Computing and Informatics, Vol. 45, 2026, 698–725, doi: 10.31577/cai_2026_3_698

SYZPRIME: ENHANCING LINUX KERNEL SECURITY FOR EDGE INTELLIGENCE APPLICATIONS VIA AUTOMATIC DISCOVERY OF HEAP LAYOUT MANIPULATION PRIMITIVES

Xingwei Li, Yunchao Wang*, Jianzhou Zhao

Information Engineering University, Zhengzhou, 450001, China
e-mail: w_yunchao@163.com

Yunchao Guan

Tsinghua University, Beijing, 650001, China

Danjun Liu

National University of Defense Technology, Changsha, 336017, China

Zhe Yang

Beijing University of Aeronautics and Astronautics, Beijing, 650001, China

Wenbin Zhang, Zehui Wu, Rongkuan Ma, Xixing Li, Qiang Wei

Information Engineering University, Zhengzhou, 450001, China

Abstract. Exploitation of the Linux kernel continues to pose significant security risks from sensitive information leakage to severe privilege escalation. While developers prioritize reducing *primitives* to limit exploitability, kernel heap exploitation remains a persistent threat that significantly threatens kernel integrity. Discovering such primitives is commonly considered an art; existing approaches rely on unsound static analysis and ad-hoc expertise, depending on expertise and limiting

* Corresponding author

practicality in the real-world Linux kernel. In this paper, we present SyzPrime, a systematic approach to automatically discover heap layout manipulation (HLM) primitives in recent kernel heap vulnerabilities, with a design that can scale to edge environments for autonomous, real-time security. The key innovation of SyzPrime lies in combining dynamic analysis via fuzzing with static instrumentation, enabling a direct correlation between primitives and fuzzing seeds. We utilize object-driven instrumentation with a restricted mutation-based fuzzer targeting specific object or cache interactions, such as allocation, deallocation and usage. By enhancing `ftrace` for precise tracking, we lay a foundation that enables the identification and generation of exploitation paths without side effects, ensuring that the discovered primitives are both reliable and practical. To demonstrate effectiveness, we evaluated SyzPrime on 40 real-world CVEs and 240 objects in the stable Linux kernel tree. Our results show that SyzPrime discovers 1.6x more primitives and identifies 4.1x more sensitive objects compared to state-of-the-art approaches.

Keywords: Edge intelligence, Linux kernel, security, exploitation primitives

1 INTRODUCTION

The security of the Linux kernel is of paramount importance in today's computing landscape. As the core component of the Linux operating system, the kernel has privileged access to hardware resources and provides critical services to user-space applications. Any vulnerabilities or weaknesses in the kernel can have severe consequences, potentially enabling attackers to gain unauthorized access, escalate privileges, or compromise the system.

Exploitation primitives, which are techniques or building blocks used in the construction of exploits, play a crucial role in understanding and mitigating kernel-level security threats. By studying these primitives and the mechanisms employed by attackers, researchers and developers can identify potential attack vectors, design effective defence strategies, and harden the Linux kernel against evolving security challenges. Discovering and identifying such exploitation primitives is not an easy task. It requires not only a deep understanding of the underlying implementation of operating systems but also an extensive knowledge of specific hardware architectures.

Current approaches to discovering Linux kernel primitives predominantly rely on identifying and exploiting vulnerable objects within the kernel space [1, 2, 3]. Notable research tools such as AlphaEXP [4], ELOISE [5], and TAODE [6] have established methodologies that focus on locating key objects as an indirect means of primitive discovery. Scatter [7] leverages control over heap objects to achieve precise management of kernel memory. Similarly, GOLLUM [8] and SHRIKE [9] also focus on memory objects, implementing accurate memory layouts across various language interpreters. While these object-centric approaches have yielded significant results in kernel exploitation research, they present several fundamental limitations that

warrant careful consideration. The first major limitation stems from the inherent dependency on specific objects' existence within the kernel. This dependency creates a significant constraint as the target object may not exist in certain kernel versions or configurations. Consequently, the primitive discovery process becomes highly contingent on the presence of specific kernel objects, limiting the methodology's applicability across different kernel versions and configurations. This limitation becomes particularly problematic when attempting to develop exploitation techniques that require consistent and reliable primitive discovery across diverse kernel environments.

A second critical limitation stems from the dependence on static analysis techniques for identifying primitives associated with these objects. While static analysis is widely used, its inherent limitations inevitably result in the generation of false positives and false negatives in its findings. False positives occur when the analysis mistakenly identifies potential primitives that are not actually exploitable, often due to the inability to model runtime conditions and object states accurately. Conversely, false negatives arise when viable primitives go undetected, particularly in cases involving complex execution paths or indirect object relationships. These inaccuracies in static analysis significantly impact the effectiveness and reliability of primitive discovery. The third significant limitation involves the management of side effects during the primitive discovery process. When attempting to allocate or manipulate target objects, unintended side effects frequently occur, including unexpected object allocations and deallocations. These side effects pose a substantial challenge as they are difficult to filter out or control effectively. The presence of such uncontrolled side effects can significantly impact the stability and reliability of any exploitation techniques developed using these primitives. Furthermore, these side effects can create complex interactions within the kernel space that may compromise the success rate of exploitation attempts. The current object-centric approach to primitive discovery often results in techniques that are neither sufficiently robust nor reliably portable across different kernel versions. This lack of reliability and portability presents a significant challenge for researchers and security practitioners attempting to develop consistent and dependable exploitation methodologies.

In this paper, we propose SyzPrime, a novel approach to discover exploitation primitives in any Linux kernel automatically, Heap Layout Manipulation (HLM) primitives in particular. Differing from others, SyzPrime employs a fuzzing-based method with a tether-based monitor for recording SLUB behavior in this process. This comprehensive monitoring approach enables precise tracking of kernel memory operations without introducing significant performance overhead or system instability. The trace-based implementation ensures minimal interference with normal kernel operations while maintaining high-fidelity data collection. It works in a primitive-free way with only kernel source code for any useable primitives. This design choice eliminates the dependency on pre-existing exploit patterns or known vulnerability types, enabling SyzPrime to discover novel exploitation techniques that traditional methods might overlook. To tackle the issue of exploitation primitives

being highly dependent on versioning, we utilize a fuzzing method across various kernel configurations to automatically and efficiently discover these primitives. This process minimizes human intervention, allowing for relatively quick and precise identification without the need for manual execution path analysis. Our fuzzing engine incorporates advanced coverage guidance and state-aware mutation strategies, ensuring thorough exploration of kernel code paths while maintaining efficiency in primitive discovery. To mitigate false positives, we employ a modified `ftrace` module to monitor kernel heap allocation and deallocation, systematically recording the allocation counts. The modified `ftrace` module incorporates advanced filtering mechanisms and precise timing measurements to identify genuine exploitation primitives while effectively eliminating spurious results accurately. This enhancement significantly alleviates the burden of manual verification and increases the reliability of the discovered primitives. On this basis, our approach enables systematic primitive discovery across different kernel versions and configurations. The methodology incorporates automated validation mechanisms and reproducibility checks to ensure the discovered primitives are practical for real-world exploitation scenarios.

We implemented a prototype of SyzPrime on the Linux kernel alongside a `ftrace`-based application for real-world Linux kernels. This implementation features comprehensive instrumentation capabilities and robust error-handling mechanisms to ensure reliable operation across diverse kernel environments and workloads.

The evaluation results show that SyzPrime can successfully discover 160 % of primitives that others found manually before. This high success rate demonstrates the effectiveness of our automated approach compared to traditional manual analysis methods. Through extensive testing across multiple kernel versions and configurations, we validate the reliability and reproducibility of our findings. On average, SyzPrime find another three primitives in the modern Linux kernel. These newly discovered primitives represent previously unknown exploitation techniques, highlighting the capability of SyzPrime to identify novel attack vectors that escaped detection by existing methodologies. The contributions of this research are summarized below.

- We identify critical limitations in current exploitation primitive discovery approaches through systematic analysis of recent research.
- We propose SyzPrime, a novel fuzzing-based framework combining static and dynamic analysis for automated and reliable primitive discovery.
- We evaluate SyzPrime on 40 real-world CVEs, demonstrating its effectiveness by identifying 108 exploitable objects in production kernel environments.

2 BACKGROUND

In this section, we will introduce the exploitations and primitives in the modern Linux kernel.

