the symbol ⋆ indicates an object tied to "file" credential whereas the symbol † represents an object associated with "cred" object. The column "Memory Cache" specifies the caches storing kernel objects. The column "Structure" represents the exploitable objects' types. The column "Offset" describes where the credential object's reference is located in the exploitable object.

CVE-ID	Observed Capability	DirtyCred
CVE-2022-27666	OOB	✔
CVE-2022-25636	Double Free ⋆	✔
CVE-2022-24122	UAF	✗
CVE-2022-0995	OOB	✔
CVE-2022-0185	OOB	✔
CVE-2021-22600	Double Free	✔
CVE-2021-4154	UAF	✔
CVE-2021-43267	OOB	✔
CVE-2021-41073	Double Free ⋆	✔
CVE-2021-34866	OOB †	✗
CVE-2021-33909	OOB †	✗
CVE-2021-42008	OOB	✔
CVE-2021-3492	Double Free	✔
CVE-2021-27365	OOB	✔
CVE-2021-26708	Double Free ⋆	✔
CVE-2021-23134	Double Free ⋆	✔
CVE-2021-22555	Double Free	✔
CVE-2021-3490	OOB †	✗
CVE-2020-14386	OOB	✔
CVE-2020-16119	Double Free ⋆	✔
CVE-2020-27194	OOB †	✗
CVE-2020-8835	OOB †	✗
CVE-2019-2215	UAF	✗
CVE-2019-1566	UAF	✗

Table 3.2: Exploitability demonstrated on real-world vulnerabilities. Note that some CVEs provide both use-after-free and double-free capabilities. Here, we categorize such vulnerabilities into double-free and mark them with a ⋆ symbol. Note that the symbol † indicates the vulnerabilities that could corrupt only data in virtual memory area.

to overwrite 8 bytes to a neighboring object at its 8th byte offset, a vulnerable object with critical data at the 8th byte would greatly simplify DirtyCred's privilege escalation process.

Additionally, Table 3.1 reveals five objects within five general caches. These enclose the reference to the credential object at the start of the objects. This suggests that even if attackers only achieve a minimal memory corruption capability (e.g., overwriting two initial bytes of a victim object with zeros), they can still utilize the identified vulnerable object to initiate a DirtyCred attack. It's important to note that Table 3.1 also differentiates between vulnerable objects referencing cred and file with distinct symbols. As we will discuss in Section 3.9, a cred object could provide

enhanced support for escaping containers. Therefore, having a sufficient number of vulnerable objects linked to cred objects indicates a greater potential for Docker escape.

Exploitability. Table 3.2 illustrates the vulnerability of DirtyCred to various attacks. As can be seen, DirtyCred successfully circumvents kernel defense mechanisms and escalates privileges in 16 out of 24 vulnerabilities, when the underlying Linux kernel activates all the exploit mitigation mechanisms discussed in Section 3.2.3. This finding suggests that DirtyCred can serve as a potent and versatile exploitation method for kernel vulnerability exploitation tasks. Of the 16 successful test cases, eight involve out-of-bound or use-after-free vulnerabilities, while the remaining eight involve double-free. DirtyCred succeeds in all double-free test cases, as a double-free capability can always be pivoted to improperly freeing a credential object.

Unsuccessful cases primarily stem from out-of-bound (OOB) and use-after-free (UAF) vulnerabilities. For OOB vulnerabilities, the failed cases exhibited memory corruption in the virtual memory area. To utilize DirtyCred, we need to identify kernel objects containing credential data. Such objects are usually allocated within the kmalloc'ed memory region rather than virtual memory, resulting in DirtyCred's inability to find the necessary objects for successful exploitation. We mark these cases with a † symbol in Table 3.2. However, as we will discuss in Section 3.9.3, the inability to exploit these cases does not imply that DirtyCred cannot exploit vulnerabilities on virtual memory. The memory corruption capabilities on virtual memory could still be pivoted to capabilities useful for DirtyCred, given suitable vulnerable objects or other capability pivoting techniques.

For the UAF failure case CVE-2022-24122, the vulnerability does not demonstrate an overwrite capability through the dangling pointer, instead, it only exhibits an over-reading ability. As discussed in Section 3.4, DirtyCred relies on either an invalid write capability or an invalid free capability. The over-reading capability of CVE-2022-24122 hampers DirtyCred from performing

a successful capability pivoting, resulting in a failed attack. For CVE-2019-2215 and CVE-2019-1566, these vulnerabilities demonstrate an overwriting capability. However, the overwriting does not occur in the critical field of exploitable objects. Without this capability, DirtyCred is unable to manipulate the necessary fields in the kernel objects to free a credential object, leading to an unsuccessful attack.

3.8 Countermeasure Against DirtyCred Attack

Given the exploitability demonstrated in the previous section, it's clear that DirtyCred represents a serious threat to existing Linux systems. Although the method of misusing the lock mechanism could be mitigated by reengineering the filesystem, this wouldn't be enough to fully block DirtyCred, as it can also be launched via another route—swapping credential objects. Thus, an effective countermeasure would be to prevent the swapping of credentials with differing privilege levels.

Looking at this issue from one angle, userspace heap defenses are inadequate for DirtyCred. The kernel requires memory allocation, freeing, and access to be as fast as possible to avoid slowing down userspace programs and the entire system. This makes the memory allocator in the kernel far simpler than that in userspace (e.g., ptmalloc), rendering userspace heap defenses unsuitable for kernel space.

From another perspective, even though the Linux kernel has introduced many defense mechanisms (e.g., CFI, SMEP, SMAP, and KASLR), none of them are effective against DirtyCred for several reasons. Firstly, DirtyCred doesn't breach any control-flow integrity, rendering efforts to protect kernel control flow pointless. Secondly, DirtyCred isn't reliant on a single exploitation component. As shown in Section 3.7, valuable objects for exploitation are found in almost all general caches, making it virtually impossible to defend against DirtyCred by eliminating exploitable objects. Thirdly, DirtyCred achieves its goal by placing a legitimate credential object in an ille-

gitimate memory location rather than altering the content of a credential object. This exploitation practice means that existing credential integrity protection techniques (e.g., Samsung Knox's Real-time Kernel Protection) are unlikely to be effective. Finally, DirtyCred escalates privileges by swapping high and low privileged credential objects, rendering many kernel object isolation schemes (e.g., AUTOSLAB and xMP) ineffective, as they separate critical kernel objects into their own memory regions based on the object type rather than their privilege level.

In light of these challenges, we propose that an effective defensive measure against DirtyCred would be to isolate high and low privileged objects in a manner that prevents them from sharing the same memory space. This would prevent DirtyCred from overlapping objects with different privileges to escalate privileges.

To achieve this, a simple approach would be to create two distinct caches, one for high privileged object storage and another for low privileged objects. Since caches are naturally isolated, this design could ensure that objects with differing privileges don't overlap. However, as discussed in Section 3.4.2, when a memory cache is destroyed, Linux's buddy allocator recycles the underlying memory page, meaning that DirtyCred could still launch its attack by exploiting this memory-page recycling feature.

Design. Considering the analysis provided above, we propose a practical defensive measure that establishes a cache for high privileged objects within the virtual memory region, while leaving low privileged objects in the standard memory area (i.e., the direct-mapped memory region). The virtual memory region pertains to dynamically allocated, virtually contiguous memory within the kernel. This region spans the memory area defined by `VMALLOC_START` to `VMALLOC_END`. As it's separate from the direct-mapped memory region, the areas assigned to high and low privileged objects won't overlap, even after the caches are destroyed and the underlying memory pages are recycled.

Implementation. We have executed our proposed defense against DirtyCred on Linux kernel

Benchmark	Vanilla	Hardened	Overhead
Phoronix			
Apache (Reqs/s)	28603.29	29216.48	-2.14%
Sys-RAM (MB/s)	10320.08	10181.91	1.34%
Sys-CPU (Events/s)	4778.41	4776.69	0.04%
FFmpeg(s)	7.456	7.499	0.58%
OpenSSL (Byte/s)	1149941360	1150926390	-0.09%
OpenSSL (Sign/s)	997.2	993.2	0.40%
PHPBench (Score)	571583	571037	0.09%
PyBench (ms)	1303	1311	0.61%
GIMP (s)	12.357	12.347	-0.08%
PostMark (TPS)	5034	5034	0%
LMBench			
Context Switch (ms)	2.60	2.57	-1.15%
UDP (ms)	9.2	9.26	0.65%
TCP (ms)	12.75	12.73	-0.16%
10k File Create (ms)	13.8	14.79	7.17%
10k File Delete (ms)	6.35	6.62	4.25%
Mmap (ms)	80.23	81.91	2.09%
Pipe (MB/s)	4125.3	4028.9	2.34%
AF Unix (MB/s)	8423.5	8396.7	0.32%
TCP (MB/s)	6767.4	6693.3	1.09%
File Reread (MB/s)	8380.43	8380.65	0%
Mmap Reread (MB/s)	15.7K	15.69K	0.06%
Mem Read (MB/s)	10.9K	10.9K	0%
Mem Write (MB/s)	10.76K	10.77K	-0.09%

Table 3.3: The performance evaluation results of the proposed defense on two different benchmarks – Phoronix and LMBench.

v5.16.15. In this implementation, we manually adjusted how `cred` objects and `file` objects are allocated in the kernel. If the allocation pertains to privileged entities, we allocate them using virtual memory. Specifically, when allocating `cred` objects, we assess the privilege based on the `UID` of the object. If the `UID` matches `GLOBAL_ROOT_UID`, indicating that the allocation is for privileged cred objects, we employ `vmalloc` as the allocator to allocate virtual memory for the object. Regarding `file` objects, we examine the file's mode. If the file is opened with write permissions, we allocate the `file` object using `vmalloc` accordingly. Our implementation is available at [94].

Technical Discussion Our suggested defense safeguards the Linux kernel by enforcing memory isolation for credential objects. As mentioned above, our implementation establishes the privilege level at the time of object allocation. However, this privilege could be modified by changing the

UID during runtime (e.g., transitioning a low privileged credential object to a high privileged one via a 'setuid' syscall). When this happens, our proposed defense may face security challenges as we only carry out object isolation at the time of allocation.

To tackle this issue, we adjusted the way kernel credential objects are altered in our implementation. Specifically, if the kernel changes a credential object's UID to GLOBAL_ROOT_UID, we will copy the high privileged credential object to the 'vmalloc' region instead of modifying the original one. Nevertheless, we anticipate potential issues if future kernel development doesn't adhere to the same pattern. Therefore, we plan to explore alternative solutions as part of our future work.

Performance Evaluation. To assess the performance of our defensive approach, we performed two benchmarks on both the standard Linux kernel and our defense-embedded kernel on a bare-metal machine (equipped with an Intel 4-Core CPU, 16GB RAM, and 1000GB HHD). Our benchmarks include a micro-benchmark from LMbench v3.0 [101] and a macro-benchmark from Phoronix Test Suite [102]. LMbench measures syscall and system I/O latency and bandwidth, whereas the Phoronix Test Suite gauges the performance of real-world applications on both Linux kernels. For LMbench, we conducted the benchmark 10 times to minimize randomness and took the average as the observed performance. For the Phoronix Test Suite, we executed the test in batch mode, which runs the test 50 times and produces average values.

Table 3.3 presents our evaluation outcomes. Firstly, we note that our proposed method generally incurs minimal performance overhead, suggesting that our defense is lightweight. Secondly, we observe some moderate performance decline for the test cases – "10k File Create" and "10k File Delete" – in LMBench. As shown in Table 3.3, our proposed defense introduces an overhead of over 4%. The performance decrease is due to file objects being allocated to the virtual memory region via vmalloc, rather than the standard memory region via kmalloc. Compared to kmalloc, vmalloc is somewhat slower as virtual memory must remap the buffer space into a virtually contiguous

range, whereas `kmalloc` doesn't require remapping.

It's worth noting that file deletion results in a smaller performance reduction than file creation (4.25% vs. 7.17%). The disparity arises because the freeing of file objects is handled by RCU, which operates asynchronously to the file deletion process. While the moderate overhead might concern some production systems, it significantly enhances kernel protection against DirtyCred. In this work, our primary goal is to raise awareness within the Linux community rather than to develop a secure, efficient defense solution. We reserve the exploration of alternate defense solutions for future research. Lastly, it should be mentioned that some cases showed a slight performance improvement after introducing defense to the Linux kernel. This is primarily due to experimental noise, although we endeavored to reduce noise as much as possible by running the benchmark multiple times and disabling CPU boost on the bare-metal machine.

3.9 Discussion and Future Work

In this section, we address some other issues not yet discussed and outline potential directions for future research.

3.9.1 Escaping Containers

DirtyCred can potentially enable both passive and active container escapes in addition to facilitating privilege escalation on Linux. As previously explained, DirtyCred exploits vulnerabilities by swapping either file or credential objects. By utilizing file objects, DirtyCred can overwrite a file with high-level privileges. However, no file within a container allows the privilege to switch the namespace. Recent research [103] has shown a potential solution to this challenge. An attacker could passively wait for the runC process and execute root commands on the host by overwriting this process. Inspired by this approach, DirtyCred could use the file object swapping mechanism

to overwrite the runC process and successfully escape the container.

On the other hand, using credential objects to enable container escape eliminates the need for passive waiting. DirtyCred could trigger a Linux kernel vulnerability, swap credential objects, and thereby escalate the attacker's privilege to `SYS_ADMIN`. With the `SYS_ADMIN` privilege, the attacker could employ a previously proposed docker escape method [104], which involves mounting a cgroup and using the `notify_no_release` mechanism to execute root commands on the host system. To showcase DirtyCred's potential for container escape, we provide a working exploit at [94]. Reviewers can download this exploit for more details on docker escape.

3.9.2 Android Rooting

Beyond container escape, DirtyCred also holds the potential for Android rooting. The Android kernel, based on the generic Linux kernel, is notoriously more challenging to exploit due to its stricter access controls and newly deployed defenses [84]. Nevertheless, DirtyCred can root Android using either of the two attack pathways discussed in this paper. It can directly swap task credentials, providing attackers with privileged task credentials and thus root privilege. Alternatively, DirtyCred could initially use its file manipulation capability to overwrite a shared system library, allowing for privilege escalation from a restricted sandbox. Then, it could overwrite kernel modules with malicious code to achieve arbitrary read/write and ultimately disable SELinux on Android. We demonstrated DirtyCred's capacity to root Android using zero-day vulnerabilities. At the time of writing this paper, we reported these vulnerabilities to Google, who acknowledged our findings.

3.9.3 Cross Version / Architecture Exploitation

Crafting an exploit with the guidance of DirtyCred can yield an exploit code that works across various kernel versions or architectures without any modification. This compatibility arises because DirtyCred doesn't need to handle Kernel Address Space Layout Randomization (KASLR), unlike other exploitation methods that require a leak of the kernel base address to bypass KASLR. As such, the exploitation code doesn't include any data specific to kernel versions or underlying architectures. Furthermore, many prior kernel exploitation methods (e.g., KEPLER [105]) heavily depend on Return-oriented Programming (ROP) for privilege escalation. Migrating these exploits to a different architecture necessitates modifying the ROP chain to maintain exploitability. As discussed throughout this paper, DirtyCred doesn't use any architecture-specific data. Therefore, once an exploit code is developed for a vulnerability, it should work on other vulnerable kernels, regardless of their versions or underlying architectures.

3.9.4 Alternative Approaches for Capability Pivoting

In Section 3.4, we presented several techniques for pivoting a memory corruption capability to a capability useful for DirtyCred. During our evaluation, we discovered that vulnerabilities occurring in virtual memory are more challenging to exploit with DirtyCred. This difficulty arises from the limited number of exploitable objects in virtual memory, which restricts the pivoting of capabilities from the original memory corruption to those useful for DirtyCred. However, this does not imply that vulnerabilities in virtual memory cannot be exploited with DirtyCred. For instance, CVE-2021-34866 exhibits an out-of-bound capability that demonstrates memory overwrite on vmalloc'ed memory. Although our pivoting approach cannot utilize this capability to deallocate a credential object, a recent writeup [106] illustrates a sophisticated method that converts this overwrite capability on vmalloc into an arbitrary read and write, enabling a double-free capability.

As discussed and demonstrated in Section 3.4.2, leveraging a double-free capability significantly increases the likelihood of successful privilege escalation by DirtyCred. Apart from capability pivoting, recent research [20] has introduced an approach to explore different capabilities of a vulnerability. We argue that these methods complement the DirtyCred attack, as demonstrated in Section 3.3.1, where DirtyCred can still be launched without pivoting capabilities. We leave the exploration of other pivoting methods for future research.

