Reviving Discarded Vulnerabilities: Exploiting Previously Unexploitable Linux Kernel Bugs Through Control Metadata Fields

Hao Zhang[†]
zhanghao1018@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China

Jian Liu[*][†]
liujian6@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China

Jie Lu[*]
lujie@ict.ac.cn
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Shaomin Chen[†]
chenshaomin@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China

Tianshuo Han[†]
hantianshuo@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China

Bolun Zhang[†]
zhangbolun@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China

Xiaorui Gong[†]
gongxiaorui@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China

Abstract

Linux kernel vulnerabilities represent a critical security threat in modern computing systems, with hundreds of new vulnerabilities discovered annually. Traditional security practices often discard vulnerabilities offering only weak primitives as "unexploitable", creating a significant blind spot in kernel security. This paper presents a novel approach to revive these previously discarded vulnerabilities by exploiting Control Metadata Fields (CMFs) within Linux objects, rather than traditional pointer manipulation.

Our CMF-based method overcomes two major limitations of existing approaches: it bypasses modern security measures like pointer authentication code and eliminates the need for precise pointer alignment that weak primitives cannot reliably achieve. Using MetaXploit, our automated analysis tool, we identified 54 exploitable CMFs across Ubuntu and Debian distributions. Empirical testing against 20 real-world vulnerabilities demonstrated successful exploitation in 18 cases, including 12 previously discarded vulnerabilities. These results challenge traditional assumptions about vulnerability exploitability and suggest that many discarded vulnerabilities in the Linux kernel warrant reevaluation.

[*]Corresponding authors.
[†]Also with School of Cyber Security, University of Chinese Academy of Sciences.

CCS Concepts

• **Security and privacy → Operating systems security**.

Keywords

OS Security; Kernel Exploitation; Weak Primitive Exploitation

ACM Reference Format:
Hao Zhang, Jian Liu, Jie Lu, Shaomin Chen, Tianshuo Han, Bolun Zhang, and Xiaorui Gong. 2025. Reviving Discarded Vulnerabilities: Exploiting Previously Unexploitable Linux Kernel Bugs Through Control Metadata Fields. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744841

1 Introduction

Linux kernel vulnerabilities are among the most critical security threats in modern computing systems, as the kernel [50] orchestrates essential system operations across a vast range of devices, from cloud servers to embedded systems. With hundreds of new vulnerabilities discovered annually [63], the sheer volume of issues challenges the Linux security community to prioritize remediation efforts. To address this, understanding and classifying vulnerability exploitability is essential, as it provides a systematic framework for identifying and mitigating the most severe threats.

To systematically evaluate vulnerability exploitability, the Linux security community relies on the concept of **exploit primitives**. These primitives refer to unintended read or write operations that attackers leverage to initiate exploits. Among these, write primitives are especially significant due to their role in enabling critical attack techniques. Write primitives are generally classified into three types based on their strength: weak primitives (restricted location or uncontrollable content, < 8 bytes), strong primitives (restricted location and controllable content, ≥ 8 bytes), and ultimate primitives (controllable location and content, ≥ 8 bytes).[1]

Historically, vulnerabilities that provide ultimate primitives have been considered the most dangerous because they enable attacks such as control flow hijacking and privilege escalation [61, 64]. Recent research [7] has demonstrated methods to escalate strong primitives to ultimate primitives, further solidifying this perspective. However, the process of escalating weak primitives into strong primitives remains underexplored. We notice that recently developed exploitation techniques have started to explore how weak primitives can be escalated and used for attacks. This underscores the pressing need for systematic research on weak primitive escalation, as weak primitive escalation is essentially a way to understand and address the security risks they may pose.

In this paper, we aim to develop methodologies for escalating weak primitives into strong ones, enabling the exploitation of previously non-exploitable vulnerabilities. While prior research [26] has explored similar objectives by manipulating pointers within Linux kernel objects, these approaches face two major technical challenges. First, modern operating systems employ advanced security mechanisms such as pointer authentication code (PAC), which introduces hardware-level protections against pointer manipulation attacks [9, 58]. Second, existing methods rely on precise pointer modifications that are challenging to achieve with weak primitives. Exploitation often requires aligning pointer values with adjacent object addresses, which involves ensuring specific pointer alignment according to the size requirements of the target object (e.g., ensuring the lower bits are zero for power-of-two-sized objects). However, weak primitives inherently lack the fine-grained control necessary for such modifications, imposing a fundamental constraint that limits the practicality of these techniques in real-world scenarios.

To address the above limitations, we shift our focus to metadata fields within kernel objects. These fields, such as flags attributes, are important but often ignored. We observe two key points: First, these fields are not well-protected by modern hardware or software defenses, unlike pointers. Second, changes to these fields can influence program control flow and memory operations. This creates a new and flexible way to escalate weak primitives.

We present our study's core definition of Control Metadata Field (CMF): CMFs are metadata fields that can alter the control flow of program when manipulated, while these changes can trigger critical memory operations or extend write capabilities. For example, an attacker can exploit a weak primitive that only allows modifying a single byte by targeting CMF that control memory write loop termination conditions. By doing so, the attacker can force the kernel to write more data than intended, such as exceeding the expected 8 bytes. This transforms a weak attack into a more powerful one. Moreover, a vulnerability with only a weak primitive can manipulate CMF to force the kernel to execute unintended free operations in the hijacked branch, resulting in use after free or double free vulnerability. This effectively demonstrates another approach to escalate weak primitives into strong ones.

[1]A detailed explanation of these categories will be provided in Section 2.1.2.

Building upon our observation, we propose a systematic methodology named MetaXploit for identifying and verifying metadata fields that can be exploited for primitive escalation in the Linux kernel. MetaXploitconsists of two complementary phases: static analysis and dynamic verification.

In the static analysis phase, we identify metadata fields that potentially influence program control flow, with a particular focus on branches involving critical memory operations such as `memcpy` or `kfree`. While static analysis provides a comprehensive list of candidate fields, its lack of runtime context can result in false positives.

In the dynamic verification phase, we implemented a two-stage dynamic testing process. The first stage generates comprehensive test cases, maximizing coverage of code paths influenced by the identified metadata. The second stage modifies metadata values to simulate real-world exploitation scenarios while monitoring execution branch changes and memory corruption, thereby validating the actual exploitation.

By combining static and dynamic approaches, MetaXploit efficiently identifies and validates the exploitability of CMFs, providing a systematic solution for weak primitive escalation. Our experimental results validate this approach: static analysis discovered 833 potential CMFs across Ubuntu and Debian distributions, with 54 verified through dynamic analysis. Further dynamic evaluation on 20 real-world vulnerabilities confirmed these metadata fields' effectiveness in escalating weak primitives to strong ones.

We further investigated 54 verified metadata fields, and categorized them into three types: flags, reference counts, and loop counters. Our work makes several key contributions in exploring these fields for primitive escalation. First, we demonstrate that flags can be exploited for primitive escalation, a technique not explored in previous work. Second, although reference counting fields were historically used for exploitation [51] and subsequently protected in recent kernel versions [28], we reveal that these protections remain insufficient (detailed in Section 3) and identify new exploitation opportunities. Third, for loop counters, while prior research [6] focused on directly modifying lengths to trigger buffer overflows, we demonstrate their effectiveness in manipulating program control flow for weak primitive escalation. Furthermore, we uncover the potential of previously unexplored types of loop counters. To facilitate future research endeavors in this domain, we have open-sourced our implementation as well as evaluation artifacts at https://github.com/Roarcannotprogramming/Weak-Primitive.

The main contributions of this paper are as follows:

- We discovered a novel category of fields - CMFs - that can be used to achieve primitive escalation. Modifying these fields can alter control flow branches to accomplish primitive escalation.
- We propose a systematic methodology - MetaXploit - that combines static and dynamic analysis to identify CMFs within the kernel and confirmed their applicability for primitive escalation.
- We identified 54 fields capable of primitive escalation. We evaluated the capabilities of CMFs on 20 real-world weak-primitives vulnerabilities in the Linux kernel over the past five years. Among these, we achieved primitive escalation on 18 weak-primitives vulnerabilities, 12 of which lack publicly available exploitation techniques that demonstrate their exploitability.

2 Problem Statement

2.1 Preliminaries

Prior to presenting our research problem, we first introduce several key concepts and background knowledge that are essential to understanding our work.

2.1.1 Primitives. In the context of kernel exploitation, primitives refer to the fundamental capabilities that an attacker can obtain through vulnerabilities. These primitives serve as essential building blocks for constructing sophisticated exploitation chains. This paper focuses on write-based primitives because they present more severe security implications and can be effectively transformed into read primitives as revealed in prior works [6, 26].

2.1.2 Weak/Strong/Ultimate Write Primitive.

Definition 2.1 (Write Primitive). A write primitive P is defined as a tuple:

$$P = (L, C, S)$$

where L denotes the write target location, C represents the written content, and S indicates the write size.

The exploitation potential of a write primitive is fundamentally determined by its capability to illegally manipulate kernel memory. We evaluate this capability using the location (L), content (C), and size (S). As illustrated in Figure 1(a), based on <L, C, S>, we classify write primitives into three categories: weak, strong and ultimate.

Definition 2.2 (Weak Primitive). A primitive is classified as weak if it satisfies the following condition:

$$((|S| < 8 \text{ bytes}) \vee (\neg C_{\text{controllable}})) \wedge (L_{\text{restricted}})$$

The 8-byte threshold corresponds to the address bus width in modern architectures, which determines the size of pointers that need to be controlled. Furthermore, the exploitation techniques presented in [10, 20, 25, 27, 55] all require a minimum of 8 bytes to achieve successful exploitation. Content being not controllable indicates that an attacker cannot control or manipulate the content being written to memory. The location restrictions typically manifest in two forms: adjacent chunk, where writes are limited to memory chunks adjacent to the vulnerable object (e.g., heap buffer overflow), and reallocated memory, where writes are confined to memory that has been reallocated to the same location (e.g., use-after-free).

In real-world kernel exploitation scenarios, these weak primitives commonly belong to Out-of-Bounds writes or Use-After-Free vulnerabilities. A representative example is CVE-2021-22555, which features a 2-byte NULL overflow vulnerability, demonstrating the practical limitations in both write size and content control that characterize weak primitives.

