CROSS-X: Generalized and Stable Cross-Cache Attack on the Linux Kernel

Paper #1891

ABSTRACT

The cross-cache attack is a fundamental component of modern Linux kernel exploits, spanning real-world attacks and recent research. Despite its importance, it is often regarded as unreliable due to its complex setup, and existing studies lack in-depth analysis of its mechanics. In this paper, we address this gap by: (1) reviewing public strategies and their limitations, (2) proposing two optimized strategies effective in varied conditions, and (3) introducing CROSS-X, an automated system that identifies suitable target objects for cross-cache attacks. We evaluated our strategies on a synthetic vulnerability and nine real-world CVEs, achieving over 99% and 85% success rates under idle and busy workloads, respectively. They also outperformed existing methods in 6 of 8 CVEs under idle workloads and 5 of 8 under busy workloads. For object identification, we define three key properties: (1) spray capability, (2) minimal interference, and (3) useful primitives. Based on these, CROSS-X identified seven versatile target objects and their relationship with interfering allocations. We believe our work will enhance public understanding of cross-cache attacks and contribute to improving Linux kernel security.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; Software security engineering.

KEYWORDS

OS Security, Kernel Exploitation, Cross-Cache Attack

ACM Reference Format:
Paper #1891. 2025. CROSS-X: Generalized and Stable Cross-Cache Attack on the Linux Kernel. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (CCS '25).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

War, war never changes. The same applies to the ongoing battle for memory safety in the Linux kernel, where vulnerabilities [45, 46, 48–50] such as Use-After-Free (UAF), Out-of-Bounds (OOB), and Double Free (DF) continue to be discovered and exploited. In 2016, the kernel introduced cache separation [13, 40], disrupting many exploit techniques. Cache separation moved key objects, previously useful to attackers, into different caches within the SLUB allocator. As a result, attackers faced increased difficulty as they could no longer easily exploit vulnerabilities by spraying specific objects.

In response, attackers devised a new concept called cross-cache attacks, which exploit the internal mechanism of physical page recycling in the page allocator—a subsystem beneath the SLUB allocator. This approach allowed attackers to use nearly *any* object for exploits, effectively bypassing cache separation. Due to their effectiveness, cross-cache attacks quickly attracted researchers' attention and became indispensable building blocks in modern Linux kernel exploits. In addition to their application in numerous exploits [1, 32, 55, 66], recent studies have also developed advanced strategies using privileged objects [33], data pages [21], and page table entries [41, 58] to further leverage cross-cache attacks.

However, cross-cache attacks currently have a significant limitation: their exact working mechanism remains unclear. Public exploits employing cross-cache attacks typically rely on predetermined object counts for spraying and freeing, identified through extensive trial and error. Although Jann Horn proposed a potentially generalizable technique [22], our analysis found it to be redundant and ineffective on half of the object caches following recent kernel updates. Additionally, there is no public guidance on selecting suitable objects for cross-cache attacks. Despite extensive databases of potentially useful objects from prior studies [4–6, 56], our manual inspection revealed many of these objects to be unsuitable for cross-cache attacks. Thus, cross-cache attacks currently have a reputation as one of the least reliable techniques [64, 65]. This perception risks unfairly undervaluing certain vulnerabilities that could be effectively exploited with robust cross-cache attacks, potentially delaying necessary fixes.

To address this issue, we present a *systematization of cross-cache attacks*, *two new strategies with high stability*, and *an automated system*, CROSS-X, designed to identify suitable objects for cross-cache attacks. Upon reviewing public techniques, we found them unsuitable for general use due to limited understanding of SLUB and the need for trial-and-error fine-tuning. We propose two new strategies: a generic approach, Complete Free, and a hybrid approach, SLUBStick + Complete Free, which incorporates the timing side-channel technique from prior work SLUBStick [41]. We conducted extensive experiments using a synthetic vulnerability and nine real-world CVEs to evaluate the effectiveness of these strategies under various conditions, such as object size and workload. The results indicate that our strategies are stable and robust, achieving success rates above 99% in idle conditions and 85% in busy conditions across all caches except one. Furthermore, our strategies outperformed others in 6 of 8 cases under idle conditions and 5 of 8 under busy conditions in real-world experiments.

For object identification, we propose the CROSS-X system, designed to automate finding suitable objects for cross-cache attacks. We define the concepts of *interfering allocations* and *noise*

CCS '25, Taipei, Taiwan

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

rates that hinder object usability and identify three essential properties for target objects: (1) spray capability, (2) minimal interfering allocations, and (3) usefulness for exploitation. Based on these properties, CROSS-X comprises three modules: `ObjectFuzzer`, `ObjectEvaluator`, and `ObjectMatcher`. These modules cooperate to filter unsuitable objects from the candidate set using kernel instrumentation, fuzzing, and symbolic execution. From the 346 objects identified in previous studies and kCTF VRP [17, 18], CROSS-X identified seven objects highly suitable for cross-cache attacks. Additionally, we found that not all interfering allocations negatively impact cross-cache attacks, contrary to common assumptions.

Contributions. This paper makes the following contributions:

- **Comprehensive Systematization.** We provide a comprehensive systematization of cross-cache attacks, including extensive SLUB background, detailed attack mechanics, desired object properties, and existing public strategies with their limitations.
- **Robust Strategies.** We propose two new strategies with higher stability and broader cache coverage than existing methods. Evaluations confirm their reliability and effectiveness across different kernel versions and system noise conditions.
- **Automated System.** We introduce CROSS-X, an automated system to identify versatile objects suitable for cross-cache attacks. CROSS-X filters candidates based on desired object properties using kernel fuzzing, instrumentation, and symbolic execution.
- **Real-world Evaluation.** We evaluate our strategies using synthetic and nine real-world vulnerabilities across various object sizes and workloads (idle and busy states). We also present object identification results and discuss key characteristics.

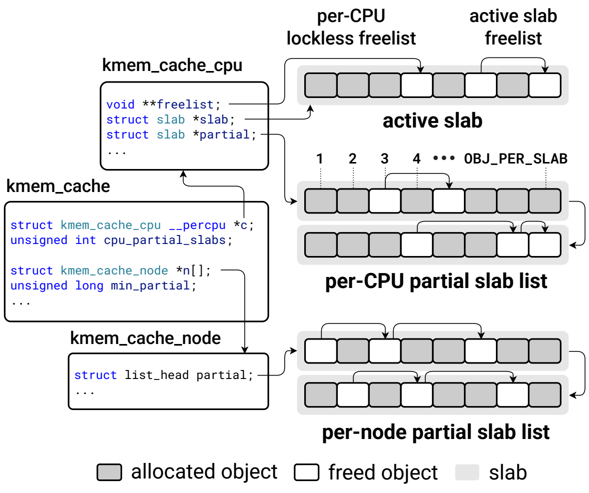
2 BACKGROUND

2.1 Linux Memory Allocator

The Linux kernel manages memory using both the page allocator and the SLUB allocator. The page allocator operates at the page level, supplying physically contiguous pages to various subsystems. The SLUB allocator works at the object level, serving as an intermediate layer to optimize small memory allocations.

2.1.1 Page Allocator. The Linux kernel manages physical pages using two types of allocators: the Buddy allocator and the Per-CPU Pageset (PCP) allocator. The Buddy allocator [27] organizes blocks of pages based on their order. The *order* of a block is an integer n, indicating that the block consists of 2^n contiguous pages. The Buddy allocator maintains freelists of blocks with the same order, allowing it to allocate a block of the required size efficiently. If no suitable block is available, the allocator splits larger blocks or merges smaller ones to meet allocation requests and reduce fragmentation.

The PCP allocator [19] was introduced later to be used alongside the Buddy allocator. As the name suggests, it provides a per-CPU freelist of blocks with small orders. Operating with the PCP allocator does not require global locks like the Buddy allocator, allowing for faster allocation and freeing of pages. Importantly, the freelists in both the Buddy and PCP allocators follow a Last-In-First-Out (LIFO) ordering. This LIFO behavior, unfortunately, benefits attackers by adding predictability to how the kernel allocates and frees pages. Through carefully crafted heap grooming, attackers can manipulate the freelists of the page allocator to insert or release the pages needed for exploitation. (see §2.2)



```
void **freelist;
struct slab *slab;
struct slab *partial;
...
```

```
struct kmem_cache_cpu __percpu *c;
unsigned int cpu_partial_slabs;

struct kmem_cache_node *n[];
unsigned long min_partial;
...
```

```
struct list_head partial;
...
```

Figure 1: Internal representation of the SLUB allocator

2.1.2 SLUB Allocator. The SLUB allocator [29] is a memory management system designed for the efficient allocation of kernel objects. Its core principle is to maintain a pre-allocated cache of objects. It uses multiple object freelists to manage *slabs*, which are blocks of pages split into object slots of the same size. Operating on top of the page allocator, it creates fresh slabs from the page allocator. When slabs are no longer needed, they are discarded and returned to the page allocator.

Object Caches. The allocator distinguishes object caches into two types: *dedicated* caches and *generic* caches. Dedicated caches are custom caches explicitly created by the kernel to serve specific kernel objects that are used frequently (e.g., `task_struct`, `inode`). These caches are usually named after the objects they hold. Generic caches, on the other hand, serve objects that do not have dedicated caches, elastic objects [5] with flexible sizes, and even non-objects such as data buffers. They are named `kmalloc-*`, where `*` denotes the maximum object size the cache can hold (e.g., `kmalloc-16`, `kmalloc-32`)[1].

Several caches can be merged into a single cache when their objects share similar characteristics, such as object size or GFP flags [44]. Conversely, generic caches can be split when specific kernel features are enabled. For example, if the memory accounting feature (`MEMCG_KMEM`) is active, objects subject to accounting are allocated from `kmalloc-cg-*` caches [40] instead of the original `kmalloc-*` caches. From an attacker's perspective, this separation can disrupt an exploit if the targeted cache is unintentionally split.

Design and Representation. Inside the SLUB allocator, objects within a slab are linked in a freelist, meaning each slab has its own freelist. Slabs can also be linked together to form a slab list. A slab fully occupied with allocated objects is a full slab, while one without

[1] Generic caches for objects larger than 1,000 bytes have been renamed—for example, `kmalloc-1024` is now `kmalloc-1k`. For brevity, we continue to use the previous names.