3.9.5 Exploitation Stability

Similar to other kernel exploitation methods, DirtyCred's exploitation stability can be influenced by two critical factors. First, when pivoting vulnerability capabilities, DirtyCred manipulates the memory layout to take control of the target memory location. The exploitation stability may vary if the memory layout manipulation is affected by system activities. Second, the stability of exploitation is also impacted by the way the kernel vulnerability is triggered. To enhance exploitation stability, recent work [107] has proposed a series of methods to stabilize kernel exploitation. In our research, our goal is to assess the exploitability of DirtyCred in real-world scenarios. We conclude that DirtyCred can successfully exploit a vulnerability as long as it demonstrates exploitability. In the future, we plan to explore the utilization of existing exploitation stabilization techniques to improve the success rate of DirtyCred's exploitation.

3.9.6 TOCTOU

As discussed earlier, DirtyCred swaps credential objects within a critical time window. It is reasonable to consider that existing TOCTOU defense mechanisms may hinder our proposed exploitation method. A recent research article [108] categorizes TOCTOU defense into source code detection, postmortem detection, system call interposition, intra/inter-process memory consistency, transac-

tional system calls, and sandbox filesystem. Source code detection analyzes the source code of the target program, which cannot be applied to defend against our exploitation since DirtyCred does not exhibit any identifiable source code patterns. Postmortem detection detects TOCTOU vulnerabilities after the attack has been executed, which does not impact our exploitation since we employ unexpected free operations that are not visible during the analysis process. System call interposition monitors the system call sequence to detect attacks, but our exploitation method does not employ a malicious system call sequence, making system call interposition ineffective against DirtyCred. Intra/inter-process memory consistency protects shared variables in multiple threads by recording operations on the variables, but our exploitation involves unexpected operations that cannot be recorded. Transactional system calls and sandbox filesystem focus on the race condition between file read and write, which is not required by our method.

3.10 Related Work

This research presents a novel method for kernel exploitation and proposes a corresponding defense mechanism to mitigate the associated threat. Therefore, the most relevant existing works are in the areas of kernel exploitation and kernel exploitation mitigation. In the following, we summarize the works in these two domains and highlight the distinctions from our proposed techniques.

Kernel Exploitation. Over time, kernel exploitation techniques have evolved alongside the development of kernel defenses. Before the introduction of Supervisor Mode Execution Prevention (SMEP) [90], the ret2usr technique [109] exploited the Linux kernel by pivoting kernel execution to the userland. However, with the widespread deployment of SMEP in Linux, this technique became ineffective since SMEP prevents kernel execution in userspace. Subsequently, Supervisor Mode Access Prevention (SMAP) [89] was proposed to further enhance the separation between the kernel and userspace by blocking direct userspace access. To bypass the protections enforced

by SMEP/SMAP, researchers introduced new exploitation methods. For example, the ret2dir technique [110] by Kemerlis et al. mirrored userspace data within the kernel address space, while KEPLER [105] by Wu et al. utilized a specific kernel code gadget to enable a long ROP chain by transforming the PC control through stack overflow.

To counter Return-Oriented Programming (ROP) attacks against the Linux kernel, KASLR (Kernel Address Space Layout Randomization) was introduced to increase the difficulty of exploitation by randomizing the kernel memory address layout. However, security experts devised practical methods [111]–[114] to circumvent KASLR. For instance, elastic objects in the kernel were utilized, as demonstrated by ELOISE [114], which disclosed sensitive kernel information by overwriting the length field of elastic objects. Additionally, Gruss et al. proposed a hardware side-channel attack leveraging pre-fetch instructions to bypass KASLR. Recent advancements even exploited vulnerabilities in processors, such as Meltdown [115] and Spectre [116], to bypass KASLR's protection on Linux. Furthermore, researchers proposed techniques to randomize the heap memory layout in the Linux kernel [117], [118], making heap layout manipulation more challenging for adversaries. However, Xu et al. introduced a memory collision technique [97] that utilized the memory reuse mechanism to exploit kernel use-after-free vulnerabilities without being hindered by heap randomization. In contrast to the aforementioned exploitation techniques, our work focuses on end-to-end exploitation without the need to bypass widely deployed kernel defenses. Unlike specific kernel protections or exploit mitigations, our method is more general and consequential. As discussed in Section 3.9, DirtyCred can even facilitate the ability to escape containers and root Android devices.

Kernel Defense. In addition to the kernel defenses introduced together with the existing exploitation methods, there are various other kernel protection and exploit mitigation mechanisms proposed by academia and industry, which have received significant attention from the security com-

munity. Here, we provide a brief overview of some recently proposed or widely adopted defenses.

To counter side-channel attacks against the Linux kernel, Gruss et al. proposed KAISER [119], a strict kernel and userspace isolation mechanism that ensures hardware does not hold any information about kernel addresses while running in user mode. To improve KASLR, Function Granular Kernel Address Space Layout Randomization (FGKASLR) [120] was introduced, randomizing the layout down to the code function level, thereby increasing the difficulty of code-reuse attacks. In the realm of control-flow hijacking prevention, researchers proposed various defense mechanisms to enforce control-flow integrity in the Linux kernel [91], [121]–[123]. For example, Yoo et al. implemented an in-kernel control-flow integrity protection using ARM's Pointer Authentication [123].

Moreover, several defense techniques have focused on protecting critical kernel data [124]–[128]. AUTOSLAB [95] and xMP [129] are examples of such kernel defenses. AUTOSLAB isolates different types of objects into different memory caches, reducing the objects available for kernel heap memory manipulation. On the other hand, xMP employs virtualization techniques to isolate sensitive data, preventing unauthorized tampering.

In terms of defense philosophy, our defense mechanism differs from works that safeguard kernel control-flow integrity. While it shares similarities with defenses that isolate critical kernel data, our defense is distinct from them in technical implementation. Rather than isolating objects based on their types or sensitivity, our defense performs memory isolation based on the privilege of kernel objects. As a result, it is more effective in defending against the threat posed by DirtyCred.

3.11 Conclusion

The Linux kernel incorporates various protection mechanisms and exploit mitigation techniques, making successful kernel exploitation challenging. To bypass these defenses, attackers must lever-

age a vulnerability with strong capabilities to circumvent protection and mitigation measures. In this work, we introduce a novel exploitation method called DirtyCred. DirtyCred enables exploitation and defense circumvention without relying on a specific kernel vulnerability. Through DirtyCred, we demonstrate that an attacker can utilize almost any heap-based kernel vulnerability to swap credential objects. This credential swap confuses the Linux kernel, leading it to treat a highly privileged file or task as if it were in a lower privileged mode. Consequently, an unprivileged user can escalate their privileges for malicious purposes. Based on our findings, we conclude that DirtyCred undermines the existing defense architecture of Linux, and if immediate action is not taken by the security community, Linux-driven systems will be at risk, potentially causing significant harm to their users. In light of this conclusion, we propose a defense mechanism in the form of privilege-object isolation. As part of our research, we have implemented a Linux kernel prototype that incorporates this defense mechanism. Our experiments demonstrate that it effectively enhances the security of Linux at a minimal to moderate cost. With this new discovery, we further emphasize the necessity of isolating memory based on object privilege to defend against the threat of DirtyCred. We have shared some of our research findings with software and hardware vendors that may be affected by DirtyCred. We are actively collaborating with them to ensure they understand and can mitigate this threat.

CHAPTER 4
CAMP: COMPILER AND ALLOCATOR-BASED HEAP MEMORY PROTECTION

4.1 Introduction

The heap, a region of computer memory dynamically allocated during runtime, is extensively utilized for dynamic memory allocation and the housing of variable-sized data structures. Given its frequent usage and intricate nature, the heap is especially susceptible to spatial and temporal memory errors.

Numerous heap protection methodologies have been deployed over the years, aiming to bolster the security of the heap. These include advancements in heap management algorithms [130] and the incorporation of layout randomization techniques [131], [132]. While these developments have bolstered the heap's security, the ongoing discovery and progression of new vulnerabilities and exploitation techniques [10], [71], [133]–[135] signify that heap exploitation is still a pertinent issue.

We propose that a key solution to efficiently combat heap exploitation is to detect and prevent heap memory corruption. Previous research endeavours such as Memcheck [136] and Address Sanitizer [137] targeted comprehensive protection, whereas others mitigated use-after-free errors [130], [138]–[143] or detected out-of-bound access [144]–[146] for partial protection. However, these solutions either introduced significant runtime overheads, like the 26x overhead introduced by MemCheck [136] in Valgrind, or provided restricted protection, such as FFmalloc [142] which only safeguards against certain use-after-free vulnerabilities.

In this work, we present `CAMP` (Compiler and Allocator-based Heap Memory Protection), an

innovative heap sanitizer for the detection of spatial and temporal heap errors. Unlike other studies [145]–[152] which require hardware support, CAMP is purely software-based, comprised of a compiler and a seglist allocator. The compiler instruments the target program to verify the pointer boundary and establish point-to relation at runtime. The custom memory allocator monitors memory ranges for each allocation, supporting the instrumented instructions, and neutralizing dangling pointers when a memory object is released. This work showcases an innovative amalgamation of a compiler and seglist allocator to achieve comprehensive heap protection with minimal runtime overhead.

The uniqueness of CAMP stems from its error detection scheme which affords significant optimization potential. Specifically, CAMP manages metadata within the allocator, enabling its runtime to present a significantly lower complexity of $O(1)$ for each pointer invalidation as opposed to existing defense solutions [138] with a complexity of $O(logN)$. Moreover, CAMP integrates three compiler optimizations that reduce runtime overhead by limiting the number of instrumentations, without compromising security. As we will detail in Section 4.4, these optimizations involve the elimination of unnecessary (and/or redundant) instrumentations and the consolidation of runtime boundary checks. As will be shown in Section 4.6, optimization plays a critical role in curtailing the runtime overhead.

This paper offers the following contributions:

- We introduce a novel approach, CAMP, which employs a customized allocator and a compiler to safeguard against heap memory corruption. Additionally, we propose optimization strategies aimed at reducing the performance overhead introduced by the instrumentation.
- We implement CAMP by customizing a segregated list allocator – tcmalloc and building our instrumentation optimization mechanism on top of the LLVM 12.0 compiler framework. Upon

the acceptance of this paper, we intend to make CAMP accessible to the community by open-sourcing it at [153].

- We carry out an exhaustive evaluation of CAMP using the real-world application Nginx, as well as the SPEC CPU 2006 and 2017 benchmarks, from both security and runtime overhead perspectives. This evaluation compares CAMP's performance with other defense solutions offering comparable levels of heap protection.

The remainder of the paper is structured as follows. Section 4.2 provides a background on memory corruption on the heap as well as heap allocators. Section 4.3 outlines the assumptions of our research and the threat model. Section 4.4 discusses the details of the proposed techniques. Section 4.5 outlines our implementation details. Section 4.6 evaluates the security and runtime overhead of our proposed techniques. Section 4.7 delves into a discussion of some related issues, followed by the related work in Section 4.8. We conclude the work in Section 4.9.

4.2 Background

This section will explore the context and characteristics of various heap allocators. We will also familiarize ourselves with two categories of heap memory corruption errors - out-of-bound and use-after-free.

4.2.1 Corruption and Protection of Heap Memory

Heap memory corruption primarily takes two forms: overflow and use-after-free. We will delve into the specifics of these two types of corruption and outline the methods employed for their protection.

Heap Overflow. Every heap object has its designated memory space. A heap overflow situation arises when the usage of a heap object surpasses its allocated memory capacity. The conventional method of detecting heap overflow involves setting aside some memory as heap cookies or red zones. A heap overflow can be detected if the magic value in the designated area is modified [154]. This approach, however, is not foolproof and can be circumvented. For instance, attackers can undermine the detection process by leaking the heap cookie [155] or overflowing the memory while keeping the red zone untouched [156]. Another tactic involves validating pointers to ensure there's no out-of-bound access [145], [146], [157]. While effective, this approach often leads to notable overhead [144].

Heap Use-After-Free. The issue of use-after-free arises when the memory space of a heap object is released, leaving behind references to the object, also known as dangling pointers. The program should avoid deferring to these dangling pointers to prevent a use-after-free situation. Several techniques have been suggested to detect use-after-free scenarios. ASAN [137], for instance, uses shadow memory to log memory status and inserts instruments into every memory access. By checking the shadow memory, access to a freed object can be detected instantly. Despite ASAN's method only resulting in reasonable overhead, its security assurance is relatively weak as attackers can alter the shadow status by reallocating the freed object. More effective strategies include never reusing freed memory [142] or postponing the freeing of memory [158], which prevents attackers from tampering with freed objects. Additionally, nullifying all existing references once an object is freed [138], [139], [159] can effectively thwart use-after-free scenarios.

4.2.2 Heap Memory Allocators

Heap memory allocators are essential for the dynamic management of ”global” memory. The effectiveness of a memory allocator is determined by its speed of memory allocation/deallocation and the minimal waste it produces. As such, various memory allocation algorithms have been developed. Here, we discuss the three most commonly used types of memory allocators.

The first type is the sequential-fit allocator. This allocator typically uses a freelist linking all the freed memory objects. Upon a request for memory allocation, the allocator scans through the freelist until it identifies a freed object with sufficient memory space. If the freed memory object is larger than required, the allocator divides the memory and reinserts the surplus back into the freelist. Sequential-fit allocators often merge neighboring freed objects to prevent fragmentation.

The second type is the Segregated List allocator (seglist allocator), which utilizes an array of freelists, each containing freed objects of the same size. During memory allocation, the seglist allocator locates the freelist of the requested size to find an appropriately sized object. Deallocating an object requires identifying its size to determine the freelist it belongs to. While the seglist allocator requires finding the corresponding freelist during allocation and deallocation, it eliminates the need for splitting and merging memory, unlike the sequential-fit allocator.

The third type of allocator, the buddy system allocator, functions similarly to the seglist allocator by maintaining different freelists for varying sizes of memory objects. However, a unique feature is that if the freelist of the requested allocation size is empty, the allocator will partition a larger object to meet the allocation requirement. Also, deallocating an object allows it to recombine the remaining portion back into a larger object.

```c
1 void main() {
2     char *buf = malloc(16);
3     buf[32] = 'x';
4     free(buf);
5     buf[0] = 'y';
6 }
```

Listing 10: A toy vulnerable example.

```c
1 void main() {
2     char *buf = malloc(16);
3     __escape(&buf, buf);
4     __check_range(buf, &buf[32], sizeof(char));
5     buf[32] = 'x';
6     // free buf, which neutralizes the dangling pointer stored in &buf
7     free(buf);
8     __check_range(buf, &buf[1], sizeof(char));
9     buf[0] = 'y';
10 }
```

Listing 11: The toy program with CAMP's protection.

4.3 Assumptions & Threat Model

CAMP is primarily centered around the detection of heap errors, encompassing both spatial and temporal heap errors. We operate under the assumption that the target program, coded in a low-level language and compiled by CAMP, harbors at least one heap-based memory vulnerability. Given that our endeavor is concentrated on securing userspace applications, the defense of lower-level kernel security falls outside our purview. In our envisioned threat landscape, we consider an attacker who is aware of CAMP's deployment, has access to the heap vulnerability, and aims to exploit this flaw for privilege escalation.

4.4 CAMP

In this section, we first illustrate CAMP's protective mechanism with an example of a vulnerable program and then delve into the specifics of its design.

4.4.1 An Toy Vulnerable Program

List 10 demonstrates a sample program that harbors two heap memory corruption vulnerabilities. The program, in particular, allocates 16 bytes of memory (line 2), then accesses the memory object at index 32, which trespasses the memory range boundary (line 3), thereby leading to a heap memory overflow. It should be noted that this heap overflow remains undetected by ASAN due to the red zone being overlooked by the overflow. In line 4, the memory object is deallocated, rendering the pointer `buf` a dangling pointer. Subsequently, the dereferencing of this dangling pointer (line 5) results in a use-after-free memory corruption.

4.4.2 CAMP's Protection Mechanism

In essence, `CAMP` fortifies the program by instrumenting it to detect memory corruption and deter exploitation. List 11 demonstrates the fortified version of the toy program. Following this, we detail how `CAMP` secures the vulnerable toy program.

Pointer Validation. `CAMP` identifies heap overflow by affirming the result pointers from pointer arithmetic, ensuring that no out-of-bound pointers are generated. This is achieved by incorporating a check instruction at the point of the pointer arithmetic to prompt the runtime and verify that the result pointer lies within the base pointer's range. As demonstrated in List 11, pointers `&buf[32]` and `&buf[0]` are derived from the buffer `buf` in lines 5 and 9, respectively. `CAMP` autonomously integrates check instructions in lines 4 and 8 to validate these pointers. Should the runtime detection ascertain that `&buf[32]` is an out-of-bound pointer, `CAMP` will terminate the execution to preclude exploitation. It's worth noting that this query necessitates that `CAMP` continuously maintains a record of each memory allocation and its corresponding memory range, which is noted during runtime for each heap allocation (line 2).