Definition 2.3 (Strong Primitive). A write primitive is classified as strong if it satisfies the following condition:

$$((|S| \geq 8 \text{ bytes}) \wedge (C_{\text{controllable}})) \wedge (L_{\text{restricted}})$$

In real-world kernel exploitation, strong primitives commonly belong to Out-of-Bounds (OOB) writes or Use-After-Free (UAF) writes. Additionally, double-free vulnerabilities in kernel exploitation also have strong primitives, as they can be readily converted to the aforementioned strong UAF write primitives [57]. As shown in Figure 1(b), after a double free, different objects (e.g., Object 1 and the user-controlled Object 2) can occupy the same memory space, allowing attacker-controlled writes.

Definition 2.4 (Ultimate Primitive). A write primitive is classified as ultimate if it satisfies the following condition:

$$((|S| \geq 8 \text{ bytes}) \wedge (C_{\text{controllable}})) \wedge (L_{\text{arbitrary}})$$

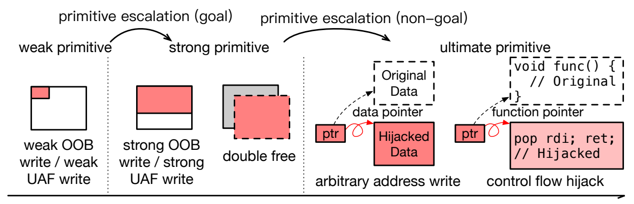
L being arbitrary means the write can target any location in memory, including critical control flow targets such as function pointers. In real-world kernel exploitation scenarios, ultimate primitives represent the most powerful class of memory corruption vulnerabilities, as they enable either unrestricted memory manipulation or direct control over kernel execution flow. A representative example is CVE-2019-2215, which demonstrates a powerful arbitrary write primitive that allows attackers to write controlled content to any kernel memory address.

2.1.3 Heap Objects, Metadata Fields and CMFs. Heap objects in the Linux kernel are memory blocks allocated through kernel memory allocation functions (such as `kmalloc`) and typically correspond to specific structures defined in the kernel source code. These objects contain both primary data and metadata fields. While primary data serves the object's main functionality, metadata fields contain supplementary information that describes the object's attributes.

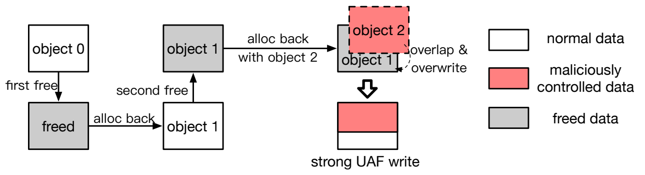
Among various metadata fields, Control Metadata Fields (CMFs) represent a special category that directly influences kernel execution flow through conditional statements. These fields, when used in conditional branches, determine the execution path of kernel code. If an attacker can manipulate these fields through a vulnerability, they can potentially alter the intended control flow of kernel execution. The hijacked branch can contain unauthorized memory operations, including unintended memory write operations and memory deallocation calls, ultimately leading to memory corruption.

2.1.4 Primitive Escalation. Primitive escalation refers to the systematic process of transforming a primitive into a more powerful one through strategic manipulation of heap objects. This transformation can be categorized into two types: (1) escalating a weak primitive to a strong primitive, or (2) escalating a strong primitive to an ultimate primitive to gain either arbitrary write capability or control flow hijacking ability. Both types rely on heap object manipulation, but they differ in their specific techniques and targets. In this paper, we focus on transforming weak primitives into strong primitives through the manipulation of CMFs in heap objects.

2.1.5 Heap Grooming and Cross-page Technique. Heap grooming is a common technique in binary exploitation and plays a crucial role in kernel vulnerability exploitation [60]. It enables the strategic positioning of vulnerable objects and victim objects in adjacent locations (in Out-Of-Bounds scenarios) or overlapping positions (in Use-After-Free scenarios). Traditional heap grooming is implemented by controlling allocation and deallocation sequences—for example, consecutively allocating vulnerable and victim objects, or



(a) Primitive Escalation from weak to strong and ultimate primitives.



(b) Converting a double-free primitive into a strong UAF write primitive.

Figure 1: Primitives and primitive escalation in the Linux kernel. The diagram utilizes a color-coded representation where white boxes denote normal data, red boxes indicate potentially maliciously controlled data, and gray boxes represent freed data.

immediately allocating a victim object after freeing a vulnerable one. The cross-page technique [14], which has gained prominence in kernel exploitation, operates on principles similar to heap grooming but extends its scope to the page level. This advancement enables the strategic arrangement of memory pages in adjacent or overlapping positions, effectively circumventing traditional memory isolation constraints. Notably, this technique facilitates the exploitation of objects that would otherwise remain segregated under normal circumstances. Exploitation techniques leveraging CMFs follow similar principles, necessitating heap grooming for strategic object positioning. The cross-page technique enhances these capabilities by extending the applicability of CMF-containing objects across diverse vulnerability scenarios. This extension is particularly significant as it overcomes the isolation constraints typically imposed by SLUB allocator mechanisms, enabling exploitation regardless of the natural segregation between vulnerable objects and those containing CMFs.

2.2 Problem Definition and Threat Model

Goal. MetaXploitaims to identify CMFs that can be modified through vulnerabilities with weak primitives, thereby altering control flow to induce illegal kernel memory operations. Specifically, these operations encompass memory writes and memory deallocations, which can be leveraged to achieve 8+ byte attacker-controlled memory operations including OOB write,UAF write, and double-free, thereby effectively escalating the primitive capabilities. To ensure effective escalation, CMFs must satisfy three fundamental rules:

- **User Accessibility** The objects containing CMFs must be creatable by unprivileged users during exploitation.
- **Conditional Statement Dependency** The CMF must be able to control conditional statements through data flow, ensuring that modifications can effectively influence program control flow.
- **Illegal Memory Operation** The hijacked control flow must lead to illegal memory operations - specifically, memory writes causing OOB writes, or memory deallocations resulting in UAF writes or double frees - enabling primitive escalation.

Non-goals. Our research scope explicitly excludes several related areas in exploit development. First, MetaXploitdoes not address the transformation techniques between strong primitives and ultimate primitives, as this topic has been extensively covered in existing literature [10, 20, 25, 27, 55]. Second, MetaXploitdoes not focus on vulnerability detection mechanisms. Finally, while MetaXploitcan identify CMFs to assist exploitation, we do not pursue the automated generation of Proof of Concept exploits that leverage these CMFs.

Threat Model. Our threat model encompasses a kernel heap vulnerability (such as OOB and UAF) that provides a limited write primitive within the Linux kernel space. In this scenario, an unprivileged user attempts to leverage this weak write capability to achieve privilege escalation to root access. The analysis considers contemporary Linux kernel deployments with comprehensive security mitigations enabled, specifically focusing on the latest stable releases of Ubuntu and Debian distributions. These security measures include Kernel Address Space Layout Randomization (KASLR), Supervisor Mode Access Prevention (SMAP), Supervisor Mode Execution Prevention (SMEP), and Kernel Page Table Isolation (KPTI). Additionally, the model accounts for advanced heap-specific protections, notably SLAB FREELIST RANDOMIZATION and SLAB FREELIST HARDENING, which were implemented to counteract previously documented exploitation techniques.

3 CMF-Based Exploitation

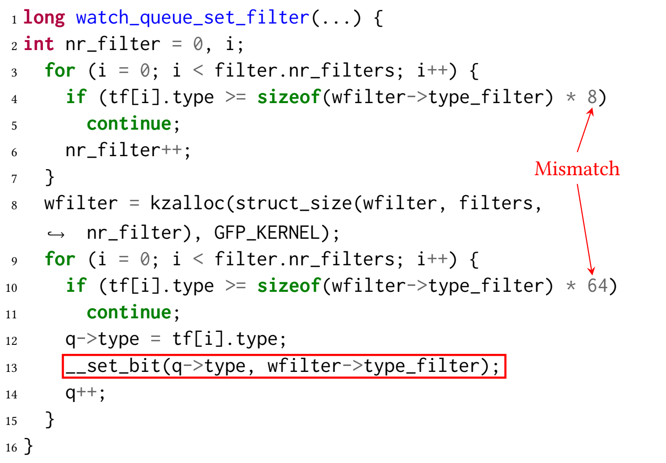
In this section, we demonstrate how to leverage a CMF to escalate the weak primitive of a vulnerability into the strong primitive. Additionally, we elaborate on the advantages of utilizing CMFs for this elevation process compared to existing approaches.

3.1 An Example of Weak Primitive Vulnerability

Figure 2 illustrates the vulnerability-related code segment (CVE-2022-0995) within the Linux kernel. The code implementation follows a two-phase approach: initially, it enumerates valid filters (lines 3–7) by validating type boundaries using an 8-multiplier constraint, followed by memory allocation (line 8), and concludes with filter bit configuration (lines 9–15). The `__set_bit` operation at line 13 assigns a value of 1 to the bit position specified by `q->type` within the `wfilter->type_filter` bitmap, specifically at the byte offset `q->type/8` and bit position `q->type%8`. A vulnerability emerges from the inconsistent boundary validation between the two iterative loops (comparing line 4 versus line 10), potentially resulting in Out-of-Bounds Write when processing tf[i].type that fall within the range [sizeof(wfilter->type_filter) × 8, sizeof(wfilter->type_filter) × 64).

Following Definition 2.1, this vulnerability can be formally defined as P = (L, C, S) where:

- **L** (Location): `wfilter->type_filter + (q->type / 8)`
  - The target byte position within the `type_filter` bitmap



```c
long watch_queue_set_filter(...) {
int nr_filter = 0, i;
  for (i = 0; i < filter.nr_filters; i++) {
    if (tf[i].type >= sizeof(wfilter->type_filter) * 8)
      continue;
    nr_filter++;
  }
  wfilter = kzalloc(struct_size(wfilter, filters,
  ↪  nr_filter), GFP_KERNEL);
  for (i = 0; i < filter.nr_filters; i++) {
    if (tf[i].type >= sizeof(wfilter->type_filter) * 64)
      continue;
    q->type = tf[i].type;
    __set_bit(q->type, wfilter->type_filter);
    q++;
  }
}
```

Figure 2: CVE-2022-0995, A OOB vulnerability with a weak primitive that can only write 1 bit.

