Cross-Cache Attacks for the Linux Kernel via PCP Massaging

Claudio Migliorelli*, Andrea Mambretti*, Alessandro Sorniotti*, Vittorio Zaccaria†, Anil Kurmus*
*IBM Research Europe – Zurich {gli, amb, aso, kur}@zurich.ibm.com
†Politecnico di Milano vittorio.zaccaria@polimi.it

***Abstract*—Kernel memory allocators remain a critical attack surface, despite decades of research into memory corruption defenses. While recent mitigation strategies have diminished the effectiveness of conventional attack techniques, we show that robust cross-cache attacks are still feasible and pose a significant threat. In this paper, we introduce PCPLOST, a cross-cache memory massaging technique that bypasses mainline mitigations by carefully using side channels to infer the kernel allocator's internal state. We demonstrate that vulnerabilities such as out-of-bounds (OOB) — and, via pivoting, use-after-free (UAF) and double-free (DF) — can be exploited reliably through a cross-cache attack, across all generic caches, even in the presence of noise. We validate the generality and robustness of our approach by exploiting 6 publicly disclosed CVEs by using PCPLOST, and discuss possible mitigations. The significant reliability (over 90% in most cases) of our approach in obtaining a cross-cache layout suggests that current mitigation strategies fail to offer comprehensive protection against such attacks within the Linux kernel.**

I. Introduction

Linux powers much of today's information technology, from everyday items such as vacuum cleaners, mobile phones, and cars to enterprise servers and critical infrastructure. Yet, memory safety issues continue to plague the Linux kernel, and allow attackers to bypass security measures [1].

Recently, much attention was given to kernel heap vulnerabilities [2–20] by academia and practitioners alike. These vulnerabilities, along with their corresponding mitigations, can broadly be categorized as *in-cache* and *cross-cache*, based on whether memory safety violations occur within the boundaries of a single *cache*, a structure shared by similar objects, or extend across caches. Hardening measures against in-cache attacks [21–24] pushed attackers towards the more complex cross-cache attacks [7, 9, 10, 12, 13, 25] leading to the introduction of new cross-cache mitigations such as `SLAB_VIRTUAL` [26].

In this paper, we show that despite the latest mitigations, cross-cache attacks remain feasible. To support this claim, we introduce PCPLOST, a novel and reliable cross-cache massaging technique that successfully circumvents current mitigation strategies [21, 22]. Our findings demonstrate that modern kernel defenses remain inadequate, as PCPLOST achieves more than 90% reliability in most exploitation scenarios, even against the experimental `SLAB_VIRTUAL` [26] protection mechanism.

Our approach builds on a detailed analysis of the kernel allocator, which uncovered hitherto overlooked interactions in its internals. These can be used to extend the detection ability of known side-channels to more effectively control the cache layout, facilitating out-of-bounds (OOB) cross-cache attacks. Furthermore, we show that the attack surface can be extended by "pivoting" temporal bugs into OOB primitives, whereas previous approaches pivoted *from* OOB to a temporal primitive instead [7, 9].

The core motivation of our research is to challenge the current belief that cross-cache exploitation is unreliable and challenging to execute [27–29]. We argue that, as kernel hardening techniques increasingly enforce strict separation between object types [21, 24, 30], cross-cache exploitation is emerging as a prevalent strategy in contemporary attacks. In particular, we show that attackers can reliably perform memory massaging to enable OOB read/write across arbitrary generic cache pairs by uniquely exploiting the allocator's interaction with PCP lists and zoned `free_area`. Unlike notable works such as PSPRAY [14], which targets in-cache attacks, or SLUBStick [9], focused on page tables and mitigated by `SLAB_VIRTUAL` [26], PCPLOST is broadly applicable and effective across generic caches, even under noise and mitigations like `SLAB_VIRTUAL`. We test PCPLOST against 6 publicly disclosed CVEs, demonstrating its effectiveness in both direct OOB exploitation and pivoted attacks derived from temporal vulnerabilities. The results demonstrate PCPLOST's broad applicability and reliability, with a success rate in achieving the desired cross-cache layout above 90% in most cases, thereby underlining the need for cross-cache mitigations for kernel security. More specifically, we make the following contributions:

- **Analysis of modern SLUB and Page Frame Allocator internals**: we examine critical, yet often overlooked, design details for cross-cache exploitation of the Linux kernel allocators, in particular presenting the inner workings of PCP lists.
- **PCP-list-based cross-cache memory massaging**: we introduce a cross-cache PCP list massaging technique that bypasses mainline mitigation strategies.