Neutralization of Dangling Pointers. CAMP deters use-after-free by neutralizing dangling pointers. During runtime, CAMP constructs the point-to relation by instrumenting the program. When a memory object is freed, CAMP can refer to the established point-to relation and identify the dangling pointers to the freed memory. By neutralizing these dangling pointers, access to use-after-free is effectively rendered impossible. Copying pointers (i.e., *pointer escapes* [160]) is tracked to construct the point-to relation. For instance, the program copies the heap pointer to the variable `buf` (line 2 of List 11), which qualifies as a pointer escape, and CAMP incorporates an escape tracking instruction after that (line 3). This escape tracking instruction inputs the address of the variable and the pointer, noting which address holds a reference to the memory allocation. Following this, the program frees the memory (line 7). Inside the free, CAMP identifies the existing dangling pointers to the freed memory and neutralizes them as non-congenial. Consequently, the variable `buf` no longer references the freed memory, and the program crashes when it is dereferenced (line 9).

4.4.3 Design Overview

Figure 4.1 depicts the principal components of CAMP, which includes a compiler and a memory allocator. The CAMP compiler, constructed on the basis of LLVM, processes the source code to produce binaries associated with the CAMP allocator. During the compilation stage, the compiler initially transforms the source code into LLVM IR before incorporating range checking and escape tracking instruction, this defends against heap memory corruption. Post this, the compiler enforces several unique compiler optimizations to minimize the protective overhead, all while ensuring security isn't compromised. When in operation, the CAMP allocator addresses heap memory allocation and deallocation requests. Furthermore, it provides support for the instrumented instructions. In particular, it keeps track of the memory range for each allocation, enabling the allocator to validate pointer bound information for every range checking query. It also manages the escape

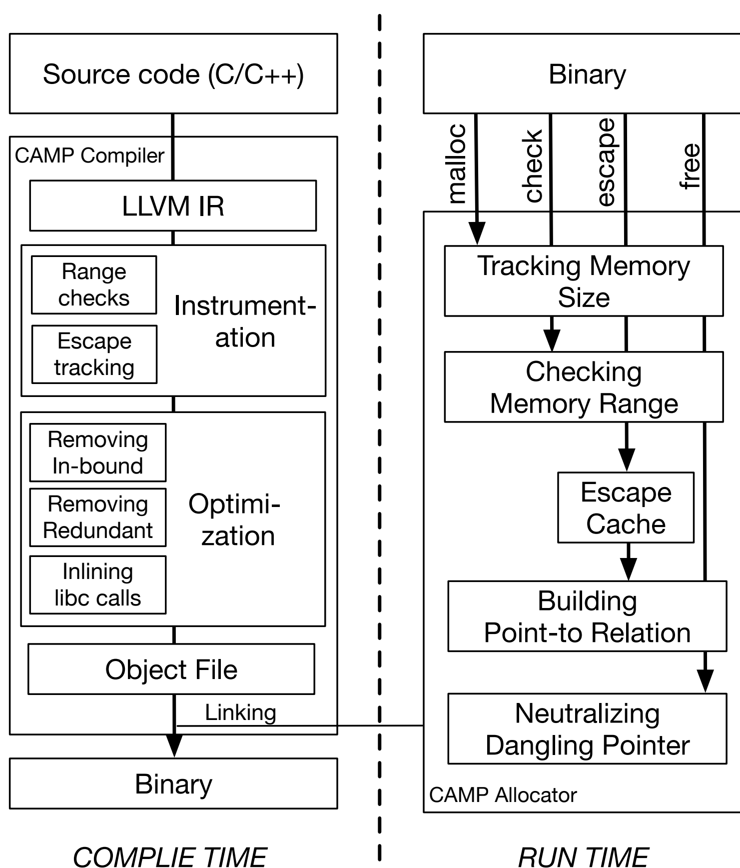

Figure 4.1: The design overview of CAMP.

tracking instruction to construct the point-to relationship. Thus, whenever a memory object is released, it is capable of neutralizing the corresponding dangling pointers, blocking the UAF access from such pointers.

4.4.4 Instrumentation by the Compiler

Range Checking Instrumentation. Given that our program model doesn't permit casting integers to pointers, all initial pointers come from explicit memory allocation (like malloc), or from addressing global or stack variables. [144] These pointers are then utilized through *pointer arithmetic* to generate new pointers, which are used to access memory. The function of `CAMP`'s range checking is to confirm that all pointers following arithmetic remain within bounds, thereby avert-

ing heap overflow. As shown in List 10, the range checking considers three arguments, namely the base pointer src, the result pointer dst of the arithmetic, and the type size $size$ of dst. The run-time will verify that the memory ranging from dst to $dst + size$ falls within src's memory range.

It's worth noting that CAMP exclusively protects heap memory, so validating non-heap memory pointers is unnecessary. To avoid needless validation, CAMP carries out dataflow and alias analysis on the LLVM IR to ascertain the point-to relationship of pointers. If a pointer is confirmed not to point to the heap at compile time, CAMP won't conduct range checking for it.

Escape Tracking Instrumentation. The escape tracking insertion allows CAMP to construct the point-to relationship of memory objects in real-time. We adopt a similar approach as presented in CARAT [160], [161] and DangNull [138] to introduce tracking after *pointer escapes* (i.e., copying pointers). As illustrated in Section 4.4.2, the tracking considers the copied pointer and its stored address as arguments. Unlike CARAT and DangNull, which instrument all potential pointer escapes, CAMP omits pointer escapes if the pointer is determined not to reference the heap during compilation. Given that CAMP's aim is to prevent heap memory errors, bypassing non-heap point-to relations doesn't risk security but contributes to better CAMP performance.

4.4.5 Runtime Support

CAMP's runtime offers foundational support for its instrumentation. The execution speed of the inserted instruction is vital to CAMP's overall performance. Suppose a program allocates m memory segments, and CAMP maintains a record of the allocated memory in a linked list. In the worst-case scenario, a single range checking operation could result in $O(m)$ time complexity, which could impose insurmountable runtime overhead. Similarly, a naive design that records n pointer escapes

in a linked list could introduce $O(n)$ time complexity when releasing an object. Next, we'll discuss how our design works in harmony with the allocator to provide fast runtime support.

Seglist Allocator. As stated in Section 4.2, a Seglist Allocator uses different free lists for varying sizes of memory objects. To free a memory object, the allocator must locate the free list of its size and insert the object into it. To ensure quick allocation and deallocation, the time complexity of finding the free list is designed to be constant. For instance, tcmalloc [162], a Segregated List Allocator developed by Google, employs *span* as the basic memory management unit. Each span handles a size class of memory objects across several continuous memory pages. The spans are kept in a page table where the page serves as the key. Whenever a memory object is allocated or deallocated, tcmalloc can locate its span, and then retrieve the freelist with constant time complexity.

The CAMP allocator uses the design of the Segregated List Allocator to provide quick runtime support. For each span, CAMP records the size of memory objects as one of the metadata. As the seglist allocator divides the memory page equally into objects, given a heap pointer denoted as ptr, we can first calculate the object's index with:

$idx = (ptr - page_base)/size$

Here, idx is the object index to the span, $page_base$ signifies the starting address of the page in the span, and $size$ refers to the size of the object. With this information, the lower boundary of the memory range can be identified as $page_base + idx * size$ and the upper boundary as $page_base + (idx + 1) * size$. This direct approach results in constant time complexity for pointer validation.

Maintaining Point-to Relation. CAMP varies from DangNull [138] and CARAT [160], [161] in how it encodes the point-to relationship. While DangNull and CARAT utilize a red-black tree

```c
1 struct obj {
2     int a;
3     int b;
4 };
5 struct obj* bar() {
6     // type-casting from void* to obj*
7     struct obj *o = malloc(sizeof(struct obj));
8     __check_range(o, o, sizeof(struct obj));
9     ...
10 }
11 int foo(struct obj *ptr) {
12     __check_range(ptr, &ptr->a, sizeof(ptr->a));
13     ptr->a = 1;
14     __check_range(ptr, &ptr->b, sizeof(ptr->b));
15     ptr->b = 2;
16 }
```

Listing 12: An example of optimizing structure field access checks.

structure, optimizing the time complexity of locating a relation to $O(logN)$, CAMP integrates the point-to information into the seglist allocator, optimizing cost to constant time complexity. Specifically, CAMP's seglist allocator maintains an escape table for each span, which is a map of object indices to their escape lists. The escape lists are linked-list structures that chain corresponding escapes to their assigned objects. When an escape tracking call is made, CAMP calculates the memory object's index in the span, retrieves its escape list from the table, and inserts a record into the list. When a memory object is freed, CAMP's allocator checks its escape list and neutralizes all the existing dangling pointers to the memory object.

To further enhance the overall performance of CAMP, we devise a cache mechanism for maintaining the point-to relationship. New point-to relations are temporarily stored in the cache until it becomes full, at which point the records are transferred to the allocator in a batch, while omitting any duplicates. This cache design enhances runtime speed and reduces memory overhead, especially in situations where the program operates repeatedly in the same block and creates similar point-to relations.

```
Input: A function F ;
Output: A set of pointer to be validated S ;
Initialize: NewPointerSet = getNewPointerSet{F} ;
In = Out = changeSet = dict() ;
foreach ptr ∈ NewPointerSet do
    if pointerMapbase[base(ptr)] == NULL then
        pointerMapbase[base(ptr)] = set() ;
    end
    pointerMap[base(ptr)].add(ptr) ;
end
In = pointerMap ;
changeSet = In − Out ;
while changeSet ≠ ∅ do
    foreach key, val ∈ In do
        if ∃ p, p' ∈ val and RedundantPair(p, p') then
            val.remove(p') ;
            val.p.offset = MAX(p.offset, p'.offset) ;
            Out[key] = val ;
            break;
        end
        else
            Out[key] = val ;
        end
    end
    changeSet = In − Out ;
    In = Out ;
end
foreach key, val ∈ Out do
    S.add(val) ;
end
```

Algorithm 1: Removing Redundant Validation

4.4.6 Compilation Optimization

CAMP is distinct in that it permits considerable optimization possibilities during the compilation phase. This attribute can considerably enhance performance while maintaining security. Here, we elucidate three optimization algorithms conceived specifically for CAMP.

Type Information-aided Optimization of Range Checks. A rudimentary strategy to prevent out-of-bound access is to implement range checks for each pointer arithmetic operation, ensuring

pointers do not trespass memory boundaries. Take for instance, the function `foo` in List 12. Here, the variable `ptr` on lines 13 and 15 undergoes two pointer arithmetics, and to maintain security, CAMP needs to insert range checks for the result pointers (lines 12 and 14). An enhanced strategy involves eliminating validation if the compiler identifies the pointer as in-bound, thereby elevating performance. However, compilers usually lack data concerning a pointer's memory range, which makes it difficult to ascertain which pointers are in bounds.

To implement the aforementioned enhancement, we use type information during the compilation process to discern the memory range of a pointer. This process involves validating not only pointer arithmetic but also type-casting operations to ascertain that the memory space pointed to by a typed pointer is sufficient for its respective type.

As an illustration, refer to the code in List 12, where function `bar` allocates a new object `obj` (line 7). The return type of `malloc` is `void*`, not `struct obj*`, leading the compiler to introduce a type-casting instruction. Following this, CAMP inserts a range check to confirm the memory space can hold the `obj` structure (line 8). With this type-casting validation, the compiler can confidently deduce that the memory space of a typed pointer is at least its type size. Consequently, the compiler can assert that pointers referring to the structure field are within bounds. As a result, CAMP can omit the range checks on lines 12 and 14. Moreover, the compiler can ensure the memory for pointer `o` (line 6) is at least the size of its type, allowing the range check on line 7 to be optimized and removed.

This optimization strategy is made viable due to CAMP's distinctive approach to security checks, which removes the risk of accessing a dangling pointer such as `ptr` in function `foo`. This allows for robust optimization of range checks for field pointers, without the fear of triggering a use-after-free vulnerability. As we will demonstrate in Section 4.6, the above-mentioned optimization technique greatly enhances CAMP's runtime performance, particularly for programs containing

```
1 struct obj {
2     char *mem;
3 };
4 void foo(struct obj *ptr, bool flag) {
5     __check_range(ptr->mem, &ptr->mem[0x100], 1);
6     ptr->mem[0x100] = 'x';
7     if (flag) {
8         ptr->mem[0x30] = 'y';
9         ptr->mem[0x1] = 'y';
10    }
11    ptr->mem[0x1] = 'z';
12    ptr->mem++;
13 }
```

Listing 13: Example codes of applying eliminating redundant optimization.

type-associated pointers. It's important to note that we do not employ the above optimization for pointers lacking type information (e.g., `void*`, `char *`) or when their type sizes cannot be determined during compilation (e.g., elastic objects [135]).

This optimization is feasible due to `CAMP`'s innovative design of security checks. The security design guarantees the absence of dangling pointers at runtime, and each typed pointer possesses sufficient memory for its structure type. Therefore, `CAMP` can aggressively optimize range checks for its field pointers without the worry of invoking a use-after-free or out-of-bound. ASAN's design cannot make this assurance; hence optimizing access to a dangling pointer could lead to a use-after-free.

Eliminating Redundant Instructions. In the context of this work, a superfluous instruction pertains to a range check that validates already validated pointers or an escape tracking that constructs a point-to relation already noted. The opportunity for optimization here lies in eliminating these unnecessary instructions to enhance performance.

Common sense suggests that if two range checks are validating the same result pointers from pointer arithmetic, one can be done away with. As an example, in List 13, the pointer `&ptr->mem[1]` on lines 9 and 11 are aliases. Hence, we could simply exclude the validation for `&ptr->mem[1]` on

line 9. Furthermore, `CAMP`'s security design allows for the merging of multiple pointer validations into a single validation. Initially, `CAMP` is required to validate the result pointers `&ptr->mem[0x100]` (line 6), `&ptr->mem[0x30]` (line 8), and `&ptr->mem[0x1]` (line 11) individually. However, if the pointer `&` ↪ `ptr->mem[0x100]` is within bounds, the other two pointers are inevitably within bounds as well. Consequently, we could do away with their validations and shift the validation of `&ptr->mem[0x100]` to line 5, as displayed in List 13. Formally, given two result pointers $ptr1$ and $ptr2$ from the same base pointer ptr, their validation can be consolidated if the following function returns True.

function REDUNDANTPAIR($ptr1, ptr2$)
  if *ptr1.offset ¿= ptr2.offset* **then**
   if *dominate(ptr1, ptr2)* ***or***
    post-dominate(ptr1, ptr2) **then**
     return True
  return False
end function

The $offset$ in the first condition symbolizes the maximum access offset from the base pointer. Hence, this condition mandates that $offset$ should be within the range of $[ptr, ptr1]$. The second condition ensures validation redundancy, where the two validations will be performed together. We utilize Algorithm 1 to eliminate redundant pointer validation. The algorithm takes a function F as input and outputs a set of pointers to be validated S. Initially, all the result pointers are gathered (line 3) and sorted into a map based on their base pointers (lines 5 to 8). The map's key is the base pointer, and the value is the set of result pointers. Then, we apply the fixed-point algorithm [163] for optimization. In each iteration, we traverse through the map's elements (line 12). If we identify two pointers that meet the redundancy condition (line 13), we remove the latter one (line 14) and update the remaining one's offset with their maximum value (line

15). Subsequently, we update the iteration output (line 16) and exit the loop to commence the next iteration. Note that if no redundancy pair is discovered, the output will be the same as the iteration input (line 19). When there is no redundancy pair left, this signifies that the fixed point has been reached. Then, we collect the remaining pointers from the out to S (lines 22 to 23). List 13 demonstrates `CAMP`'s instrumentation after implementing this optimization, where only the validation on line 5 is retained, but the security guarantee remains intact. Note that applying this optimization to redzone-based protection could easily result in false negatives, as the out-of-bound access may be removed, and the subsequent access may bypass the redzone, thereby escaping detection.

In line 12 of List 13, the pointer `ptr->mem` is self-updated. This statement involves three operations when broken down. First, the program retrieves the pointer stored at the address `&ptr->mem`. Then, it generates a new pointer based on the retrieved pointer, inserting a range check in this process. This check ensures that the newly created pointer refers to the same memory object as the old one. The new pointer is then copied into the address `&ptr->mem`, after which `CAMP` inserts an escape tracking for the pointer copying. A key observation is that the address `&ptr->mem` should have been initialized earlier, meaning such a point-to relation should have been recorded. Because the memory object of the new pointer doesn't change, recording the same point-to relation is redundant. Thus, we can optimize this escape tracking for performance without compromising precision. Our optimization strategy is to identify those escape pairs involved in self-updating.