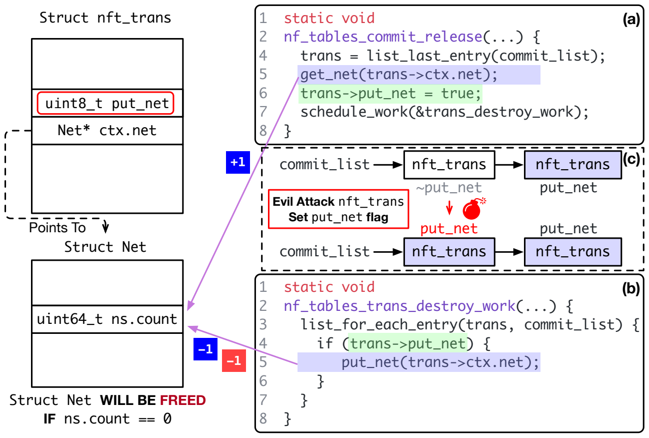
  - Offset is determined by q->type, converted to byte addressing through division by 8
  - Affects the bit position determined by q->type % 8 within the target byte
- **C** (Content): 1
  - The __set_bit operation sets the specified bit to 1
- **S** (Size): 1 bit
  - Each write operation modifies exactly one bit through the __set_bit operation.

Following Definition 2.2, this vulnerability exhibits characteristics consistent with a weak primitive, i.e., the write operation is constrained to writing value 1, with a size of 1 bit, and can only be performed at restricted memory locations.

3.2 CMF-Based Primitive Escalation

CMF introduces a novel approach to vulnerability exploitation. Instead of attempting to directly convert weak primitives into ultimate primitives, CMF first escalates weak primitives into strong primitives. Then, it utilizes existing techniques to transform these strong primitives into ultimate primitives. Below, we present a motivating example identified by MetaXploit that demonstrates how CMF can exploit the weak primitive vulnerability discussed in Section 3.1.

Figure 3 illustrates the nft_trans object in the netfilter table subsystem. This object represents a specific user-submitted transaction (such as creating or deleting packet filtering rules). It contains a put_net flag and a ctx.net pointer that points to a Net object. A group of transactions within the same network namespace share the same ctx.net pointer pointing to the same Net object. To manage this group of transactions, the put_net flag was introduced. Before the execution starts at line 7 in Figure 3(a), the put_net flag of the last transaction in the group is set to true (line 6), and the reference count of Net is incremented through get_net to prevent premature release of Net. The flags of other transactions remain false. When this group of transactions completes, based on whether the put_net flag is true, the reference count is decremented through put_net (Figure 3(b) line 5) .



```c
1  static void
2  nf_tables_commit_release(...) {
4    trans = list_last_entry(commit_list);
5    get_net(trans->ctx.net);
6    trans->put_net = true;
7    schedule_work(&trans_destroy_work);
8  }
```

```c
1  static void
2  nf_tables_trans_destroy_work(...) {
3    list_for_each_entry(trans, commit_list) {
4      if (trans->put_net) {
5          put_net(trans->ctx.net);
6      }
7    }
8  }
```

Figure 3: Constructing a double-free by exploiting the put_net metadata in the nft_trans object.

However, as shown in Figure 3(c), we can modify the put_net flag of another non-last transaction to true using the write-1 operation from CVE-2022-0995 (discussed in Section 3.1). This causes line 5 in Figure 3(b) to execute an additional time, resulting in an extra decrement of the reference count, creating an imbalance in the reference counting system, which can lead to a UAF vulnerability in the Net object. Following the exploitation process detailed in Appendix[2], attackers can transform this UAF into a double free vulnerability. As defined in Definition 2.3, a double free constitutes a strong primitive, which can be further leveraged to achieve an ultimate primitive for successful exploitation.

3.3 Advantages of CMF in Primitive Escalation

We demonstrate the advantages of CMF-based primitive escalation from two perspectives.

3.3.1 Comparison with pointer-based primitive escalation. First, CMF's pointer-independent nature enables it to bypass all kernel pointer-based mitigations. For instance, ARM64 architecture, widely used in mobile devices running Android with its Linux kernel foundation, has implemented Pointer Authentication Code support. This technology verifies pointer integrity during pointer access, effectively preventing pointer-based escalation techniques under full PAC protection. Second, pointer-based escalation demands stricter requirements on the content of primitives, due to pointer validity and memory alignment constraints. When exploiting nft_trans::put_net, attackers may only need to ensure the content is non-zero, as shown in the put_net example in Figure 3.

3.3.2 Bypassing Defense Mechanisms. The exploitability of the CMF stems from insufficient defense mechanisms against metadata field manipulation in the Linux kernel. While the kernel mainline implements various security measures [1, 11, 17, 18, 29–32] targeting metadata fields (Table 1), they focus on countering known exploitation techniques. Specifically, they protect against illegal modifications to critical structures such as list_head, scatterlist,

[2]https://github.com/Roarcannotprogramming/Weak-Primitive/blob/master/appendix/exploit_strategy_for_reference_count.md

Table 1: Defenses for metadata field in the Linux kernel

Defense Mechanisms	Exist in the Mainline Kernel	Enabled by Default in	
		Ubuntu	Debian
DEBUG_LIST	✓	✗	✓
DEBUG_SG	✓	✗	✗
DEBUG_CREDENTIALS	✓	✗	✗
DEBUG_NOTIFIERS	✓	✗	✗
DEBUG_VIRTUAL	✓	✗	✗
BUG_ON_DATA_CORRUPTION	✓	✗	✓
STATIC_USERMODEHELPER	✓	✗	✗

cred, and notifier_block. The STATIC_USERMODEHELPER mechanism provides additional protection for the global kernel variable modprobe_path. Notably, these security measures do not extend to the CMF field, leaving it vulnerable to exploitation even when all defenses are enabled.

4 Identifying CMFs in Kernel

4.1 Overview

MetaXploit operates in two distinct phases: (a) Static Analysis for CMF Candidates Identification and (b) Dynamic Validation of CMF. In the first phase, MetaXploit employs data flow analysis to identify CMFs capable of influencing program control flow. When critical memory operations exist within hijacked branches, attackers can potentially exploit weak primitives to corrupt these fields, triggering unauthorized memory operations and gaining enhanced memory control capabilities, ultimately leading to strong primitives. The second phase involves dynamic validation, where MetaXploit verifies whether metadata corruption can actually generate strong primitives during execution by simulating potential attack paths, thus eliminating false positives identified during static analysis.

However, both phases face significant challenges. First, **filtering metadata fields in the kernel that affect control flow but do not lead to strong primitives poses a substantial challenge**. Memory operations in hijacked branches do not necessarily yield strong primitives; we propose a systematic analysis approach to filter such cases (details in 4.2.2). Second, **generating effective test cases that trigger the execution of conditional statements influenced by CMFs presents another challenge**. The complex path from user space to kernel space makes it difficult to generate effective test cases, and traditional directed fuzzing approaches are time-consuming. To address this, we propose a novel solution utilizing large language models for test case generation (details in 4.3.1).

4.2 Static Analysis

In this phase, we aim to identify metadata fields that satisfy the three rules outlined in Section 2.2. As illustrated in Figure 4(a), our methodology begins by identifying metadata fields within allocatable kernel objects that satisfy the *Conditional Statement Dependency* rules. This means there exists a data flow propagation path from the metadata access statement to the conditional statement. Subsequently, as shown in Figure 4(b), we analyze memory

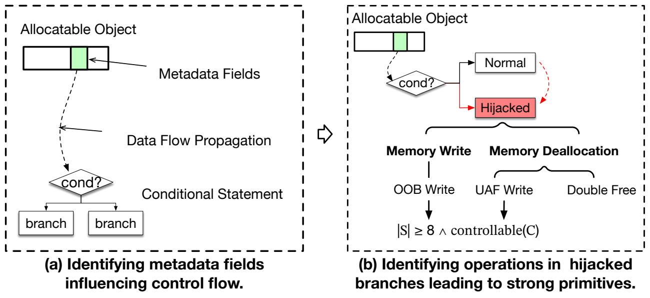


Figure 4: The workflow of static analysis. $|S|$ represents the number of bytes can be written, $controllable(C)$ means the written content is controllable.

operations that could potentially generate strong primitives. Specifically, we focus on two types of memory operations: memory writes and memory deallocation. Memory writes may lead to additional Out-of-Bounds writes, while memory deallocation operations could result in Use-After-Free writes or Double Free.

The verification of *User Accessibility* is deferred to the dynamic phase due to the complexity of metadata access permissions, which involves multiple implicit kernel-level checks that are challenging to detect through static analysis.

4.2.1 *Identifying metadata fields influencing control flow.*

We first identify allocatable objects in the kernel, then systematically analyze their associated metadata to determine which of these fields can influence control flow execution.

Algorithm 1 presents our object identification process. This algorithm takes a set of memory allocation functions as input and produces a set of allocatable object types as output. The algorithm starts by initializing an empty set SallocObj and then processes each allocation function in two ways: for functions that don't return "void *", it directly adds their return type to the result set (for example, when a function returns a specific structure pointer type like io_kiocb *); for functions returning "void *", it examines each call site to determine the actual type through context analysis. The type detection is handled by the RetTypeofAlloc procedure, which considers three specific contexts: direct assignment (where the type is determined from the left-hand side of an assignment), variable declaration (where the type comes from the declarator), and return statement context (where the type is inferred from the enclosing function's return type). This systematic approach ensures comprehensive identification of all possible allocatable object types, whether they're explicitly typed in function returns or implicitly determined through usage context, making it particularly effective for discovering all allocatable objects.

After identifying all objects, we locate the metadata fields within these objects and their access instances in the kernel. Beyond the metadata fields directly within objects, we also identified the fields within nested objects, mainly fields within embedded structs and unions. We treat the metadata fields of nested objects as part of the current object because the memory they occupy is a portion of the memory allocated for the current object. To analyze how these

Algorithm 1: Identify Allocatable Object

```
Input: $S_{allocFunction}$
Output: $S_{allocObj}$
▷ $S_{allocFunction}$: The set of memory allocation functions
▷ $S_{allocObj}$: The set of allocatable objects
1  $S_{allocObj} \leftarrow \emptyset$;
2  for $F_{alloc} \in S_{allocFunction}$ do
3      if ReturnTypeOf($F_{alloc}$) $\neq$ "void *" then
4          $S_{allocObj} \leftarrow S_{allocObj} \cup$ ReturnTypeOf($F_{alloc}$)
5      else
6          foreach $FC_{alloc}$ such that TargetOf($FC_{alloc}$) = $F_{alloc}$ do
7              $S_{allocObj} \leftarrow S_{allocObj} \cup$ RetTypeofAlloc($FC_{alloc}$)

8  Procedure RetTypeofAlloc($allocFunctionCall$)
       /* Case 1: Direct assignment context */
9      if $allocFunctionCall \in AssignmentExpression$ then
10         $retType \leftarrow TypeOf(AssignmentExpression.leftHand)$
       /* Case 2: Variable declaration context */
11     else if $allocFunctionCall \in DeclarationStatement$ then
12         $retType \leftarrow TypeOf(DeclarationStatement.declarator)$
       /* Case 3: Return statement context */
13     else if $allocFunctionCall \in ReturnStatement$ then
14         $retType \leftarrow$ ReturnTypeOf($EnclosingFunction$)
15     return $retType$
```

metadata influence control flow, we perform taint analysis where we define the sources as the values read from these metadata fields, and the sinks as the uses of tainted values in control statements(such as `if`, `switch`, `while`, `for`, and `do...while`).