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240862
www.ndss-symposium.org

- **Novel use of software side-channel at the page allocator level**: building on previous research on software side-channels in the SLUB allocator [9, 14] we introduce a novel use of such techniques to reveal interactions between SLUB, PCP lists and `free_area`.
- **Amplification of cross-cache massaging applicability with pivoting strategy**: by leveraging the pivoting technique, we successfully expand the attack surface to vulnerabilities other than OOB.
- **Evaluation against real-world CVEs and extensive reliability analysis**: we validate our PCP list massaging technique using publicly disclosed CVEs and conduct a thorough examination of its applicability against generic (kmalloc) caches.

Threat model. As in previous kernel heap massaging attacks, we consider an unprivileged local attacker that can interact with the Linux kernel through system calls. For the purpose of the side channel, the attacker needs to be capable of performing fine-grained time measurements, for instance by using the userspace-available instruction `rdtsc`. To achieve privilege escalation or arbitrary read or write capability, the attacker needs to combine this layout with a temporal or spatial kernel heap memory safety vulnerability. Therefore, we assume the presence of such a vulnerability.

II. Background

In Section II-A and Section II-C, we provide the necessary background on kernel memory allocation. In Section II-C, we delve into the lesser-known optimization details used by the kernel to allocate memory pages: these inner workings are central to the success and reliability of PCPLost.

A. SLUB Allocator

The Linux kernel employs the SLUB allocator for efficient dynamic object allocation in kernel space [31, 32]. This memory management system operates through two main allocation APIs: `kmem_cache_alloc()` and `kmalloc()`. The `kmem_cache_alloc()` function allocates objects of specific types (such as `task_struct` instances), while `kmalloc()` serves as a general-purpose wrapper for allocating untyped memory buffers of specified sizes (Table V in Appendix A).

As shown in Figure 1, the SLUB architecture is centered around the `kmem_cache` object which groups physically contiguous, possibly pre-initialized instances of objects of a specific type into contiguous sequences of 2^k pages called *slabs*, and, within each slab, free objects are linked in a list (`freelist`). To minimize locking overhead, each CPU is assigned its own main slab (via a per-CPU variable of type `kmem_cache_cpu`) for lockless access. When the current slab of a CPU is exhausted, the allocator swaps it with a slab from a `partial` slabs list (i.e., a list of slabs with one or more free objects), to ensure a continuous supply of objects. As these per-CPU partial slabs become fully utilized, SLUB extends its search to partial slabs on the current NUMA node, thereby preserving allocation efficiency (via `kmem_cache_node`). When

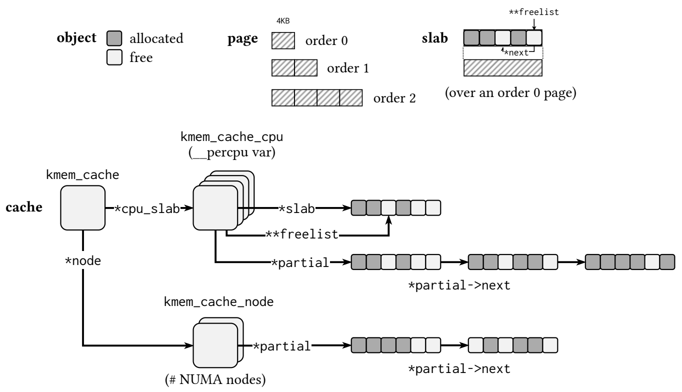


Fig. 1: Overview of the SLUB allocator architecture. Objects are stored in slabs, which are set of contiguous memory pages. Each slab is managed by the kernel using the `kmem_cache_cpu` data structure. This data structure is a per-CPU version of the main `kmem_cache`, which stores metadata to correctly manage kernel objects.

all slabs are consumed, SLUB obtains new pages from the Page-Frame Allocator, described in II-B.