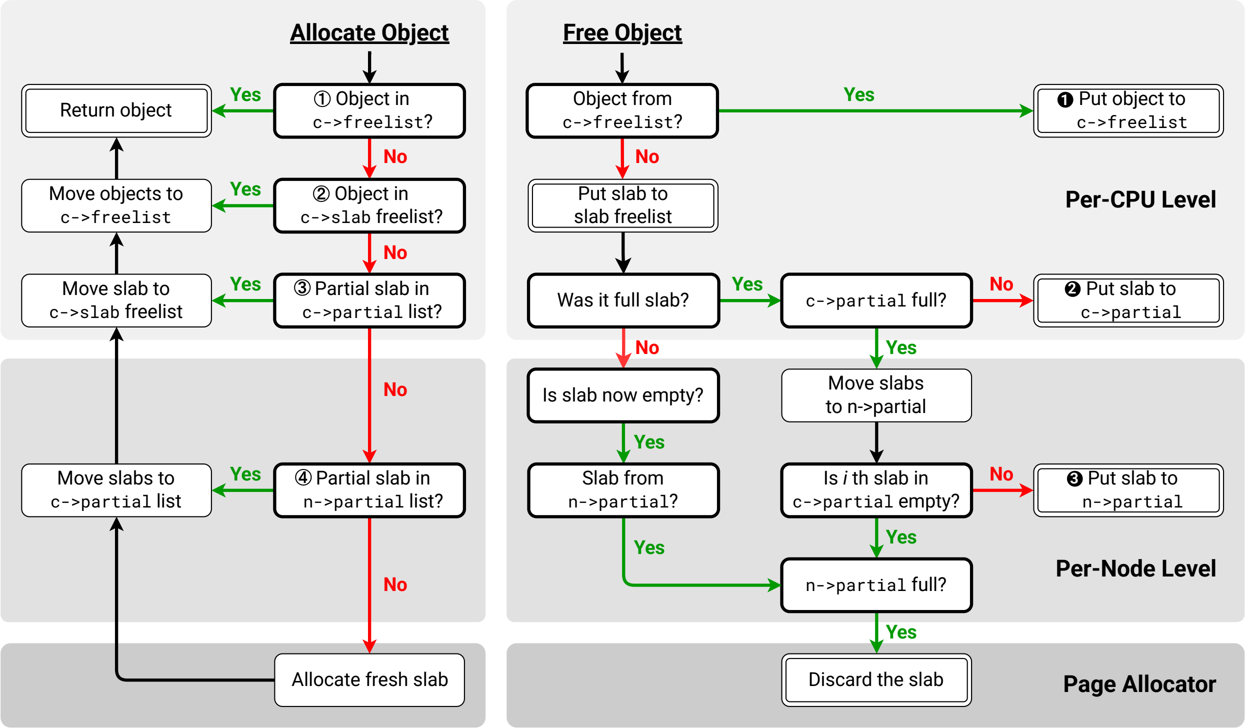


Figure 2: Allocation and deallocation paths of the SLUB allocator. Nodes with thick borders indicate decision points, while double-bordered nodes represent terminal states where the procedure completes. Execution time increases as the process traverses downward in the diagram.

any allocated objects is empty. A slab containing both allocated and free objects is called a *partial* slab. The SLUB allocator primarily manages partial slabs, removing them from the list once they become full. Additionally, the slab currently used for allocations is known as the *active* slab.

Figure 1 illustrates the SLUB allocator's internal representation. The `kmem_cache` structure represents an object cache with a two-level mechanism: `kmem_cache_cpu` and `kmem_cache_node`. The `kmem_cache_cpu` structure manages the cache state for each CPU, maintaining the active slab (`c->slab`) and a per-CPU partial slab list (`c->partial`). It also includes a per-CPU lockless freelist (`c->freelist`) within the active slab. The `kmem_cache_node` structure manages the cache state for each NUMA node, maintaining a per-node partial slab list (`n->partial`) that supplies slabs to all CPUs in the node. Lengths of the per-CPU and per-node partial slab lists are regulated by `cpu_partial`[2] and `min_partial`, respectively, and are considered full when exceeding these thresholds. Finally, we denote the number of objects in a slab as `OBJ_PER_SLAB`. `OBJ_PER_SLAB` depends on object size, as the object size determines the slab size (i.e., how many pages constitute a slab).

Allocation and Deallocation. Figure 2 illustrates a simplified allocation and deallocation process of the SLUB allocator. During allocation, the allocator begins by traversing the per-CPU freelist ① and the active slab freelist ②, searching for an available object in the active slab. If none are found, it checks the partial slabs to promote one to the active slab, searching the per-CPU partial list ③ and the per-node partial list ④. When no suitable slabs are available, the allocator requests a page to create a new slab. The newly acquired slab then becomes the active slab.

Upon deallocation, the object is placed into the per-CPU lockless freelist ❶ if it was allocated from there. Otherwise, it is added to its slab freelist, and the slab is moved to the per-CPU partial slab list ❷. If this list is full, the SLUB allocator migrates slabs to the per-node partial slab list ❸. If that list also becomes full during migration, the slab being processed is discarded to the page allocator. Additionally, a slab is discarded if deallocating an object makes it empty and it resides in a full per-node partial slab list.

2.2 Cross-Cache Attacks

As described in §2.1, the SLUB allocator discards slabs to and allocates fresh slabs from the page allocator, enabling internal slab reuse. Cross-cache attacks are sophisticated exploit strategies that take advantage of this behavior, as illustrated in Figure 3. These attacks consist of two phases: *recycling* and *reclaiming*. In the recycling phase, the attacker forces the SLUB allocator to discard the slab containing a vulnerable object to the page allocator. In the reclaiming phase, the attacker reuses that slab by allocating a completely different object from another cache.

[2] This parameter has since been renamed to `cpu_partial_slabs` in recent kernel updates. For consistency, we continue to refer to it by its legacy name.

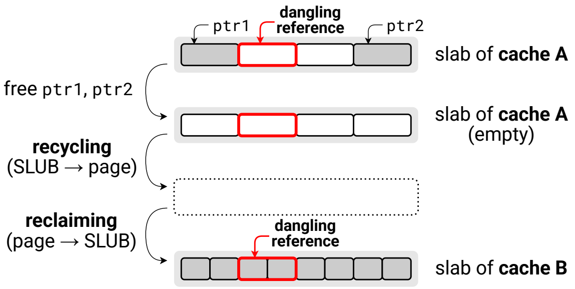


Figure 3: Cross-cache attack exploiting a UAF vulnerability. The object with a red border is the vulnerable object with a dangling reference. The attacker first frees `ptr1` and `ptr2` to empty the slab of cache A, then triggers *recycling* to discard it. Next, the attacker allocates multiple objects in cache B to *reclaim* the slab. The dangling reference remains.

Cross-cache attacks are widely used in modern exploits [1, 32, 55, 66] due to their ability to bypass cache separation. Recent research on exploit strategies has also leveraged these attacks to reclaim slabs as privileged objects [33], data pages [21], and page table entries [41, 58]. However, in-depth analysis of how cross-cache attacks work remains limited, making the attack procedure hypothetical and obscure. Public exploits often rely on incorrect information or heuristic strategies regarding the SLUB allocator, reducing the reliability and stability of these attacks (see §3.2).

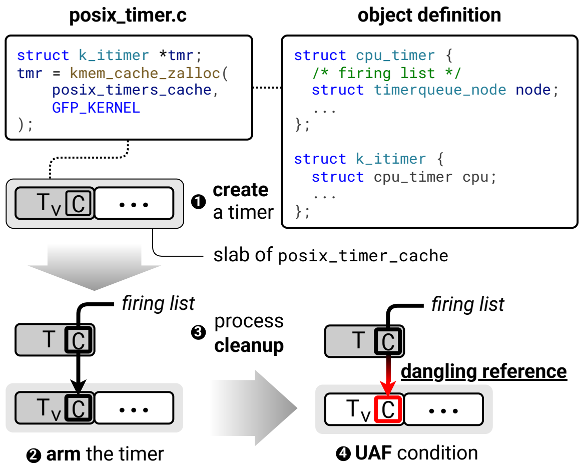
Terminologies. We define an object in which the vulnerability occurs as a *vulnerable object*. The slab containing this object, which is the target of recycling, is the *vulnerable slab*. The cache to which the vulnerable slab belongs is referred to as the *vulnerable cache*. Objects used to reclaim the vulnerable slab are called *target objects*, while their corresponding cache is termed the *target cache*.

2.3 Threat Model

We assume an attacker with unprivileged code execution and a kernel UAF vulnerability (or another vulnerability such as OOB or DF pivoted to UAF [33, 41]). Mitigations such as SMEP [26], SMAP [9], KPTI [10], and kCFI [2, 15] are enabled, following prior works [21, 56]. We also consider SLUB-targeted defenses such as object freelist protections [7, 14], randomized caches (`RANDOM_KMALLOC_CACHES`) [11], and the bucket allocator (`SLAB_BUCKETS`) [8]. Some of these protections are present in major distributions and significantly hinder exploits within the same cache, necessitating cross-cache attacks. However, they do not affect the feasibility of cross-cache attacks (see §7). The analyzed kernel versions are three LTS releases: v5.15, v6.1, and v6.6, as `cpu_partial` values differ starting from v6.1, and v6.6 introduces `RANDOM_KMALLOC_CACHES`. Finally, we do not consider microarchitectural vulnerabilities [20, 24, 37].

3 TECHNICAL OVERVIEW & CHALLENGES

This section describes how we successfully exploited CVE-2022-2585 using the cross-cache attack, which serves as the motivational example for our work. We then discuss two technical challenges in



posix_timer.c

```
struct k_itimer *tmr;
tmr = kmem_cache_zalloc(
    posix_timers_cache,
    GFP_KERNEL
);
```

object definition

```
struct cpu_timer {
  /* firing list */
  struct timerqueue_node node;
  ...
};

struct k_itimer {
  struct cpu_timer cpu;
  ...
};
```

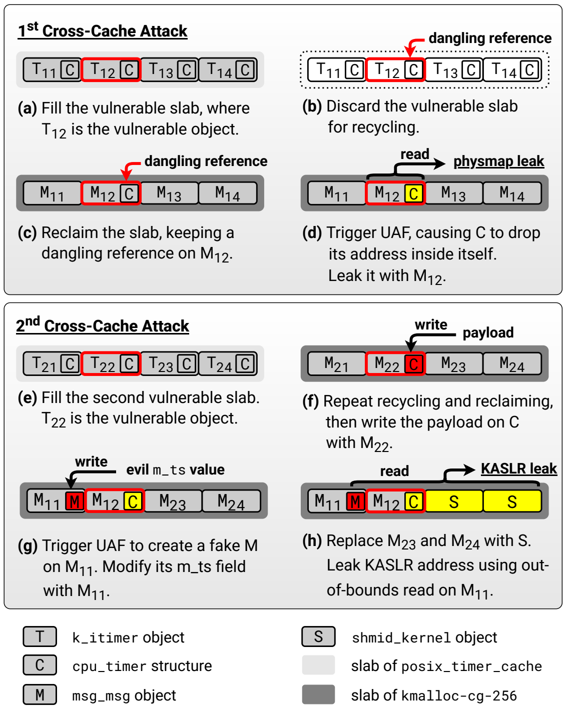
Figure 4: Step-by-step example of triggering CVE-2022-2585. T denotes a `k_itimer` object, and C a `cpu_timer` structure. T_v is the vulnerable `k_itimer` object.

executing a successful cross-cache attack and explain the limitations of public exploits in addressing them effectively.

3.1 Motivational Example

CVE-2022-2585 [47], illustrated in Figure 4, represents a real-world UAF vulnerability previously considered not exploitable using object-based techniques, thus requiring page spraying [21]. When a user creates a CPU timer, a `k_itimer` object containing a `cpu_timer` is allocated in the dedicated cache called `posix_timers_cache` ❶. The user can then arm the timer, inserting the `cpu_timer` into the *firing list* ❷. This list should be cleared when the process is cleaned up, freeing all associated timers ❸. However, due to a bug, freed timers remained on the list if a non-leader thread called `execve`, resulting in a UAF condition ❹. The vulnerability could then be triggered by invoking `exit` to remove the `cpu_timer` from the list or by waiting for the timer to fire at the configured interval.