Consolidating Runtime Calls. Ideally, for the same memory pointer (including aliases), the validation can be optimized into a single range check using the aforementioned method. However, if the pointer arithmetic is dynamic, i.e., the result of pointer arithmetic can't be determined at the compilation time, `CAMP` has to instrument separate range checks for them. List 14 depicts

```
1 void foo(char *ptr, int i, int j) {
2     unsigned int start, end;
3     __get_range(ptr, &start, &end);
4     assert(&ptr[i]>=start && &ptr[i]+1<end);
5     ptr[i] = 'x';
6     assert(&ptr[j]>=start && &ptr[j]+1<end);
7     ptr[j] = 'y';
8 }
```

Listing 14: Example codes of applying merging runtime calls optimization.

such an example. Lines 5 and 7 access the same array `ptr` lacking type information. Furthermore, the access is dynamic, based on runtime values `i` and `j`, making the optimization of removing redundancy inapplicable. Consequently, `CAMP` must instrument lines 5 and 7 separately. Each time a range check is executed, `CAMP` must switch its context into the library, query the memory range, and validate the pointer. This process is time-intensive. An optimization strategy is to merge the range checks as they share the same base pointer. This could minimize the frequent context switching and save time on querying the memory range.

To implement this, we first follow the same method as in Algorithm 1 to construct a *pointer map*, where pointer arithmetic with the same base pointer is grouped together. Then, for each group, we traverse the function's CFG and find their nearest ***dominator*** instruction. Here, a range query is inserted to initialize the memory chunk's range variables. Subsequently, the original range check is replaced with an assertion to ensure the boundary. List 14 also presents the result after applying this strategy. Line 3 queries the memory range of `ptr` and initializes the memory ranges into variables `start` and `end`. After this, validation for `&ptr[i]` and `&ptr[j]` is accomplished through two assertions on lines 4 and 6.

4.5 Implementation Details

This section presents the specifics of our implementation for `CAMP`'s compiler and its allocator.

4.5.1 CAMP Compiler

The CAMP compiler, built on the LLVM 12.0.0 compiler framework, integrates instrumentation and optimization as an LLVM pass loadable via clang.

To arm CAMP with a defense mechanism against heap overflow, all pointer arithmetic instructions and type-casting instructions are instrumented. For pointer arithmetic, the compiler scans all bitcode, collecting the getelementptr instruction from LLVM IR - the sole pointer arithmetic instruction since CAMP prohibits integers casting to pointers. As discussed in Section 4.4.4, CAMP inserts a range checking instruction for each pointer arithmetic, taking three input arguments - the base pointer, the result pointer, and the size of the result pointer's type. In our implementation, the base pointer is the pointer operand of the getelementptr instruction, with the result pointer being the GEP's outcome value, and the third argument being the size of the result type. These arguments constitute a CALL instruction, inserted following the getelementptr instruction. To determine if the source operand references the heap, it is backtracked through LLVM's SSA to identify its source. If the source is found to be a stack or global variable, it is concluded that it does not refer to the heap, thus skipping such getelementptr instructions.

In addition to instrumenting getelementptr instruction, the bitcast instruction, which denotes type-casting in LLVM, is also instrumented. For each type-casting instruction, CAMP inserts a checking instruction with two input arguments - the result pointer and its type size. We employ the pointer operand as the result pointer and the size of the result pointer's type to guarantee sufficient memory for the object.

We employ CARAT's [160], [161] method for tracking escapes. Namely, CAMP compiler instruments store instructions if the type of its value operand is a pointer. The pointer and value operand are extracted from the store instruction and used to build an escape tracking CALL instruction, inserted before the store instruction. The aforementioned method is followed to deter-

mine if the escape is on heap memory. If the value from the store on the stack or global region is concluded, the instrumentation is skipped. Note that when implementing the redundancy removal optimization, loops are treated as a special case. We utilized LLVM's `LoopInfo` pass to identify the boundary of variables accessed within the loop, moving check instructions outside the loop.

4.5.2 CAMP Memory Allocator

The allocator is implemented as a shared library, replacing the default memory allocator and based on tcmalloc [162]. In what follows, we outline our significant modifications to support `CAMP`'s runtime checks.

Tcmalloc is a segregated list allocator that upholds a page table, mapping the page address to its `span`. Note that a span can manage several continuous memory pages for larger memory objects. The original tcmalloc records only the first and last page in the page map. To ensure any heap pointer can locate its `span`, we document every memory page used by tcmalloc in the page map. Each span holds two metadata pieces to support `CAMP`'s runtime checks. One represents the object size used for memory range checks, and the other is a reference to the escape pointer array containing chained escapes to the objects it maintains.

Tcmalloc preserves a page table, the *size class map*, mapping each page address to its size class. For every page, we compress its span's start address and its size class into an 8 bytes unit and map it to the *size class map*. Specifically, the lower 16 bits store the size class, and the remaining bits store the page address. With this design, each range checking by `CAMP` can retrieve the required data in constant time, rapidly validating the pointers. If a span isn't recorded in the size map[1], `CAMP` reverts to the original routine to retrieve the `span` and then retrieve the required address and size for pointer validation. It's worth noting that in certain cases, a typical program might use

[1] In tcmalloc, large objects have their unique spans, thus aren't recorded in the size map.

CWE (number)	Good Test (Selected/Total/Passed)	Bad Test (Selected/Total/Passed)
Buffer Overflow(122)	3870/3870/3870	2308/3870/2308
Double Free(415)	820/820/820	820/820/820
Use After Free(416)	394/394/394	288/288/288
Invalid Free(761)	288/288/288	288/288/288

Table 4.1: Security evaluation of CAMP on Juliet Test Suite.

out-of-bound pointers as memory boundaries. Such coding style is incompatible with `CAMP`'s protection design. To address this, we reserve one-byte memory at each memory allocation's end so that the out-of-bound pointer used as memory boundary will still be in-bound.

For each pointer escape, `CAMP` generates a record allocated by tcmalloc's metadata allocator. The record, containing the pointer's location, is stored in the span of the referenced object. Escape records for the same object are linked. When a memory object is freed, `CAMP` traverses its escape lists, checking if the pointer in the recorded location still references the object, and `CAMP` will neutralize the pointer. Along this process, all the records related to the object are freed. The cache mechanism implemented in `CAMP` is a temporal escape array. Each time a new escape track is invoked, `CAMP` scans the array to find if there is a matching record. If not, the new record is appended to the array. When the array is full, all the records are committed to the span, and the array is cleared. Note that the records in the cache are also examined at free to ensure all dangling pointers are neutralized.

4.6 Evaluation

This section primarily assesses `CAMP`'s proficiency in identifying heap overflow and the utilization of memory after it has been freed, utilizing a standard vulnerability benchmark and several real-world vulnerabilities. Subsequently, we delve into the detailed protective measures offered by

CVE/Issue ID	Application	Bug Type	CAMP	ASAN --	Memcheck	DangNull	MarkUs	Delta Pointer
CVE-2015-3205	libmimedir	Use-After-Free	✔	✔	✔	✔	✔	/
CVE-2015-2787	PHP 5.6.5	Use-After-Free	✔	✔	✔	✖	✖	/
CVE-2015-6835	PHP 5.4.44	Use-After-Free	✔	✔	✔	✔	✖	/
CVE-2016-5773	PHP 7.0.7	Use-After-Free	✔	✔	✔	✔	✖	/
Issue-3515 [164]	mruby	Use-After-Free	✔	✔	✔	Build Fail	✖	/
CVE-2020-6838	mruby	Use-After-Free	✔	✔	✔	Build Fail	✖	/
CVE-2021-44964	Lua	Use-After-Free	✔	✔	✔	Build Fail	✔	/
CVE-2020-21688	FFmpeg	Use-After-Free	✔	✔	✔	✖	✔	/
CVE-2021-33468	yasm	Use-After-Free	✔	✔	✔	✔	✔	/
CVE-2020-24978	nasm	Use-After-Free	✔	✔	✔	✖	✔	/
Issue-1325664 [165]	Chome	Use-After-Free	✔	✔	✔	Build Fail	✖	/
CVE-2022-43286	Nginx	Use-After-Free	✔	✔	✔	✖	✔	/
CVE-2019-16165	cflow	Use-After-Free	✔	✔	✔	✖	✔	/
CVE-2021-4187	vim	Use-After-Free	✔	✔	✔	✖	✔	/
CVE-2022-0891	libtiff	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2022-0924	libtiff	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2020-19131	libtiff	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2020-19144	libtiff	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2021-4214	libpng	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2021-3156	sudo	Heap Overflow	Run Well	✔	✔	/	/	Build Fail
CVE-2018-20330	libjpeg-turbo	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2020-21595	libde265	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2020-21598	libde265	Heap Overflow	✔	✔	✔	/	/	Build Fail
Issue-5551 [166]	mruby	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2022-0080	mruby	Heap Overflow	Run Well	✔	✔	/	/	Build Fail
CVE-2019-9021	PHP	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2022-31627	PHP	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2021-32281	gravity	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2020-15888	Lua	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2021-26259	htmldoc	Heap Overflow	✔	✖	✔	/	/	Build Fail
CVE-2022-28966	Wasm3	Heap Overflow	✔	✔	✔	/	/	Build Fail

Table 4.2: The security evaluation results of CAMP and related tools on real-world vulnerabilities. ✔ represents that the corresponding tool successfully detected the memory corruption in the vulnerability. ✖ indicates the tool failed to detect the memory corruption that happened. ”/” represents the tool does not support protecting the corresponding type of vulnerability. ”Run Well” means the application runs well without causing any memory corruption with the PoC input. ”Run Fail” represents that the tool failed to run due to compatibility issues. ”Build Fail” means the tool failed to compile the targeted application to enforce protection.

CAMP using two case studies on real-world vulnerabilities. Following this, we discuss the security capabilities of CAMP in comparison to other existing works. Ultimately, we demonstrate CAMP's performance and memory overhead using SPEC CPU benchmarks and real-world applications, showcasing its superiority over the most recent research tools. The tests were carried out on a bare-metal machine equipped with an Ubuntu 22.04 system, a 12th Gen Intel i7-12700 CPU running at 4.9 GHz, 32GB of RAM, and 1T of SSD storage.

4.6.1 Security Evaluation

Juliet Test Suite. To assess the protective measures offered by CAMP, we carried out experiments using the Juliet Test Suite, following the pattern set by recent works [141], [152]. The Juliet Test Suite comprises various test programs for different types of vulnerabilities, with each having both good and bad tests. The proof-of-concept (PoC) in the bad tests triggers the corresponding vulnerability, whereas the PoC in the good tests does not.

Considering CAMP's primary focus is to prevent heap memory corruption, we only included heap-related vulnerability types from the Juliet Test Suite. An important note is that CAMP's overflow prevention mechanism relies on the size of the memory object. Hence, heap overflows that do not exceed the memory boundary are deemed harmless as they don't corrupt other memory objects. Following this logic, we employed a customized ASAN[2] to exclude the bad tests wherein the overflow is contained within the memory object.

When it comes to CAMP's use-after-free protection, it neutralizes dangling pointers, and rather than reporting an error, the dereference of a dangling pointer leads to the program's termination without any report. To evaluate the test cases in the use-after-free category, we employed a gdb

[2]A customized ASAN that rounds up each allocation and includes a large red zone to prevent overflows from impacting adjacent objects and escaping detection. When evaluating tests for overflow, if ASAN does not report any issues, it indicates that the overflow occurs within an object. In such cases, the test can be safely removed from the test suite.

script to confirm that the termination of the program was due to the dereference of neutralized dangling pointers.

Table 4.1 displays the results of the selected tests, showing the chosen vulnerability type, the number of selected tests, the total number of tests, and the number of tests passed. A few of the tests, originally classified as Heap-Based Buffer Overflow, do not contain heap overflow, such as cases `Heap_Based_Buffer_Overflow__c_src_char_cat_*`. These tests trigger a buffer overflow when copying data from the heap to the stack, resulting in a stack overflow instead of a heap overflow. These cases were excluded from the selected test cases using the customized ASAN. CAMP passed all selected tests without generating any false positives or negatives.

Real-world Applications. Aside from the Juliet Test Suite, we also evaluated CAMP's security using a set of real-world vulnerabilities. We incorporated all real-world vulnerabilities used in [142]. Additionally, we collected other types of vulnerabilities from the CVE database [167]–[169]. Table 4.2 lists the selected vulnerabilities. Our dataset comprises 14 use-after-free and 18 heap overflow vulnerabilities across 19 applications, including language interpreters, commonly used libraries, browsers, web servers, and commonly used UNIX tools. The aim was to evaluate CAMP's effectiveness in preventing various heap memory corruption vulnerabilities and its scalability over a variety of real-world applications. For comparison, we also evaluated related tools, including ASAN/ASAN-- [170], Memcheck [136], DangNull [138], MarkUs [140], and Delta Pointer [171] for their effectiveness in detecting and preventing heap errors.

Table 4.2 presents the security evaluation results on real-world applications. CAMP successfully detected and prevented all use-after-free vulnerabilities. In the case of heap overflow vulnerabilities, CAMP was able to detect 16 out of 18 and report them. The other two ran smoothly without causing any reports or crashes. However, upon manual investigation using a debugger, we found

that the overflow had occurred, but the memory boundaries were not exceeded due to CAMP's seglist allocator's rounded-up memory allocation. As a result, the overflow is mitigated, and no memory corruption occurred. We argue that these two cases do not count as false negatives of CAMP as the exploitation is prevented.

For tools providing a comparable level of heap protection, Memcheck was able to detect all the heap errors in the dataset. ASAN-- reported all the heap errors except CVE-2021-26259 [172]. The reason behind this is that ASAN uses a red zone to detect heap overflow vulnerabilities. However, if a non-linear heap overflow occurs and skips the red zone, ASAN's detection will be defeated. The CVE-2021-26259 showcases this scenario. Unlike ASAN, CAMP was able to detect this case successfully as it detects heap overflow based on the memory boundary, making it impossible for non-linear heap overflows to bypass its protection. We argue that CAMP's protection is stronger and more robust than those two tools. As discussed in prior work [152], ASAN and Memcheck's use-after-free protection could be defeated if the attacker fills up the quarantined memory and re-occupies the freed memory that the dangling pointer refers to, as such, the access from the dangling pointer will not be detected, thus enabling a possible exploitation against use-after-free vulnerabilities. However, CAMP tracks all the pointers referencing heap memory and prevents use-after-free vulnerabilities by neutralizing the dangling pointer, which fundamentally mitigates the vulnerability.

For tools offering partial heap protection, DangNull and Delta Pointer showed limited compatibility support. 4 out of 14 use-after-free cases were not able to be built with DangNull. Among the 10 use-after-free cases that could be successfully compiled, only 4 of them were detected. The others just crashed with the PoC input as if there is no protection. Note that DangNull has a similar use-after-free protection scheme as CAMP, but it fails to detect 6 cases that CAMP could detect (e.g., CVE-2022-43286 [173], CVE-2019-16165 [174], and CVE-2021-4187 [175]). This

Benchmark	Time and Memory Overhead				
	CAMP	**ASAN−−**	**ASAN**	**ESAN**	**Memcheck**
600.perlbench_s	237.95% / 2241.12%	76.95% / 366.92%	143.59% / 358.20%	644.00% / 4.15%	3496.46% / 138.97%
602.gcc_s	78.56% / 135.52%	83.61% / 63.42%	99.47% / 62.77%	-	2888.13% / 30.42%
605.mcf_s	14.62% / 31.55%	24.45% / 3.61%	27.88% / 3.61%	109.33% / -4.24%	601.05% / 22.68%
623.xalancbmk_s	138.94% / 1220.66%	107.86% / 428.07%	109.41% / 433.51%	81.67% / 8.60%	4962.60% / 98.81%
625.x264_s	75.07% / 12.68%	62.26% / 13.52%	75.92% / 13.26%	90.94% / -3.55%	2070.57% / 56.96%
631.deepsjeng_s	1.58% / 0.00%	44.23% / -0.23%	64.08% / -0.23%	18.85% / -0.25%	3251.34% / 25.34%
641.leela_s	3.02% / 514.19%	13.97% / 2832.83%	17.33% / 2833.72%	6.65% / -17.52%	4163.69% / 262.82%
657.xz_s	7.79% / 0.00%	17.45% / 2.98%	13.40% / 2.98%	14.61% / -0.70%	718.87% / 24.45%
619.lbm_s	1.34% / 0.01%	37.32% / 5.94%	29.38% / 5.94%	34.14% / -0.36%	2907.53% / 25.98%
638.imagick_s	45.47% / 0.07%	17.23% / 4.46%	28.56% / 4.47%	21.70% / -2.00%	4452.66% / 22.93%
644.nab_s	62.55% / 26.13%	35.18% / 67.52%	35.14% / 66.63%	1988.66% / -1.34%	3722.35% / 31.80%
Geomean	21.27% / 127.47%	38.27% / 104.72%	44.78% / 104.35%	65.31% / -1.94%	2546.88% / 56.49%

Table 4.3: The relative time and memory overhead of CAMP, ASAN --, ASAN, ESAN, and Memcheck on SPEC CPU2017. ”-” indicates the tool failed to run the corresponding benchmark.

is because DangNull only tracks the point-to relation from heap to structured object on the heap. If the use-after-free is caused by dangling pointers on the stack/global memory, or the use-after-free object has no type information, DangNull will miss the detection, thus making exploitation possible. Delta Pointer showed even worse compatibility support, which could only compile 5 out of 17 cases, but all the out-of-bound in compatible cases were detected. MarksUs showed better compatibility support but failed to detect 6 out of 14 use-after-free vulnerabilities.