4.2.2 *Identifying hijacked branches leading critical memory operations.*

Following the identification of metadata-affected conditions, we investigate whether the resulting hijacked branches can lead to critical memory operations through a systematic analysis approach. Our methodology begins with static analysis of memory operations within these hijacked branches, specifically focusing on operations that could potentially create strong primitives: memory write and memory deallocation. The presence of such operations serves as an initial indicator of potential strong primitives.

Memory write operations. To identify memory write operations with strong primitives, three essential conditions must be satisfied: first, the memory write must occur within a branch where the conditional expression is influenced by metadata fields, such that increased metadata field values cause hijacked branches leading to out-of-bounds writes; second, the write operation must exceed 8 bytes in length; and third, the written data must be user-controllable. Our static analysis methodology identifies writes satisfying these conditions by first locating all memory write statements and memory copy functions, verifying whether destination data (left-hand values in assignments and copied data) is influenced by loop variables; then analyzing destination data types to determine size requirements, directly verifying basic types like `long`, and examining structure fields for composite types; and finally confirming data controllability by identifying if source data originates from user-kernel interaction functions (including `copy_from_user` function family and system call parameters) through taint analysis. Additionally, if the source data is heap-stored and subject to out-of-bounds reads, attackers can achieve indirect control over the input data through carefully crafted heap layouts, making such source data user-controllable as well.

Memory deallocation When analyzing hijacked branches containing memory deallocation operations, Write-After-Free (WAF) and Double-Free conditions manifest when the freed objects encounter additional write or free operations across different kernel APIs. Our methodology implements a type-based alias analysis approach to identify these conditions by determining whether objects in subsequent operations are aliases of the freed objects. This approach considers all objects of the same type as potential aliases, trading precision for efficiency.

Once potential WAF or Double-Free conditions are identified, we proceed to determine whether they lead to strong primitives. For WAF conditions, we apply the same methodology used in memory write operation analysis. The identification of Double-Free conditions requires additional consideration, as not all free operations on a previously freed object lead to Double-Free conditions. Specifically, free operations occurring in either (a) object destruction functions (such as destroy or cleanup routines) or (b) error handling paths should be excluded from consideration. This exclusion is justified because both scenarios mark the natural end of an object's lifecycle—once an object is freed at these points, its kernel-space lifecycle concludes, preventing subsequent free operations [54].

Beyond these considerations, another two specific scenarios warrant careful attention in eliminating false positives:

- When both normal and hijacked branches contain deallocation operations targeting the same object, metadata modifications do not introduce additional deallocation operations. We employ local alias analysis to identify such cases by examining whether both branches free the same object. For example, in Figure 5(a), the value of `nft_trans::msg_type` can be either `NFT_MSG_DELRULE` or `NFT_MSG_DESTROYRULE`. In both branches, `nf_tables_rule_destroy` is called to free the same object. Since there is no different deallocation between both branches, changing `nft_trans::msg_type` does not lead to any additional memory release.
- In cases where the kernel implements proper memory management practices - such as using `list_del` functions before deallocation or nullifying pointers after deallocation - to remove global memory references, the risk of UAF or double-free vulnerabilities is mitigated. Consequently, these hijacked branches do not result in strong primitives. For example, in Figure 5(b), the value of `x25_forward::lci` controls whether the `x25_forward` object is freed. However, when the object is freed, the `list_del` function removes the global reference to this object. Without global references, this object cannot be freed a second time. We employ dominance analysis to verify that the execution of deallocation operations necessarily implies the execution of global reference removal operations, thereby eliminating such cases.

4.3 Dynamic Verification

To ensure the applicability of our static analysis results in real-world exploit scenarios, we employ dynamic analysis techniques for comprehensive verification. This phase serves two critical purposes: eliminating false positives from static analysis and validating the practical exploitability of identified vulnerabilities. As outlined in Section 4.2, static analysis exhibits two primary limitations that contribute to false positives: the inability to determine whether

```c
static void nft_commit_release(
↪   struct nft_trans *trans)
{
  switch (trans->msg_type) {
  ...
  case NFT_MSG_DELRULE:
  case NFT_MSG_DESTROYRULE:
    nf_tables_rule_destroy(
      &trans->ctx,
      nft_trans_rule(trans)
    );
    break;
  ...
  }
}
```

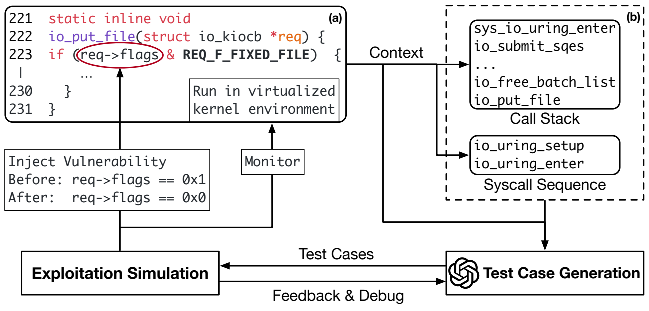
(a) An example of both branches contain free operations.

```c
void x25_clear_forward_by_lci(unsigned int
↪   lci)
{
  struct x25_forward *fwd, *tmp;

  write_lock_bh(&x25_forward_list_lock);
  list_for_each_entry_safe(fwd, tmp,
  ↪   &x25_forward_list, node) {
    if (fwd->lci == lci) {
      list_del(&fwd->node);
      kfree(fwd);
    }
  }
  write_unlock_bh(&x25_forward_list_lock);
}
```

(b) An example of simultaneous global reference removal and object deallocation.

Figure 5: Examples of memory operations in hijacked branch failing to obtain strong primitives.



```
221  static inline void
222  io_put_file(struct io_kiocb *req) {
223  if (req->flags & REQ_F_FIXED_FILE)  {
 |     ...
230    }
231  }
```

```
Inject Vulnerability
Before: req->flags == 0x1
After:  req->flags == 0x0
```

Figure 6: The workflow of Dynamic Verification.

non-privileged users can invoke specific kernel functions (User Accessibility), and the challenge of verifying whether Illegal Memory Operations can materialize under actual runtime conditions. To address these constraints, as illustrated in Figure 6, we implement a two-stage dynamic validation process: *Test Case Generation* and *Exploitation Simulation.*

4.3.1 Test Case Generation. In this phase, we aim to achieve comprehensive dynamic execution of conditional statements influenced by CMFs, with the requirement that these executions must be triggered from user space programs. However, reaching these conditional statements presents substantial technical challenges in terms of both coverage and execution efficiency. While the kernel's native test suite provides baseline coverage, it addresses only a limited subset of metadata fields. Although goal-directed fuzzing techniques, exemplified by SyzDirect [49], offer a systematic approach, they face several critical limitations. First, the computational complexity of analyzing the extensive number of conditional statements renders this approach prohibitively time-intensive for practical implementation. Second, they cannot guarantee reproducible test cases due to the stateful nature of OS kernels. Third, they generate test cases that are difficult for humans to understand and reason about, which hinders manual intervention in *Exploitation Simulation.*

To address these limitations, we explored the possibility of directly generating test cases to trigger the execution of condition statements. Leveraging the fact that Large Language Models (LLMs) are trained on extensive Linux kernel code and documentation, we proposed an LLM-based approach for test case generation. Our methodology involves extracting the target function (Figure 6(a)) containing condition statements and annotating them with *[target line]* comments to indicate specific execution targets. We then query the LLM using the prompt shown in Table 2. First, we instructed the LLM to generate C code that could reach our target methods. If the generated code failed to compile or encountered runtime errors, we provided the error messages back to the LLM for error correction and code regeneration. Note that while LLM-generated code successfully reaches the target methods, the condition statements may not be executed. Instead of requesting more test cases from the LLM, we address this challenge in the `Exploitation Simulation` phase through manual debugging.

To enhance the effectiveness of test case generation, we incorporate another two contexts, as shown in Figure 6(b): (1) the complete call stack from target functions to syscall entries, derived through static analysis, and (2) successful syscall sequences that triggered target function execution, obtained from Syzkaller [16]. This contextual information is vital because syscall execution often requires specific kernel states, which must be established through precise sequences of preliminary syscalls. For example, as illustrated in Figure 6(b), while `io_uring_enter` directly triggers our target function, it requires preparatory syscalls like `io_uring_setup` to establish the necessary kernel state.

4.3.2 Exploitation Simulation. During exploitation, weak primitives are expected to modify CMF and trigger illegal memory operations to obtain strong primitives. To validate this approach, as shown in Figure 6, we execute generated test cases within a virtualized kernel environment provided by QEMU [4], where we systematically modify CMFs while monitoring illegal memory operations.

Inject Vulnerability. Before accessing CMF, we modify its value to simulate exploitation primitives and observe whether execution branches change. Typically, we simulate initial vulnerability primitives by flipping each bit of the original value of CMF. However, during the propagation of CMF in the data flow, there may be checks on the range of its value. When such checks occur, we record these ranges and constraints. We then manually verify whether CMF can influence the control flow while satisfying these constraints.