As shown in Figure 5, our exploit achieves privilege escalation through two consecutive cross-cache attacks. We selected `msg_msg` from the `kmalloc-cg-256` cache as the target object because it provides powerful OOB read capabilities [35], achievable by corrupting its `m_ts` field, which indicates message length. First, we trigger a UAF by exiting the thread that allocated the vulnerable object, causing `cpu_timer` to write its own address during the list deletion logic. Next, we leak this address via `msg_msg` to obtain a physmap address (Figure 5a~5d). Using this leaked address, we craft a payload and trigger the UAF again, creating a fake `msg_msg` object that leads to a KASLR leak using `shmid_kernel` objects (Figure 5e~5h). Finally, we perform an unlinking attack [39] to corrupt `modprobe_path`, achieving privilege escalation. For interested readers, we have published our end-to-end exploit online [12].



(a) Fill the vulnerable slab, where T_{12} is the vulnerable object.

(b) Discard the vulnerable slab for recycling.

(c) Reclaim the slab, keeping a dangling reference on M_{12}.

(d) Trigger UAF, causing C to drop its address inside itself. Leak it with M_{12}.

(e) Fill the second vulnerable slab. T_{22} is the vulnerable object.

(f) Repeat recycling and reclaiming, then write the payload on C with M_{22}.

(g) Trigger UAF to create a fake M on M_{11}. Modify its m_ts field with M_{11}.

(h) Replace M_{23} and M_{24} with S. Leak KASLR address using out-of-bounds read on M_{11}.

Figure 5: Overview of exploiting CVE-2022-2585. The subscript x and y (e.g., T_{xy}) denotes an object in the y-th slot of the x-th vulnerable slab. Yellow objects indicate leak sources, while red objects represent deliberately written malicious payloads. The unlinking attack, which follows step (h), is omitted for simplicity.

3.2 Technical Challenges

The example in Section §3.1 demonstrates the effectiveness of a cross-cache attack, enabling an end-to-end exploit from a single UAF within a dedicated cache. In many real-world exploits like this, an attacker must ensure that the target object overlaps with the vulnerable one to leverage key primitives such as address leakage and unlinking attacks. Consequently, the exploit's reliability heavily depends on successful object recycling and reclaiming. However, one crucial question remains: is this process as straightforward as it seems? Multiple cycles of recycling and reclaiming are often necessary, as shown in this example, and the instability of these phases can significantly impact reliability. To address this challenge, we explore two key aspects: *reliable recycling of the vulnerable slab* and *identifying suitable target objects*.

Reliable Recycling. To trigger the recycling of a vulnerable slab, we must reach the code path that discards it in the SLUB allocator. Previous research and public exploits have tackled this problem, proposing various methods. However, we find that these methods are not always effective and apply only in specific scenarios. For instance, Guo et al. [21] suggest that freeing all objects in the vulnerable slab will lead to its recycling. However, as shown in Figure 2, this approach often fails when the per-CPU or per-node partial slab list is not full. In Section §4, we analyze public exploits to understand how they attempt to achieve recycling using different approaches. Based on these insights, we propose an improved strategy that enhances reliability across diverse scenarios.

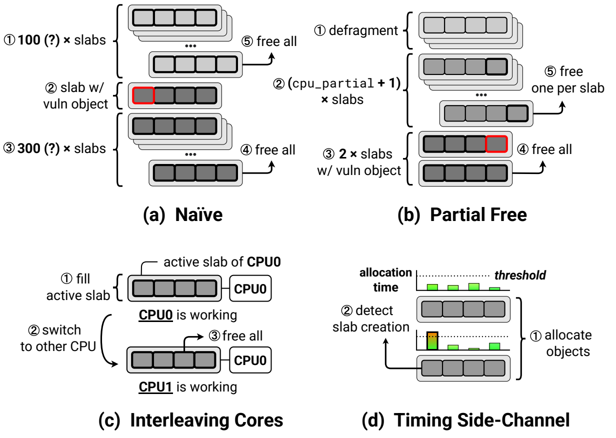


Figure 6: Strategies used in public exploits to achieve recycling of the vulnerable slab. Objects with the bold border indicates they are freed after allocation.

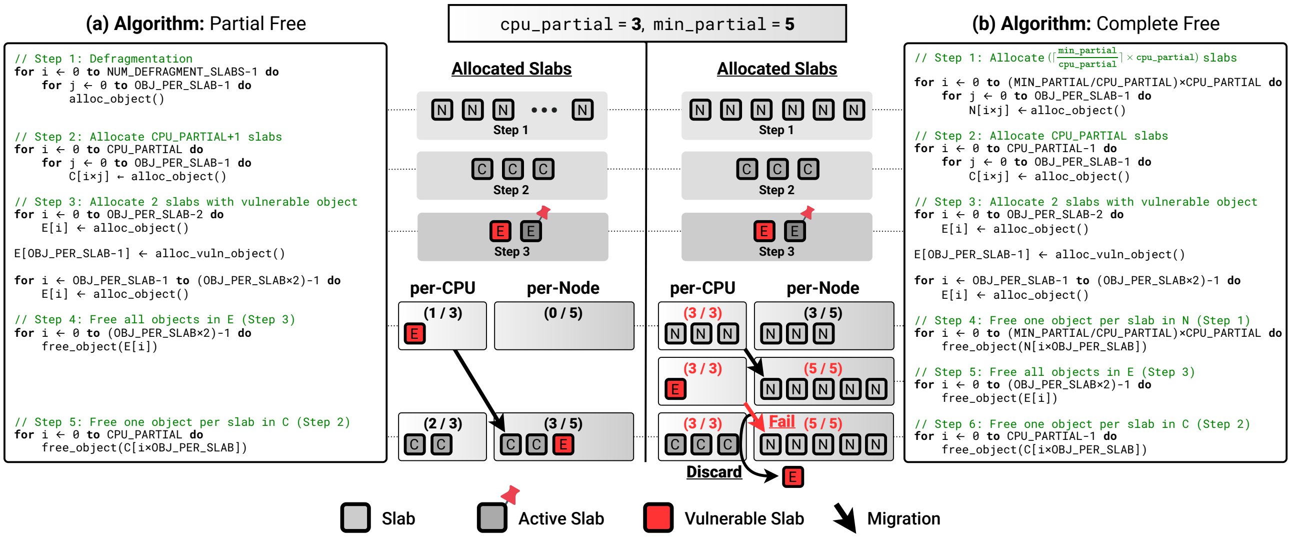
Identifying Suitable Objects. Selecting the right object is crucial, as objects provide essential primitives such as address leaks and memory writes. The security community has extensively studied this topic [4–6, 56], discovering useful objects across various caches and curating an online database [54]. However, our manual review of these objects revealed that many are unsuitable for cross-cache attacks. For instance, some objects require privileges for allocation (e.g., `kexec_segment`), are freed immediately after allocation (e.g., `poll_list`, `dentry`), or cannot be allocated multiple times (e.g., `ctl_table`), making them impractical for reclaiming. In Section §5, we outline the desired characteristics of target objects. We then introduce our automated system, which systematically selects suitable objects from a pool of candidates and explains its implementation.

4 ACHIEVING RELIABLE RECYCLING

4.1 Public Exploits Review

Public exploits involving cross-cache attacks attempt to recycle the vulnerable slab using various methods. To understand how these exploits work and their limitations, we reviewed public write-ups and studies on recent cross-cache attacks. Specifically, we examined four public write-ups [1, 22, 52, 55] and three studies [30, 41, 57] published since 2020, where cross-cache attacks are central to the exploit and clearly demonstrate each strategy. We then grouped these methods into four strategies. Figure 6 illustrates how each strategy works.

Naïve. The Naïve strategy [55] allocates many objects around the vulnerable one, hoping that releasing them leads to recycling. The number of these objects is arbitrary and often determined through trial and error. This makes general usage difficult. Additionally, the Naïve strategy may fail if migration of per-CPU partial slabs occurs



(a) Algorithm: Partial Free

```
// Step 1: Defragmentation
for i ← 0 to NUM_DEFRAGMENT_SLABS-1 do
    for j ← 0 to OBJ_PER_SLAB-1 do
        alloc_object()

// Step 2: Allocate CPU_PARTIAL+1 slabs
for i ← 0 to CPU_PARTIAL do
    for j ← 0 to OBJ_PER_SLAB-1 do
        C[i×j] ← alloc_object()

// Step 3: Allocate 2 slabs with vulnerable object
for i ← 0 to OBJ_PER_SLAB-2 do
    E[i] ← alloc_object()

E[OBJ_PER_SLAB-1] ← alloc_vuln_object()

for i ← OBJ_PER_SLAB-1 to (OBJ_PER_SLAB×2)-1 do
    E[i] ← alloc_object()

// Step 4: Free all objects in E (Step 3)
for i ← 0 to (OBJ_PER_SLAB×2)-1 do
    free_object(E[i])

// Step 5: Free one object per slab in C (Step 2)
for i ← 0 to CPU_PARTIAL do
    free_object(C[i×OBJ_PER_SLAB])
```

(b) Algorithm: Complete Free

```
// Step 1: Allocate (⌈min_partial / cpu_partial⌉ × cpu_partial) slabs

for i ← 0 to (MIN_PARTIAL/CPU_PARTIAL)×CPU_PARTIAL do
    for j ← 0 to OBJ_PER_SLAB-1 do
        N[i×j] ← alloc_object()

// Step 2: Allocate CPU_PARTIAL slabs
for i ← 0 to CPU_PARTIAL-1 do
    for j ← 0 to OBJ_PER_SLAB-1 do
        C[i×j] ← alloc_object()

// Step 3: Allocate 2 slabs with vulnerable object
for i ← 0 to OBJ_PER_SLAB-2 do
    E[i] ← alloc_object()

E[OBJ_PER_SLAB-1] ← alloc_vuln_object()

for i ← OBJ_PER_SLAB-1 to (OBJ_PER_SLAB×2)-1 do
    E[i] ← alloc_object()

// Step 4: Free one object per slab in N (Step 1)
for i ← 0 to (MIN_PARTIAL/CPU_PARTIAL)×CPU_PARTIAL do
    free_object(N[i×OBJ_PER_SLAB])

// Step 5: Free all objects in E (Step 3)
for i ← 0 to (OBJ_PER_SLAB×2)-1 do
    free_object(E[i])

// Step 6: Free one object per slab in C (Step 2)
for i ← 0 to CPU_PARTIAL-1 do
    free_object(C[i×OBJ_PER_SLAB])
```

Figure 7: Algorithm of (a) Partial Free and (b) Complete Free, and their behavior when (`cpu_partial`, `min_partial`) is (3, 5). `per-CPU` and `per-Node` represent the per-CPU and per-node partial slab lists, with red indicating full capacity. Symbols `N`, `C`, and `E` inside slabs denote the object arrays used in each algorithm. Only Complete Free successfully discards the vulnerable slab.

immediately before freeing the vulnerable object. Such situations arise due to incorrect calculations.