We argue that `CAMP` provides a much more comprehensive heap error detection capability when compared to similar tools. Our evaluation demonstrates that `CAMP` outperforms the combination of partial heap protection tools (such as Delta Pointer + DangNull/MarkUS) as well as ASAN--, which exhibits false negatives in the case of non-linear overflow.

4.6.2 Performance Evaluation

In this section, we will evaluate `CAMP`'s performance using two SPEC CPU benchmarks. We'll also evaluate how each of `CAMP`'s design components contributes to its overall effectiveness, including

Benchmark	Metric	CAMP	LowFat	Delta Pointer	DangNull	FreeGuard	MarkUs	FFMalloc
SPEC CPU2006	Time	54.92%	160.62%	37.39%	39.99%	10.40%	15.84%	9.50%
	Mem	237.67%	38.60%	0.01%	158.52%	70.89%	2.96%	27.57%
SPEC CPU2017	Time	21.27%	96.96%	-	28.61%	8.23%	11.90%	10.94%
	Mem	127.47%	54.35%	-	314.13%	29.23%	32.35%	62.00%

Table 4.4: The relative time and memory overhead of CAMP, LowFat, Delta Pointer, DangNull, FreeGuard, MarkUs, and FFmalloc on SPEC CPU2006 and SPEC CPU2017.

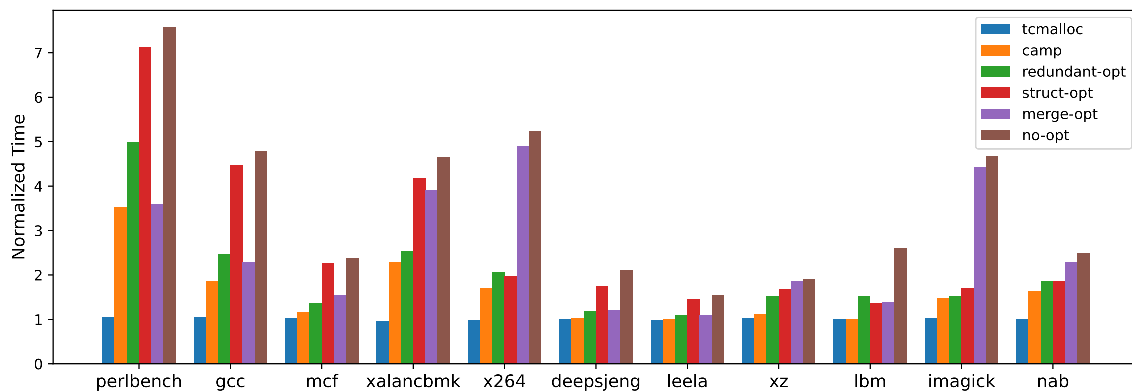

Figure 4.2: Evaluation result of CAMP breakdown on SPEC CPU2017. From left to right, the bars show the normalized time of tcmalloc replacement, CAMP, CAMP with each optimization disabled, and CAMP without optimization.

compiler optimization and the customized allocator. Finally, we'll gauge `CAMP`'s performance in two real-world applications.

SPEC CPU Benchmark. We evaluated the performance of `CAMP` in comparison with other related tools[3] like ASAN [137], ASAN --[170], ESAN [176], Softbound+CETS [144], and Memcheck [136]. The SPEC CPU benchmark suite was used to measure the performance overhead. All programs in the suite were compiled with default configurations and used reference input. To prevent termination, all tools were configured to ignore detected errors.

[3]These tools were designed to provide protection against both out-of-bound and use-after-free errors.

To ensure a fair comparison, only the heap error detectors of ASAN -- and ASAN were enabled. Unfortunately, ASAN and Memcheck could not compile and run the `omnetpp` and `dealII` programs and were excluded from the evaluation. Softbound+CETS showed limited support, failing to compile all programs in SPEC CPU2017 and supporting only 7 programs in SPEC CPU2006, so we only compared it in SPEC CPU2006. We excluded PACMem [152] from our evaluation as it needs specialized hardware (ARM PA) to detect heap memory errors. To avoid the performance gain from `CAMP`'s customized allocator, we used tcmalloc as the default allocator for the baseline. It should be noted that the benchmark was run 10 times and we reported the average result to minimize randomness.

The evaluation results are presented in Table 4.3 for the SPEC CPU2017 benchmark suite. Each row represents a specific application benchmark, with the benchmark name listed first and subsequent columns showing the relative time and memory measured compared to a baseline. In accordance with the most recent works [141], [152], we utilized the geometric mean value to represent the average overhead of each tool. `CAMP` demonstrated the best runtime speed compared to other tools, with an average overhead of 21.27%, while ASAN --, ASAN, and ESAN had overhead rates of 38.27%, 44.78%, and 65.31%, respectively. Memcheck had the worst runtime performance, introducing a 2546.88% overhead compared to the baseline. In terms of memory overhead, `CAMP` had a higher rate of 127.47% compared to roughly 104% for ASAN and ASAN --
↪ . ESAN had the best memory overhead performance, with -1.94%. `CAMP` still outperforms all other tools in terms of runtime speed in the SPEC CPU2006 benchmark. Specifically, it introduces an overhead of 54.92%, while ASAN --, ASAN, ESAN, SoftBound+CETS, and Memcheck have overheads of 56.77%, 67.02%, 123.08%, 319.75%, and 1990.02%, respectively.

In comparison with other tools offering partial memory protection, including LowFat [146], Delta Pointer [171], DangNull [138], FreeGuard [130], MarkUs [140], and FFMalloc [142], `CAMP`

Benchmark	Time (s)		Overhead
	Native	CAMP	
cfrac	2.91	3.82	31.27%
espresso	3.62	3.62	0.00%
barnes	1.35	1.34	-0.74%
redis	2.66	2.68	0.75%
leanN	25.35	26.18	3.27%
alloc-test1	2.99	3.06	2.34%
alloc-testN	2.91	3.42	17.53%
sh6benchN	2.41	2.39	-0.83%
sh8benchN	5.79	8.3	43.35%
xmalloc-testN	2.664	2.306	-13.44%
cache-scratchN	0.43	0.44	2.33%
Geomean	-	-	9.79%

Table 4.5: Time Overhead on mimalloc-bench. Native represents using the default allocator – ptmalloc, CAMP means using its customized seglist allocator based on tcmalloc.

demonstrates superior performance on the SPEC CPU Benchmarks (results in Table 4.4). In summary, `CAMP` provides the most comprehensive heap protection among the compared tools. It can detect both spatial and temporal heap errors, unlike tools that offer only partial heap protection. Even when compared to the combined overhead of Delta Pointer (for OOB protection) and FF-Malloc (for UAF protection), `CAMP` performs comparably (54.92% vs. 46.89%) while providing superior error detection capabilities and less restrictive memory constraints.

Components Evaluation. `CAMP` is composed of several elements, such as compiler optimization and a seglist allocator, each playing a part in the overall performance. We carried out evaluations on distinct `CAMP` configurations to establish the influence of each component.

We examined the effect of `CAMP`'s tailor-made allocator by running an evaluation with mimalloc-bench to juxtapose different allocator performances. The results, displayed in Table 4.5, include the ”Native” column, which showcases the performance of the system's default allocator (ptmal-

loc), and the "CAMP " column, indicating the performance with CAMP's custom allocator. Both allocators demonstrated varying behavior across different tests. For instance, in the cfrac test, the Native allocator outperformed CAMP by 31.27

In order to determine the efficacy of compiler optimization, we individually disabled each optimization within various CAMP configurations, which we label as struct-opt, redundant-opt, merge ↪ -opt, and no-opt in Figure 4.2. To comprehend the role of the allocator cache design, we assessed the performance of CAMP with the allocator cache disabled. Lastly, to gauge the impact of the seglist allocator, we compared these results to a baseline that used tcmalloc. These configurations and the results they produced on SPEC CPU2017 are illustrated in Figure 4.2.

Our analysis reaffirms that the seglist allocator contributes minimally to the overall performance. The baseline using tcmalloc demonstrates a similar runtime speed to the default ptmalloc, with a nominal average speed improvement of just 2.26%. We then inspected the role of the allocator cache design in CAMP. During this evaluation, three programs (perlbench, gcc, xalancbmk) encountered memory depletion and were unable to finish the test, thus they were omitted from Figure 4.2. The remaining programs showed an average overhead of 40.34%, nearly twice the overhead when the cache is enabled (20.94%). Our investigations showed that these three programs created numerous repeated point-to relationships, causing high memory usage for metadata management. Notably, all these programs house a language interpreter that forms ASTs during input parsing, resulting in connections for which CAMP has to maintain point-to records. This showcases the utility of the cache design, which allows repeated point-to relations to be stored, thereby reducing both time and memory overhead.

We also discovered that the proposed compiler optimization considerably diminished the performance overhead. CAMP without compiler optimization bore an overhead of 204.09%, whereas the standard CAMP bore an overhead of 20.94%. Among the evaluated programs, imagick benefitted

System	Output (req/s)	Latency (μs)				
		Average	**50%**	**75%**	**90%**	**99%**
Native	150,368	643.23	625	635	649	910
CAMP	108,322	880	850	870	910	1070
ASAN	97,095	970	900	930	1040	1910

Table 4.6: CAMP and ASAN's output and latency evaluation results on Nginx. In the Latency column, the "Average" represents the average latency of the requested connection, the others show the latency distribution.

most from optimization, with its overhead reduced from 368.18% to 48.47%. Upon disassembling each optimization, we discovered that structure optimization exerted the most significant impact. Disabling this optimization increased the overhead from 20.94% to 113.39%. Similarly, deactivating redundant optimization and structure optimization led to respective overheads of 64.20%, and 95.73%.

Nginx. To evaluate the performance of `CAMP` on a large-scale, real-world application, we conducted experiments on Nginx v1.22.1 using the wrk v4.2.0 benchmarking tool. For these experiments, we configured the tool with 8 threads, 100 connections, and a test duration of 60 seconds. To ensure consistency, we repeated the test 30 times and recorded the average results. The findings are presented in Table 4.6. On average, CAMP introduces a 27.96% overhead on Nginx's request output, In terms of latency, `CAMP` adds 36.81% more time. The results reflex `CAMP`'s efficiency on real-world applications with mild overhead. As a comparison, ASAN incurs a 35.43% overhead on request output and has a latency overhead of 50.80%.

Chromium. Apart from Nginx, we conducted an evaluation of `CAMP`'s performance on Chromium. Our assessment employed three widely-used browser benchmarks: Kraken, SunSpider, and Lite Brite. Additionally, we measured the loading time of various websites, as this parameter directly

Benchmark	**Time (ms)**		**Overhead**
	Native	**CAMP**	
kraken	1069	1722	61.09%
sunspider	521	813	56.05%
Lite Brite	2930	5520	88.40%
Geomean	-	-	67.14%
google.com	1101	1427	29.61%
facebook.com	831	1199	44.28%
amazon.com	2298	3120	35.77%
openai.com	1444	1791	24.03%
twitter.com	1479	1708	15.48%
gmail.com	1691	2032	20.17%
youtube.com	2143	2628	22.63%
wikipedia.org	984	1535	56.00%
Geomean	-	-	28.59%

Table 4.7: CAMP's performance evaluation results on the Chromium browser. In the Benchmark column, kraken, sunspider and Lite Brite are three browser benchmarks, whereas the following are websites used to measure the loading time of the browser.

impacts the user's browsing experience. To track the loading time, we applied a browser extension and recorded the mean loading time for the top 8 websites according to the Top Websites Ranking [177]. We performed each benchmark test 30 times to average out potential inconsistencies or anomalies. The outcomes of the evaluation are documented in Table 4.7, where the average overhead of CAMP is denoted by the geometric mean.

Our observation indicates that CAMP incurs a 67.14% overhead on the three browser benchmarks. However, the webpage loading time only escalated by an average of 28.59%. It's essential to realize that these benchmarks target specific browser elements, which may not wholly represent the total performance. In contrast, webpage loading involves the execution of the JavaScript engine, DOM processing, and other factors that more accurately mirror real-world browsing. As such, we maintain that the 28.59% overhead induced by `CAMP` is relatively minor in this scenario.

4.7 Discussion

In this section, we discuss some other issues we have not yet discussed.

Potential False Positives and Negatives. As CAMP uses a pointer-based protection approach, it has some shared vulnerabilities with previous studies [144]–[146]. Here, we discuss the scenarios that could lead to these situations. Firstly, in C/C++, out-of-bound pointers can be utilized as memory boundaries, these pointers don't function for memory access, but for memory boundary checks. As such pointers are out-of-bounds, CAMP can produce false positives. To alleviate this, we reserve additional memory space for each allocation so that these boundary pointers remain within the bounds, as mentioned in Section 4.5. Altering the memory layout this way effectively counteracts the problem, hence no false positives were observed in our evaluation. We view this more as a compatibility concern, best resolved at the source code level. This not only eliminates the likelihood of false positives but also bolsters the security of the code.

Secondly, it should be highlighted that in C/C++, integers may be employed for pointer arithmetic instead of actual pointers. This constitutes a problem for CAMP as it can lead to undetected out-of-bounds pointer accesses. To tackle this, we impose a compiler policy that restricts the conversion of integers to pointers. By preventing such conversions, the creation of out-of-bounds pointers can be avoided. If developers need to execute such casts, they could denote their intention explicitly using compiler attributes to locally disable the policy.

In-bound Overflow Prevention. CAMP's overflow detection design is rooted in memory boundaries, implying it only recognizes overflows that breach the boundaries as violations. Thus, CAMP's design shares a similar vulnerability with previous works [144], [145], [178], as it doesn't guard against in-bound heap overflows. However, we discovered that, with an appropriate implementa-

tion, CAMP could diminish some in-bound overflow. Specifically, if pointer arithmetic is executed on an array of a structure, we can use the array size to validate the pointer and ensure that the result pointer remains within the structure's array. This mitigates in-bound overflow with type information. However, we don't claim this as a capability of CAMP to prevent in-bound overflow. This is because type information isn't always accessible, and the array might be dynamic, leaving CAMP without a method to validate it and ensure no overflow is inside the structure. Therefore, we regard the prevention of in-bound overflows as a future avenue for research.

4.8 Related Work

Safe Allocator. In the battle against heap memory corruption attacks [10], [71], [133]–[135], numerous safe heap allocators [141], [142], [179]–[182] have been suggested. For instance, FF-malloc [142] introduces fast-forward allocation for one-time memory allocation, thereby eliminating use-after-free access. DieHarder [132] uses heap address space randomization to thwart heap exploitation. Oscar [179] uses shadow memory for each heap object to preclude temporal memory errors, hence access via dangling pointers is detected. Markus [140] adopts a strategy that quarantines freed memory, removing any remaining dangling pointers. DangZero [141] conceives an allocator design capable of directly interfacing with the kernel page table, paving the way for an alias-based Use-After-Free (UAF) detection mechanism, thereby precluding virtual memory reuse. FreeGuard [130] enhances performance with a freelist and optimizes the shadow memory scheme. Unlike these works, CAMP's protection isn't entirely dependent on allocators.

Pointer Invalidation. Various works [138], [143], [144], [171], [178], [183], [184] detect memory errors via pointer invalidation. For example, CETS [143] uses a lock-and-key identifier-based system to track distinct metadata for each pointer to detect dangling pointers. Undangle [183] em-

ploys dynamic taint tracking to pinpoint and eradicate unsafe dangling pointers. DangNull [138] uses runtime point-to relation and nullifies pointers when objects are freed. For spatial safety, Redfat [184] combines redzone and low-fat pointers to identify buffer overflow. Delta Pointer [171] introduces a new pointer tag, ensuring that any overflow pointer becomes invalid, thereby preventing the error. Softbound [144] uses shadow memory to monitor the bounds of each allocated memory block and impose access control, with runtime checks added for efficient overflow detection. Unlike these works, CAMP integrates with the compiler and the allocator to optimize pointer invalidation and deliver swift runtime support, offering comprehensive heap protection.

Memory Sanitizer. While most memory sanitizers offer complete heap error detection instead of partial heap protection, ASan [137] uses shadow memory and redzone to detect temporal and spatial memory errors. To minimize its overhead, ASan -- [170] and SANRAZOR [185] suggest several compiler optimizations to lessen instrumented checks on memory access while maintaining the same security level. FuZZan [186] presents new metadata structures to cut down the memory management overhead for sanitizers. CUP [187] proposes a hybrid metadata scheme supporting all program data, including globals, heap, and stack. EffectiveSan [176] displays a dynamic type system for detecting memory errors, though it has some limitations on detecting temporal errors. Memcheck [136], a part of Valgrind [188], detects full memory errors but operates on binary code instead of source code. Unlike these methods, CAMP distinguishes itself by its protection scheme, thereby providing superior speed.