Monitor. Once we confirm that modifying CMF can affect the control flow to reach unintended branches, we further evaluate whether this leads to unexpected memory operations. This helps us determine if strong primitives can be achieved. In this step, we mainly use KASAN [12] to monitor memory operations after the control flow changes. If a memory operation triggers an out-of-bound write or a use-after-free write, KASAN provides information about the location and memory context of the operation. This allows us to quickly confirm whether the memory operation results in a strong primitive. However, if the OOB and UAF issues reported by KASAN are not caused by the critical memory operations identified in static analysis, but rather by other unrelated kernel memory operations, we disable KASAN and manually debug using GDB

Table 2: Prompts used for test-case code generation

	Code Generation	Code Debug
System Prompt	As a Linux kernel security researcher, I ask questions with a certain level of expertise. Please answer my questions from a professional Linux kernel security perspective. Please strictly follow the format requirements in your response. Based on the target kernel function call stack and the system call sequence that leads to this kernel function, which I will provide next, follow my instructions step by step to write a user-space program. The user-space program you write should serve as a test case, aiming to trigger the kernel to invoke this kernel function. If code is involved, it should be written in C language as a single file.	As a Linux kernel security researcher, I ask questions with a certain level of expertise. Please answer my questions from a professional Linux kernel security perspective. Please strictly follow the format requirements in your response. In the code you generated, there are compilation errors or runtime errors. Please fix the errors based on the error messages. You may need to check: 1. Syntax errors. 2. Header file paths - if you encounter "header file not found" errors, even with dependencies installed, the header file paths provided by the dependencies might differ from those in the code. 3. System call parameters - check both the number of parameters and their meanings, especially for parameters passed by pointers.
User Prompt	Kernel Function: {target_function} Call Stack: {call_trace} System call sequence: {syscall_sequence} Response Format (Please strictly follow the format): <think process> Step-by-step explanation of your thought process and necessary explanations. </think process> <pkg install cmd> Compilation dependency installation command that can be executed directly in the shell, starting with apt install or pip install, based on the Debian system. If unnecessary, leave it blank, do not output anything here. </pkg install cmd> <code> Your C language code, do not add any markdown syntax prefixes or suffixes. </code> <code name> The file name of the C language code, which should follow the format test_case_*.c. </code name> <compile cmd> Compilation command that can be executed directly in the shell, with the working directory being the directory where the C language code is located. Do not add any markdown syntax prefixes or suffixes (e.g., gcc -lpthread test_case_netfilter.c -o test_case_netfilter). </compile cmd>	Error in C code: {code} C code filename: {code_name} Installed dependencies: {installed_pkg} Error message: {err_msg} Response format (please strictly follow the format): <think process> Step by step description of your error analysis process and necessary explanations </think process> <pkg install cmd> Compilation dependency installation commands that can be directly executed in shell, starting with apt install or pip install, based on debian system, without any markdown syntax prefix/suffix (e.g., apt install xxx). If not necessary, leave empty, do not output anything here </pkg install cmd> <code> Modified C code, without any markdown syntax prefix/suffix </code> <compile cmd> Modified compilation command, if no modification needed, keep original. Without any markdown syntax prefix/suffix. </compile cmd>

to verify whether the target critical memory operations can be executed.

5 Implementation

Here we elaborate on the implementation details of techniques used in Section 4.

Static analysis. Our static analysis is primarily based on CodeQL [2, 37]. Before conducting C/C++ program analysis, CodeQL requires compilation of the target program to gather fundamental program information, which is then stored in a database. So prior to analyzing the kernel code, we compile it using distribution-specific build configurations and generate the corresponding database. Throughout the static analysis process, we employ CodeQL's built-in TaintTracking::Global<> module to perform interprocedural data flow taint analysis, which is flow-sensitive, context-sensitive, and field-sensitive. To optimize analysis performance and precision, we define taint barriers that prevent taint propagation through specific program components. These barriers exclude analysis of non-essential code sections such as debugging facilities (e.g., kprobe.c) and architecture or virtualization-specific implementations (such as code in arch/riscv/ and arch/x86/hyperv/ directories).

Dynamic Verification. Our dynamic verification requires two key elements from the kernel source code: the system call sequence and call stack for the target function. The system call sequence is obtained through analysis of coverage data from 36 Syzkaller [16] instances maintained on the Syzbot [13] platform. For call stack identification, we first constructed reverse call graphs where edges point from callee to caller functions. By traversing these graphs from target functions, we systematically identified all possible call chains, effectively reconstructing the complete call stacks that could lead to target functions.

For test case generation, we implemented an LLM-driven approach utilizing LangChain [21] framework in conjunction with the OpenAI o1-mini model [36]. The system incorporates an iterative refinement mechanism: when generated test cases fail either during compilation or fail to reach the intended target location during execution, the model receives specific error feedback and automatically regenerates the test case. To maintain computational efficiency while ensuring adequate refinement opportunities, we implemented a maximum threshold of four iteration cycles for the self-correction process. In cases where errors persist after automated refinement attempts, the system escalates to human expert intervention for manual analysis and resolution.

6 Evaluation

In this section, we evaluate our tool MetaXploit, along with the CMFs to address the following research questions:

RQ1: How effective is MetaXploit in identifying CMFs?
RQ2: How can CMFs be exploited in real-world vulnerability attacks?
RQ3: How does CMF-based primitive escalation technique compare to existing primitive escalation techniques?
RQ4: How effective are the strategies adopted by MetaXploit (ablation study)?

Experimental Setup. To address RQ1, we evaluated the effectiveness of MetaXploit across two Linux kernel distributions: Ubuntu 22.04 (Linux v6.5.13) and Debian Bookworm (Linux v6.1.112). For RQ2, we conducted a comprehensive analysis of real-world Linux

Table 3: Summary of MetaXploit analysis stages and results for Ubuntu and Debian kernels.

Target	Stage	Allocatable Objects	CMFs	Conditional Statements	Time Cost
Ubuntu	Initial Collection	7,091	67,563	-	6s
	Static Analysis	633	833	1,937	170s
	Dyn. Verification	49	54	61	17h
Debian	Initial Collection	5,879	58,038	-	6s
	Static Analysis	546	736	1,694	164s
	Dyn. Verification	46	51	57	17h

kernel vulnerabilities involving weak primitives reported within the past five years. We assessed the potential for exploiting these vulnerabilities using CMF-based primitive escalation technique. In RQ3, we performed a comparative analysis of CMF-based primitive escalation technique against two established primitive escalation techniques: Elastic Object [6] and Thanos Object [26]. To address RQ4, we evaluated the efficacy of our LLM-based test case generation strategy by comparing it against two alternative approaches: kernel self-tests and Syzkaller-generated test cases. Additionally, we conducted detailed ablation studies to systematically analyze the individual strategies of our LLM-based methodology. All experiments were conducted on a server equipped with an AMD Ryzen 9 7950X processor and 128GB of RAM.

6.1 RQ1: Identifying CMFs

6.1.1 Overall Results. As shown in Table 3, the evaluation process consists of three stages: Initial Collection, Static Analysis, and Dynamic Verification. In the initial collection stage, we collect 7,091 and 5,879 allocatable objects, along with 67,563 and 58,038 metadata fields in Ubuntu and Debian, respectively. Through static analysis, we reduce the number of objects to 663 and 546, with associated metadata fields decreasing to 833 and 736, which affected 1,937 and 1,694 conditional statements. Finally, through dynamic verification, we confirmed the actual CMFs, involving 49 and 46 objects, 54 and 51 metadata fields, which affected 61 and 57 conditional statements in Ubuntu and Debian, respectively. The static analysis took approximately 3 minutes to complete, while the dynamic verification process required 17 hours for both Ubuntu and Debian targets.

Our further analysis revealed several root causes of false positives in static analysis:

- **User Accessibility Violations** As discussed in Section 4.2, static analysis does not determine whether users can access specific metadata fields, leading to false positives. This manifests in two ways: certain metadata field access functionalities are not enabled by default within the kernel (377 false positives); permission checks along access paths prevent the utilization of these objects (215 false positives);
- **Illegal Memory Operation Violations** False positives arise when the analysis identifies potentially critical memory operations along hijacked branches without confirming their actual impact. Specifically, write operations may not exceed the 8-byte threshold, and normal execution flows might not necessarily introduce free operations. Although we implemented heuristic rules to filter out some false positives, the inherent limitations of static analysis—particularly its inability to execute code dynamically—prevent complete elimination of these false positives.

6.1.2 Detailed Results. The detailed results are presented in Table 4. As shown in Column 5, through manual analysis, we classified the identified CMFs into three functional categories: flags, reference counters, and loop counters, based on their logical functionality.

9 CMFs functioning as flags, the kernel implements boolean checks on these fields or their individual bits to determine execution paths. Attackers can manipulate these flag fields through weak primitives to trigger unauthorized memory write or memory deallocation operations within the hijacked branch, potentially leading to OOB write, UAF write or double-free.

32 CMFs are reference counters that are used for kernel object lifecycle management. These counters implement a tracking mechanism where the count increments when a new instance references an object and decrements when an instance terminates its reference. Under normal operation, the object is deallocated when its reference count reaches zero. By artificially reducing the reference count below its legitimate value, attackers can create a scenario where the actual number of references exceeds the stored count. This misalignment enables multiple deallocation attempts on the same object, resulting in a double-free primitive.

13 CMFs represent loop counters, which regulate the number of iterations in kernel loops. These counters present a potential attack surface where attackers can manipulate the counter values to force excessive loop iterations. Such manipulation can trigger unauthorized memory operations within the loop, potentially resulting in illegal memory operation.

6.2 RQ2: Real-world Exploitation

In this section, we present our analysis of matching CMFs with real-world vulnerabilities and evaluate their practical exploitability.

Matching CMFs to Real-World Vulnerabilities. The exploitation of real-world vulnerabilities using CMFs requires careful analysis of vulnerability primitives and their capabilities. Here, we enhance a vulnerability primitive as a quadruple (type, offset, size, value), where we introduce *type* denoting the vulnerability classification. For instance, CVE-2021-22555 exhibits an out-of-bounds write vulnerability primitive, enabling the writing of two bytes `\x00` at offset 0 of the adjacent object, represented as (OOB write, 0, 2, 0). For a real-world vulnerability to be compatible with CMFs, it must satisfy two conditions: (1) the presence of CMFs within the range defined by (offset, offset + size), and (2) the ability to write values that can effectively influence control flow. Notably, the vulnerable object and victim object must be adjacent (for OOB) or allocated in the same space (for UAF). While we do not explicitly distinguish between objects in cache versus page allocation, this is because we can employ the Cross Page technique [14] to force the object containing CMF and the vulnerable object to satisfy this memory constraint.