Partial Free. The Partial Free strategy in the CVE-2020-29661 exploit [22] intentionally triggers *migration* of slabs to achieve deterministic recycling. Here, migration refers to moving slabs from the per-CPU to the per-node partial slab list. This step is critical, as overflowing the per-node partial slab list allows the vulnerable slab to be discarded upon its migration. To accomplish this, Partial Free first performs defragmentation ①, allocates (`cpu_partial+1`) slabs ②, and then allocates two additional slabs, placing the vulnerable object between them ③. Next, it frees all objects allocated in step ③, placing the vulnerable slab into the per-CPU partial slab list ④. It then frees only one object per slab from step ②, adding more slabs to the list. Since the number of slabs allocated in step ② exceeds `cpu_partial`, migration eventually occurs. When the per-node partial slab list overflows as expected, the discard of the vulnerable slab succeeds.

Partial Free is the first strategy to use a generalisable formula. It has also been adopted in other exploits [1, 52], albeit with heuristic modifications to the formula (such as allocating `2×(cpu_partial+1)` slabs) due to its failure on recent kernels. We identified the root cause of this issue and propose the improved strategy, Complete Free, which operates with higher stability on newer kernels. We discuss this in Section §4.2.

Interleaving Cores. This strategy is a variant of Partial Free and offers an alternative way to place slabs into the per-CPU partial slab list. It first fills the active slab of the current CPU to produce a full slab ①, then uses CPU pinning to switch to another core ②. When all objects allocated in step ① are freed, the slab moves to the per-CPU partial slab list of the switched CPU ③. An attacker can repeat these steps as necessary. The Interleaving Cores strategy is effective when the maximum number of objects allocated at once is limited, such as in systems with restricted resources [57].

Timing Side-Channel. Pspray [30] introduced the concept of a SLUB timing side-channel, where allocations that trigger slab creation take longer. SLUBStick [41] extended this into a precise measuring technique by allocating objects with deliberately incorrect arguments, causing them to be immediately freed within the same system call and isolating SLUB-relevant timing. Using this primitive, SLUBStick groups objects across a predetermined number of slabs and frees them to trigger recycling. However, this measurement depends on two preconditions, though implicitly stated in the original paper. First, the object must support *immediate free*; while SLUBStick uses the `add_key` syscall in generic caches, not all objects in dedicated caches follow this pattern. Second, the object must not be *deferred freed*, such as via RCU or a separate thread, since SLUBStick assumes the free operation is fast enough to ignore. When this assumption is violated, the timing becomes unreliable. For these reasons, we applied SLUBStick selectively to augment the Complete Free strategy only when both conditions were met, resulting in meaningful improvements in our stability experiments.

4.2 Our Approach: Complete Free

Partial Free was expected to be generally applicable, but it consistently failed for six generic caches in recent kernels, as demonstrated in the stability experiment (§6.1). These caches showed deterministic failures (0% success) for recycling. Through careful reasoning, we identified the root cause as the *reversal* of the size relationship between the `cpu_partial` and `min_partial` values, as shown in Table 1. From v5.15 to v6.1, almost half of the generic caches began to have relatively smaller `cpu_partial` values. This phenomenon was clearly uncommon in earlier versions,

Table 1: Comparison of `cpu_partial` and `min_partial` values in object caches for Linux kernel versions 5.15 and 6.1. Caches where `cpu_partial` fell below `min_partial` in v6.1 are marked †, and those below in both versions are marked ★.

Object Cache	`cpu_partial` vs. `min_partial` (v5.15 → v6.1)
`kmalloc-16` †	(30 > 5) → **(1 < 5)**
`kmalloc-32` †	(30 > 5) → **(2 < 5)**
`kmalloc-64` †	(30 > 5) → **(4 < 5)**
`kmalloc-96`	(30 > 5) → (6 > 5)
`kmalloc-128`	(30 > 5) → (8 > 5)
`kmalloc-192`	(30 > 5) → (12 > 5)
`kmalloc-256`	(13 > 5) → (7 > 5)
`kmalloc-512`	(13 > 5) → (7 > 5)
`kmalloc-1024` †	(6 > 5) → **(3 < 5)**
`kmalloc-2048` †	(6 > 5) → **(3 < 5)**
`kmalloc-4096` ★	**(2 < 6)** → **(2 < 6)**

where it was not applied to any caches except `kmalloc-4096`. Why does this reversal affect Partial Free? Take Figure 7a, for example, which illustrates the Partial Free algorithm and its behavior when (`cpu_partial`, `min_partial`) is (3, 5)—the same setup used for `kmalloc-1024` and `kmalloc-2048` after v6.1. Since `min_partial` is larger than `cpu_partial`, Partial Free cannot overflow the per-node partial slab list (Step 5), leaving the vulnerable object inside the list rather than discarding it.

Based on these observations, we propose the *Complete Free* strategy, an improved version of Partial Free that addresses these reversal cases with higher stability. Additionally, we introduce a hybrid strategy SLUBStick + Complete Free that combines the SLUBStick timing side-channel with Complete Free to further improve the recycling success rate.

Complete Free. The Complete Free strategy in Figure 7b adds two steps to Partial Free to ensure slab discard even in reversal scenarios. First, Complete Free populates ($\left\lceil \frac{\texttt{min\_partial}}{\texttt{cpu\_partial}} \right\rceil \times \texttt{cpu\_partial}$) slabs at the beginning (Step 1). This step serves a dual purpose: defragmentation to avoid dependence on specific SLUB states, and ensuring per-node slab list overflow later. Next, before freeing the vulnerable slab (Step 5), one object is freed per slab allocated in Step 1 (`N` slabs in Figure 7b). At this point, the per-CPU partial slab list overflows in reversal cases (`cpu_partial < min_partial`), triggering migration (Step 4) to reserve space. The vulnerable slab then lands in the per-CPU partial slab list upon release (Step 5). Finally, in Step 6, the per-CPU partial slab list overflows again (by `C` slabs in Figure 7b), resulting in migration of the vulnerable slab. Since the per-node partial slab list is already full, the migration ultimately fails, forcing the vulnerable slab to be successfully discarded.

To the best of our knowledge, Complete Free is the first strategy that is generalizable and adaptable to all SLUB caches. Although it relies on fixed kernel parameters such as `cpu_partial` and `min_partial`, this information is publicly available and can be easily obtained (e.g., by reading the kernel source or `/proc/slabinfo`). We also evaluated its stability under various conditions (synthetic/real-world vulnerabilities, different kernel versions, system noise states) in Sections §6.1 and §6.3. In both experiments, Complete Free consistently worked for all caches, unlike other methods, and demonstrated stable success rates under different conditions.

SLUBStick + Complete Free. We also present a hybrid strategy, SLUBStick + Complete Free, which combines the Complete Free strategy with SLUBStick's timing side channel. In this hybrid approach, we apply the measurement technique from Section §4.1 during slab allocations (Steps 1–3 in Figure 7b). This helps filter out unexpected allocations in the vulnerable cache, which can be detected when the `OBJ_PER_SLAB`-th object allocation does not result in an observable timing gap. Compared to the original SLUBStick artifacts, where the exploit uses a fine-tuned "allocate all, and free all" Naive strategy, SLUBStick + Complete Free can also enhance portability for SLUBStick attacks. We evaluated SLUBStick + Complete Free under conditions where SLUBStick's two measurement requirements (immediate free possible, no deferred free) are met, and it achieved even higher success rates in certain scenarios.

5 AUTOMATED OBJECT IDENTIFICATION

5.1 Desired Properties of Target Objects

Selecting an appropriate target object for cross-cache attacks is challenging, as discussed in Section §3.2. Not all interesting objects identified in previous works are suitable for various reasons. What then qualifies as a versatile target object? We categorized three desired properties of kernel objects that make them suitable targets for cross-cache attacks:

(1) **Spray Capability.** The object must be allocatable multiple times from userland; in other words, it should have the capability to be sprayed. This property is necessary during the reclaiming phase, where the object must be sprayed to occupy the vulnerable slab rather than allocated just once or twice.

(2) **Minimal interfering Allocations.** The object should minimize allocations of unrelated objects, as these might interfere with slab reclamation. We refer to such unintended allocations as *interfering* allocations. Additionally, we introduce the concept of *noise rates* to measure these interfering allocations.

(3) **Useful Primitives.** The object should provide useful primitives for exploit writers, such as delivering payloads to the kernel or leaking addresses. The specific primitives required may vary depending on the targeted vulnerability, but understanding what primitives each object provides is important.

Unfortunately, manually testing all candidate objects against these properties would require significant effort. Therefore, we developed CROSS-X, an automated system designed to perform end-to-end target object identification.

5.2 System Overview

From a high-level perspective, as shown in Figure 8, the CROSS-X system takes a set of candidate objects as input. These objects pass through three modules that filter out unsuitable candidates and select versatile target objects. The modules are `ObjectFuzzer`, `ObjectEvaluator`, and `ObjectMatcher`; each assess one of the characteristics described above. The `ObjectFuzzer` uses kernel instrumentation and fuzzing to verify if an object can be allocated. Then, the `ObjectEvaluator` checks whether the allocated objects actually appear in memory and measures the number of interfering allocations to assess their practical usability. Finally, the `ObjectMatcher` applies symbolic execution to identify useful primitives among the

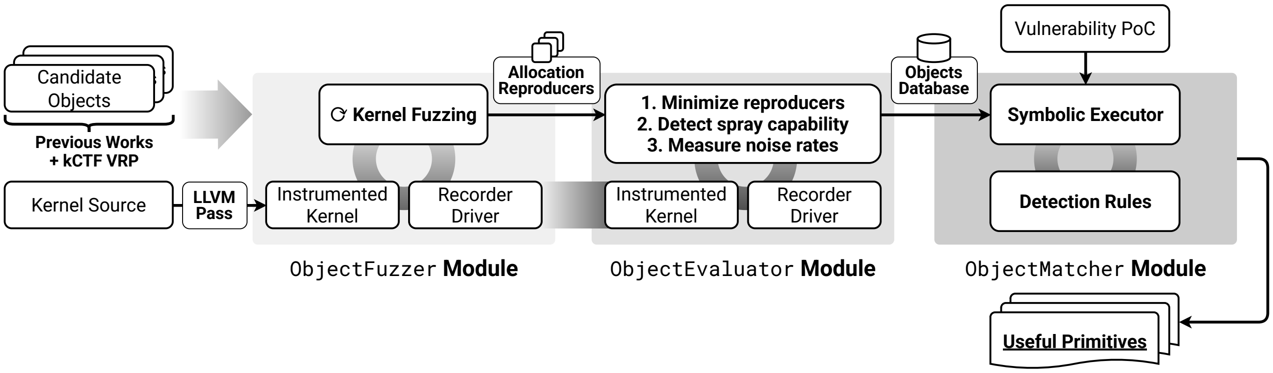


Figure 8: Executive workflow of the CROSS-X system.

remaining objects. The detailed design and implementation of these modules are described in the following section.

5.3 Design and Implementation

5.3.1 Fuzzing Allocation Reproducers. The `ObjectFuzzer` identifies kernel objects that can be allocated from userland and produces *allocation reproducers*, which are programs confirmed to allocate these objects. This process involves three components: an LLVM pass for kernel instrumentation, a recorder driver to detect allocations at runtime, and Syzkaller [16] for reproducer generation.