Hardware-assisted Protection. Numerous works [145]–[152] leverage hardware for enforcing memory safety. For instance, Low-Fat [146] enriches the pointer representation with base and bounds information so that runtime or hardware can prevent spatial safety violations. In-Fat [145] enhances the hardware-assisted tagged-pointer scheme, using three complementary object meta-

data schemes to reduce the required pointer tag bits. Some works, like PtAuth [147], AOS [149], and PACMem [152], utilize the Pointer Authentication Code (PAC) feature of ARM to detect memory errors. HeapCheck [148] uses unused pointer bits in 64-bit systems to hold an index of a bounds table, aiding the detection of out-of-bounds and use-after-free errors. By storing the bounds information in an 8 KB on-chip SRAM cache, HeapCheck introduces minimal overhead. BOGO [151] uses Intel MPX to ensure spatial and temporal memory safety. CHEx86 [189] introduces an innovative process architecture aimed at preventing memory errors by instrumenting the code at the microcode level. Unlike these methods, CAMP doesn't require additional hardware support.

4.9 Conclusion

Thwarting memory corruption in the heap is a challenging endeavor. Current methodologies to tackle heap memory corruption either offer restricted protection or incur substantial runtime overhead, which hampers their deployment in real-world products. CAMP, through the use of meticulously constructed code instrumentation and a bespoke allocator, furnishes a comprehensive defense against heap memory corruption. Although the instrumentation adds some runtime overhead, our findings show that this overhead can be drastically cut back through a sequence of optimization strategies that eliminate or consolidate unneeded instrumentations. An appraisal of CAMP, conducted using a large-scale real-world application and SPEC CPU Benchmarks, underscores the significant reduction in performance impact. The minimal overhead coupled with CAMP's capacity to proficiently detect and avert heap memory corruption, underscores its potential as an efficacious solution for protecting programs against heap memory corruption.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

This dissertation presents three critical steps I have done toward understanding and reducing exploitability in the Linux kernel to improve security.

To analyze the capability of kernel vulnerability, we propose an object-driven kernel fuzzing method which allows security analysts to explore various contexts and paths toward a kernel bug, revealing many potential error behaviors. Through this method, we can escalate the bug's exploitability, providing a deeper understanding of kernel vulnerabilities. Our current focus is on Linux kernel bugs; however, future directions include extending this method to other kernels such as XNU, FreeBSD. Although our design is not specifically for Linux, intensive engineering efforts will be necessary for these expansions due to varying support for debugging features and the predominant design of existing fuzzing tools for Linux.

We introduce a novel exploitation method called DirtyCred, compositing a critical piece of kernel exploitation. DirtyCred facilitates exploitation and defense circumvention irrespective of a specific kernel vulnerability. DirtyCred confuses the Linux kernel by swapping credential objects, which allows an unprivileged user to escalate their privileges. This finding exposes a significant weakness in the existing defense architecture of Linux, warranting urgent action. To mitigate this threat, we propose a defense mechanism called privilege-object isolation and have implemented a Linux kernel prototype that integrates this defense. The prototype shows promising results in terms of enhanced security at a minimal to moderate cost.

To reduce the exploitability of memory corruption vulnerability, we propose a comprehensive defense method called CAMP. Despite initial runtime overhead, we discovered that through a series of optimization strategies, this overhead can be significantly reduced. Performance testing, using real-world applications and SPEC CPU Benchmarks, validated our solution's efficacy, demonstrating CAMP's potential as an effective solution to protect programs against heap memory corruption.

These advancements provide valuable contributions to kernel security, offering more robust defenses against vulnerabilities and novel exploitation methods. Moreover, they pave the way for future research and development in other OS kernels.

5.2 Future research

One of the future directions I plan to explore is extending CAMP from userspace to kernel space. As we have demonstrated in this dissertation, CAMP's security design effectively prevents heap memory corruption and introduces only a reasonable overhead. I envision that applying the same protection scheme in kernel space would also be effective, although several unique challenges arise when the context is switched to the kernel. Additionally, there is room for more compiler optimization to further reduce the overhead. For instance, one could design an optimization that considers data flow across functions, thereby reducing additional checks.

Another direction I intend to pursue is the analysis of kernel exploitation composability. Owing to its complexity, it is challenging to apply a general formal algorithm that can fundamentally solve this problem. Therefore, a more practical approach is to progressively investigate composability, enabling the construction of a comprehensive understanding of exploitation.

REFERENCES

[1] R. Hat, *Cve-2021-3715*, https://access.redhat.com/security/cve/cve-2021-3715, 2021.

[2] Syzbot, *Warning: Odebug bug in route4$_c$hange*, https://syzkaller.appspot.com/bug?id=1bba967ec4596833317399ba8d6f7d655bd655e8, 2020.

[3] *Cve-2021-3715 patch*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ef299cc, 2021.

[4] Z. Lin, Y. Chen, X. Xing, and K. Li, *Your trash kernel bug, my precious 0day*, https://zplin.me/talks/BHEU21_trash_kernel_bug.pdf, 2021.

[5] Microsoft, *A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003*, https://www.betaarchive.com/wiki/index.php/Microsoft_KB_Archive/875352, 2005.

[6] S. Bratus, "What hacker research taught me," Rss, 2009.

[7] S. Bratus, J. Bangert, A. Gabrovsky, A. Shubina, D. Bilar, and M. E. Locasto, "Composition patterns of hacking," in *Proceedings of the 1st International Workshop on Cyber Patterns*, 2012, pp. 80–85.

[8] T. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 391–403, 2017.

[9] Z. Lin, Y. Chen, Y. Wu, *et al.*, "Grebe: Unveiling exploitation potential for linux kernel bugs," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 2078–2095.

[10] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1963–1976.

[11] D. Vyukov, *Syzkaller*, https://github.com/google/syzkaller, 2020.

[12] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, “kAFL: Hardware-assisted feedback fuzzing for os kernels,” in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[13] D. Jones, *Trinity*, https://github.com/kernelslacker/trinity, 2020.

[14] T. Blazytko, M. Schlögel, C. Aschermann, *et al.*, “AURORA: Statistical crash analysis for automated root cause explanation,” in *Proceeding of the 28th USENIX Security Symposium (USENIX Security)*, 2020.

[15] syzbot, *General protection fault in hrtimer_active*, https://syzkaller.appspot.com/bug?id=5d3cce34cc09f722e859ae2037801f5b0d67c5c9, 2017.

[16] *Linux kernel design patterns - part 2*, https://lwn.net/Articles/336255/, 2009.

[17] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” 1998.

[18] syzbot, *KASAN: Slab-out-of-bounds Read*, https://syzkaller.appspot.com/bug?id=7022420cc54310220ebad2da89e499bdb1f0f5e8, 2019.

[19] syzbot, *BUG: Unable to handle kernel paging request*, https://syzkaller.appspot.com/bug?id=692a8c2104416b219c0036b0a566eb88f73b1dd5, 2018.

[20] Z. Lin, *Grebe’s source code*, https://github.com/Markakd/GREBE, 2021.

[21] K. Lu and H. Hu, “Where Does It Go? Refining indirect-call targets with multi-layer type analysis,” in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[22] Y. Chen, Z. Lin, and X. Xing, “A systematic study of elastic objects in kernel exploitation,” in *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[23] I. Ben-Gal, “Outlier detection,” in *Data mining and knowledge discovery handbook*, Springer, 2005, pp. 131–146.

[24] S. Pailoor, A. Aday, and S. Jana, “MoonShine: Optimizing os fuzzer seed selection with trace distillation,” in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[25] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the linux kernel," in *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS)*, 2020.

[26] W. You, P. Zong, K. Chen, *et al.*, "SemFuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[27] J. Corina, A. Machiry, C. Salls, *et al.*, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[28] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "KRace: Data race fuzzing for kernel file systems," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[29] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[30] syzbot, *WARNING: Refcount bug in crypto_mod_get*, https://syzkaller.appspot.com/bug?id=bdeea91ae259b3a42aa8ed8d8c91afd871eb5d80, 2020.

[31] syzbot, *WARNING: Refcount bug in nr_insert_socket*, https://syzkaller.appspot.com/bug?id=521a764b3fc8145496efa50600dfe2a67e49b90b, 2019.

[32] syzbot, *General protection fault in delayed_uprobe_remove*, https://syzkaller.appspot.com/bug?id=229e0b718232b004dfddaeac61d8d66990ed247a, 2019.

[33] *Full performance results of syzkaller, syzkaller variant, grebe without mutation optimization and grebe*, https://tinyurl.com/x9ky26ms, 2021.

[34] syzbot, *BUG: Unable to handle kernel paging request*, https://syzkaller.appspot.com/bug?id=d1baeb189d38d5ba53517876c89b20d4e6857bc6, 2017.

[35] syzbot, *BUG: Corrupted list in __neigh_create*, https://syzkaller.appspot.com/bug?id=de28cb0e686acfa1c9dbad1e11cbb0ac9b05caf2, 2019.

[36] syzbot, *Warning: Refcount bug*, https://syzkaller.appspot.com/bug?id=8eceaff64a35a9f02c1315bbf12b7f262a0b4f08, 2020.

[37] syzbot, *General protection fault in qrtr_endpoint_post*, https://syzkaller.appspot.com/bug?id=f56bbe6668873ee245986bbd23312b895fa5a50a, 2020.

[38] syzbot, *WARNING in get_pi_state*, https://syzkaller.appspot.com/bug?id=bb7fa48ebde0db8e3fc683a47bb69ab1dca895bc, 2017.

[39] syzbot, *BUG: Corrupted list in kobject_add_internal*, https://syzkaller.appspot.com/bug?id=f0ec9a394925aafbdf13d0a7e6af4cff860f0ed6, 2020.

[40] syzbot, *KASAN: General protection fault in crypto_chacha20_crypt*, https://syzkaller.appspot.com/bug?id=d767177245c54af614d5241159cce56995eef0db, 2018.

[41] syzbot, *WARNING: ODEBUG bug in io_sqe_files_unregister*, https://syzkaller.appspot.com/bug?id=460cc948740aa1e715156c0edf5d5d397401d557, 2020.

[42] syzbot, *WARNING in vhost_dev_cleanup*, https://syzkaller.appspot.com/bug?id=0df4c1a9c14776f5fd163180e3580ad88b32649a, 2018.

[43] syzbot, *General protection fault in vb2_mmap*, https://syzkaller.appspot.com/bug?id=4cf5ee79b52a4797c5bd40a58bd6ab243d40de48, 2019.

[44] syzbot, *General protection fault in strlen*, https://syzkaller.appspot.com/bug?id=502c872feb9bbb5ad6494c349c7faa87a9f1777b, 2018.

[45] syzbot, *WARNING in dma_buf_vunmap*, https://syzkaller.appspot.com/bug?id=163388d1fb80146cd3ba22a11a5a1995c3eaaafe, 2019.

[46] syzbot, *BUG: Unable to handle kernel paging request*, https://syzkaller.appspot.com/bug?id=b36d7e444fe532685b683ae7980f4e3a184f0ad8, 2020.

[47] syzbot, *General protection fault in scatterwalk_copychunks*, https://syzkaller.appspot.com/bug?id=1fd1d44caf96ca464e1c1f19299d1f3e7558f6e5, 2018.

[48] syzbot, *BUG: Corrupted list in mousedev_release*, https://syzkaller.appspot.com/bug?id=b9b37a7aaeb4a4e2357b2dfdd1f689e3ffa66282, 2020.

[49] syzbot, *General protection fault in bpf_tcp_close*, https://syzkaller.appspot.com/bug?id=695527bd03b09f741819baddcd231c16fe014a48, 2018.

[50] syzbot, *General protection fault in hci_event_packet*, https://syzkaller.appspot.com/bug?id=0d93140da5a82305a66a136af99b088b75177b99, 2020.

[51] syzbot, *BUG: Unable to handle kernel paging request*, https://syzkaller.appspot.com/bug?id=85fd017144b9b1d6761870ff71852d15e4cdd44e, 2020.

[52] syzbot, *General protection fault in kernel_accept*, https://syzkaller.appspot.com/bug?id=b0e30ab5186d097b8e3e23e8ca971fbf1cf54659, 2019.

[53] syzbot, *WARNING: Odebug bug in tcf_queue_work*, https://syzkaller.appspot.com/bug?id=6a039858238a38cbc7f372607fc5d49f4469cf2c, 2020.

[54] syzbot, *WARNING: Bad unlock balance in ucma_event_handler*, https://syzkaller.appspot.com/bug?id=d5222b3e1659e0aea19df562c79f216515740daa, 2020.

[55] syzbot, *General protection fault in syscall_return_slowpath*, https://syzkaller.appspot.com/bug?id=575a090948f98f28593563c9d9d9b343eb39bbb4, 2020.

[56] syzbot, *KASAN: Slab-out-of-bounds read in bitmap_ip_add*, https://syzkaller.appspot.com/bug?id=3a6c9972ff471c4dbc3f45e83dd5fa2f18cb82a4, 2020.

[57] syzbot, *KASAN: Use-after-free read in ip6_dst_destroy*, https://syzkaller.appspot.com/bug?id=27ae1ae5c54e09f8c86dd9428df048e7886be6dc, 2020.

[58] syzbot, *KASAN: Use-after-free read in sctp_auth_free*, https://syzkaller.appspot.com/bug?id=cbb289816e728f56a4e2c1b854a3163402fe2f88, 2020.

[59] syzbot, *KASAN: Slab-out-of-bounds read*, https://syzkaller.appspot.com/bug?id=4bf11aa05c4ca51ce0df86e500fce486552dc8d2, 2020.

[60] syzbot, *KASAN: Slab-out-of-bounds write in sha512_final*, https://syzkaller.appspot.com/bug?id=e4be30826c1b7777d69a9e3e20bc7b708ee8f82c, 2018.

[61] syzbot, *KASAN: Use-after-free read in cma_bind_port*, https://syzkaller.appspot.com/bug?id=3b7409f639067d927b8ad1b11a5e08bae27061af, 2018.

[62] syzbot, *KASAN: Use-after-free read in tipc_nl_node*, https://syzkaller.appspot.com/bug?id=ddaf58be21bc0aacece5a53ab1ae5db7e89f5bb0, 2019.

[63] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[64] D. Song, F. Hetzelt, D. Das, *et al.*, "PeriScope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, 2019.

[65] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2011.

[66] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, 2008.

[67] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[68] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[69] Y. Shoshitaishvili, R. Wang, C. Salls, *et al.*, "SoK:(state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[70] N. Stephens, J. Grosen, C. Salls, *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[71] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[72] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[73] Y. Wang, C. Zhang, X. Xiang, *et al.*, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 25nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[74] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[75] W. Xu, J. Li, J. Shu, *et al.*, "From Collision To Exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[76] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[77] K. Lu, M.-T. Walter, D. Pfaff, and S. Nürnberger and Wenke Lee and Michael Backes, "Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[78] H. Cho, J. Park, J. Kang, *et al.*, "Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers," in *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[79] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[80] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[81] W. Wu, Y. Chen, J. Xu, X. Xing, W. Zou, and X. Gong, "FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[82] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[83] J. Edge, *Kernel address space layout randomization*, 2013.

[84] google, *Kernel control flow integrity*, 2022.

[85] T. M. Corporation, *Cve-2022-0847*, 2022.

[86] M. Kellermann, *The dirty pipe vulnerability*, 2022.

[87] Google, *Roses are red, violets are blue, giving leets more sweets. all of 2022!* 2022.

[88] D. Howells, *Credentials in linux*, 2022.

[89] J. Corbet, *Supervisor mode access prevention*, 2012.

[90] M. Jurczyk, *Smep: What is it, and how to beat it on windows*, 2011.

[91] J. Edge, *Control-flow integrity for the kernel*, 2020.

[92] J. Corbet, *The current state of kernel page-table isolation*, 2017.

[93] T. M. Corporation, *Cve-2021-4154*, 2021.

[94] Anonymous, *Dirtycred exploit*, 2022.

[95] Z. Lin, *How autoslab changes the memory unsafety game*, 2021.

[96] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[97] W. Xu, J. Li, J. Shu, *et al.*, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[98] Linux, *Userfaultfd's introduction in the linux kernel user's and administrator's guide*, 2022.

[99] Linux, *Fuse's introduction in the linux kernel user's and administrator's guide*, 2022.

[100] J. Horn, *Linux: Uaf via double-fdput*, 2022.

[101] L. McVoy and C. Staelin, *Lmbench - tools for performance analysis*, 2022.

[102] P. Media, *Open-source, automated benchmarking*, 2022.