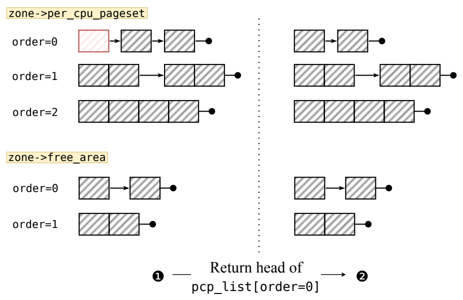
B. Page Frame Allocator

The Page Frame (Buddy) Allocator [31, 32] is used by the Linux kernel to manage physical memory and reduce fragmentation. Memory is allocated in pages, typically 4096 bytes each, and allocation is order-based: an order-k page consists of 2^k contiguous pages, with k ranging from 0 to `MAX_PAGE_ORDER`. Within a NUMA node, the Linux kernel organizes physical memory into distinct *zones* shared among CPU-cores. Each zone is independently managed by the (zoned) Buddy Allocator and maintains its collection of free page lists (the `free_area`) organized by order, as shown in Figure 2. Moreover, each page has a specified migration type (`migrate_type`), which determines the policy for page migration: the kernel can choose to move the physical location of pages within a NUMA system, while maintaining their virtual addresses unchanged. Therefore, migration types group pages with similar migration properties (e.g., movable or unmovable).

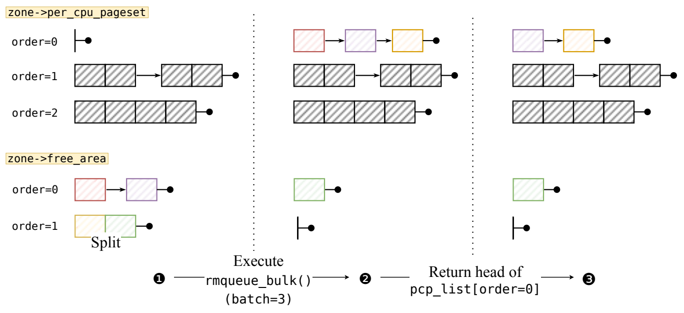
When the kernel requests order-k pages, all physical pages of that order can become exhausted. To obtain more pages, the allocator performs a page split: it examines the next higher-order list (order-$k+1$), removes a page, and splits it into two order-k pages. The first chunk is returned to the caller, while the second is added to the order-k list in the `free_area`. We refer to the two order-k page chunks involved in a split as *buddy pages*. The first page is referred to as the *left-buddy* (L-buddy) and the second page as the *right-buddy* (R-buddy).

C. Per-CPU-Pageset (PCP) lists

To mitigate the latency associated with allocations coming from the `free_area`, which is shared among CPU-cores, the kernel uses auxiliary structures known as *Per-CPU-Pageset*



(a) Non-empty PCP list. Whenever SLUB needs to allocate a new slab page of a certain order (say 0), it invokes the underlying Page Frame Allocator. If possible, the latter allocates the page from the head of the PCP list of the corresponding order (❶ - ❷) through the `rmqueue_pcplist()` function. This pool of memory pages is per-CPU, therefore allowing a fast allocation path (no locking).



(b) Empty PCP list. If the PCP list for the requested order and migrate type is empty, the kernel executes `rmqueue_bulk()`. Here, `rmqueue_bulk()` is called with `batch=3`: the PCP list is to be expanded with three order-0 pages (transition ❶ - ❷). The request is satisfied with two pages coming from the `free_area` directly, and the third by splitting a higher order page. These 3 pages are *reserved* in the PCP list, with the first one being returned to the caller (❷ - ❸).

Fig. 2: Main allocation paths taken by the Page Frame Allocator when requesting an order-0 page. Figure 2a illustrates such an allocation when the PCP list is not empty, whereas Figure 2b shows the allocation when the PCP list is empty. For simplicity, figures 2a and 2b show pages of the same migrate type (e.g., `UNMOVABLE`).

(PCP) lists, i.e., a list of fresh pages for each page order and migration type that is local to the CPU core.

When the kernel asks for a free order-k page, it checks first the corresponding PCP list. If the list is not empty, an order-k page is returned immediately from it (through `rmqueue_pcplist()`), as shown in Figure 2a. If the list is empty, the kernel invokes the Buddy Allocator to acquire multiple (=`batch`) fresh pages through `rmqueue_bulk()` (see Figure 2b). When the `free_area` for order-k pages is emptied, the Buddy Allocator inspects higher-order lists in the `free_area` and performs splits until it obtains the required batch of order-k pages which are moved to the order-k PCP list (*batch allocation*). Finally, the first order-k page in the batch allocation gets removed from the PCP list and returned to the kernel.

The Linux kernel attempts to maintain pages as close as possible to the CPU core by utilizing these batch allocations from the `free_area` and moving pages to the PCP list. Our empirical analysis shows that pages with orders supported by PCP lists (up to `PAGE_ALLOC_COSTLY_ORDER = 3`) exhibit strong locality per CPU core, leading to sparse utilization of the corresponding `free_area` lists and thereby better multicore scaling.