`ObjectFuzzer` runs as follows: First, the LLVM pass instruments candidate object allocation sites by inserting calls to the recorder driver whenever pointers returned by allocation functions (e.g., `kmalloc`, `kmem_cache_alloc`) are cast to typed pointers using `BitCastInst`. Syzkaller then runs alongside the recorder driver, which monitors object deallocation by hooking functions such as `kfree` and `kmem_cache_free`. After each fuzzing session, the `ObjectFuzzer` checks for objects still allocated. If objects remain allocated, the module deliberately crashes the kernel to collect the corresponding allocation reproducer for further analysis.

5.3.2 Versatility Evaluation. The `ObjectEvaluator` assesses the versatility of objects confirmed as allocatable by the `ObjectFuzzer`. Using allocation reproducers with the instrumented kernel and recorder driver, it performs three tasks:

- **Reproducer Minimization.** For each allocation reproducer, the `ObjectEvaluator` minimizes the system call sequence by removing unrelated calls. To filter these calls, `ObjectEvaluator` first wraps each `syscall` function invocation in a wrapper that encodes the system call's position inside templates. It then applies delta-debugging: each system call is disabled one at a time using string replacements, and the module checks whether the target object is still allocated. If the object allocation succeeds without a particular system call, that call is removed from the reproducer.
- **Spray Capability Detection.** After minimization, the `ObjectEvaluator` again utilizes the recorder driver to detect the spray capability of each object. The goal of this step is to identify objects that genuinely remain in memory, filtering out false positives (e.g., objects quickly freed after deferred operations but mistakenly marked allocatable). Each reproducer is executed repeatedly in a loop, counting the number of allocations and frees. If the allocation count exceeds the free count by a certain threshold, the object is considered to have spray capability.

Table 2: Detection rules for identifying potentially useful primitives. Superscripted rules request additional assumptions about the traits of the targeted vulnerability.

Results	Potential Primitives
Symbolic value write on a function pointer	Control Flow Hijacking
Symbolic value write on a data pointer	Arbitrary Read & Write[1]
Kernel address write on user-provided data	Kernel Address Leak
Memory write on lethal data	Use-After-Free & Out-of-bounds Write[2]
Memory write on a data pointer	Invalid Free[3]

[1] Assumes the pointer is a data buffer readable and writable through the object.
[2] Assumes the reference count and buffer length are compromised, respectively.
[3] Assumes the pointer references an object that is also freed upon release.

- **Noise Rate Measurement.** For objects identified as sprayable, their noise rate is measured to assess practical usability. The noise rate is calculated by averaging the number of interfering allocations during a single allocation of the target object. For more fine-grained analysis, the noise rates are categorized into those from objects in the same cache (vulnerable cache), caches with the same order, and caches with different orders. This categorization reflects how the page allocator manages freelists according to page order. Noise rate results are detailed in Section 6.2.

5.3.3 Excavating Useful Primitives. The `ObjectMatcher` identifies potentially useful primitives among the remaining objects within the context of the targeted vulnerability. It operates using the vulnerability's proof-of-concept (PoC) program and employs symbolic execution similar to Syzscope [67] to analyze memory states after cross-cache attacks. Although the required primitives may vary depending on the vulnerability or exploit strategy, the `ObjectMatcher` provides validated results that illustrate how each object might be useful in exploiting the vulnerability.

Table 3: Performance of recycling strategies across various kernel versions and system workloads. For each object size and workload, the highest success rates are both bolded and underlined. The bottom row, marked with 🏆, indicates the number of winning cases where each strategy outperforms the others. The highest count is underlined.

(a) v5.15 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.13%	**100.00%**	**100.00%**	**100.00%**
32	80.03%	**100.00%**	93.33%	0.00%
64	40.07%	**100.00%**	**100.00%**	**100.00%**
96	0.60%	95.97%	**96.67%**	46.63%
128	10.10%	93.33%	93.33%	**100.00%**
192	0.43%	80.00%	86.67%	**99.87%**
256	3.33%	86.67%	96.67%	**100.00%**
512	3.33%	73.33%	93.33%	**99.97%**
1024	30.40%	73.33%	86.67%	**90.03%**
2048	17.03%	**100.00%**	76.67%	43.40%
4096	0.00%	0.00%	60.00%	**100.00%**
🏆	**0/11**	**4/11**	**3/11**	**8/11**

(c) v6.1 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	99.93%	**100.00%**
32	0.07%	0.00%	99.97%	**100.00%**
64	83.80%	0.00%	**100.00%**	**100.00%**
96	7.17%	99.70%	**100.00%**	**100.00%**
128	93.27%	86.67%	96.67%	**100.00%**
192	99.40%	96.67%	**100.00%**	**100.00%**
256	72.60%	96.67%	**100.00%**	**100.00%**
512	56.53%	96.67%	93.33%	**99.97%**
1024	81.30%	0.00%	96.67%	**100.00%**
2048	99.03%	0.00%	**100.00%**	**100.00%**
4096	0.00%	0.00%	96.67%	**99.97%**
🏆	**0/11**	**0/11**	**8/11**	**11/11**

(e) v6.6 in Idle State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	96.53%	**99.60%**
32	0.00%	0.00%	**100.00%**	99.70%
64	**99.97%**	0.00%	**99.97%**	98.87%
96	5.27%	93.30%	90.00%	**97.97%**
128	98.07%	86.67%	83.33%	**98.50%**
192	**98.33%**	83.33%	80.00%	95.60%
256	**99.90%**	86.67%	93.33%	96.30%
512	99.90%	93.33%	**100.00%**	96.10%
1024	96.87%	0.00%	86.67%	**97.37%**
2048	**99.10%**	0.00%	80.00%	98.33%
4096	0.00%	0.00%	86.67%	**97.70%**
🏆	**4/11**	**0/11**	**3/11**	**5/11**

(b) v5.15 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	84.63%	**91.13%**	0.00%
32	0.00%	**83.67%**	67.17%	0.27%
64	7.77%	93.20%	93.03%	**97.73%**
96	38.93%	96.37%	**97.90%**	39.57%
128	41.10%	96.10%	**96.93%**	96.57%
192	38.00%	79.60%	92.77%	**99.67%**
256	22.77%	72.27%	49.70%	**99.80%**
512	24.77%	75.90%	92.03%	**99.07%**
1024	38.50%	**59.97%**	43.73%	29.23%
2048	11.67%	**84.53%**	79.47%	14.77%
4096	0.00%	0.00%	85.87%	**98.27%**
🏆	**0/11**	**3/11**	**3/11**	**5/11**

(d) v6.1 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	**85.83%**	0.00%
32	3.80%	1.37%	**33.20%**	6.27%
64	9.93%	0.03%	96.10%	**97.50%**
96	9.87%	80.30%	**97.53%**	97.20%
128	35.40%	88.83%	92.03%	**98.37%**
192	40.57%	**99.07%**	96.20%	98.67%
256	34.17%	92.73%	90.47%	**99.37%**
512	49.90%	84.90%	89.73%	**98.77%**
1024	62.50%	0.60%	**91.60%**	58.07%
2048	78.37%	0.00%	92.60%	**97.47%**
4096	0.00%	0.00%	89.63%	**98.60%**
🏆	**0/11**	**0/11**	**4/11**	**6/11**

(f) v6.6 in Busy State

Size	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
16	0.00%	0.00%	52.27%	**79.40%**
32	66.60%	0.00%	81.10%	**85.27%**
64	66.93%	0.00%	**90.53%**	39.30%
96	54.53%	85.20%	**89.57%**	77.87%
128	16.10%	9.50%	**93.70%**	77.70%
192	33.33%	**80.53%**	75.27%	56.03%
256	31.83%	**87.57%**	15.87%	5.37%
512	0.73%	92.10%	**92.40%**	88.17%
1024	17.60%	0.00%	88.87%	**93.17%**
2048	0.17%	0.37%	75.13%	**94.00%**
4096	0.00%	0.00%	69.00%	**94.50%**
🏆	**0/11**	**2/11**	**4/11**	**5/11**

Unlike Syzscope, which assumes attackers control the entire object memory, the `ObjectMatcher` conducts a more detailed analysis using the detection rules listed in Table 2 combined with object layout information. Specifically, we categorize each target object's fields into four types: (1) function pointers, (2) data pointers, (3) lethal data such as reference counts and buffer lengths, and (4) user-provided data. The values within user-provided data are symbolized only to reflect more realistic scenarios. By combining these classifications with the detection rules, the `ObjectMatcher` automatically identifies potentially useful primitives.

6 EVALUATION

In this section, we perform three experiments—stability, object identification, and real-world exploitability—to evaluate how our findings support effective cross-cache attacks.

(1) **Stability Experiment.** We assess whether our proposed strategies, Complete Free and SLUBStick + Complete Free, reliably conduct cross-cache attacks by comparing them to existing methods using a synthetic vulnerability.
(2) **Object Identification.** We run CROSS-X to verify if our system correctly identifies suitable target objects. We then discuss how different noise rates affect the versatility of these objects.
(3) **Real-World Exploitability.** We perform real-world experiments involving 9 CVEs, including UAF and DF vulnerabilities, to confirm whether our strategy and the identified target objects are feasible in practical scenarios.

For Stability and Real-World experiments, we used stabilization techniques from prior work K(H)EAPS [63], such as CPU pinning when possible. Although these techniques are primarily for same-cache exploits, some are complementary for cross-cache attacks, such as helping prevent unwanted CPU migrations.

6.1 Stability Experiment

To evaluate the stability of the Complete Free and SLUBStick + Complete Free strategies, we developed a kernel module with a synthetic UAF vulnerability. This module allocates and frees objects in the `kmalloc-*` and `kmalloc-cg-*` caches, which serve as the vulnerable and target caches in our experiment. It also allows reads after freeing, thus exposing a UAF vulnerability. Using this module, we built an experimental program that attempts to recycle the vulnerable slab in the `kmalloc-*` cache and reclaim it with `kmalloc-cg-*` objects. The module inserts a marker string when allocating target objects, serving as a success oracle for cross-cache attacks. If the program triggers a UAF read on the vulnerable object after reclamation and retrieves the marker string, it is considered successful.

With this setup, we compared the Naïve, Partial Free, Complete Free, and SLUBStick + Complete Free strategies by creating program variants for each. These were executed on kernel versions v5.15, v6.1, and v6.6 to include reversal cases in Section §4.2 and the adoption of the `RANDOM_KMALLOC_CACHES` mitigation (see §7). We tested the strategies under idle and busy workloads to assess resilience against external noise, consistent with previous studies [30, 41]. The busy workload was simulated using `stress-ng` with CPU, memory, process, file I/O, and other noise options. Each program was executed 100 times to measure the success rate, followed by a reboot. We repeated this 30 times to calculate the average success rate. The Interleaving Cores strategy was excluded since it is a minor variant of Partial Free. We also conducted a comparison between the original SLUBStick artifacts (reclaiming with page table) [42] and Complete Free by porting our strategy to their environment. The details of this experiment are discussed in Appendix §A.1 due to space limits.