[103] Datadog, *Using the dirty pipe vulnerability to break out from containers*, 2022.

[104] A. Chapman, *Privileged container escape control groups release_agent*, 2020.

[105] W. Wu, Y. Chen, X. Xing, and W. Zou, "{Kepler}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[106] HexRabbit, *Cve-2021-34866 writeup*, 2021.

[107] K. Zeng, Y. Chen, H. Cho, *et al.*, "Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability," in *Proceedings of the 31st USENIX Conference on Security Symposium*, 2022.

[108] R. Raducu, R. J. Rodrıguez, and P. Álvarez, "Defense and attack techniques against file-based TOCTOU vulnerabilities: A systematic review," *IEEE Access*, vol. 10, pp. 21 742–21 758, 2022.

[109] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "{Kguard}: Lightweight kernel protection against {return-to-user} attacks," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[110] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[111] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[112] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[113] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Jump over aslr: Attacking branch predictors to bypass aslr,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[114] Y. Chen, Z. Lin, and X. Xing, “A systematic study of elastic objects in kernel exploitation,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[115] M. Lipp, M. Schwarz, D. Gruss, *et al.*, “Meltdown: Reading kernel memory from user space,” in *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018.

[116] P. Kocher, J. Horn, A. Fogh, *et al.*, “Spectre attacks: Exploiting speculative execution,” in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 2019.

[117] T. Garnier, *Mm: Slab freelist randomization*, 2016.

[118] D. Williams, *Randomize free memory*, 2018.

[119] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, “Kaslr is dead: Long live kaslr,” in *International Symposium on Engineering Secure Software and Systems*, 2017, pp. 161–176.

[120] K. C. Accardi, *Function granular kaslr*, 2020.

[121] J. Criswell, N. Dautenhahn, and V. Adve, “Kcofi: Complete control-flow integrity for commodity operating system kernels,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[122] X. Ge, N. Talele, M. Payer, and T. Jaeger, “Fine-grained control-flow integrity for kernel software,” in *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*, 2016.

[123] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, “In-kernel control-flow integrity on commodity oses using arm pointer authentication,” *arXiv preprint arXiv:2112.07213*, 2021.

[124] D. McKee, Y. Giannaris, C. O. Perez, *et al.*, “Preventing kernel hacks with hakc,” in *Proceedings 2022 Network and Distributed System Security Symposium*.

[125] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, “Enforcing kernel security invariants with data flow integrity.,” in *Proceedings 2016 Network and Distributed System Security Symposium*, 2016.

[126] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, “Pt-rand: Practical mitigation of data-only attacks against page tables.,” in *Proceedings of the 2017 Network and Distributed Systems Security Symposium*, 2017.

[127] Q. Chen, A. M. Azab, G. Ganesh, and P. Ning, “Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[128] K. Huang, Y. Huang, M. Payer, *et al.*, “The taming of the stack: Isolating stack data from memory errors,” in *Proceedings of the 2022 Network and Distributed Systems Security Symposium*, 2022.

[129] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, “Xmp: Selective memory protection for kernel and user space,” in *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, 2020.

[130] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, “Freeguard: A faster secure heap allocator,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[131] Z. Lin, R. D. Riley, and D. Xu, “Polymorphing software by randomizing data structure layout,” in *Detection of Intrusions and Malware, and Vulnerability Assessment: 6th International Conference, DIMVA*, Springer, 2009.

[132] G. Novark and E. D. Berger, “Dieharder: Securing the heap,” in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[133] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, “{Maze}: Towards automated heap feng shui,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1647–1664.

[134] I. Yun, D. Kapil, and T. Kim, “Automatic techniques to systematically discover new heap exploitation primitives,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1111–1128.

[135] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[136] N. N. Julian Seward, *Memcheck: a memory error detector*, https://valgrind.org/docs/manual/mc-manual.html.

[137] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[138] B. Lee, C. Song, Y. Jang, *et al.*, "Preventing use-after-free with dangling pointers nullification.," in *NDSS*, 2015.

[139] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[140] S. Ainsworth and T. M. Jones, "Markus: Drop-in use-after-free prevention for low-level languages," 2020.

[141] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "Dangzero: Efficient use-after-free detection via direct page table access," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[142] *Preventing Use-After-Free Attacks with Fast Forward Allocation.* 2021.

[143] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for c," in *Proceedings of the 2010 international symposium on Memory management*, 2010.

[144] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[145] S. Xu, W. Huang, and D. Lie, "In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[146] G. J. Duck and R. H. Yap, “Heap bounds protection with low fat pointers,” in *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[147] R. M. Farkhani, M. Ahmadi, and L. Lu, “{Ptauth}: Temporal memory safety via robust points-to authentication,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1037–1054.

[148] G. Saileshwar, R. Boivie, T. Chen, B. Segal, and A. Buyuktosunoglu, “Heapcheck: Low-cost hardware support for memory safety,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 1, pp. 1–24, 2022.

[149] Y. Kim, J. Lee, and H. Kim, “Hardware-based always-on heap memory safety,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 1153–1166.

[150] J. Woodruff, R. N. Watson, D. Chisnall, *et al.*, “The cheri capability model: Revisiting risc in an age of risk,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 457–468, 2014.

[151] T. Zhang, D. Lee, and C. Jung, “Bogo: Buy spatial memory safety, get temporal memory safety (almost) free,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 631–644.

[152] Y. Li, W. Tan, Z. Lv, *et al.*, “Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[153] anonymous, *The source code of CAMP*, https://anonymous.4open.science/r/CAMP-EE27.

[154] J. M. Mark Dowd and J. Schuh, *Magic Value: Potential Mitigations for Heap Overflow*, https://cwe.mitre.org/data/definitions/122.html.

[155] *Heap Cookies for memory protection*, https://fuzzysecurity.com/tutorials/mr_me/3.html.

[156] M. Phillips, *Design of Redzone in Address Sanitizer*, https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm.

[157] J. M. Mark Dowd and J. Schuh, *Protect Out-Of-Bound by Validating Pointer*, https://cwe.mitre.org/data/definitions/823.html.

[158] T. Yamauchi and Y. Ikegami, "Heaprevolver: Delaying and randomizing timing of release of freed memory area to prevent use-after-free attacks," in *Network and System Security: 10th International Conference, NSS 2016, Taipei, Taiwan, September 28-30, 2016, Proceedings 10*, Springer, 2016.

[159] O. Corporation, *Nullify references after reclaiming memory*, https://docs.oracle.com/cd/E19159-01/819-3681/abebi/index.html, 2010.

[160] B. Suchy, S. Campanoni, N. Hardavellas, and P. Dinda, "Carat: A case for virtual memory through compiler-and runtime-based address translation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[161] B. Suchy, S. Ghosh, D. Kersnar, *et al.*, "Carat cake: Replacing paging via compiler/kernel cooperation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[162] G. Inc., *Design of TCMalloc from Google*, https://google.github.io/tcmalloc/overview.html.

[163] V. Berinde and F. Takens, *Iterative approximation of fixed points*. Springer, 2007, vol. 1912.

[164] C. Smith, *Heap use-after-free in mruby*, https://github.com/mruby/mruby/issues/3515.

[165] *Issue 1325664: Security: pdfium use-after-free in v8*, https://bugs.chromium.org/p/chromium/issues/detail?id=1325664.

[166] *Heap-based Buffer Overflow in mruby*, https://huntr.dev/bounties/4458e0b9-0ad3-4036-a032-1b3c4705b889/.

[167] *CVE details: The ultimate security vulnerability datasource*, https://www.cvedetails.com/.

[168] *CVE program*, https://cve.mitre.org/.

[169] *VulnDB: The most comprehensive vulnerability database and timely source of intelligence available*, https://vuldb.com/.

[170] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[171] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–14.

[172] *CVE-2021-26259: A flaw was found in htmldoc in v1.9.12. Heap buffer overflow*, https://nvd.nist.gov/vuln/detail/CVE-2021-26259.

[173] *CVE-2022-43286: Nginx NJS v0.7.2 was discovered to contain a heap-use-after-free bug*, https://nvd.nist.gov/vuln/detail/CVE-2022-43286.

[174] *CVE-2019-16165: GNU cflow through 1.6 has a use-after-free in the reference function in parser.c*, https://nvd.nist.gov/vuln/detail/CVE-2019-16165.

[175] *CVE-2021-4187: vim is vulnerable to Use After Free*, https://nvd.nist.gov/vuln/detail/CVE-2021-4187,

[176] G. J. Duck and R. H. Yap, "Effectivesan: Type and memory error detection using dynamically typed c/c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 181–195.

[177] *Top Websites Ranking*, https://www.similarweb.com/top-websites/.

[178] H. Liu, R. Tian, B. Ren, and T. Liu, "Prober: Practically defending overflows with page protection," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[179] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical {page-permissions-based} scheme for thwarting dangling pointers," in *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 815–832.

[180] Microsoft, *Daan Leijen. 2020. Mimalloc*, https://github.com/microsoft/mimalloc.

[181] B. Liu, P. Olivier, and B. Ravindran, "Slimguard: A secure and memory-efficient heap allocator," in *Proceedings of the 20th International Middleware Conference*, 2019.

[182] D. V. Kostya Serebryany, *Scudo Hardened Allocator*, https://llvm.org/docs/ScudoHardenedAllocator.html.

[183] J. Caballero, G. Grieco, M. Marron, and A. 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*, 2012.

[184] G. J. Duck, Y. Zhang, and R. H. Yap, "Hardening binaries against more memory errors," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 117–131.

[185] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "Sanrazor: Reducing redundant sanitizer checks in c/c++ programs.," in *OSDI*, 2021.

[186] Y. Jeon, W. Han, N. Burow, and M. Payer, "{Fuzzan}: Efficient sanitizer metadata design for fuzzing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 249–263.

[187] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cup: Comprehensive user-space protection for c/c++," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 381–392.

[188] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[189] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 762–775.

[190] ooo, *DC29 final scoreboard*, https://scoreboard.ooo/scores.html, 2021.

[191] Anonymous, *Behavior representing exploitability*, https://forms.gle/vdPiASeYycqEzEi29, 2021.

[192] syzbot, *KASAN: Global-out-of-bounds read in fbcon_resize*, https://syzkaller.appspot.com/bug?id=ebcbbb6576958a496500fee9cf7aa83ea00b5920, 2020.

[193] syzbot, *Kernel BUG at security/keys/keyring.c:line!* https://syzkaller.appspot.com/bug?id=f7649aa07ffca82dc93dc5cebc00c665849f5138, 2019.

[194] syzbot, *WARNING in snd_info_get_line*, https://syzkaller.appspot.com/bug?id=27ea7ae6337aef698924e3eac5aa2b925374ca4c, 2020.

[195] syzbot, *KASAN: Use-after-free read*, https://syzkaller.appspot.com/bug?id=b7f48618d1139d02d0faba8e5932c51eec329b65, 2020.

[196] syzbot, *WARNING: Refcount bug in qdisc_put (2)*, https://syzkaller.appspot.com/bug?id=badc9136121e634336bcdb31592a4b70b064e421, 2020.

[197] syzbot, *KASAN: Use-after-free read in do_madvise*, https://syzkaller.appspot.com/bug?id=33913c931f51814eeb2ecdbe03af91d1d6d73520, 2020.

[198] syzbot, *Warning in snd_usbmidi_submit_urb/usb_submit_urb*, https://syzkaller.appspot.com/bug?id=28741ff1906f93db2a398bc40e082da51828ec5b, 2020.

[199] Linux, *File management in the linux kernel*, 2022.

APPENDIX A
APPENDIX

A.1 Additional Details of GREBE's Evaluation

A.1.1 Detail of User Study

To find the relationship between a kernel bug's error behavior and its exploitation potential, we designed a survey (i.e. Figure A.1) and conducted a user study with IRB approval. In our survey, we first asked the subjects' backgrounds, including their occupations and expertise levels. Following the background inquiry, we investigated whether the participants agree that, in most scenarios, the kernel error behaviors like double free, use-after-free, and out-of-bound access imply higher exploitation potential than the kernel error behaviors such as BUG, GPF, WARN, and NULL pointer dereference. We drew comparison between different error behaviors in the survey and provided examples to help the participants understand the context. For each comparison pair, the participant is required to briefly explain the reason if he/she disagrees with our classification.

We started our recruitment from CTF players in top-tier teams [190]. The invited players were encouraged to distribute our survey to knowledgeable experts further. We offer a $10 gift card for each participant to motivate the completion of our survey. In total, we managed to recruit 21 security experts participating in our survey. Among these human subjects, 12 of them claim themselves as CTF players or exploit practitioners. 14 are researchers in academia. 2 are members of blue team in an enterprise. Note that one participant can have multiple roles. Besides, 10 participants have experience in crafting Linux kernel exploits, 4 reading write-ups or exploits, and 7 debugging kernel and developing patches. As the subjects participating in the survey have

diverse backgrounds, we deem that the survey results reflect the viewpoint of most security experts regarding how to assess the exploitability of bugs according to their error behaviors.

Our survey showed that 18 out of 21 participants agree that, for all comparison pairs, error behaviors like double-free, use-after-free, and out-of-bound access imply higher exploitation potential. For the comparison between double-free and BUG/GPF, use-after-free and BUG/GPF, out-of-bound access, and BUG/GPF/WARN/NULL ptr deref, there are 1/2, 2/3, and 3/2/1/1 participants who disagree with our classification, respectively. They explained that, in the situation where the attacker can control the corruption range, errors like GPF/BUG/WARN could imply higher exploitation potential. In our user study, we further contacted those participants for further clarification. In the follow-up interview, they conceded that though they have encountered some particular cases, they agree that our classification works in most situations. As such, we carefully conclude that there is a shared sense among security analysts. That is, compared with error behaviors like GPF/WARN/BUG/NULL ptr deref, kernel error behaviors such as double free, use-after-free, and out-of-bound imply higher exploitation potential.

A.1.2 Procedure of Error Triaging

When exploring multiple error behaviors for a target bug, `GREBE` may hit other bugs, demonstrating error behaviors that do not result from the target bug. To ensure the newly identified error behaviors are truly tied to the bug of our interest, error triaging is needed. As we mentioned earlier, there has not yet been accurate error triaging methods. We, therefore, seek the help of kernel professionals.

In this work, our professional team performed error triaging by following the procedure below. Given a bug of our interest, the team first finds the bug's patch and applies it to the corresponding kernel image. Then, for each newly identified error behavior, the team executes the PoC program

1. Which of the following roles do you identify yourself as (multiple choices)?
   a. CTF player/exploit practitioner
   b. Academia researcher
   c. Security analyst in enterprise blue team
   d. Official in government agency

2. How's your experience in Linux kernel exploitation
   a. I've debugged kernel or developed kernel patches but done nothing about exploitation
   b. I've read some writeups
   c. I've written some exploits for CTF challenges or real-world vulnerabilities

3. What's the easiest way to get in touch with you? We ask this question for gift card sending and potential follow-up question. We promise to keep privacy and won't identify you via the contact.

4. Do you agree that, without going into details, double free behavior implies higher exploitability than BUG in most cases?
   a. Yes
   b. No
   c. I don't know

Figure A.1: Sampled questions from the exploitability survey form [191]

tied to that newly discovered error behavior. If the patch fails to block the demonstration of the error (i.e., the patched kernel still crashes), the team excludes that error behavior with the conclusion that it is not associated with the target bug. Otherwise, the team will put their effort into inspecting the execution of the PoC program. In the inspection phase, the team will manually examine the bug patch and extract the condition of the bug triggering. With this triggering condition in hand, the team further examines the execution of the PoC program. If the execution aligns with the triggering condition, the team safely concludes the newly discovered error is tied to the bug of our interest. To minimize the possible human mistake, we asked the team to form a unanimous agreement before we associate that new error behavior to the bug of our interest.

It should be noted that, like Syzkaller, when `GREBE` triggers a bug and demonstrates an error behavior, it may not generate a PoC program allowing the team to follow the procedure above. In this situation, the team will take a close look at the call stack of the kernel panic. Following the call stack, the team will manually track the kernel execution reversely and infer if the panic results from the same root cause. In this manual analysis phase, the team utilizes several heuristics to align an error with the bug. First, the team will confirm the functions in the call stack are relevant to the functions where the patch is applied. Second, the team will ensure the panic site is related to the variables that the patch influences.

As we can see, the rationale of the triaging procedure above is as follows. We assume that the patch could successfully block the triggering of the bug and thus prevent it from exhibiting the corresponding errors. If the patched kernel image still demonstrates errors, the manifested error is not likely to associate with the bug of our interest. However, it should be noted that the procedure above might mistakenly exclude some error behaviors tied to the bug of our interest simply because the patch might not be correct, and we falsely rule out the corresponding error behaviors. As a result, we emphasize that the error behaviors we identified could mean only the lower bound of

the total number of all possible error behaviors. However, as we showed in Section 2.6, the lower bound still provides good utility, helping a security researcher explore multiple error behaviors for a given kernel bug.