ID	Year	Conference	Short Title	Vulnerability	Target	Technique
1	2017	NDSS	Stack-Spray [10]	UUM	kernel	S2E Trinity
1	2018	USENIX	FUZE [11]	UAF	kernel	Syzkaller
2	2019	USENIX	Kepler [12]	CFHP	kernel	Angr
3	2019	CCS	Slake [13]	UAF/OOB/DF	kernel	LLVM Syzkaller
4	2020	USENIX	KOOBE [14]	OOB	kernel	S2E Angr
5	2020	CCS	ELOISE [5]	UAF/OOB/DF	kernel	LLVM
6	2023	TIFS	TAODE [6]	UAF/OOB	kernel	LLVM
7	2023	USENIX	AlphaEXP [4]	ALL	kernel	KINT Syzkaller
8	2019	CCS	Gollum [8]	BOF	interpreters	SELF
9	2023	NDSS	Bagua [15]	HLM	user	SELF
10	2021	USENIX	MAZE [16]	HLM	user	SELF
11	2018	NDSS	SHRIKE [9]	HLM	user	SELF
12	2023	USENIX	Scatter [7]	HLM	user	FUZZ

Table 1. List of a selection of 16 papers discussing (non)-reproducibility in kernel fuzzing

2.1 Kernel Heap Vulnerability

Kernel heap vulnerabilities continue to pose security threats to modern operating systems, particularly within the Linux kernel's SLUB allocator, which employs complex memory management mechanisms. However, these complexities can inadvertently introduce subtle flaws that malicious adversaries can exploit. The SLUB allocator's intricate handling of memory includes various operations such as object allocation, deallocation, and the maintenance of complex data structures, which can lead to a range of memory corruption issues. Among the most concerning vulnerabilities are Use-After-Free (UAF), Out-of-Bounds (OOB) writes, and Double-Free (DF) errors. The sophisticated interplay between allocation and deallocation processes, alongside the various data structures utilized by the SLUB allocator, creates an environment where identifying and analyzing exploit primitives becomes increasingly challenging. This is further complicated by the dynamic nature of heap operations, which can vary significantly depending on runtime conditions and workloads. Despite extensive research and continuous improvements in security practices, systematically identifying and analyzing exploit primitives derived from heap vulnerabilities remains a significant challenge.

2.2 Kernel Heap Exploitation

Recent research in kernel heap exploitation has primarily focused on discovering exploitable objects in the Linux kernel. Various approaches have emerged targeting different types of objects: ELOISE [5] introduces elastic objects that contain length fields and leak channels, while AlphaEXP [4] identifies security-sensitive kernel objects by simulating adversarial behaviour through knowledge graph analysis. TAODE [6] proposes Thanos objects with heap pointers for transforming weak primitives into strong ones. Despite these advances in object identification, existing

approaches tend to focus on specific object types:

- Elastic objects for information leakage,
- Thanos objects for primitive transformation,
- Security-sensitive objects for exploit generation.

However, none provides a comprehensive framework for discovering and analyzing HLM primitives, which are crucial for exploitation.

2.3 Kernel Exploitation Objects

Exploit primitives [17] refer to capabilities granted during a reasonably generic exploit, include write, read and execution capabilities. Research on the security of kernel heap vulnerability exploitation is primarily based on the **primitives**. Exploitable vulnerabilities utilize powerful primitives that can read or write any byte, referred to as "AAW" or "AAR". Many such primitives exist, yet the powerful ones alone are not what matter; weaker primitives are equally essential, often serving as the foundation for stronger ones. These primitives are used to connect the exploitable states. Kepler [12] and Retspill [18] is control-flow hijacking primitive is one of the most exploitable gadgets in Linux kernel.

ID	Struct	Cache	Size	LK	LH	LS	CFH	RW
1	signalfd_ctx	kmalloc-8	8	×	✓	×	×	×
2	setxattr∗	elastic	$< 65\,536$	×	×	×	✓	✓
3	ldt_struct	kmalloc-16	16	×	×	×	×	✓
3	shm_file_data	kmalloc-32	32	✓	✓	×	×	✓
4	subprocess_info	kmalloc-128	96	✓	×	×	✓	×
5	seq_operations	kmalloc-32	32	✓	×	×	✓	×
6	user_key_payload	elastic	[24, 0x7fff + 0x18]	✓	✓	×	×	✓
7	msg_msg	elastic	$[0x30, 0x1000]$	×	✓	×	×	✓
8	sendmsg∗	elastic	[44, 0x7fffffff]	×	×	×	×	✓
9	file	kmalloc-256	232	✓	✓	×	✓	×
11	timerfd_ctx	kmalloc-256	216	✓	✓	×	×	×
11	tty_struct	kmalloc-1k	[0x290, 0x2e0]	✓	✓	×	✓	×
12	pipe_buffer	elastic	[0x28, 0x1000]	✓	✓	×	✓	✓
13	packet_sock	elastic	[1 024, 2 048]	✓	×	×	✓	×
14	sk_buff	elastic	[512, . . .]	✓	×	×	✓	×
15	cred†	kmalloc-192	192	×	×	×	×	×

Table 2. List of a selection of 15 objects in kernel fuzzing. ∗ means the object is allocation in this syscall not the object is the syscall name. † means the `cred` object is special, it contains the ultimate goal to PrivilegeEscalator. LK means leak kernel address, LH means leak heap address, LS means leak stack address, CFH means control flow hijacking (can hijack RIP), and RW means readable and writable. The method is one of the method to alloc the targeted object.

3 HEAP SPRAY AND EXPLOITATION

3.1 Heap Spray

The heap spray primitive is crucial in Linux kernel exploitation. Heap spray is a critical primitive in Linux kernel exploitation, serving as a fundamental technique for achieving reliable exploitation outcomes. Heap spraying serves as a key technique in the exploitation process, where the kernel's SLUB allocator is used to deliberately allocate multiple objects of specific sizes to fill the kernel heap in a controlled manner. This technique is especially important when manipulating SLUB cache layouts or when precise object placement is necessary for successful exploitation. Unlike user-space heap spraying, kernel heap spraying must carefully consider the characteristics of the SLUB allocator, cache selection mechanisms, and potential interference from concurrent kernel operations. HLM is an important aspect of the heap spraying techniques in Linux kernel exploitation. Within the context of the SLUB allocator, HLM focuses on precisely placing target objects at specific SLUB cache locations to create a controlled heap layout that aids in exploitation. This technique is essential for various exploitation scenarios: when exploiting SLUB buffer overflows, attackers must position objects containing function pointers or critical kernel data structures adjacent to the vulnerable object; for use-after-free vulnerabilities, it is crucial for attackers to ensure that freed SLUB objects are reallocated with carefully crafted target objects of matching sizes.

Success in HLM often necessitates a deep understanding of the SLUB allocator's free list management. Attackers must strategically exhaust specific SLUBs while maintaining precise control over object allocation sequences and mitigating interference from standard kernel operations.

3.2 Exploitation Model

In kernel heap exploitation, vulnerable objects serve as the initial attack vector, while victim objects contain targetable data structures, such as function pointers. In this context, spray objects are crucial for achieving reliable manipulation. The utilization of effective spray objects is crucial for exploitation, as these objects must provide precise control over memory layout while minimizing interference with normal kernel operations. Successful spray objects typically share common characteristics: they can be allocated in large quantities without side effects, their life cycle can be controlled through user-space operations, and their size properties allow for strategic cache manipulation. This emphasis on spray object properties reflects their fundamental role in bridging the gap between vulnerability triggers and successful exploitation.