Evaluation of Real-World Exploitability.To validate the effectiveness of CMF, we conducted a comprehensive evaluation of real-world exploitability. We first manually identified potential vulnerability candidates from CVE reports [35] and Syzbot crash contexts [13] spanning the past five years, specifically focusing

Table 4: CMFs identified in Linux Kernel. Column 1 provides sequential numbering, Columns 2 displays the identified CMFs and their corresponding container objects. Column 3 shows object sizes and field offsets, where a "+" symbol after a size value indicates a variable-length object with that value as its minimum size. Column 4 displays target values associated with control-flow branches that produce strong primitives. Here, v_{mod} represents the attacker-modified value, while v_{orig} denotes the original value. Column 5 indicates the role of CMF in kernel objects, where ♣ represents a flag, ♦ represents a reference counter, and ♠ represents a loop counter.

#	CMF	Size/Off	Target Value	Role
1	`nft_trans::put_net`	88+/ 36	v_{mod}!=0	♣
2	`rfcomm_dev::flags`	504/408	v_{mod}&1==1	♣
3	`shmid_kernel::shm_nattch`	256/136	v_{mod}==0	♣
4	`proc_dir_entry::mode`	176/168	(v_{mod}>>13)&5==5	♣
5	`rpc_task::tk_flags`	224/216	(v_{mod}>>15)&1==1	♣
6	`ceph_inode_xattr::should_free_val`	64/ 60	v_{mod}!=0	♣
7	`ceph_inode_xattr::should_free_name`	64/ 56	v_{mod}!=0	♣
8	`io_kiocb::flags`	240/ 68	v_{mod}&1==0	♣
9	`pipe_buffer::flags`	40+/ 24	(v_{mod}>>4)&1==1	♣
10	`nft_set::refs`	256/ 32		♦
11	`lec_arp_table::usage`	208/200		♦
12	`unix_address::refcnt`	8+/ 0		♦
13	`io_sq_data::refs`	144/ 0		♦
14	`ovl_aio_req::ref`	64/ 48		♦
15	`gss_cl_ctx::count`	96/ 0		♦
16	`fsnotify_group::refcnt`	296/ 8		♦
17	`fsnotify_mark::refcnt`	72/ 4		♦
18	`rpc_clnt::cl_count`	480/ 0	$1 \leq v_{\text{mod}} \land v_{\text{mod}} < v_{\text{orig}}$	♦
19	`rpc_task::tk_count`	224/ 0		♦
20	`sctp_chunk::refcnt`	256/ 16		♦
21	`sctp_transport::refcnt`	704/ 32		♦
22	`sctp_association::base.refcnt`	2,112/8		♦
23	`mountpoint::m_count`	40/ 32		♦
24	`configfs_fragment::frag_count`	56/ 0		♦
25	`posix_acl::a_refcount`	32+/ 0		♦
26	`tcf_proto::refcnt`	104/ 64		♦
27	`xfrm_state::refcnt`	744/ 72		♦
28	`xfrm_policy::refcnt`	832/ 48		♦
29	`pid::count`	96/ 0		♦
30	`nsproxy::count`	72/ 0		♦
31	`user_struct::__count`	144/ 0		♦
32	`user_namespace::ns.count`	624/260		♦
33	`ucounts::count`	160/ 28		♦
34	`cgroup_namespace::ns.count`	48/ 20		♦
35	`time_namespace::ns.count`	88/ 36	$1 \leq v_{\text{mod}} \land v_{\text{mod}} < v_{\text{orig}}$	♦
36	`key::usage`	224/ 0		♦
37	`cred::usage`	176/ 0		♦
38	`pid_namespace::ns.count`	144/132		♦
39	`Qdisc::refcnt`	384+/100		♦
40	`files_struct::count`	704/ 0		♦
41	`io_sq_data::refs`	144/ 0		♦
42	`key::datalen`	224/126		♠
43	`sched_gate_list::num_entries`	328/288		♠
44	`nft_pipapo_match::field_count`	48+/ 0		♠
45	`nft_pipapo_match::bsize_max`	48+/ 24		♠
46	`ip_set::dsize`	136/ 88		♠
47	`nft_rule::dlen`	24+/ 21		♠
48	`sfq_slot::qlen`	56/ 16	$v_{\text{mod}} > v_{\text{orig}}$	♠
49	`poll_list::len`	16+/ 8		♠
50	`tcp_fastopen_context::num`	56/ 32		♠
51	`mask_array::max`	40+/ 20		♠
52	`ip_sf_socklist::sl_count`	24/ 4		♠
53	`cfg80211_rnr_elems::cnt`	8+/ 0		♠
54	`cfg80211_mbssid_elems::cnt`	8+/ 0		♠

on cases likely to contain weak primitives. Subsequently, three domain experts meticulously analyzed these vulnerabilities to develop proof-of-concept (PoC) exploits—either by locating existing public exploits or constructing new ones when necessary. Cases where experts could not successfully extract or demonstrate primitives within a two-day timeframe were excluded from our analysis. This thorough analysis process required two man-months.

Ultimately, as illustrated in Table 5, our research evaluated the exploitability of 20 Linux kernel vulnerabilities disclosed in the past five years, comprising 9 CVE-listed vulnerabilities and 11 kernel crashes identified through Syzbot (Column 1). Among these cases, 6 vulnerabilities had existing public exploits, while the remaining 14 cases(marked with †) had no public exploitation demonstrations prior to our work. These vulnerabilities and crashes initially presented as weak primitives (Column 2), and the table details the corresponding CMF fields utilized in our analysis (Column 3).

Of the 20 analyzed vulnerabilities, we classified 7 as Out-of-Bounds vulnerabilities and 13 as Use-After-Free vulnerabilities. Our CMF-based technique successfully achieved primitive escalation in 18 cases, facilitating complete end-to-end exploitation. However, we encountered two failures in our approach: first, in CVE-2023-35788, the vulnerability's impact was confined to memory corruption within the vulnerable object itself, unable to extend to adjacent objects. Second, in vulnerability 29c47e9e5189, no CMFs were present within the accessible range (offset, offset + size) of the adjacent object's memory space.

6.3 RQ3: Comparison

We conducted a comparative analysis of existing exploitation techniques—Elastic Object [6] and Thanos Object [26]—against our CMF-based technique for primitive escalation in real-world vulnerabilities. As shown in Table 5, Elastic Object and Thanos Object successfully achieved primitive escalation in 8 and 13 out of the 20 vulnerabilities, respectively, while our method demonstrated superior effectiveness with 18 successful cases. While the combined coverage of traditional methods (15 cases) might appear comparable to our approach, it's crucial to note that Thanos Object becomes completely ineffective when PAC is enabled, reducing the total coverage of traditional methods to only 8 cases.

This enhanced performance can be attributed to fundamental differences in approach requirements. Elastic Object's effectiveness is constrained by its reliance on a length field, which must exceed expected values to enable broader memory control. Thanos Object faces limitations as it requires vulnerability primitives capable of completely corrupting the entire pointer field, whereas some vulnerabilities only permit modification of specific bits. In contrast, our CMF-based primitive escalation technique requires only partial value modifications that influence control flow, implementing more relaxed constraints on target values.

Case Study. We demonstrate the superiority of CMF in primitive escalation through CVE-2022-32250. This vulnerability provides an 8-byte write primitive, however, the written content is limited to an uncontrollable pointer.

Table 5: Leveraging CMFs to exploit 20 real-world Linux kernel vulnerabilities. The first column shows the CVE number, where "†" indicates vulnerabilities without public exploits. The second column describes the initial primitive capabilities, with parentheses containing the type, write offset, write length, and writable bytes respectively. The "*" indicates that the written offset can be controlled within a certain range. The third column lists the CMFs used for escalating primitives. The last three columns indicate whether CMF, Elastic Object, and Thanos Object can escalate primitives, where ✓ represents "yes" and ✗ represents "no".

CVE ID or Syzbot ID	Initial Primitive	CMF	Exploitable Using		
			CMF	Elastic	Thanos
CVE-2024-36025†	(OOB write, 0, 8, uncontrollable int)	`io_sq_data::refs`	✓	✓	✗
CVE-2024-38578†	(OOB write, *, 1, 0)	`io_kiocb::flags`	✓	✗	✓
CVE-2023-35788†	(OOB write, 454, 1, arbitrary)	-	✗	✗	✗
CVE-2022-0995	(OOB write, *, 1bit, true)	`pipe_buffer::flags`	✓	✓	✓
CVE-2022-25636	(OOB write, 0, 4, 4/5)	`cred::usage`	✓	✓	✓
CVE-2022-32250	(UAF write, 32, 8, uncontrollable ptr)	`nft_trans::put_net`	✓	✗	✗
CVE-2021-4200	(OOB write, *, 3, arbitrary)	`unix_address::refcnt`	✓	✗	✓
CVE-2021-22555	(OOB write, 0, 2, 0)	`cred::usage`	✓	✗	✓
CVE-2021-26708	(UAF write, 40, 4, arbitrary)	`poll_list::len`	✓	✓	✓
4c0d0c4cde78 [41]†	(UAF write, 128, 4, increased by 1)	`cfg80211_rnr_elems::cnt`	✓	✓	✓
3f0a39be7a20 [43]†	(UAF write, 54, 2, 5)	`nft_rule::dlen`	✓	✓	✗
29c47e9e5189 [38]†	(UAF write, 1776, 4, decreased by 1)	-	✗	✗	✗
cf54c1da6574 [40]†	(UAF write, 16, 4, decreased by 1)	`unix_address::refcnt`	✓	✓	✓
b53a9c0d1ea4 [39]†	(UAF write, 776, 8, uncontrollable data)	`poll_list::len`	✓	✓	✓
c041b4ce3a6d [44]†	(UAF write, 8, 1bit, 0)	`fsnotify_group::refcnt`	✓	✗	✓
07bb74aeafc8 [42]†	(UAF write, 1216, 4, decreased by 1)	`unix_address::refcnt`	✓	✗	✓
e1dc29a4daf3 [45]†	(UAF write, 200, 4, decreased by 1)	`lec_arp_table::usage`	✓	✗	✓
0fd4135ebd71 [48]†	(UAF write, 228, 4, uncontrollable data)	`nft_trans::put_net`	✓	✗	✗
6b0df248918b [46]†	(UAF write, 200, 4, decreased by 1)	`pipe_buffer::flags`	✓	✗	✓
55d58a05f0b5 [47]†	(UAF write, 1440, 8, decreased by 1)	`pid::count`	✓	✗	✗

When attempting exploitation through Elastic Object, an attacker needs to overwrite a length field within an Elastic Object. However, the uncontrollable pointer, with its high-order bits set to 0xFFFF, would result in a length value exceeding the length of the entire kernel space if used to overwrite the length field. Such a large value would trigger a system crash when the kernel attempts to access memory using this length.