Stability Results. Table 3 summarizes the average success rates. Among public strategies, the Naïve strategy exhibited inconsistent performance and dropped sharply on busy states, especially in kernel v6.1 (Tables 3c and 3d) and v6.6 (Tables 3e and 3f). Partial Free showed better scores, performing best in 4 (Table 3a), 3 (Table 3b), and 2 (Table 3f) caches out of 11. However, it deterministically failed (0% success) for `kmalloc-4096` in v5.15 and five additional caches in v6.1 and v6.6, where the caches correspond to the reversal cases in Table 1. These 0% rates slightly increase to 0.03%~1.37% in busy states (Tables 3d and 3f), indicating that the original formula for Partial Free clearly no longer works and requires external allocations to break through the migration wall.

In contrast, our proposed strategies demonstrated significant stability and resilience to system noise. The Complete Free strategy consistently achieved high success rates across all tested scenarios, confirming its reliability for cross-cache attacks. Specifically, in v6.1 idle states (Table 3c), Complete Free achieved at least 93.33% for `kmalloc-512` and over 96.67% success in all others. In v6.6 idle states (Table 3e), Complete Free achieved over 80.00% success in all caches, indicating that `RANDOM_KMALLOC_CACHES` introduces slight noise but does not affect our strategy. Even under busy conditions, success rates remained above 85% except for `kmalloc-32` in v6.1 and above 52% except for `kmalloc-256` in v6.6, both of which were significantly impacted by external interference. Still, Complete Free managed the highest success rate (33.20%) for `kmalloc-32` and was the only effective strategy for both `kmalloc-16` and `kmalloc-32` in v6.1 busy (Table 3d), while other methods achieved less than 7% success. This underscores the reliability and adaptability of Complete Free for cross-cache attacks, independent of object size, vulnerability traits, and system workload.

The SLUBStick + Complete Free strategy further improved Complete Free, consistently achieving the highest overall success across all environments. Notably, in v6.1 idle states (Table 3c), it reached 100% success in nine caches and 99.97% in two additional caches. These results indicate that using a timing side-channel effectively protects recycling periods from interference when external noise is moderate or low. In contrast, during v5.15 and v6.1 busy states (Tables 3b and 3d) involving `kmalloc-16` and `kmalloc-32`, SLUBStick + Complete Free frequently encountered timeout failures due to high external noise. The smaller objects in these caches required more allocations to fill slabs, prolonging the attack and increasing vulnerability to interference. The dropdowns were restored in v6.6 (Table 3f), suggesting that added cache separation from `RANDOM_KMALLOC_CACHES` was helpful. Yet SLUBStick + Complete Free also suffered in `kmalloc-256` on busy v6.6, showing a 10%p greater drop than Complete Free. Based on these results, we suggest that simpler approaches such as Complete Free might offer better reliability when dealing with small-object caches under high system workloads.

Table 4: Summary of Object Identification Results. Each row builds on previous findings. For example, objects with high success rates are verified as sprayable.

Description	Number of Objects
Total Candidate Objects	346
↪ Present in the Target Kernel	248
↪ Allocation Reproducers Generated	53
↪ Confirmed Spray Capability	**11**
↪ Exhibited High Success Rates	**7**

Table 5: Results of cross-cache experiments for sprayable objects. NR_{SC}, NR_{SO}, and NR_{DO} represent noise rates for objects in the same cache, same-order cache, and different-order cache, respectively. Objects shaded in gray exhibit low NR_{SC} and NR_{SO} values. (NR_{SC}= 0, NR_{SO}< 3.5)

Object	NR_{SC}	NR_{SO}	NR_{DO}	Success Rate
sock_fprog_kern	1	52.3	366.52	0.0%
sk_security_struct	0	15.31	102.55	0.0%
snd_info_buffer	1	9.35	7.61	28.7%
kernfs_open_file	0	1.64	0.71	99.7%
timerfd_ctx	0	1.85	1.82	99.8%
io_ring_ctx	0	1.66	0.00	10.9%
msg_msg	0	0.15	0.35	99.7%
pipe_inode_info	0	3.42	9.98	94.2%
pipe_buffer	0	0.00	0.68	99.9%
fsnotify_group	0	2.20	3.16	99.9%
shmid_kernel	0	0.89	3.02	99.8%

6.2 Object Identification

For the object identification experiment, we applied CROSS-X to kernel objects of interest and evaluated their versatility for cross-cache attacks. We collected 346 candidate objects from prior works [5, 6, 33, 34, 56] and public exploits from kctf VRP [17, 18]. We ran the `ObjectFuzzer` module on a v6.1 kernel with `defconfig` for 2 hours, repeating the process 10 times. This duration was chosen after observing convergence in object discovery during preliminary testing of 0.5 hours repeated 5 times, as shown in Figure 9 in the Appendix. For objects that passed the fuzzing stage and demonstrated spray capability, we developed a program for the cross-cache experiment using these objects as targets, similar to the stability experiment (Section §6.1). Each program was executed 1,000 times, and we calculated average success rates along with three types of noise rates—same cache, same order cache, and different order cache—as described in Section §5.3.

Identification Results. Table 4 summarizes the object identification results at each stage. Of the initial candidate objects, 248 were present in the target kernel, and 53 objects could be allocated by unprivileged users. After the `ObjectEvaluator` stage, CROSS-X identified 11 objects with spray capability, which were then tested in the cross-cache experiment. The results of this experiment are shown in Table 5, where we identified 7 versatile target objects suitable for cross-cache attacks. These objects had high success rates, ranging from 94.2% (pipe_inode_info) to over 99.7% for the other six objects. Due to space constraints, detailed metadata of these objects, including affiliating cache, allocation site, and system call, is provided Table 8 in the appendix.

Furthermore, we observed that these 7 objects had low noise rates for the same cache ($\text{NR}_{\text{SC}} = 0$) and same order cache ($\text{NR}_{\text{SO}} < 3.5$). Interestingly, the different order cache (NR_{DO}) did not significantly affect the success of cross-cache attacks, as observed with pipe_inode_info (9.98), fsnotify_group (3.16), and shmid_kernel (3.02). This is because pages of different orders are retrieved from different freelists within the page allocators, preventing interference with each other. However, io_ring_ctx is an exception; despite having low noise rates in all categories, it exhibited a notably low success rate (10.9%). This occurred because it involved multiple page-level allocations that our ObjectEvaluator module did not detect. Consequently, multiple interfering allocations took place, competing with our reclaiming phase. Nonetheless, our findings help refine the existing understanding that all types of interfering allocations negatively affect cross-cache attacks [36].

6.3 Real-World Exploitability

In line with related works on kernel exploit strategies [21, 33, 41, 63], we conducted a real-world exploitability evaluation on nine CVEs. Eight CVEs were published after 2022, when kmalloc-cg-* separation is enabled by default, along with CVE-2020-29660, where the Partial Free strategy was first suggested to exploit a race UAF. The CVEs include seven UAF and two DF vulnerabilities pivoted to UAF, covering different cache types, object sizes, and purposes of cross-cache attacks. We developed four PoCs that perform critical exploitation steps (e.g., control hijack, address leak) via cross-cache attacks. The PoCs differ only in the recycling strategies described in Section §6.1. Each PoC was executed 300 times under both idle and busy workloads to measure success rates, rebooting the system after each execution. Note that SLUBStick + Complete Free was applied to three out of the nine CVEs; the others were excluded due to mismatching requirements (e.g., lack of a measurement primitive or use of deferred frees). For detailed information about the CVEs, see Table 9 in the Appendix.

Exploitability Results. Complete Free showed strong overall performance, achieving success rates above 91% in 7 out of 9 CVEs under idle workloads (Table 6a) and ranking best in 6 out of 9 under busy workloads (Table 6b). It also produced robust results for CVE-2023-2235, a real-world reversal case in which Partial Free consistently failed. SLUBStick + Complete Free outperformed Complete Free in CVE-2023-3609 under idle workloads but not under busy ones. Three cases showed degraded performance: in CVE-2023-20938, the hidden allocation of another object in the same cache required strategy adjustments; and in CVE-2022-2588 and CVE-2022-32250, high external noise led to timeouts. Meanwhile, the Naïve strategy outperformed Complete Free in CVE-2022-2585. This exceptional case was due to the vulnerability characteristics, where the exploit required two consecutive cross-cache attacks. While this was not a problem under idle conditions—since we do not depend on a specific SLUB state—external allocations after the first cross-cache attack disrupted the second under busy workloads. For such situations, techniques that can isolate task execution momentarily (e.g., Context Conservation in K(H)EAPS [63]) would be helpful to mitigate disruptions.

Table 6: Results of real-world experiments. Gray-highlighted CVEs had limitations (e.g., no measurement, deferred free) in applying SLUBStick + Complete Free.

(a) In Idle State

CVE	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
CVE-2023-20938	0.00%	36.67%	**38.00%**	19.33%
CVE-2023-3609	0.00%	94.67%	95.33%	**99.33%**
CVE-2023-5345	0.00%	85.33%	**86.67%**	83.00%
CVE-2020-29660	0.00%	75.67%	**77.00%**	N/A
CVE-2022-2588	28.67%	**92.67%**	91.00%	N/A
CVE-2022-32250	1.67%	**99.33%**	98.33%	N/A
CVE-2022-2585	99.67%	98.67%	**100.00%**	N/A
CVE-2023-3910	0.00%	**100.00%**	99.67%	N/A
CVE-2023-2235	99.67%	0.00%	**100.00%**	N/A

(b) In Busy State

CVE	Naïve	Partial Free	Complete Free	SLUBStick + C.F.
CVE-2023-20938	0.00%	32.00%	**40.67%**	25.33%
CVE-2023-3609	0.00%	**98.00%**	97.67%	95.00%
CVE-2023-5345	0.00%	82.00%	**83.00%**	78.33%
CVE-2020-29660	0.00%	73.67%	**75.33%**	N/A
CVE-2022-2588	2.00%	21.33%	**34.67%**	N/A
CVE-2022-32250	19.00%	**63.67%**	60.33%	N/A
CVE-2022-2585	**23.00%**	10.33%	12.00%	N/A
CVE-2023-3910	1.33%	3.33%	**3.67%**	N/A
CVE-2023-2235	31.00%	0.00%	**98.67%**	N/A

7 DISCUSSION & FUTURE WORKS

Mitigations. Cross-cache attacks are not mitigated by SMEP [26], SMAP [9], KPTI [10], or kCFI [2, 15], as these defenses preserve kernel execution integrity, which is not violated in cross-cache scenarios. To mitigate SLUB-based abuses, several protections have been proposed, some of which have already been adopted into the mainline Linux kernel. In the following, we discuss the effectiveness of these mitigations in the context of cross-cache attacks.

- **Slab Freelist Protections.** Freelist protections (SLAB_FREELIST_RANDOM, SLAB_FREELIST_HARDENED) [7, 14] (in v4.8 and v4.14) shuffle object freelists and XOR-encrypt next pointers, respectively. Since these protections were designed to mitigate freelist corruptions and OOBs, they do not block cross-cache attacks.
- **Randomised Slab Caches.** RANDOM_KMALLOC_CACHES [11] (in v6.6) splits objects in generic caches (kmalloc-*) into 16 different caches (kmalloc-rnd-#-*) based on allocation callsites. This could be troublesome when same-sized but different objects, apart from the vulnerable object, are required (e.g., due to resource constraints), as these objects will highly likely fall into different caches. However, as evidenced in our stability experiments on v6.6 (§6.1), randomized caches cannot block cross-cache attacks by design; they only add slight noise to some caches.