Out-of-Bounds vulnerability. OOB is a vulnerability that allows access beyond a predetermined size (e.g., heap buffer overflow, heap out-of-bound write). This vulnerability classifies two types, OOB-Object and OOB-Freelist [3], which

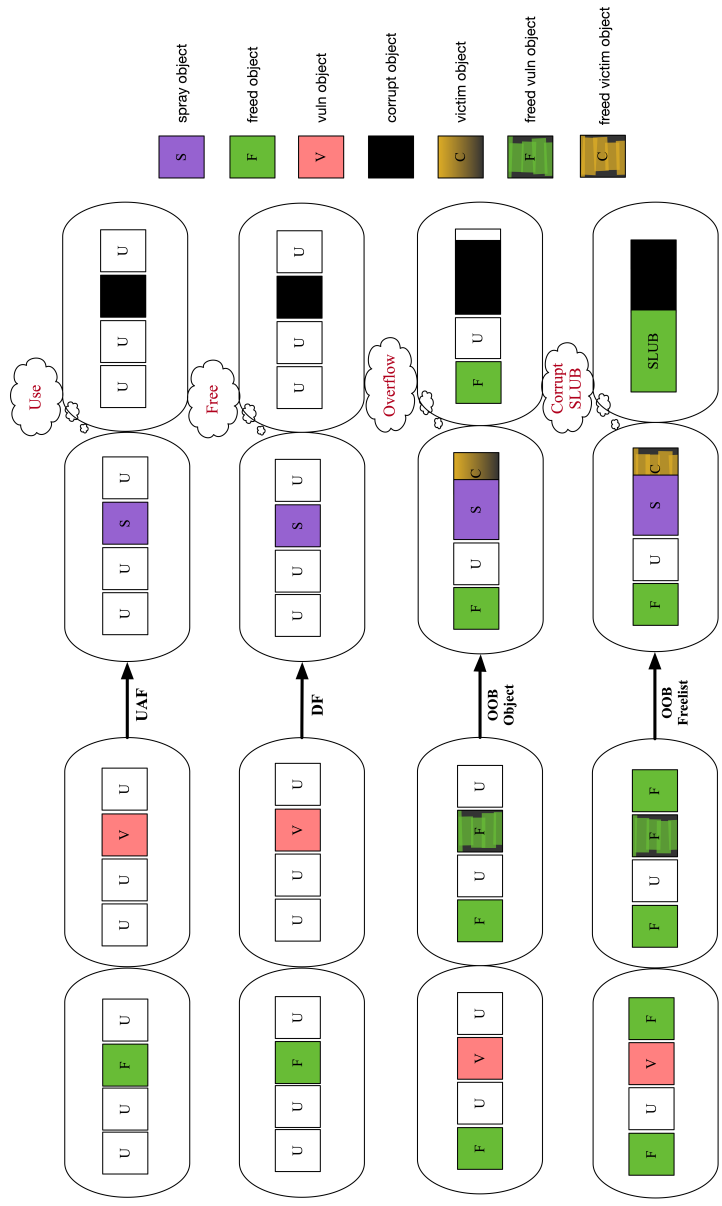


Figure 1. The exploitation model for kernel heap vulnerabilities centres on the precise manipulation of memory layouts, with particular emphasis on heap spray techniques and layout manipulation phases. Note that the purple "S" indicates spray objects, which must be precisely manipulated alongside specific vulnerable objects to enable deterministic and controlled exploitation outcomes.

means the overflowed object is an in-use object or a freed object; it can have totally different exploitation methods. OOB-Object enables us to manipulate the critical fields of a victim object following the sprayed object and overwrite the fields to gain primitives. The attacker needs to hijack the SLUB free list, which stores metadata in the adjacent freed slot. By manipulating our target object, the attacker can overwrite the metadata of the freed slot to hijack the last allocation.

Use-after-Free vulnerability. Use-after-Free vulnerability occurs when an object is accessed after its memory has been released. This vulnerability typically unfolds in two stages. The first stage involves freeing the vulnerable object while neglecting to set the dangling pointer to NULL. The second stage involves another part of the program attempting to use this object. For exploitation, the attacker needs to allocate a target object to manipulate the freed object. In the second stage, there is an opportunity to utilize powerful primitives in the inconsistent state of our target object. The most important part is that we need to allocate an object in the proper time slot and avoid the side effects. (e.g., allocate another same cache object in the period.)

Double-Free vulnerability. Double-Free (DF) vulnerability often arises from issues such as reference counting bugs or logic errors. DF is similar to UAF but occurs when an attacker frees an object that has already been freed. To exploit a DF vulnerability, first, the attacker allocates a victim object at the target. As a result, the attacker can hijack the program counter in the same approach as described above in Figure 1.

4 SYZPRIME

In this section, we will introduce our framework SyzPrime and show how it works in addressing HLM puzzles in the Linux kernel exploitation.

Listing 1 show the `sendmsg` primitive. We should notice that the core logic of allocation is `sock_kmalloc(sock->sk, ctl_len, ...)` denoted as control sendmsg that we can allocate size from 0x28–0x7fffffff (0x28 is the size of `sizeof(struct cmsghdr)` and 0x7fffffff is the `INT_MAX`).

4.1 Object-Driven Instrumentation

Building upon a comprehensive collection of security-sensitive objects, we introduced an object-driven instrumentation method for object tracking within the Linux kernel. Specifically, we augmented these objects with technological markers through compiler modifications (e.g., GCC, LLVM). Shown as Listing 1, the security sensitive object is `ctl_buf` in line 23. This instrumentation enables us to track all basic blocks that allocate, (de)allocate, or directly manipulate (e.g., Read or Write) the `ctl_buf` during kernel compilation. As a result, each tracked basic block returns a kernel block address annotated with our custom markers (e.g., basic block address

```
void *sock_kmalloc(...){
  int max = READ_ONCE(sock_net(sk)->core.sysctl_max);
  if ((int)size <= max && atomic_read(&sk->omem_alloc)+size
      < max){
    void *mem;
    atomic_add(size, &sk->omem_alloc);
    mem = kmalloc(size, priority);
    if (mem)
      return mem;
      atomic_sub(size, &sk->omem_alloc);
  }
  return NULL;
}
EXPORT_SYMBOL(sock_kmalloc);

static int ____sys_sendmsg(...){
  char ctl[sizeof(struct cmsghdr)]
  char *ctl_buf = ctl;
  int ctl_len;
  ssize_t err;
  ctl_len = msg_sys->msg_controllen;
  if (ctl_len) {
    if (ctl_len > sizeof(ctl)){
      ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL)
         ; [1]
      if (ctl_buf == NULL)
        return -ENOBUFS;
    }
    err = copy_from_user(ctl_buf, msg_sys->msg_control_user
       , ctl_len); [2]
    if (err)
      return -EFAULT;
    msg_sys->msg_control = ctl_buf;
    msg_sys->msg_control_is_user = false;
  }
  return 0;
}
```

Listing 1. Simplified `sendmsg` primitive code snippet emphasizing `sock_kmalloc` in lastest stable branch 6.11.5, including sock_kmalloc and ____sys_sendmsg two functions

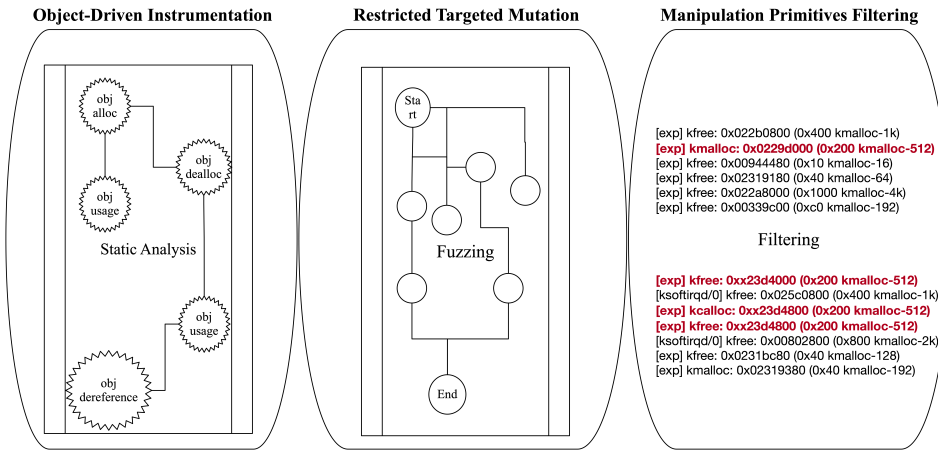


Figure 2. The architecture of SyzPrime comprises three components that work together to discover reliable heap manipulation primitives. The red-highlighted sections in the "Filtering" represent the interactions of objects that need to be filtered.

with a magic number in the first 16 bits), facilitating comprehensive object lifecycle monitoring.

SyzPrime start our analysis using static taint analysis to identify the critical data handling the locations as mentioned earlier. Specifically, in our example, the object `ctl_buf`, which is allocated at line 23 in Listing 1 and pinpointed as the source of the taint. It is notably and obviously utilized in line 27 of the convention of `copy_from_user`, serving as the sink to give the adversary total control of the memory of the object. Considering implicit data-flow analysis, the assignment `msg_sys->msg_control = ctl_buf` at line 30 xx represents an implicit sink, highlighting a secondary propagation pathway that could extend the influence of `ctl_buf` to other locations interacting with `msg_sys`. The data dependency analysis implemented in SyzPrime adopts a methodology akin to traditional data flow analysis that meticulously tracks the initial definition points and maintains distinct records for each usage location of these defined variables. Furthermore, SyzPrime's strategy enhances precision by mapping out the connections between variable definitions and their subsequent utilization across the function.