For exploitation via Thanos Object, success requires manipulating an object's pointer to reference an adjacent object, creating dual pointer references to the same object. This requires the uncontrollable pointer value to exactly match the adjacent object's address in the Thanos Object, which has a very low probability of occurrence.

In contrast, the CMF-based approach only requires modifying the `nft_trans::put_net` field, where the associated conditional statement merely checks whether a single byte of this field is non-zero for normal execution. To achieve control flow hijacking, we only need the pointer value to overwrite this single byte, with a probability of 255/256 for a non-zero value. This significantly higher probability makes exploitation substantially more feasible.

6.4 RQ4:Ablation Study

MetaXploit uses LLMs to generate test cases for dynamic verification. Previously, test cases were primarily created using self-test code within the kernel or generated via fuzzing tools like Syzkaller, which attempt to produce reproducible code, though not always successfully. We evaluated the efficiency of these methods as a comparison baseline for our approach. Additionally, we conducted ablation experiments to assess the impact of system call sequence information and call trace on the efficiency of our method.

Table 6 presents our evaluation results. In the first row, we show the coverage achieved by all methods for the required test cases. In the second row, using self-tests as a baseline, we evaluate the performance of all methods on test cases that are harder to cover. The third row lists the average tokens consumed by LLM-based methods per test case, regardless of success or failure.

Syzkaller has limitations in generating C-language reproducible code for all covered paths, achieving only 24.56% coverage compared to the baseline self-test's 25.53%. Even with only the target location's code provided, the LLM achieved a 29.98% coverage of the required paths, demonstrating the potential of LLM-based approaches. Adding system call sequences increased coverage to 31.33%, while incorporating call trace information led to a substantial improvement, reaching 41.97%. When both call trace and system call sequences were included as LLM input prompts, we achieved the highest coverage of 43.32%.

It is important to note that while the improvement from system call sequences is small, they help the LLM generate test cases not covered by self-tests. One reason for the small improvement is that not all required paths are covered by Syzbot. For uncovered paths, there is no system call sequence information to include in prompts. On the other hand, system call sequences guide the LLM in generating test cases that involve combinations of system calls.

Table 6: Evaluation of test case generation methods for dynamic verification

	Self-Test	Syzkaller (C Reproducible code)	LLM (Raw)	LLM + Call Trace	LLM + System Call Sequence	LLM + All Context
Coverage (%)	25.53%	24.56%	29.98%	41.97%	31.33%	43.32%
Hard-to-Cover Cases	-	24	53	90	67	96
Avg. Token Cost per Test Case	-	-	3,630.3	4,085.2	3,404.5	3,740.0

All LLM approaches maintained reasonable token efficiency, with the full context requiring only a 3% increase in tokens (3,740.0 vs. 3,630.3) while achieving a 44% improvement in coverage.

7 Discussion

In this section, we discuss the implications of our research from multiple perspectives and address several key considerations

Automated Kernel Exploitation. Although this study does not focus directly on automated exploitation techniques, the identified metadata fields and our methods for recognizing these fields can be applied to the primitive escalation step of automated exploitation. Our work can also enhance existing attack path search techniques, such as AlphaExp [52], an automated system for constructing attack paths based on expert systems and knowledge graphs. Currently, AlphaExp does not consider metadata fields when searching for potential exploitation targets. By integrating our findings, AlphaExp could improve its ability to identify exploitation targets and generate more comprehensive attack paths.

False Positives and False Negatives. In the implementation of MetaXploit, we eliminated false positives caused by static analysis through a dynamic validation step. However, dynamic analysis may introduce some false negatives. Due to the lack of ground truth, it is difficult to accurately measure the false negative rate. To address this, we invited three human experts to inspect the results of our static analysis and identify potential false negatives. However, the experts did not find any such cases.

Other Metadata Fields for Primitive Escalation. In addition to the CMF fields identified in this work, other metadata fields may also have potential for primitive escalation. One example is the lock fields in kernel objects. If an attacker could modify such fields to change a lock from a "locked" state to an "unlocked" state, new race conditions could be introduced, potentially enabling more advanced exploitation primitives. However, even though prior work [15, 22] has explored controlling the timing of race condition windows during vulnerability triggering, achieving precise timing for stable exploitation remains a challenge. We consider this a direction for future work.

CMF-based Exploitation and Data-oriented Programming. Data-oriented Programming (DOP) [8] achieves program behavior manipulation by modifying non-control data without breaking control flow integrity. For example, in Linux kernel exploitation, modifying `modprobe_path` to execute malicious code with root privileges [34] represents a common DOP technique. CMF-based exploitation represents a specialized form of DOP that specifically addresses challenges associated with weak primitives in kernel exploitation. Unlike other DOP techniques such as data-only attacks, CMF-based exploitation does not emphasize using data attacks to directly achieve privilege escalation, but rather focuses on escalating the capabilities of exploitation primitives. Consequently, CMF-based exploitation can achieve more sophisticated exploitation outcomes by combining with other exploitation techniques after obtaining sufficiently powerful primitives, rather than merely elevating to root privileges. This approach offers unique advantages in exploitation scenarios targeting container escapes.

Defenses Against CMF-based Exploitation. As discussed in Section 3.3.2, existing kernel protections for metadata are insufficient against these exploits, while other defenses that disrupt heap layouts can be bypassed by advanced techniques like cross-page attacks. A potential defense approach could leverage shadow objects, a concept demonstrated in prior research [59] to protect critical kernel fields. This method would maintain a separate copy of sensitive fields in isolated memory areas, requiring attackers to compromise both the original object and its shadow copy. By focusing protection specifically on CMF and other identified sensitive fields rather than all kernel objects, the implementation could balance security improvements with performance considerations. However, this approach would inevitably introduce additional overhead in both performance and memory usage, which would need careful evaluation in real-world scenarios.

Applications to Other Systems. Metadata fields exist in all operating systems, and CMFs are also present in systems like XNU, FreeBSD, and Windows. Adapting our methods to these systems would require additional effort. Static analysis on closed-source systems, such as Windows, remains particularly challenging. Identifying CMFs in such systems is considered future work.

Responsible Disclosure. Although our research did not reveal any new vulnerabilities, we responsibly notified and shared our findings and artifacts with the Linux kernel maintainers, and proceeded with public disclosure with their permission.

8 Related Work

In this section, we review existing research related to Linux kernel exploitation across different stages.

Exploring the Exploitability of Vulnerabilities. When an attacker discovers a potential vulnerability, the first step is to determine whether it can be exploited and what primitives it can provide. FUZE [56] uses context-aware fuzzing to explore new usage points for use-after-free vulnerabilities and applies symbolic execution to identify possible exploitation primitives. KOOBE [5] investigates methods to derive various exploitation primitives from heap overflow write capabilities. GREBE [24] leverages fuzzing techniques to examine how vulnerabilities can cause memory corruption in different contexts. Similarly, SyzScope [64] uses symbolic execution to analyze the memory corruption capabilities of kernel memory.

Expanding Exploitation Capabilities. (Most Relevant to This Research) Expanding exploitation capabilities is a critical step in kernel exploitation. This involves using the primitives provided by the vulnerability and combining them with appropriate kernel objects to gain more memory control over the kernel. SLAKE [7] identifies kernel objects containing function pointers that can be used for control flow hijacking. ELOISE [6] focuses on elastic kernel objects that enable information leakage when combined with overflow capabilities. K-LEAK [23] proposes a general method to search for kernel memory leaks using data flow analysis. SLUBStick [33] leverages a combination of side-channel attacks across cache boundaries and page table overwrites to transform kernel memory vulnerabilities into arbitrary read-write primitives. Beyond Control [62] identified exploitable objects within filesystem subsystems that can be manipulated to execute data-only attacks. SCAVY [3] identifies fields that can expand exploitation capabilities by affecting privileged system resources. These works represent explorations of kernel data-oriented exploitation techniques.

Recent studies have also explored kernel exploitation under constrained primitives. The DirtyPipe [19] vulnerability demonstrated how flag corruption can lead to severe consequences, while Liu et al. [26] proposed a method to create double-free conditions by modifying the lower bits of pointers. These studies provide indirect insights that support our work.

Additionally, other research has focused on techniques for kernel heap layout. KHeaps [60] addresses reliability issues in SLAB heap layouts and proposes a stabilization technique. Guo et al. [14] further explore effective page layout strategies, which we also use when evaluating CMF's effectiveness on real-world vulnerabilities.

Privilege Escalation Techniques. Privilege escalation is the final goal after obtaining strong exploitation primitives. Traditional methods involve hijacking control flow using Return-Oriented Programming (ROP) techniques. RetSpill [61] discovered that certain kernel memory regions can be controlled by users to arrange gadgets for ROP. DirtyCred [25] introduced a data-only attack method by replacing regular privilege credentials with root credentials to achieve privilege escalation. USMA [27] and DirtyPagetable [53] enable arbitrary kernel code modification by remapping kernel memory to user space to achieve data-only attack.

Automated Kernel Exploitation Frameworks. AlphaEXP [52] uses knowledge graphs to infer kernel exploitation paths and identify sensitive objects along these paths, making progress in exploring automated kernel exploitation. Since CMFs can escalate primitives, they could expose additional exploitation paths for such techniques.

9 Conclusion

This paper challenges conventional wisdom in kernel security by demonstrating how Control Metadata Fields (CMFs) can be leveraged to exploit previously discarded vulnerabilities. By targeting CMFs that influences control flow rather than pointers, our approach bypasses modern security measures while benefiting from more relaxed exploitation constraints. Our tool, MetaXploit, through static analysis and dynamic verification, identified 54 exploitable CMFs in Linux kernel. The practical effectiveness of our approach is demonstrated by successfully escalating 18 out of 20 real-world vulnerabilities, including 8 previously unexploitable cases.

Acknowledgments

We thank all reviewers for their valuable feedback and extend our gratitude to the shepherd for their guidance throughout the review process. This work was supported by the National Natural Science Foundation of China (62202452).