- **Dedicated Bucket Allocator.** SLAB_BUCKETS [8] (in v6.11) introduced more dedicated caches (msg_msg-*, memdup_user-*) to confine msg_msg objects and memdup_user-allocated buffers, which have been prime targets for delivering user payloads. However, this protection still has two limitations. First, payload-ingestable APIs remain outside the SLAB_BUCKETS fence, specifically Netlink helpers (nla_memdup, nla_strdup), eBPF loaders (bpf_prog_load, bpf_map_create), and string operations (memdup_user_nul). Second, it is still susceptible to internal slab reuse, as the bucket allocator is not a completely different mechanism but shares most SLUB codebases. Therefore, although it adds some difficulty to exploit design, SLAB_BUCKETS cannot block cross-cache attacks.
- **Slab Virtual.** SLAB_VIRTUAL [51] is being discussed as a direct mitigation for cross-cache attacks. It pins the address spaces of slab pages to prevent the recycling phase of the attack. However, SLAB_VIRTUAL has not yet been merged into the mainline due to performance overhead and compatibility problems [38]. Assuming its adoption, page-level UAFs [3, 25] are considered a promising direction for exploit writers to bypass SLUB involvement entirely. PageJack [64] suggests pivoting strategies from UAF and OOB to create dangling page objects and reclaim them as security-critical objects, similar to DirtyCred [33]. ObjectEvaluator in CROSS-X can help measure the noise characteristics of these objects to better understand and fine-tune reclaiming strategies for PageJack attacks.

Other Aspects of Cross-Cache Attacks. In this paper, our scope is limited to exploiting UAF with same order cache objects. However, real-world exploits include scenarios beyond this scope—specifically, (1) cross-cache overflows, where buffer overflows span slabs of different caches, and (2) reclaiming with objects from caches of different orders. Objects identified by CROSS-X are capable of performing the reclaiming phase in these scenarios. However, they require additional strategies for precise page-level heap grooming to ensure slabs are contiguous (for overflow) and to trigger necessary splitting or merging (for different orders). Although some studies [36, 57, 60] suggest strategies for these scenarios, systematic evaluation under various conditions—such as different combinations of vulnerable and target cache orders—is still needed.

Page-Oriented Kernel Exploits. The CROSS-X system currently does not consider page allocations beyond the SLUB allocator, which is a limitation. In future work, we plan to evaluate these allocations and related strategies. Meanwhile, several studies [21, 41, 58, 64, 65] have already used such allocations for non-slab page reclamation in cross-cache attacks. For example, they attempt to reclaim vulnerable slabs using pages allocated for data buffers, such as those in the pipe and io_uring subsystems, or by leveraging page table entries targeting specific memory regions for compromise. However, these page-level allocations and exploit methods are relatively rare and often require additional permissions, such as access to particular subsystems or devices. Therefore, they can be effectively mitigated by specifically designed protections [23, 53, 62], as demonstrated by Guo et al. [21], or by deliberately restricting permissions or removing relevant subsystems [28].

8 RELATED WORK

Application of Cross-Cache Attacks. Several modern exploits use cross-cache attacks as a core technique. The possibility of such attacks was first proposed by Xu et al. [61]. DirtyCred later introduced a privilege escalation method by swapping unprivileged file and cred objects with privileged ones, using cross-cache attacks to turn vulnerabilities into UAF conditions. Non-slab reclamation exploits further expanded the scope. Guo et al. [21] extensively analyzed page spraying attacks, exploring root causes and alternative reclamation strategies. SLUBStick [41] proposed a general strategy combining page table reclamation [58] with memory write primitives. Our work improves the reliability of these approaches amid varying cross-cache strategies.

Exploit Reliability Engineering. ExpRace [31] enhanced race condition exploit stability by manipulating kernel interrupts. Playing for K(H)EAPS [63] studied heap exploit reliability, exposing expert misconceptions and proposing a combined technique for improvement. Pspray [30] used a timing side channel to detect fresh slab allocations and infer heap layouts, increasing kernel heap exploit success. Recently, Maar et al. [43] presented a TLB side-channel kernel leak technique to overcome new defense-induced constraints, reviving exploit primitives like unlinking attacks and constrained writes. In contrast, our work focuses on stabilizing cross-cache attacks by analyzing instability sources and proposing enhanced strategies.

Automated Systems for Exploit Generation. FUZE [59] and KOOBE [4] created automated frameworks to diversify UAF and OOB exploits using static and dynamic methods. SyzScope [67] elevated low-risk kernel bugs into severe threats like control flow hijacking or arbitrary writes via fuzzing and symbolic execution. Other studies targeted useful kernel objects. SLAKE [6] modeled exploitable objects by modifying slab layouts. ELOISE [5] identified elastic kernel objects to bypass KASLR and heap cookies. Alpha-EXP [56] developed an expert system to classify kernel objects by simulating attacker behavior and building a knowledge graph to infer attack paths. Building on these, our system CROSS-X identifies versatile objects well-suited for cross-cache attacks from a database of public objects, based on target object properties.

9 CONCLUSION

The reliability of cross-cache attacks has been limited due to an incomplete understanding and reliance on heuristics, despite their significance in modern exploits. This work presents new insights to address this problem by proposing two optimized and generalized attack strategies, along with an automated system, CROSS-X, for identifying versatile objects. We implemented a prototype of CROSS-X and evaluated the identified objects using our proposed strategies both on synthetic and nine real-world vulnerabilities. As a result, we identified seven suitable target objects for cross-cache attacks and achieved high stability across various environments.

References

[1] 2023. Escaping the Google kCTF Container with a Data-Only Exploit. https://h0mbre.github.io/kCTF_Data_Only_Exploit/. Online, Accessed: 2025-04-13.

[2] Android. 2025. Kernel control flow integrity. https://source.android.com/docs/security/test/kcfi. Online, Accessed: 2025-07-26.

[3] Oriol Castejón. 2024. Mind the Patch Gap: Exploiting an io_uring Vulnerability in Ubuntu. https://blog.exodusintel.com/2024/03/27/mind-the-patch-gap-exploiting-an-io_uring-vulnerability-in-ubuntu/. Online, Accessed: 2025-04-13.

[4] 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)*. 1093–1110.

[5] 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*. 1165–1184.

[6] 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*. 1707–1722.

[7] Kees Cook. 2017. mm: Add SLUB free list pointer obfuscation. https://patchwork.kernel.org/project/linux-hardening/patch/20170726041250.GA76741@beast/. Online, Accessed: 2025-04-13.

[8] Kees Cook. 2024. slab: Introduce dedicated bucket allocator. https://lwn.net/Articles/980302/. Online, Accessed: 2025-07-26.

[9] Jonathan Corbet. 2012. Supervisor mode access prevention. https://lwn.net/Articles/517475/. Online, Accessed: 2025-04-13.

[10] Jonathan Corbet. 2017. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/. Online, Accessed: 2025-04-13.

[11] Jonathan Corbet. 2023. Randomness for kmalloc(). https://lwn.net/Articles/938637/. Online, Accessed: 2025-04-13.

[12] CROSS-X. 2025. CVE-2022-2585 Exploit. https://github.com/crossx-1891/CROSS-X/blob/main/exploits/cve-2022-2585/exploit.c. Online, Accessed: 2025-04-13.

[13] Vladimir Davydov. 2016. slab: add SLAB_ACCOUNT flag. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=230e9fc2860450fbb1f33bdcf9093d92d7d91f5b. Online, Accessed: 2025-04-13.

[14] Thomas Garnier. 2016. mm: SLAB freelist randomization. https://lwn.net/Articles/685047/. Online, Accessed: 2025-04-13.

[15] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 179–194.

[16] Google. 2015. Syzkaller. https://github.com/google/syzkaller.

[17] Google. 2020. kCTF VRP Setup. https://google.github.io/kctf/vrp.html. Online, Accessed: 2025-04-13.

[18] Google. 2022. Kernel Exploits Recipes Notebook. https://docs.google.com/document/d/1a9uUAISBzw3ur1aLQqKc5JOQLaJYiOP5pe_B4xCT1KA. Online, Accessed: 2025-04-13.

[19] Mel Gorman. 2004. *Understanding the Linux virtual memory manager*. Vol. 352. Prentice Hall Upper Saddle River.

[20] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 368–379.

[21] 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. *arXiv preprint arXiv:2406.02624* (2024).

[22] Jann Horn. 2021. How a simple Linux kernel memory corruption bug can lead to complete system compromise. https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html. Online, Accessed: 2025-04-13.

[23] Huawei. 2020. Emui 11.0 security technical white paper. https://consumer.huawei.com/content/dam/huawei-cbg-site/common/campaign/privacy/whitepaper/emui_11.0_security_technical_white_paper_v1.0.pdf. Online, Accessed: 2025-04-13.

[24] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205.

[25] Xingyu Jin and Clement Lecigene. 2024. CVE-2024-44068: Samsung m2m1shot_scaler0 device driver page use-after-free in Android. https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2024/CVE-2024-44068.html. Online, Accessed: 2025-04-13.

[26] Mateusz Jurczyk and Gynvael Coldwind. 2011. SMEP: What is it, and how to beat it on Windows. https://j00ru.vexillium.org/2011/06/smep-what-is-it-and-how-to-beat-it-on-windows/. Online, Accessed: 2025-04-13.

[27] Kenneth C Knowlton. 1965. A fast storage allocator. *Commun. ACM* 8, 10 (1965), 623–624.

[28] Tamás Koczka. 2023. Learnings from kCTF VRP’s 42 Linux kernel exploits submissions. https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html. Online, Accessed: 2025-04-13.

[29] Christoph Lameter. 2007. SLUB: The unqueued slab allocator V6. https://lwn.net/Articles/229096/. Online, Accessed: 2025-04-13.

[30] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. 2023. Pspray: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA.

[31] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. 2021. ExpRace: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*. 2363–2380.

[32] Zhenpeng Lin. 2023. CVE-2022-2588. https://github.com/Markakd/CVE-2022-2588. Online, Accessed: 2025-04-13.

[33] 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*. 1963–1976.

[34] Danjun Liu, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, Zhenhao Luo, Tai Yue, and Baosheng Wang. 2022. From release to rebirth: Exploiting thanos objects in Linux kernel. *IEEE Transactions on Information Forensics and Security* 18 (2022), 533–548.

[35] William Liu. 2021. corCTF 2021 Fire of Salvation Writeup: Utilizing msg_msg Objects for Arbitrary Read and Arbitrary Write in the Linux Kernel. https://www.willsroot.io/2021/08/corctf-2021-fire-of-salvation-writeup.html. Online, Accessed: 2025-04-13.

[36] William Liu. 2022. Reviving Exploits Against Cred Structs - Six Byte Cross Cache Overflow to Leakless Data-Oriented Kernel Pwnage. https://www.willsroot.io/2022/08/reviving-exploits-against-cred-struct.html. Online, Accessed: 2025-04-13.

[37] William Liu, Joseph Ravichandran, and Mengjia Yan. 2023. EntryBleed: A Universal KASLR Bypass against KPTI on Linux. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*. 10–18.

[38] LKML.org. 2023. Re: [RFC PATCH 00/14] Prevent cross-cache attacks in the SLUB allocator. https://lkml.org/lkml/2023/9/18/1047. Online, Accessed: 2025-04-13.