By leveraging a targeted approach to data dependency tracking, SyzPrime aims to strike a balance between thoroughness and efficiency, making it well-suited for large-scale Linux kernel where managing complex data structures and interactions is critical for reliability and performance. SyzPrime leverages the fuzzing capabilities of Syzkaller to conduct comprehensive kernel fuzzing based on the above instrumentation. When a syscall is generated by Syzakller, the associated program is executed in the instrumentated kernel environment (as detailed in Section 4.2). Throughout this process, an object-driven instrumentation mechanism leverages `ftrace` [19]

to meticulously capture a comprehensive execution trace. When modifications are made to the kernel, ensuring that `ftrace` specifically captures the behavioural dynamics at the targeted points is crucial for thorough analysis. Our instrumented approach allows us to configure `ftrace` to actively monitor and log syscalls and object allocations at these critical locations. This specialized logging is pivotal as it enables the identification and documentation of all the behaviour directly related to the object areas. By doing so, SyzPrime enhances our ability to gather relevant data that assists in pinpointing the underlying causes of any discrepancies or errors.

4.2 Restricted Targeted Mutation

Our restricted targeted mutation approach aims to generate focused mutations around specific syscalls $\mathcal{S}$ to enrich object-reaching behaviour while maintaining precise control over the fuzzing process. The "restricted" aspect refers to our modifications to Syzkaller's mutation process to limit its overly flexible mutation strategies. On the other hand, "targeted" ensures that each generated or mutated seed is precisely engineered to reach our instrumentation locations consistently.

Initially, SyzPrime constrains the mutation scope to a carefully selected set of highly relevant syscalls. In this scenario, Syzkaller utilizes templates to guide mutation across two dimensions: syscall mutation and parameter mutation. Upon identifying a program $\mathbb{P}$ (specifically, the syscall $\mathcal{S}$ within $\mathbb{P}$) that aligns with our objectives, we designate it as a candidate program. Subsequently, we halt any further mutations at the syscall level to ensure that all subsequent mutations are confined solely to parameter levels.

Furthermore, SyzPrime leverages Syzkaller's choice table[1] $\mathbb{CT}$ to effectively multiply static and dynamic priorities, thereby identifying syscalls that exhibit the strongest correlations with specific outcomes. By meticulously analyzing Syzkaller's corpus (e.g., enriched corpus[2]), we strategically select the n most pertinent syscalls $\mathcal{S}_i$ as candidate syscalls based on their historical interaction patterns in the choice table. SyzPrime precisely confine the mutation process to these most relevant n syscalls, significantly streamlining the search and enhancing the efficiency of identifying a robust number of promising candidates $\mathbb{P}$s.

Lastly, SyzPrime elaborates on parameter mutation through fine-grained transformations (e.g., hint mutations). Our focus intensifies particularly on the last triggering syscall $\mathcal{S}_m$, implementing targeted mutations while continuously evaluating their feedback from our instrumentation, as elaborated in Section 4.3. We utilize a feedback-driven evaluation mechanism to assess the quality of each mutation, ensuring that we achieve both coverage and precision in our targeted fuzzing. Moreover, SyzPrime optimizes parameter variations based on historical success patterns,

[1] `https://github.com/google/syzkaller/pull/4674`

[2] `https://github.com/cmu-pasta/linux-kernel-enriched-corpus`

which, when combined with our selective mutation strategy, ensures the diversifying of the behaviour of target objects. Recognizing the enhanced value of mutating parameters of $\mathcal{S}$ for triggering diverse `ftrace` recordings, SyzPrime increases the parameter mutation probability for the last triggering syscall $\mathcal{S}_m$. From the parameter perspective, SyzPrime ensures that mutations adhere to the constraints specified in the template, thus enhancing the overall effectiveness of fuzzing.

4.3 Manipulation Primitives Filtering

To facilitate targeted analysis of a specific object, we propose a manipulation primitives filtering algorithm, which involves modifying `ftrace` to support tracing the allocation and deallocation of a particular cache for a specific object size. By instrumenting all instances of the object, following the methodology presented in Section 4.1, SyzPrime enables fine-grained filtering and analysis of the object's life cycle. By selectively applying `ftrace` to individual objects, fuzzing-based tracing instrumentation provides powerful feedback to identify and understand the root causes of unexpected side effects related to specific objects.

Through extensive fuzzing of relatively similar seeds with a mutation in Section 4.2, we developed an enhanced tracing and filtering mechanism by modifying the `ftrace` to obtain a more granular understanding of how syscalls impact behaviour. Our modified tracing framework provides detailed insights into the kernel object interactions, particularly focusing on the allocation, deallocation, and utilization patterns of critical objects, including objects sharing the same cache space. This comprehensive tracking enables a more nuanced approach to primitive filtering during the continuous fuzzing.

Our enhanced filtering mechanism utilizes a rapid assessment approach to evaluate the capabilities of each syscall concerning primitive operations. SyzPrime extends `ftrace`, focusing on both target objects $\mathcal{O}$ and their associated objects $\mathcal{O}'$ within the same cache, facilitating comprehensive tracking of allocation and deallocation patterns. Native `ftrace`, whether configured for `function-graph` tracing, generates extensive logging data that requires sophisticated filtering strategies. To address this challenge, SyzPrime adapted the core `ftrace` algorithm to monitor memory operations as `object` tracer selectively, maintaining visibility into critical object interactions. The modification lays the groundwork for efficient log processing, employing binary search techniques to sift through the voluminous trace data rapidly.

This strategy enables the construction of a detailed call relationship network by correlating object lifecycle events across various kernel subsystems. Through this refined methodology, SyzPrime precisely tracks the target objects and efficiently manages trace data. The assessment metrics encompass multiple dimensions of object interaction, including allocation frequency, deallocation patterns, associated object types, specific allocation sizes, and the number of exploitable bytes. This enhanced primitive filtering approach enables us to construct a detailed capability profile for each syscall, offering insights into both direct and indirect effects on kernel memory management. By maintaining detailed statistics about object interactions,

we can more effectively guide the fuzzing process toward promising execution paths that exhibit interesting memory manipulation patterns. The granular nature of our tracing allows us to identify subtle interactions between different kernel subsystems, particularly in cases where object allocations and deallocations might have security implications.

5 IMPLEMENTATION

In this section, we implemented our idea as a prototype and named it SyzPrime. As mentioned above, our technique integrates both static and dynamic analysis through a comprehensive framework incorporating compiler modifications, fuzzer extensions, and kernel tracing enhancements.

The static analysis leverages compiler adjustments in both LLVM and GCC frameworks. For LLVM, we implemented lightweight static taint propagation analysis to track object interactions across function boundaries. In GCC, we enhance the `sancov_pass` and incorporate instrumentation functionality to monitor object life cycle with our instrumentated code via `ftrace`. These modifications facilitate precise tracking mechanisms while maintaining reasonable performance overhead.

The dynamic analysis component is realized through extensive modifications to the Syzkaller fuzzer framework written in over 1 000 lines of Go code, building upon its architecture to support syscall configuration, parameter filtering and targeted mutation. SyzPrime incorporates several concepts from SyzDirect to improve the precision of our targeted fuzzing approach. Our modifications enable fine-grained mutation over syscall selection and parameter generation through a prioritized scheduling strategy, where mutation rates are dynamically adjusted based on observed object interactions, allowing us to focus on sequences that exhibit promising manipulation patterns during recording and filtering.

Our manipulation filtering component extends the kernel's `ftrace` infrastructure with 500+ lines of custom C extensions, enhancing kernel-side tracing capabilities and a user-space interface to precisely track object interactions, allocation events, and memory operations. The core mechanism processes `ftrace` output through Python scripts to construct object interaction graphs efficiently. This integration of compiler instrumentation and enhanced tracing enables efficient identification and evaluation of kernel object interactions, providing metrics that guide the fuzzing process based on allocation patterns and memory operations.

6 EVALUATION

In this section, we conducted experiments to evaluate SyzPrime's performance and answer the following questions:

RQ1: Can SyzPrime effectively generate primitives in a real-world stable production Linux kernel environment?

RQ2: Can SyzPrime discover more primitives compared to state-of-the-art approaches?

RQ3: Can SyzPrime effectively identify exploitable primitives in real-world cases?