A.1.3 Detail of Distance Measurement & Hypothesis Validation

Section 2.6.2 hypothesizes that the distance (number of basic blocks) between a bug's root cause and the corresponding error site may correlate with the false negatives of GREBE. Here, we detail how we measure the distance and present our hypothesis testing result.

It is challenging to measure the distance between the root cause and the error site for a given bug. The Linux kernel is a multi-process system. The system call that triggers the root cause could be different from the one that brings about the error. As a result, we address this issue as follows. First, we identify all the lines of the kernel code that the patch changes. Second, we examine which of these lines is executed first when replaying the PoC program. In this work, we treat that line as our root cause site. If the patch site and the error site share the same system call, we simply count the basic blocks in between. For the kernel bug, the root cause of which and the error site reside in different system calls, we combine the basic blocks of both system calls. More specifically, we take the total number of basic blocks that the error-site-related system call has executed. Then, we add this number to the number of basic blocks that the root-cause-related system calls have executed (right after the root cause is triggered and before the error occurs).

In Table A.1 ("N of BB" column), we show the distance measure for the error manifested in the bug's original report. We mark the distance measure with a star sign if that bug's root cause and error site do not share the same system call. As we can observe from the table, there is no clear relation between the distance and the false positive. GREBE demonstrates false positives regardless of whether the distance is long or short enough. In addition, the false-negative occurrence does

```
1 struct key *request_key_auth_new(...)
2 {
3     struct request_key_auth *rka;
4
5     ...
6
7     /* allocate a auth record */
8     rka = kzalloc(sizeof(*rka), GFP_KERNEL);
9
10    ...
11    rka->cred = get_cred(irka->cred);
12    ...
13 }
14
15 static void free_request_key_auth(struct request_key_auth *rka)
16 {
17    ...
18    if (rka->cred)
19        put_cred(rka->cred);
20    ...
21    kfree(rka);
22 }
```

Listing 15: The allocation and deallocation sites for the object in the type of "struct request_key_auth".

not depend upon whether root cause and error site share (or not share) the same system call. With these observations, we safely reject our hypothesis.

A.2 Identifying Credential Object

As is mentioned in Section 3.7.1, we designed and implemented an automated tool to facilitate the identification of those exploitable objects (*i.e.,* the objects enclosing a reference to credential objects). Here, we discuss how we design this automated tool step-by-step and then describe our implementation in detail.

Table A.1: This table shows false negative analysis results. The ”SYZ ID” is the case ID, with the second column showing the basic block count between the crash and root cause site. A following $\star$ indicates the sites are from different syscalls; otherwise, the same. The third and fourth columns indicate new behaviors found manually and via GREBE, respectively. A ”-” signifies no behavior found.

SYZ ID	N of BB	New Behaviors Discovered Manually	New Behaviors Discovered by GREBE
1fd1d44[47]	5128 $\star$	general protection fault in skcipher_walk_done	general protection fault in skcipher_walk_done
695527b[49]	2313 $\star$	KASAN: use-after-free Write in bpf_tcp_close	KASAN: use-after-free Write in bpf_tcp_close
		BUG: unable to handle kernel paging request in qlist_free_all	BUG: unable to handle kernel paging request in qlist_free_all
		WARNING: ODEBUG bug in sock_hash_free	-
ebcbbb6[192]	1	-	-
f7649aa[193]	38	-	-
6a03985[53]	2	general protection fault in hfsc_unbind_tcf	general protection fault in hfsc_unbind_tcf
		WARNING: refcount bug in __tcf_action_put	WARNING: refcount bug in __tcf_action_put
		KASAN: use-after-free Read in route4_get	KASAN: use-after-free Read in route4_get
		WARNING: refcount bug in route4_destroy	WARNING: refcount bug in route4_destroy
		KASAN: null-ptr-deref Read in route4_destroy	KASAN: null-ptr-deref Read in route4_destroy
		KASAN: use-after-free Read in route4_destroy	KASAN: use-after-free Read in route4_destroy
27ea7ae[194]	1	-	-
d5222b3[54]	652 $\star$	WARNING: bad unlock balance in ucma_destroy_id	WARNING: bad unlock balance in ucma_destroy_id
		general protection fault in rdma_listen	general protection fault in rdma_listen
		KASAN: use-after-free Read in addr_handler	-
		KASAN: use-after-free Read in cma_cancel_operation	-
		KASAN: use-after-free Read in rdma_listen	KASAN: use-after-free Read in rdma_listen
		BUG: corrupted list in rdma_listen	BUG: corrupted list in rdma_listen
de28cb0[35]	2	BUG: corrupted list in neigh_mark_dead	BUG: corrupted list in neigh_mark_dead
		KASAN: use-after-free Read in neigh_mark_dead	KASAN: use-after-free Read in neigh_mark_dead
		KASAN: slab-out-of-bounds Read in neigh_mark_dead	KASAN: slab-out-of-bounds Read in neigh_mark_dead
		KASAN: use-after-free Read in ___neigh_create	KASAN: use-after-free Read in ___neigh_create
		KASAN: slab-out-of-bounds Read in ___neigh_create	KASAN: slab-out-of-bounds Read in ___neigh_create
		KASAN: use-after-free Read in neigh_change_state	KASAN: use-after-free Read in neigh_change_state
f56bbe6[37]	2	KASAN: slab-out-of-bounds Read in qrtr_endpoint_post	KASAN: slab-out-of-bounds Read in qrtr_endpoint_post
b7f4861[195]	1	-	-
e4be308[60]	857 $\star$	KASAN: slab-out-of-bounds Write in tgr192_final	KASAN: slab-out-of-bounds Write in tgr192_final
		KASAN: slab-out-of-bounds Write in tgr160_final	KASAN: slab-out-of-bounds Write in tgr160_final
		KASAN: slab-out-of-bounds Write in crypto_sha3_final	KASAN: slab-out-of-bounds Write in crypto_sha3_final
		KASAN: slab-out-of-bounds Write in rmd320_final	KASAN: slab-out-of-bounds Write in rmd320_final
		KASAN: slab-out-of-bounds Write in wp384_final	KASAN: slab-out-of-bounds Write in wp384_final
		KASAN: slab-out-of-bounds Write in sha512_finup	KASAN: slab-out-of-bounds Write in sha512_finup
		KASAN: slab-out-of-bounds Write in sha1_finup	KASAN: slab-out-of-bounds Write in sha1_finup
		KASAN: slab-out-of-bounds Write in sha1_final	KASAN: slab-out-of-bounds Write in sha1_final
		KASAN: slab-out-of-bounds Write in sha256_final	KASAN: slab-out-of-bounds Write in sha256_final
		KASAN: slab-out-of-bounds Write in rmd160_final	KASAN: slab-out-of-bounds Write in rmd160_final
		KASAN: slab-out-of-bounds Write in sha256_finup	KASAN: slab-out-of-bounds Write in sha256_finup
d1baeb1[34]	1043 $\star$	general protection fault in skb_release_data	general protection fault in skb_release_data
		general protection fault in skb_clone	-
		KASAN: wild-memory-access Read in skb_copy_ubufs	KASAN: wild-memory-access Read in skb_copy_ubufs
		KASAN: slab-out-of-bounds Write in pskb_expand_head	KASAN: slab-out-of-bounds Write in pskb_expand_head
7022420[18]	64	KASAN: slab-out-of-bounds Write in default_read_copy_kernel	KASAN: slab-out-of-bounds Write in default_read_copy_kernel
0df4c1a[42]	553	KASAN: use-after-free Read in remove_wait_queue	KASAN: use-after-free Read in remove_wait_queue
		KASAN: use-after-free Read in corrupted	KASAN: use-after-free Read in corrupted
		KASAN: use-after-free Read in eventfd_release	KASAN: use-after-free Read in eventfd_release
badc913[196]	1510 $\star$	-	-
33913c9[197]	1	KASAN: use-after-free Read in do_madvise	KASAN: use-after-free Read in do_madvise
28741ff[198]	2	WARNING in snd_usbmidi_submit_urb/usb_submit_urb	WARNING in snd_usbmidi_submit_urb/usb_submit_urb
0df4c1a[42]	6	BUG: unable to handle kernel paging request in pcpu_freelist_populate	BUG: unable to handle kernel paging request in pcpu_freelist_populate
		BUG: unable to handle kernel paging request in htab_map_alloc	BUG: unable to handle kernel paging request in htab_map_alloc
		BUG: unable to handle kernel paging request in bpf_lru_populate	BUG: unable to handle kernel paging request in bpf_lru_populate
		KASAN: vmalloc-out-of-bounds Write in pcpu_freelist_populate	KASAN: vmalloc-out-of-bounds Write in pcpu_freelist_populate
		KASAN: vmalloc-out-of-bounds Write in bpf_lru_populate	KASAN: vmalloc-out-of-bounds Write in bpf_lru_populate
		KASAN: vmalloc-out-of-bounds Write in htab_map_alloc	KASAN: vmalloc-out-of-bounds Write in htab_map_alloc
		KASAN: vmalloc-out-of-bounds Read in htab_free_elems	KASAN: vmalloc-out-of-bounds Read in htab_free_elems
		BUG: unable to handle kernel paging request in htab_free_elems	-

A.2.1 Design

Step 1: analyzing structure definition. Recall that an exploitable object should include a pointer referencing a credential object. Therefore, the first step in identifying an exploitable object is to analyze the definition of kernel data structures. This analysis could bound our consecutive analysis in a scope, avoiding unnecessary analysis of irrelevant data structures in the following steps.

Given a data structure, we go through each field based on the definition. If a field is a nested structure or union type, we also extract its fields and examine them accordingly. In this work, we follow this procedure recursively until all the structure fields are thoroughly analyzed. Along with this analysis, we also examine the type information of pointers. If a field pointer references a credential structure type (e.g., file or cred), we record the field's offset in the enclosed structure and mark the structure type as a candidate.

Step 2: identifying allocation sites. As is described in the main text, DirtyCred performs privilege escalation by exploiting heap-based memory corruption vulnerabilities. Therefore, we need to ensure that the objects in the identified structure type can be allocated on the kernel heap. To do it, we first pinpoint each code site that allocates an object on the kernel heap (see the object allocation example code snippet in List 15). Second, we examine the return value of the heap (de)allocation function. Following the data flow of the allocated object, we extract the object's type information and check if it matches the structure candidates identified in the first step. We record the allocation site for the corresponding object if a match exists.

Step 3: pinpointing free sites. Recall that DirtyCred needs to deallocate credential objects. As a result, we also need to guarantee that when deallocating an exploitable object candidate, the kernel could also free the corresponding credential object along the way. In the Linux kernel, credential objects have their unique properties. Their deallocation is carried out through dedicated,

standard kernel API functions [88], [199]. In this work, we summarize these deallocation functions manually and then use these functions as the starting point to perform our analysis. Specifically, for each code site where the credential-object deallocation function is invoked, we taint corresponding arguments and then perform a backward data-flow analysis. Our backward analysis terminates until it identifies a site where the credential object is initialized. We examine the initialization site and extract the type information of the initialized object. If the object type matches our structure candidate, we record the deallocation site and conclude that the object candidate has the potential to facilitate DirtyCred's privilege escalation.

Step 4: tracking down reachable objects. Note that the objects identified above may not be the ones under the user's control. For example, the objects might be capable of allocation only at the booting phase of the Linux. In this sense, DirtyCred cannot use these objects for memory manipulation and thus pivot a vulnerability's capability. Therefore, the last step is to examine whether the candidate objects can be allocated and freed through user-permitted system calls. To do it, we leverage a kernel fuzzer to explore the reachability of candidate objects' (de)allocation sites. Given a candidate object identified in the first three steps, if the fuzzer could trigger its allocation and deallocation sites using permitted system calls, we mark it as a valuable object for DirtyCred. For those that kernel fuzzer cannot reach out to the (de)allocation sites, we rely on our manual effort to analyze their reachability.

A.2.2 Implementation

To enable the first three analysis steps, we implemented a static analysis tool on top of LLVM. The analysis tool takes as input the kernel bitcode. To prevent the bitcode from being optimized, which might lose type information and make the data flow complex, we used a customized clang to generate bitcode before any code optimization is invoked. The tool contains 3,382 lines of C++

code in total. Our implementation is available at [94].

To complete the last step of the analysis, we utilized the state-of-the-art kernel fuzzer – Syzkaller [11]. To enable Syzkaller to report (de)allocation sites' reachability, we inserted a panic function at each site where a candidate object is (de)allocated. Once the Syzkaller generates an input that reaches the site, the kernel will experience panic, informing Syzkaller that the site has been reached. Benefited from the advance of kernel fuzzing, Syzkaller will also output a minimized input reachable to the corresponding (de)allocation sites if it triggers the corresponding panic functions. It should be noted that Syzkaller relies on syscall templates to generate the input and thus dynamically test the kernel. For some kernel modules, the templates have not yet been supported by Syzkaller. In this situation, we confirm the existence of (de)allocation manually.

A.3 CAMP's Performance on SPEC CPU2006

Benchmarks	Time					
	CAMP	**ASAN−−**	**ASAN**	**ESAN**	**Softbound+CETS**	**MEMCHECK**
400.perlbench	300.23%	256.68%	285.49%	629.16%	-	3407.29%
401.bzip2	48.90%	43.98%	48.22%	114.70%	354.91%	912.38%
403.gcc	67.86%	175.04%	173.97%	600.95%	-	1877.21%
429.mcf	27.25%	24.06%	34.85%	158.27%	634.40%	303.66%
433.milc	9.04%	38.82%	48.64%	50.76%	239.02%	1194.67%
445.gobmk	28.14%	36.86%	38.84%	52.46%	356.30%	2418.05%
456.hmmer	119.45%	90.07%	89.83%	270.44%	477.35%	1647.21%
458.sjeng	10.26%	40.23%	48.64%	13.35%	264.45%	2329.69%
462.libquantum	18.74%	14.75%	20.11%	197.61%	-	553.84%
464.h264ref	126.17%	89.88%	125.67%	326.47%	-	2689.47%
470.lbm	7.39%	25.82%	28.41%	51.63%	141.17%	5236.58%
482.sphinx3	114.08%	44.70%	52.97%	199.55%	-	4207.37%
444.namd	90.43%	75.60%	82.01%	57.64%	-	4076.65%
450.soplex	64.42%	42.82%	44.45%	128.66%	-	1518.52%
453.povray	113.95%	105.89%	150.95%	266.29%	-	5385.34%
473.astar	112.80%	30.62%	37.97%	80.75%	-	1085.92%
483.xalancbmk	297.90%	166.85%	203.05%	48.80%	-	5158.20%
Geomean	54.92%	56.77%	67.02%	123.08%	319.75%	1990.02%

Table A.2: Time overhead of CAMP, ASAN--, ASAN, ESAN, Softbound+CETS, MemCheck on the SPEC CPU2006. ”-” means the case where the tool failed to run the benchmark.

Benchmarks	Memory					
	CAMP	**ASAN−−**	**ASAN**	**ESAN**	**Softbound+CETS**	**MEMCHECK**
400.perlbench	1522.09%	339.64%	285.49%	1.90%	-	163.14%
401.bzip2	0.05%	2.75%	48.22%	-0.98%	127.90%	33.59%
403.gcc	109.95%	187.98%	173.97%	-6.78%	-	44.12%
429.mcf	51.66%	12.02%	34.85%	-0.56%	396.77%	2.29%
433.milc	379.00%	36.77%	48.64%	-1.58%	89.01%	10.50%
445.gobmk	143.05%	612.69%	38.84%	-9.29%	638.64%	230.80%
456.hmmer	4.32%	1061.04%	89.83%	-39.34%	-12.09%	197.22%
458.sjeng	-0.10%	-1.69%	48.64%	-3.26%	1.18%	44.82%
462.libquantum	-0.41%	185.86%	20.11%	-21.56%	-	48.34%
464.h264ref	18.82%	330.51%	125.67%	-19.84%	-	120.71%
470.lbm	-0.01%	11.47%	28.41%	-1.41%	-1.55%	34.39%
482.sphinx3	4093.04%	650.28%	52.97%	-4.03%	-	240.28%
444.namd	3.63%	9.93%	82.01%	-8.45%	-	101.86%
450.soplex	12.65%	44.26%	44.45%	-20.79%	-	21.20%
453.povray	7101.36%	2020.28%	150.95%	-12.61%	-	1188.33%
473.astar	1840.76%	176.09%	37.97%	5.29%	-	50.27%
483.xalancbmk	1800.53%	187.89%	203.05%	14.54%	-	77.51%
Geomean	237.67%	181.81%	180.94%	-8.45%	102.25%	97.14%

Table A.3: Memory overhead of CAMP, ASAN--, ASAN, ESAN, Softbound+CETS, MemCheck on the SPEC CPU2006. ”-” means the case where the tool failed to run the benchmark.