[39] Lam Jun Long. 2022. io_uring - new code, new bugs, and a new exploit technique. https://starlabs.sg/blog/2022/06-io_uring-new-code-new-bugs-and-a-new-exploit-technique/. Online, Accessed: 2025-04-13.

[40] Waiman Long. 2021. mm: memcg/slab: create a new set of kmalloc-cg-<n> caches. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=494c1dfe855ec1f70f89552fce5eadf4a1717552. Online, Accessed: 2025-04-13.

[41] 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)*. 4051–4068.

[42] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. 2024. SLUBStick Artifacts. https://github.com/isec-tugraz/SLUBStick/blob/main/exploits/userspace/exploit_key.c. Online, Accessed: 2025-04-13.

[43] Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard. 2025. WHEN GOOD KERNEL DEFENSES GO BAD: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In *34rd USENIX Security Symposium: USENIX Security 2024*. USENIX Association.

[44] Vitaly Nikolenko Michael S. 2022. Linux kernel heap feng shui in 2022. https://duasynt.com/blog/linux-kernel-heap-feng-shui-2022. Online, Accessed: 2025-04-13.

[45] MITRE. 2021. CVE-2021-22555. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22555. Online, Accessed: 2025-04-13.

[46] MITRE. 2022. CVE-2022-0185. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0185. Online, Accessed: 2025-04-13.

[47] MITRE. 2022. CVE-2022-2585. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-2585. Online, Accessed: 2025-04-13.

[48] MITRE. 2022. CVE-2022-2602. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-2602. Online, Accessed: 2025-04-13.

[49] MITRE. 2022. CVE-2022-32250. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-32250. Online, Accessed: 2025-04-13.

[50] MITRE. 2023. CVE-2023-5345. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-5345. Online, Accessed: 2025-04-13.

[51] Matteo Rizzo. 2023. Prevent cross-cache attacks in the SLUB allocator. https://lore.kernel.org/linux-mm/202309151425.2BE59091@keescook/. Online, Accessed: 2025-04-13.

[52] Javier P Rufo. 2024. CVE-2022-22265. https://soez.github.io/posts/CVE-2022-22265-Samsung-npu-driver/. Online, Accessed: 2025-04-13.

[53] Samsung. 2024. Real-time Kernel Protection (RKP). https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-for-android/core-platform-security/real-time-kernel-protection/. Online, Accessed: 2025-04-13.

[54] Looker Studio. [n. d.]. Interesting Kernel Objects. https://lookerstudio.google.com/u/0/reporting/68b02863-4f5c-4d85-b3c1-992af89c855c/page/n92nD. Online, Accessed: 2025-04-13.

[55] SSD Secure Disclosure technical team. 2022. SSD Advisory – Linux CLOCK_THREAD_CPUTIME_ID LPE. https://ssd-disclosure.com/ssd-advisory-linux-clock_thread_cputime_id-lpe/. Online, Accessed: 2025-04-13.

[56] 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)*. 4229–4246.

[57] Le Wu. 2024. Game of Cross Cache: Let's win it in a more effective way!. In *Blackhat USA*.
[58] Nicolas Wu. 2023. Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. https://web.archive.org/web/20241017044748/https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html. Online, Accessed: 2025-04-13.
[59] 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)*. 781–797.
[60] Zhiyun Qian Xiaochen Zou. 2022. CVE-2022-27666: Exploit esp6 modules in Linux kernel. https://etenal.me/archives/1825. Online, Accessed: 2025-04-13.
[61] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 414–425.
[62] Jun Yao. 2018. arm64/mm: move idmap_pg_dir,tramp_pg_dir,swapper_pg_dir to .rodata section. https://patchwork.kernel.org/project/linux-hardening/patch/20180620085755.20045-2-yaojun8558363@gmail.com/. Online, Accessed: 2025-04-13.
[63] 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)*. 71–88.
[64] Jiayi Hu Zhiyun Qian, Jinmeng Zhou, Qi Tang, and Wenbo Shen. 2024. PageJack: A Powerful Exploit Technique With Page-Level UAF. In *Blackhat USA*.
[65] 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 preprint arXiv:2401.17618* (2024).
[66] Gulshan Singh Zi Fan Tan and Eugene Rodionov. 2024. Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938. https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938/. Online, Accessed: 2025-04-13.
[67] 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)*. 3201–3217.

A Appendix

A.1 Stability Experiment with SLUBStick

We compared our Complete Free strategy with SLUBStick [41], the state-of-the-art in cross-cache attacks. SLUBStick employs an optimized "allocate all, free all" approach, where fine-tuned parameters (e.g., total number of slabs, index of the vulnerable slab) enhance reliability. As their original setup differs from our stability experiments, we ported our Complete Free strategy to their environment. Specifically, we used the v6.2 kernel with identical mitigations and measured the success rate of reclamation into page table entries using the provided kernel module with SLUBStick's synthetic vulnerability. Unfortunately, we were unable to reproduce SLUBStick artifacts for objects larger than 256 bytes, which rely on separate PoC programs using huge pages. Therefore, we report results only for `kmalloc-16` to `kmalloc-256`, covering single-order slab caches and `cpu_partial` reversal cases. All programs were run 100 times then rebooted, repeating 30 times same as in our stability experiments (see §6.1). We also simulated busy workloads with `stress-ng` using identical options.

Results. Table 7 shows the success rates of Complete Free and SLUBStick under idle and busy workloads. Complete Free achieved higher success in 4 out of 7 cases under idle workloads (Table 7a) and 6 out of 7 under busy workloads (Table 7b). SLUBStick exhibited more performance drops, especially in small-size caches (`kmalloc-16~kmalloc-96`). This contrasts with the original SLUBStick paper, where almost no drop occurred except for `kmalloc-64`, but it was expected since we apply 12 noise options for the busy workload, compared to only 3 (CPU, I/O, and memory) in their study. Overall, the result aligns with our stability experiments, where timing side-channels are susceptible to external noise with small objects, showcasing Complete Free's adaptability to various exploitation plans with cross-cache attacks.

Table 7: Results of stability experiments with SLUBStick.

(a) In Idle State

Size	SLUBStick	Complete Free
16	86.40%	**100.00%**
32	93.03%	**100.00%**
64	96.03%	**99.13%**
96	97.40%	**100.00%**
128	**97.67%**	96.67%
192	**98.73%**	96.67%
256	**97.70%**	93.33%

(b) In Busy State

Size	SLUBStick	Complete Free
16	38.10%	**96.67%**
32	46.47%	**87.23%**
64	69.23%	**93.33%**
96	77.23%	**96.60%**
128	85.27%	**96.57%**
192	88.90%	**89.90%**
256	**90.40%**	87.10%

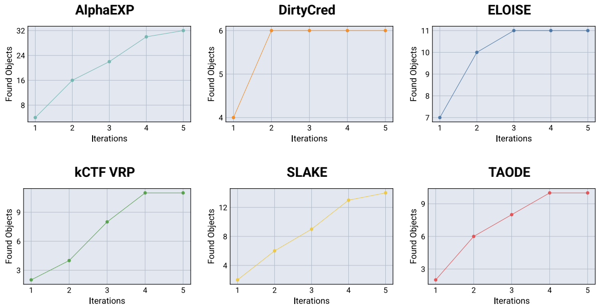


Figure 9: Accumulated number of discovered objects during preliminary testing of the `ObjectFuzzer` module. The module was run for 0.5 hours per iteration, repeated 5 times. Convergence was observed in 4 out of 6 object candidate sets (DirtyCred, ELOISE, kCTF VRP, TAODE), and near convergence in the remaining 2 (AlphaEXP, SLAKE).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Table 8: Detailed information on objects with confirmed spray capabilities from experiments. Objects with a grey background exhibited low NR_{SC} and NR_{SO}, while bolded objects represent the final confirmed suitable selections.

Object	Slab Cache	Allocation Site	Allocation System Call
`sock_fprog_kern`	`kmalloc-16`	`bpf_prog_store_orig_filter` in net/core/filter.c	`setsockopt`
`sk_security_struct`	`kmalloc-32`	`selinux_sk_alloc_security` in security/selinux/hooks.c	`io_uring_setup`
`snd_info_buffer`	`kmalloc-32`	`snd_info_text_entry_open` in sound/core/info.c	`openat`
`kernfs_open_file`	`kmalloc-192`	`kernfs_fop_open` in fs/kernfs/file.c	`openat`
`timerfd_ctx`	`kmalloc-256`	`sys_timerfd_create` in fs/timerfd.c	`timerfd_create`
`io_ring_ctx`	`kmalloc-2048`	`io_ring_ctx_alloc` in io_uring/io_uring.c	`io_uring_setup`
`msg_msg`	`kmlloc-cg-64`[1]	`alloc_msg` in ipc/msgutil.c	`msgsnd`
`pipe_inode_info`	`kmalloc-cg-192`	`alloc_pipe_info` in fs/pipe.c	`pipe`
`fsnotify_group`	`kmalloc-cg-256`	`inotify_new_group` in fs/notify/inotify/inotify_user.c	`inotify_init`
`shmid_kernel`	`kmalloc-cg-256`	`newseg` in ipc/shm.c	`shmget`
`pipe_buffer`	`kmalloc-cg-1024`	`alloc_pipe_info` in fs/pipe.c	`pipe`

[1] `msg_msg` is indeed an elastic object that can grow up to 4,096 bytes in size. The above is merely based on `ObjectFuzzer` results.

Table 9: Detailed environmental setup for each CVE evaluated in the real-world experiment. The Objective column specifies the critical operation whose successful execution indicates an effective exploit, accomplished through cross-cache attacks with objects listed in the Target Object column.

CVE	Vulnerability Type	Vulnerable Cache	Deallocation	Target Object	Objective	SLUBStick+C.F. Availability
CVE-2023-20938	UAF	`kmalloc-128`	Direct	`fsnotify_group`	Kernel Heap Leak	✓
CVE-2023-3609	UAF	`kmalloc-128`	Direct	`msg_msg`	Control Flow Hijacking	✓
CVE-2023-5345	DF	`kmalloc-128`	Direct	`msg_msg`	Kernel Heap Leak	✓
CVE-2020-29660	UAF	`pid`[1]	Direct	page table entries	Page Table Corruption	✗[2]
CVE-2022-2588	DF	`kmalloc-192`	Worker	`msg_msg`	Payload Delivery	✗
CVE-2022-32250	UAF	`kmalloc-64`	Worker	`msg_msg`	Kernel Heap Leak	✗
CVE-2022-2585	UAF	`posix_timer_cache`	RCU	`msg_msg`	Kernel Heap Leak	✗
CVE-2023-3910	UAF	`filp`	Worker	`pipe` buffer[3]	Kernel Base Leak	✗
CVE-2023-2235	UAF	`perf_event`	RCU	`msg_msg`	Kernel Heap Leak	✗

[1] This cache is merged with the `seq_file` and `eventpoll_epi` caches, enabling the use of `seq_file` and `epitem` objects during exploitation.
[2] Despite not using deferred frees, we could not find a method to enable measurement primitives (immediate free within the same system call) for this exploit.
[3] For CVE-2023-3910, we used the page spraying technique employed in the original exploit.