6.1 Setup

Dataset. We use the 20 primitives in Figure 6. For the corpus, we will use the corpus dump from Syzbot[3] and enriched-corpus. Syzkaller executes corpus, and their coverage is up to 90 % of the max coverage. Experiments were conducted on a server running Debian 12, equipped with a 2 × Intel(R) Xeon(R) Gold 5 218 CPU (2.30 GHz, 32 cores) and 512 GB of RAM. To maintain equitable conditions, each fuzzing virtual machine was allocated an identical configuration (4 cores, 8 GB of memory) and ran multiple times, with results averaged to account for variability.

Baselines. We compare SyzPrime with five other approaches. Since SyzPrime focuses on discovering HLM primitives that are different from others, we choose some approaches similar to ours. SOTA AlphaEXP [4] is our baseline, which focuses on the finding of security-sensitive objects by constructing a knowledge graph. SLAKE [13] is a pioneering approach designed to facilitate Linux kernel exploitation by systematically manipulating sensitive kernel objects but not only focusing on manipulating. KOOBE [14] extracts the capabilities of OOB vulnerabilities through capability-guided fuzzing and symbolic tracing, then matches these capabilities with potential target objects to evaluate exploitability. ELOISE [5] systematically analyzes elastic objects in the Linux kernel that contain length fields and leak channels, then pairs them with vulnerability capabilities to evaluate their potentials. TAODE [6] Thanos objects in the Linux kernel to transform weak exploit primitives into strong exploit primitives by exploiting their heap pointers and release paths to create vulnerable overlapping.

Prototype System. Our prototype system runs on Debian 12 with 256 GB RAM. Our evaluation process is performed on the same stable kernel v4.15 and v5.10 with default config in GCD and other AWS production environments.

6.2 Primitives Generation (RQ1, RQ3)

To answer **RQ1**, we use SyzPrime to generate *HLM primitives* for those 40 real-world CVEs of objects as listed in Table 6. For more real-world cves, we choose the real CVEs and their exploitation for evaluation as listed in Table 3. In this experiment, SyzPrime generates 80 % (32 out of 40) primitives than other tools 55 % (22 out of 40), which we try all the approaches in the baselines.

[3] `https://syzkaller.appspot.com/upstream`

ID	CVE	Type	Cache	Capability	Object	SyzPrime	Ohters	Primitives
1	CVE-2019-15666	UAF	kmalloc-1k	(8, 8, NULL)	setattr∗	✓	✓	1
2	CVE-2017-6074	DF	kmalloc-2k	(0, 2 048, arb)	sk_buff	✓	✓	21
3	CVE-2017-7184	OOB	kmalloc-[32,512]	(0, *, arb)	cred	✓	✗	4
4	CVE-2017-8890	DF	kmalloc-256	(0, 256, arb)	ipv6_mc_socklist	✗	✗	–
5	CVE-2017-11176	DF	kmalloc-1k	(0, 1 024, arb)	sendmsg∗	✓	✗	1
6	CVE-2019-15666	UAF	kmalloc-1k	(8, 8, NULL)	setxattr∗	✓	✗	1
7	CVE-2021-22555	DF	kmalloc-1k	(0, 1 024, arb)	pipe_buffer	✓	✗	2
8	CVE-2021-22600	DF	kmalloc-256	(0, 256, arb)	pipe_buffer	✓	✗	2
9	CVE-2021-23134	DF	kmalloc-1k	(0, 1 024, arb)	io_buffer	✗	✗	–
10	CVE-2021-26708	DF	kmalloc-64	(0, 64, arb)	sk_buff	✓	✓	21
11	CVE-2021-41073	UAF	kmalloc-32	(0, 32, arb)	io_buffer	✓	✓	1
12	CVE-2021-42008	OOB	kmalloc-4k	(0, 4 096, arb)	msg_msg	✓	✓	6
13	CVE-2021-43267	OOB	kmalloc-1k	(0, 1 024, arb)	tty_struct	✓	✗	1
14	CVE-2022-0185	OOB	kmalloc-4k	(0, 4 096, arb)	msg_msg	✓	✓	6
15	CVE-2022-0995	DF	kmalloc-1k	(0, 1 024, arb)	msg_msg	✓	✗	6
16	CVE-2022-1015	OOB	–	–	–	✗	✗	–
17	CVE-2022-2588	DF	kmalloc-[192,256]	(0, 192-256, arb)	file	✓	✗	5
18	CVE-2022-2639	OOB	kmalloc-64k	(0, 65 536, arb)	msg_msg	✓	✓	6
19	CVE-2022-24122	UAF	kmalloc-1k	(0,128,arb)	msg_msg	✓	✗	6
20	CVE-2022-25636	OOB	kmalloc-[32-4k]	–	msg_msgseg	✗	✗	–
21	CVE-2022-27666	OOB	8-page	(0, 32 768, arb)	msg_msg	✓	✓	6
22	CVE-2022-32250	UAF	kmalloc-64	(24, 8, hptr)	user_key_payload	✓	✓	5
23	CVE-2023-0461	UAF	kmalloc-512	(0, 64, arb)	user_key_payload	✓	✗	5
24	CVE-2023-3390	UAF	kmalloc-cg-512	(0, 512, arb)	msg_msg	✓	✓	1
25	CVE-2023-34918	OOB	kmalloc-64	(0, 64, arb)	user_key_payload	✓	✓	5
26	CVE-2023-3611	OOB	kmalloc-128	(0, 128, arb)	subprocess_info	✓	✗	1

...

...

ID	CVE	Type	Cache	Capability	Object	SyzPrime	Ohters	Primitives
26	CVE-2023-3611	OOB	kmalloc-8k	(0, 8 192, arb)	qdisc	✓	✓	2
27	CVE-2023-3777	UAF	kmalloc-cg-16	(0, 16, arb)	nft_expr	✓	✓	8
27	CVE-2023-3777	UAF	kmalloc-cg-64	(0, 16, arb)	nft_rule	✓	✗	1
28	CVE-2023-4004	UAF	kmalloc-192	(0, 64, arb)	nft_table	✗	✓	–
30	CVE-2023-4015	UAF	kmalloc-cg-16	(0, 16, arb)	nft_expr	✓	✗	8
31	CVE-2023-4206	UAF	kmalloc-128	(0, 64, arb)	socket	✓	✓	1
32	CVE-2023-4244	UAF	kmalloc-cg-128	(0, 64, arb)	nft_rule	✗	✗	1
33	CVE-2023-52447	UAF	kmalloc-1k	(0, 1 024, arb)	array_of_maps	✗	✗	–
34	CVE-2023-6817	UAF	kmalloc-128	(0, 128, arb)	nft_set_elem	✗	✗	–
35	CVE-2023-6932	OOB	kmalloc-192	(0, 64, arb)	user_key_payload	✓	✓	5
36	CVE-2024-0193	UAF	kmalloc-cg-16	(0, 16, arb)	nft_expr	✓	✗	8
37	CVE-2023-4569	UAF	kmalloc-128	(0, 128, arb)	nft_set_elem	✗	✗	–
38	CVE-2024-26581	UAF	kmalloc-cg-16	(0, 16, arb)	nft_expr	✓	✗	1
39	CVE-2024-26808	UAF	kmalloc-cg-4k	(0, 4 096, arb)	net_device	✓	✗	2
39	CVE-2024-26808	UAF	kmalloc-cg-192	(0, 192, arb)	msg_msg	✓	✓	1
40	CVE-2024-26925	UAF	kmalloc-cg-256	(0, 256, arb)	nft_set	✗	✗	–

Table 3. The summary of exploitable primitives discovering in 40 real-world vulnerabilities. There CVEs have three root causes as detailed in Section 3 of OOB, UAF and DF, "EXEC" means the vulnerability have a strong capabiliry to exexute kernel code. In the "Capability" column, *arb* denotes that the vulnerability can write an arbitrary value, *NULL* denotes the capability to write NULL char as the same content as TAODE [6]. "Primitives" column means the number of primitives of interaction of object without side effects. Note that the "Object;; column is one of exploitation used this object for heap spary.

Notice that, the vulnerability in the Netfilter subsystem, the explosion often uses `nft*` object for heap spray or heap layout manipulation since the flexibility in these object allocation and deallocation without side effects. SyzPrime can extend to search the HLM primitives in their objects, but others, like ELOISE, do not identify the nft_expr as an elastic object and can not discover the primitives, as does TAODE. In Table 3, we could found msg_msg is the most commonly used for exploitation due to its elastic and strong primitives and SyzPrime almost discovering his primitives totally, others also. However, when encountering less commonly used kernel objects such as `socket`, `nft_expr`, and `timerfd_ctx`, others often fail to identify them due to their reliance on inflexible strategies that are almost entirely based on static analysis. SyzPrime based on both static and dynamic analysis, we can find more primitives than others only find one. To answer **RQ3**, we need to focus on the to enquotePrimitives that represents the HLM primitives without side effects; more than one seems that SyzPrime can change the exploit with our found primitives and densify the exploitation when is important for assessment.