References

[1] Popov Alexander. 2018. Linux Kernel Defence Map. https://a13xp0p0v.github.io/2018/04/28/Linux-Kernel-Defence-Map.html Accessed: 2025-01-08.
[2] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:25. doi:10.4230/LIPIcs.ECOOP.2016.2 https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2016.2.
[3] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitras. 2024. SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 7141–7158. https://www.usenix.org/conference/usenixsecurity24/presentation/avllazagaj
[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.
[5] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1093–1110. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng
[6] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. 2020. A Systematic Study of Elastic Objects in Kernel Exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1165–1184. doi:10.1145/3372297.3423353 https://doi.org/10.1145/3372297.3423353.
[7] Yueqi Chen and Xinyu Xing. 2019. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1707–1722. doi:10.1145/3319535.3363212
[8] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. 2021. Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches. 24, 4, Article 26 (Sept. 2021), 36 pages. doi:10.1145/3462699
[9] Jonathan Corbet. 2017. ARM pointer authentication. https://lwn.net/Articles/718888/ Accessed: 2025-01-08.
[10] Boneh Dan. 2018. Control Hijacking. https://crypto.stanford.edu/cs155old/cs155-spring18/lectures/02-ctrl-hijacking.pdf Accessed: 2025-01-08.
[11] Jones Dave. 2006. Re: memory corruptor in .18rc1-git. https://lore.kernel.org/lkml/20060714043112.GA20478@redhat.com/ Accessed: 2025-01-08.
[12] Linux Kernel Documentation. 2025. The Kernel Address Sanitizer (KASAN). Linux Kernel Documentation. https://docs.kernel.org/dev-tools/kasan.html Accessed: 2025-01-08.
[13] Google. 2024. syzbot - Kernel Fuzzing Dashboard. Web. https://syzkaller.appspot.com Accessed: 2025-01-08.
[14] Ziyi Guo, Dang K Le, Zhenpeng Lin, Kyle Zeng, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, Adam Doupé, and Xinyu Xing. 2024. Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1189–1206. https://www.usenix.org/conference/usenixsecurity24/presentation/guo-ziyi
[15] Tianshuo Han, Xiaorui Gong, and Jian Liu. 2024. CARDSHARK: Understanding and Stablizing Linux Kernel Concurrency Bugs Against the Odds. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 6203–6218. https://www.usenix.org/conference/usenixsecurity24/presentation/han-tianshuo
[16] Google Inc. 2024. Syzkaller - kernel fuzzer. https://github.com/google/syzkaller Accessed: 2025-01-08.
[17] Axboe Jens. 2007. Add CONFIG_DEBUG_SG sg validation. https://lore.kernel.org/lkml/1193076664-13652-11-git-send-email-jens.axboe@oracle.com/ Accessed: 2025-01-08.

[18] Cook Kees. 2016. bug: Provide toggle for BUG on data corruption. https://lore.kernel.org/all/1471470132-29499-5-git-send-email-keescook@chromium.org/ Accessed: 2025-01-08.

[19] Max Kellermann. 2022. The Dirty Pipe Vulnerability. https://dirtypipe.cm4all.com Accessed: 2025-01-08.

[20] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*. 957–972.

[21] LangChain Contributors. 2024. LangChain: Building context-aware reasoning applications. https://github.com/langchain-ai/langchain Accessed: 2025-01-08.

[22] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. 2021. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2363–2380. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yoochan

[23] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. 2024. K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel. In *31st Annual Network and Distributed System Security Symposium, NDSS*.

[24] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. 2022. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2078–2095. doi:10.1109/SP46214.2022.9833683

[25] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1963–1976. doi:10.1145/3548606.3560585

[26] Danjun Liu, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, Zhenhao Luo, Tai Yue, and Baosheng Wang. 2023. From Release to Rebirth: Exploiting Thanos Objects in Linux Kernel. *IEEE Transactions on Information Forensics and Security* 18 (2023), 533–548. doi:10.1109/TIFS.2022.3226906

[27] Yong Liu. 2022. USMA: Share Kernel Code. https://i.blackhat.com/Asia-22/Thursday-Materials/AS-22-YongLiu-USMA-Share-Kernel-Code.pdf Slides presented at Black Hat Asia 2022.

[28] LKDDB. 2018. CONFIG_REFCOUNT_FULL: Perform full reference count validation at the expense of speed. https://cateee.net/lkddb/web-lkddb/REFCOUNT_FULL.html Accessed: 2025-01-08.

[29] LKDDB. 2024. CONFIG_DEBUG_CREDENTIALS: Debug credential management. https://cateee.net/lkddb/web-lkddb/DEBUG_CREDENTIALS.html Accessed: 2025-01-08.

[30] LKDDB. 2024. CONFIG_DEBUG_NOTIFIERS: Debug notifier call chains. https://cateee.net/lkddb/web-lkddb/DEBUG_NOTIFIERS.html Accessed: 2025-01-08.

[31] LKDDB. 2024. CONFIG_DEBUG_VIRTUAL: Debug VM translations. https://cateee.net/lkddb/web-lkddb/DEBUG_VIRTUAL.html Accessed: 2025-01-08.

[32] LKDDB. 2024. CONFIG_STATIC_USERMODEHELPER: Force all usermode helper calls through a single binary. https://cateee.net/lkddb/web-lkddb/STATIC_USERMODEHELPER.html Accessed: 2025-01-08.

[33] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. 2024. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4051–4068. https://www.usenix.org/conference/usenixsecurity24/presentation/maar-slubstick

[34] Midas. 2021. Linux Kernel Exploitation Technique: Overwriting modprobe_path. https://lkmidas.github.io/posts/20210223-linux-kernel-pwn-modprobe/ Accessed: 2025-04-01.

[35] MITRE.org. [n. d.]. https://cve.mitre.org/

[36] OpenAI. 2024. OpenAI o1 System Card. https://cdn.openai.com/o1-system-card.pdf Accessed: 2025-01-08.

[37] Semmle. 2021. CodeQL. https://codeql.github.com/ Accessed: 2025-01-08.

[38] syzbot. 2023. KASAN: slab-use-after-free Write in gfs2_qd_dealloc. https://syzkaller.appspot.com/bug?extid=29c47e9e51895928698c Accessed: 2025-01-08.

[39] syzbot. 2023. KASAN: slab-use-after-free Write in mini_qdisc_pair_swap. https://syzkaller.appspot.com/bug?extid=b53a9c0d1ea4ad62da8b Accessed: 2025-01-08.

[40] syzbot. 2023. KASAN: slab-use-after-free Write in sco_chan_del. https://syzkaller.appspot.com/bug?extid=cf54c1da6574b6c1b049 Accessed: 2025-01-08.

[41] syzbot. 2023. KASAN: slab-use-after-free Write in sco_sock_timeout. https://syzkaller.appspot.com/bug?extid=4c0d0c4cde787116d465 Accessed: 2025-01-08.

[42] syzbot. 2023. KASAN: use-after-free Write in j1939_sock_pending_del. https://syzkaller.appspot.com/bug?extid=07bb74aeafc88ba7d5b4 Accessed: 2025-01-08.

[43] syzbot. 2024. KASAN: slab-use-after-free Write in __hci_acl_create_connection_sync. https://syzkaller.appspot.com/bug?extid=3f0a39be7a2035700868 Accessed: 2025-01-08.

[44] syzbot. 2024. KASAN: slab-use-after-free Write in l2tp_session_delete. https://syzkaller.appspot.com/bug?extid=c041b4ce3a6dfd1e63e2 Accessed: 2025-01-08.

[45] syzbot. 2025. KASAN: slab-use-after-free Write in io_submit_one. https://syzkaller.appspot.com/bug?extid=e1dc29a4daf3f8051130 Accessed: 2025-03-31.

[46] syzbot. 2025. KASAN: slab-use-after-free Write in recv_work. https://syzkaller.appspot.com/bug?extid=6b0df248918b92c33e6a Accessed: 2025-03-31.

[47] syzbot. 2025. KASAN: slab-use-after-free Write in sco_conn_put. https://syzkaller.appspot.com/bug?extid=55d58a05f0b5fd2ea0c7 Accessed: 2025-03-31.

[48] syzbot. 2025. KASAN: slab-use-after-free Write in sk_skb_reason_drop. https://syzkaller.appspot.com/bug?extid=0fd4135ebd713ba973dc Accessed: 2025-03-31.

[49] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1630–1644. doi:10.1145/3576915.3623146

[50] Linus Torvalds. 2024. The Linux Kernel. https://www.kernel.org Accessed: 2025-01-08.

[51] Nikolenko Vitaly. 2016. CVE-2016-6187: Exploiting Linux kernel heap off-by-one. https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit Accessed: 2025-04-01.

[52] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. 2023. AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4229–4246. https://www.usenix.org/conference/usenixsecurity23/presentation/wang-ruipeng

[53] Nicolas Wu. 2023. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html Accessed: 2025-01-08.

[54] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2041–2058. https://www.usenix.org/conference/usenixsecurity21/presentation/wu-qiushi

[55] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1187–1204. https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei

[56] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 781–797. https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei

[57] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. 2023. Mitigating Security Risks in Linux with KLAUS: A Method for Evaluating Patch Correctness. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4247–4264. https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yuhang

[58] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 89–106. https://www.usenix.org/conference/usenixsecurity22/presentation/yoo

[59] Zheng Yu, Ganxiang Yang, and Xinyu Xing. 2024. ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 7177–7193. https://www.usenix.org/conference/usenixsecurity24/presentation/yu-zheng

[60] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2022. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 71–88. https://www.usenix.org/conference/usenixsecurity22/presentation/zeng

[61] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2023. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 3093–3107. doi:10.1145/3576915.3623220

[62] Jinmeng Zhou, Jiayi Hu, Ziyue Pan, Jiaxun Zhu, Wenbo Shen, Guoren Li, and Zhiyun Qian. 2024. Beyond Control: Exploring Novel File System Objects for Data-Only Attacks on Linux Systems. arXiv:2401.17618 [cs.CR] https://arxiv.org/abs/2401.17618

[63] Xiaochen Zou, Yu Hao, Zheng Zhang, Juefei Pu, Weiteng Chen, and Zhiyun Qian. 2024. SyzBridge: Bridging the Gap in Exploitability Assessment of Linux Kernel Bugs in the Linux Ecosystem. In *31st Annual Network and Distributed System Security Symposium, NDSS*.

[64] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. 2022. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3201–3217. https://www.usenix.org/conference/usenixsecurity22/presentation/zou