6.3 Comparsion with State-of-the-Art (RQ2, RQ3)

To answer **RQ2**, we compare SyzPrime against five state-of-the art tools: SLAKE [13], KOOBE [14], EILOSE [5], AlphaEXP [4] and TAODE [6]. We utilize AlphaEXP as our primary benchmark due to its comprehensive approach – it simulates adversarial analysis and exploitation by constructing a knowledge graph to represent kernel source code and infer attack paths for identifying exploitable objects as detailed in Table 4. Its comprehensive knowledge graph representation makes it a strong baseline for our evaluation.

The comparative performance evaluation between SyzPrime and the current state-of-the-art approaches is presented in Table 4. The results demonstrate that SyzPrime successfully identifies more objects (108), significantly outperforming most existing approaches, though remaining second to AlphaEXP, which identifies 240 sensitive objects in total. In comparison, SLAKE identifies 33, EILOSE 40 and KOOBE only 17, respectively.

This performance disparity can be attributed to the specialized focus of existing approaches. EILOSE targets elastic objects exclusively, which need a length in the object, KOOBE focuses solely on out-of-bounds related objects, and SLKAE emphasizes object identification rather than slab manipulation. While SyzPrime primarily focuses on HLM primitives, we enhanced its capabilities by incorporating additional capabilities captured in our filtering modules and extending our analysis across the entire corpus using Syzkaller, specifically targeting primitives without side effects and their corresponding objects. The higher quantity of HLM primitives identified by SyzPrime correlates directly with its superior ability to discover relevant objects, demonstrating its more comprehensive coverage compared to existing approaches. Our evaluation conclusively demonstrates that SyzPrime not only effectively identifies exploitable primitives but also discovers a broader range of objects and primitives compared to existing approaches. This comprehensive

identification capability makes SyzPrime a more versatile tool for security analysis.

7 DISCUSSION

Our approach demonstrates several significant advantages in kernel HLM primitives fuzzing. The integration of static instrumentation with dynamic analysis provides a comprehensive view of object lifecycles, enabling more precise targeting of potential vulnerabilities. The restricted mutation strategy significantly reduces the search space while maintaining effectiveness, and the primitives filtering mechanism proves particularly effective in identifying meaningful object interactions, allowing for more efficient resource allocation during the fuzzing process. Moreover, the modification of Syzkaller and `ftrace`, rather than building entirely new frameworks, ensures our approach can be readily integrated into existing workflows.

However, our implementation has limitations that suggest directions for future work. First, our static analysis could be enhanced to better handle complex pointer aliasing and inter-procedural flows, as our implementation may miss subtle object relationships, especially those involving function pointers or callback invocations. Second, although the performance overhead of our tracing approach remains acceptable for single-object fuzzing, there is room for optimization, particularly in large-scale scenarios. Additionally, our primitive filtering mechanism currently relies on manual configuration. Developing automated methods would significantly enhance the system's adaptability and scalability. Finally, extending SyzPrime to handle object interactions better poses unique challenges in state management within the Linux kernel. We seem to have a promising direction for future work, as capturing these complex interactions would significantly assess the system's vulnerability exploitation capabilities.

8 RELATED WORK

8.1 Kernel Exploitation

Many Linux Kernel Exploitation works have proliferated due to its increasing security relevance as the kernel gains widespread adoption. Recent works, such as SlubTrick [1], PageSpary [2] and PageJack[20], investigate the Linux kernel exploitation techniques, targeting the SLUB allocator layer, the page allocator layer, and data-only attack style exploitation, respectively. Other studies have focused on various key aspects. ExpRace [21], KHeap [3], and PSPRAY [22] examine the stability in the exploitation, while AEM [23] and SyzBridge [24] explore the exploitation migration techniques to expand the potential impacts. Additionally, SyzScope [25], GREBE [26] and KLAUS [27] aim to analyze multi-behavioral patterns to identify the potential exploitability, yet DirtyCred [28] and Retspill [18] demonstrate innovative methods to compromise the kernel directly. In contrast, SyzPrime only focuses

Objects in Linux v4.15 with default configuration	
Write	keyctl_update_key◇, msg_msg◇, add_key◇, ip_options_get_from_user◇, scsi_request, drm_ioctl, __get_filter, hiddev_ioctl_usage, proc_ioctl, kexec_segment, do_ipv6_setsockopt, do_semtimedop, vt_do_kdgkb_ioctl, setxattr, xt_table_info, gss_pipe_downcall, snd_ctl_elem_info, simple_transaction_argresp, ethtool_set_eeprom, sock_filter, proc_do_submiturb, snd_ctl_elem_id, map_lookup_elem, move_addr_to_kernel, compat_agpioc_reserve_wrap, drm_syncobj_array_find, fb_sys_write, fb_write, drm_mode_dirtyfb_ioctl, ipv6_txoptions, elf_prpsinfo, sk_buff, usblp_write, drm_crtc, drm_property_blob, drm_i915_gem_object, tty_struct, agpioc_reserve_wrap, create_entry, drm_syncobj_array_wait_timeout, kernfs_fop_write, kexec_segment, map_update_elem, proc_bulk, sendmsg, simple_attr_write, rawv6_seticmpfilter, snd_info_buffer, memfd_create×, drm_syncobj_timeline_signal_ioctl×, raw_seticmpfilter⋆, cpumask⋆
Read	ipv6_opt_hdr⋆, sock_fprog_kern⋆, policy_load_memory⋆, ldt_struct⋆, ip_options⋆, cfg80211_wowlan_tcp⋆, seq_file⋆, xfrm_policy⋆, xfrm_algo_aead⋆, xfrm_algo⋆, ip_sf_socklist⋆, proc_dir_entry⋆, station_info⋆, cfg80211_pkt_pattern⋆, user_key_payload⋆, xfrm_replay_state_esn⋆, ext4_dir_entry_2⋆, mon_reader_bin⋆, sg_header⋆, tc_cookie⋆, inotify_event_info⋆, audit_rule_data⋆, fb_info⋆, cfg80211_sched_scan_request⋆, fb_cmap_user⋆, cache_request⋆, fname⋆, ieee80211_mgd_auth_data⋆, cfg80211_bss_ies⋆, cache_reader⋆, mon_reader_text⋆, tcp_fastopen_context⋆, request_key_auth⋆, xfrm_algo_auth⋆, cfg80211_scan_request⋆, msg_msg⋆, tcp_sock×, user_element, neighbour, pneigh_entry, net_device, netdev_phys_item_id, rchan_buf, netlink_ext_ack, cfg80211_nan_match_params, wiphy, wiphy_iftype_ext_capab, wireless_dev, usb_device, urb, hidraw_report, hid_device, sg_request, usblp, drm_crtc, drm_plane, cfg80211_connect_resp_params, fb_cmap, beacon_data, probe_resp, cfg80211_roam_info, cfg80211_wowlan_wakeup, cfg80211_ssid, drm_master, seq_buf, cfg80211_mgmt_tx_params, ieee80211_mgd_assoc_data, kobj_uevent_env, rpc_pipe_msg, geneve_opt, __kfifo, cfg80211_ft_event_params, fat_ioctl_filldir_callback, key_params, drm_property_blob, tcp_fastopen_cookie, cfg80211_pmsr_ftm_result×, cfg80211_update_owe_info×, cfg80211_fils_resp_params×, sg_scsi_ioctl⋆
Exec	seq_operations◇, perf_event_context◇, linux_binprm◇, vmap_area◇, tty_struct◇, seq_file◇, avc_node◇, kioctx_table◇, snd_seq_timer◇, tty_ldisc◇, sk_security_struct◇, assoc_array_edit◇, cgroup_namespace◇, file◇, ext4_allocation_context◇, tty_file_private◇, subprocess_info◇, timerfd_ctx◇, ccid◇, ip_options◇, kioctx◇, ip_sf_socklist◇, request_key_auth◇, pid_namespace◇, k_itimer◇, ip_mc_list◇, sock◇, ip_mc_socklist◇×, key◇×, packet_sock×, fsnotify_group×, blk_plug_cb, blk_stat_callback, snd_timer, hci_dev, snd_pcm_runtime, drm_i915_gem_object, vga_device, nfs_io_completion, snd_pcm, udp_sock, tracer, snd_timer_instance, dio, snd_hwdep, snd_kcontrol, link_master, snd_ketl_ioctl, scsi_cmnd, fbcon_ops, sony_sc, clk_fractional_divider, snd_pcm_hw_rule, snd_seq_device, snd_info_entry, snd_card, snd_jack, net_device, pipe_buffer, shm_file_data, acpi_cpufreq_data, hid_device, ahci_host_priv, snd_seq_client_port, kprobe, sched_domain_topology_level, loop_device, input_polled_dev, hashtab, iommu_group, crypto_ahash, serio, hda_jack_callback, akcipher_instance, ahash_request_priv, crypto_tfm, skcipher_instance, journal_s, input_dev_poller, alps_data, nf_conntrack_expect, crypto_skcipher, aead_instance, crypto_acomp, ubuf_info, psmouse, ml_device, rpc_task, kthread_create_info, proc_inode, proc_dir_entry, fib6_walker, aio_kiocb, simple_attr, inet_connection_sock, nfs_server, ring_buffer, async_entry, filter_pred, nfs_renamedata, nfs_commit_data, nfs_pgio_header, hda_codec, rtnl_link, rpc_rqst, flow_block_cb×, flow_indr_block_cb×, tcf_filter_chain_list_item×, io_wq×, execute_cb×, context_barrier_task×

Table 4. ◇: Identified by SLAKE as well, ⋆: Identified by ELOISE as well, ×: Identified by KOOBE as well, aaa: identified by SyzPrime as well

on specific stages, particularly the HLM stage in the Linux kernel heap exploitation, as heap exploitation remains prevalent among today's exploitable vulnerabilities.

8.2 Automated Exploit Generation

Automated Exploit Generation (AEG) remains a challenging research domain. Early AEG research primarily focused on stack-based vulnerabilities, with significant advances driven by the DARPA Cyber Grand Challenge [29]. These initial approaches leveraged symbolic execution (e.g., angr [30]) to generate exploits for relatively straightforward memory corruption vulnerabilities. The field has expanded significantly to address more complex heap-based vulnerabilities, beginning with SHRIKE [9] and Gollum [8] to exploit PHP and Python heap vulnerabilities automatically. Subsequently, ArcHeap [31], HeapHopper [32] and Evocatio [33] further advanced heap exploitation automation by modelling heap operations and identifying primitives systematically. Revery [34] introduced a novel approach for exploiting through state exploration, while Hack [35] demonstrated automated exploitation in more complex scenarios. Recent research has increasingly focused on root cause analysis [36, 37, 38, 39] to enhance the understanding before exploitation and introduce sophisticated approaches for vulnerability analysis and classification, enabling more precise and reliable exploit generation. SyzPrime builds on these developments, offering deeper insights into kernel heap vulnerabilities to support targeted, efficient primitives.

8.3 Heap Layout Manipulation

HLM has emerged as a significant and formidable challenge in the exploitation of heap-based vulnerabilities. Sean Heelan et al. first systematically address this issue by introducing Gollum [8] and SHRIKE [9], targeting HLM within language interpreters like PHP, Python, Perl etc.. Building upon this foundation, researchers have developed more specialized approaches for HLM. Maze [16] further advances by developing `Dig` & `Fill` algorithm to model HLM as a Linear Diophantine Equation math problem. Scatter [7] addresses HLM by implementing precise heap layout mechanisms for overflow via manipulation distance-guided fuzzing, while BaGua [15] introduces novel techniques under symbolic execution for manipulating heap objects through heap operation dependence graph. However, HLM faces distinctly different challenges in the Linux kernel, where factors like complex object allocation primitives and numerous side effects introduce obstacles that previous research has not fully addressed. SyzPrime aims to tackle these obstacles by focusing on HLM techniques specifically tailored to SLUB allocators.

9 CONCLUSION

In this paper, we have presented a systematic approach to Linux kernel heap manipulation via integrating static instrumentation with dynamic fuzzing. Our frame-

work SyzPrime implements precise tracking through static instrumentation, targeted syscall mutation strategies, and efficient primitive filtering fuzzing mechanisms. Through extensive evaluation, we demonstrate that SyzPrime effectively discovers Linux kernel HLM primitives and sensitive kernel objects in the real-world Linux kernel environments while maintaining reasonable overhead.

REFERENCES

[1] Maar, L.—Gast, S.—Unterguggenberger, M.—Oberhuber, M.—Mangard, S.: SLUBStick: Arbitrary Memory Writes Through Practical Software Cross-Cache Attacks Within the Linux Kernel. 33rd USENIX Security Symposium (USENIX Security 24), 2024, pp. 4051–4068, https://www.usenix.org/conference/usenixsecurity24/presentation/maar-slubstick.

[2] Guo, Z.—Le, D. K.—Lin, Z.—Zeng, K.—Wang, R.—Bao, T.—Shoshitaishvili, Y.—Doupé, A.—Xing, X.: Take a Step Further: Understanding Page Spray in Linux Kernel Exploitation. 2024, pp. 1189–1206, https://www.usenix.org/conference/usenixsecurity24/presentation/guo-ziyi.

[3] Zeng, K.—Chen, Y.—Cho, H.—Xing, X.—Doupé, A.—Shoshitaishvili, Y.—Bao, T.: Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 71–88, https://www.usenix.org/conference/usenixsecurity22/presentation/zeng.

[4] Wang, R.—Chen, K.—Zhang, C.—Pan, Z.—Li, Q.—Qin, S.—Xu, S.—Zhang, M.—Li, Y.: AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects. 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 4229–4246, https://www.usenix.org/conference/usenixsecurity23/presentation/wang-ruipeng.

[5] Chen, Y.—Lin, Z.—Xing, X.: A Systematic Study of Elastic Objects in Kernel Exploitation. Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), 2020, pp. 1165–1184, doi: 10.1145/3372297.3423353.

[6] Liu, D.—Wang, P.—Zhou, X.—Xie, W.—Zhang, G.—Luo, Z.—Yue, T.—Wang, B.: From Release to Rebirth: Exploiting Thanos Objects in Linux Kernel. IEEE Transactions on Information Forensics and Security, Vol. 18, 2022, pp. 533–548, doi: 10.1109/TIFS.2022.3226906.

[7] Zhang, B.—Chen, J.—Li, R.—Feng, C.—Li, R.—Tang, C.: Automated Exploitable Heap Layout Generation for Heap Overflows Through Manipulation Distance-Guided Fuzzing. 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 4499–4515, https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-bin.

[8] Heelan, S.—Melham, T.—Kroening, D.: Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), 2019, pp. 1689–1706, doi: 10.1145/3319535.3354224.

[9] Heelan, S.—Melham, T.—Kroening, D.: Automatic Heap Layout Manipulation for Exploitation. 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 763–779, https://www.usenix.org/conference/usenixsecurity18/presentation/heelan.

[10] Lu, K.—Walter, M. T.—Pfaff, D.—Nürnberger, S.—Lee, W.—Backes, M.: Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. 2017, doi: 10.60882/cispa.25434535.

[11] Wu, W.—Chen, Y.—Xu, J.—Xing, X.—Gong, X.—Zou, W.: FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 781–797, https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei.

[12] Wu, W.—Chen, Y.—Xing, X.—Zou, W.: KEPLER: Facilitating Control-Flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 1187–1204, https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei.

[13] Chen, Y.—Xing, X.: SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), 2019, pp. 1707–1722, doi: 10.1145/3319535.3363212.

[14] Chen, W.—Zou, X.—Li, G.—Qian, Z.: KOOBE: Towards Facilitating Exploit Generation of Kernel Out-of-Bounds Write Vulnerabilities. 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1093–1110, https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng.

[15] Li, R.—Zhang, B.—Chen, J.—Lin, W.—Feng, C.—Tang, C.: Towards Automatic and Precise Heap Layout Manipulation for General-Purpose Programs. Network and Distributed System Security Symposium (NDSS 2023), 2023, doi: 10.14722/ndss.2023.23232.

[16] Wang, Y.—Zhang, C.—Zhao, Z.—Zhang, B.—Gong, X.—Zou, W.: MAZE: Towards Automated Heap Feng Shui. 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1647–1664, https://www.usenix.org/conference/usenixsecurity21/presentation/wang-yan.

[17] Azad, B.: A Survey of Recent iOS Kernel Exploits. Project Zero, Google, https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html.

[18] Zeng, K.—Lin, Z.—Lu, K.—Xing, X.—Wang, R.—Doupé, A.—Shoshitaishvili, Y.—Bao, T.: RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), 2023, pp. 3093–3107, doi: 10.1145/3576915.3623220.

[19] Rostedt, S.: Ftrace Linux Kernel Tracing. Linuxcon Japan, 2010, https://events.static.linuxfound.org/slides/2010/linuxcon_japan/linuxcon_jp2010_rostedt.pdf.

[20] Zhou, J.—Pan, Z.—Hu, J.—Zhu, J.—Shen, W.—Chang, R.—Li, G.—Qian, Z.: Beyond Control: Exploring Novel File System Objects for Data-Only

Attacks on Linux Systems. IEEE Transactions on Dependable and Secure Computing, Vol. 23, 2026, No. 3, pp. 7242–7258, doi: 10.1109/TDSC.2026.3672847.

[21] Lee, Y.—Min, C.—Lee, B.: ExpRace: Exploiting Kernel Races Through Raising Interrupts. 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 2363–2380, https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yoochan.

[22] Lee, Y.—Kwak, J.—Kang, J.—Jeon, Y.—Lee, B.: Pspray: Timing Side-Channel Based Linux Kernel Heap Exploitation Technique. 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 6825–6842, https://www.usenix.org/conference/usenixsecurity23/presentation/lee-yoochan.

[23] Jiang, Z.—Zhang, Y.—Xu, J.—Sun, X.—Liu, Z.—Yang, M.: AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. 2023 IEEE Symposium on Security and Privacy (SP), 2023, pp. 2122–2137, doi: 10.1109/SP46215.2023.10179286.

[24] Zou, X.—Hao, Y.—Zhang, Z.—Pu, J.—Chen, W.—Qian, Z.: SyzBridge: Bridging the Gap in Exploitability Assessment of Linux Kernel Bugs in the Linux Ecosystem. Network and Distributed System Security Symposium (NDSS 2024), 2024, doi: 10.14722/ndss.2024.24926.

[25] Zou, X.—Li, G.—Chen, W.—Zhang, H.—Qian, Z.: SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux Kernel. 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 3201–3217, https://www.usenix.org/conference/usenixsecurity22/presentation/zou.

[26] Lin, Z.—Chen, Y.—Wu, Y.—Mu, D.—Yu, C.—Xing, X.—Li, K.: GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. 2022 IEEE Symposium on Security and Privacy (SP), 2022, pp. 2078–2095, doi: 10.1109/SP46214.2022.9833683.

[27] Wu, Y.—Lin, Z.—Chen, Y.—Le, D. K.—Mu, D.—Xing, X.: Mitigating Security Risks in Linux with KLAUS: A Method for Evaluating Patch Correctness. 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 4247–4264, https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yuhang.

[28] Lin, Z.—Wu, Y.—Xing, X.: DirtyCred: Escalating Privilege in Linux Kernel. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), 2022, pp. 1963–1976, doi: 10.1145/3548606.3560585.

[29] Song, J.—Alves-Foss, J.: The DARPA Cyber Grand Challenge: A Competitor's Perspective. IEEE Security & Privacy, Vol. 13, 2015, No. 6, pp. 72–76, doi: 10.1109/MSP.2015.132.

[30] Wang, F.—Shoshitaishvili, Y.: Angr – The Next Generation of Binary Analysis. 2017 IEEE Cybersecurity Development (SecDev), 2017, pp. 8–9, doi: 10.1109/SecDev.2017.14.

[31] Yun, I.—Kapil, D.—Kim, T.: Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1111–1128, https://www.usenix.org/conference/usenixsecurity20/presentation/yun.

[32] Eckert, M.—Bianchi, A.—Wang, R.—Shoshitaishvili, Y.—Kruegel, C.—Vigna, G.: HeapHopper: Bringing Bounded Model Checking to Heap Im-

plementation Security. 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 99–116, https://www.usenix.org/conference/usenixsecurity18/presentation/eckert.

[33] Jiang, Z.—Gan, S.—Herrera, A.—Toffalini, F.—Romerio, L.—Tang, C.—Egele, M.—Zhang, C.—Payer, M.: Evocatio: Conjuring Bug Capabilities from a Single PoC. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), 2022, pp. 1599–1613, doi: 10.1145/3548606.3560575.

[34] Wang, Y.—Zhang, C.—Xiang, X.—Zhao, Z.—Li, W.—Gong, X.—Liu, B.—Chen, K.—Zou, W.: Revery: From Proof-of-Concept to Exploitable. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), 2018, pp. 1914–1927, doi: 10.1145/3243734.3243847.

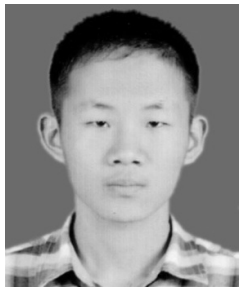
[35] Gennissen, J.—O'Keeffe, D.: Hack the Heap: Heap Layout Manipulation Made Easy. 2022 IEEE Security and Privacy Workshops (SPW), 2022, pp. 289–300, doi: 10.1109/SPW54247.2022.9833896.

[36] Blazytko, T.—Schlögel, M.—Aschermann, C.—Abbasi, A.—Frank, J.—Wörner, S.—Holz, T.: AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 235–252, https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko.

[37] Park, Y.—Lee, H.—Jung, J.—Koo, H.—Kim, H. K.: Benzene: A Practical Root Cause Analysis System with an Under-Constrained State Mutation. 2024 IEEE Symposium on Security and Privacy (SP), 2024, pp. 1865–1883, doi: 10.1109/SP54263.2024.00074.

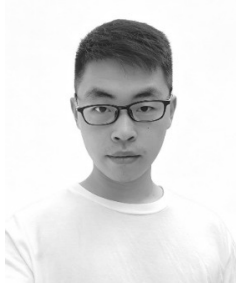
[38] Yagemann, C.—Pruett, M.—Chung, S. P.—Bittick, K.—Saltaformaggio, B.—Lee, W.: ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1989–2006, https://www.usenix.org/conference/usenixsecurity21/presentation/yagemann.

[39] Jiang, Z.—Jiang, X.—Hazimeh, A.—Tang, C.—Zhang, C.—Payer, M.: Igor: Crash Deduplication Through Root-Cause Clustering. Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), 2021, pp. 3318–3336, doi: 10.1145/3460120.3485364.



Xingwei Li received his B.Sc. and M.Sc. degrees in 2018 and 2020, respectively, from the Information Engineering University. His research interests include offensive security techniques.

Yunchao Wang has his Ph.D. and is a lecturer in the Information Engineering University. His research interests include software and system security.



Jianzhou Zhao received his B.Sc. degree from the Information Engineering University, Zhengzhou, China in 2022. He is currently working toward his Ph.D. degree in cyberspace security with the Information Engineering University, Zhengzhou, China. His research interests include kernel security techniques, fuzzing and software analysis.

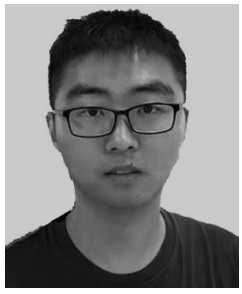
Yunchao Guan is a Master student at the Network Security Lab (NISL), Institute for Network Sciences and Cyberspace, Tsinghua University. His research primarily focuses on software and system security, with a special focus on IoT security and mobile application security.

Danjun Liu received his B.Sc., M.Sc., and Ph.D. in computer science and technology from the National University of Defense Technology. His research interests include operating systems and software security.

Zhe Yang received his B.Sc. degree from the Information Engineering University. He is currently pursuing his M.Sc. degree at the School of Cyber Science and Technology, Beihang University. His research interests include IoT device security analysis and large language model system applications.

Wenbin Zhang is a Ph.D. candidate, whose primary research interest is network protocol security analysis.

Zehui Wu obtained his Ph.D. degree and now serves as Assistant Professor at the Information Engineering University. His research interests include software and system security.



Rongkuan Ma obtained his Ph.D. and currently serves as a lecturer. His main research interests include program analysis and software and Web security.

Xixing Li is a lecturer at the Information Engineering University. He received his Ph.D. degree from the Information Engineering University, Zhengzhou, China, in 2023. He is a System Security Researcher. His current research interests include embedded system security, program analysis, and vulnerability discovery.

Qiang Wei is Professor at the Information Engineering University. His main research interests include software analysis.