III. Related Work

In Section III-A, we introduce previous work on SLUB attacks, and in Section III-B, we present related defenses.

A. SLUB Attacks

Memory corruption exploits in the SLUB allocator involve two object types: vulnerable objects, which are susceptible to a memory error such as an out-of-bounds (OOB) write, and target objects, whose access or corruption can get an attacker closer to their goal, such as obtaining privilege escalation or simply greater kernel memory access. Exploits typically progress incrementally, using vulnerable objects to corrupt targets and expand control. To ensure reliability, attackers may employ *heap feng-shui* or *memory massaging* [15, 33, 34], which allows to strategically arrange the memory layout of the SLUB allocator. These techniques categorize based on the kmem-caches storing the objects. The following two sections describe these different attack types and provide examples of such techniques in related work.

1) In-Cache attacks: Efforts to exploit the SLUB allocator have primarily focused on corrupting target objects that are stored within the same `kmem_cache` as vulnerable objects, a scenario commonly known as an *in-cache* attack. In these attacks, an adversary often employs a technique called *spraying* [11, 35–38], which involves inserting numerous instances of a target object into memory. When dealing with OOB vulnerabilities, this method increases the likelihood that a target object will be allocated adjacently to an object vulnerable in the same slab cache. In contrast, the exploitation of use-after-free (UAF) vulnerabilities requires an attacker to reclaim for the target object a memory slot that, prior to freeing, was used to store the vulnerable object. Different UAF exploitation strategies then exist depending on how the freed object is subsequently used.

In either case, inferring the current state of the SLUB allocator can assist attackers in determining if the memory layout is favorable for an attack. Timing attacks serve as a practical method for this inference. These attacks, extensively studied for cross-CPU, cross-VM data leakage [39–41], and KASLR circumvention [42], have recently been adapted to reveal the SLUB allocator state. PSPRAY [14] shows that system call timings can be used to infer the state of the SLUB allocator, and discard those cases where the memory layout is not favorable for the vulnerability at hand. In particular, PSPRAY distinguishes between allocations via the per-CPU slab free-list (fastest), from partial slabs (slower), and from

Technique	Cross-cache	Page adjacency layout	Side-channel based	Bypasses SV [26]
SLUBStick [9]	✓	✗	✓	✗
PSPRAY [14]	✗	✗	✓	N/A
BridgeRouter [29]	✓	✗	✗	?
PCPLOST	✓	✓	✓	✓

TABLE I: Comparison with previous work. `SV` stands for `SLAB_VIRTUAL` (without guard pages). The "?" sign indicates we believe it could be used to bypass `SLAB_VIRTUAL`, but it is not claimed nor demonstrated by BridgeRouter. For PSPRAY, "N/A" indicates that `SLAB_VIRTUAL` is not applicable in this case (in-cache).

the Buddy Allocator (slowest) and improve the success rate of the attack.

2) Cross-Cache attacks: Due to recent kernel slab mitigations that increasingly segregate user-controlled objects from critical kernel objects [43, 44] – and also objects of different types in more general terms [21, 24, 30] – , the probability of finding both vulnerable and target objects within the same cache has significantly diminished. For instance, `struct msg_msg`, extensively used in many kernel exploits, is not allocated in `kmalloc-*` caches anymore, rather in their accounted version (`kmalloc-cg-*`). In response, attackers have shifted their focus towards a more general attack vector known as *cross-cache* attacks [7, 9, 10, 12, 13]. In these attacks, the vulnerable object and the target object are allocated in separate caches, providing attackers with a broader range of possibilities in terms of object types. However, this extended attack surface introduces additional challenges to memory massaging.

When exploiting cross-cache OOB vulnerabilities, the memory pages backing the caches of both vulnerable and target object must be adjacent in memory. Attackers might use heap massaging techniques [13, 34, 45, 46], to obtain the desired (contiguous) memory layout through a careful sequence of allocations triggered by appropriate system calls.

In contrast, cross-cache use-after-free (UAF) attacks rely on page recycling [7, 9, 10, 12, 16, 47], i.e., forcing the page containing the vulnerable object to be freed, and reusing it for a target object. In this context, spraying involves flooding the memory and selectively reclaiming some controlled objects so that the SLUB allocator is tricked into freeing an entire slab. Given the last-in-first-out (LIFO) handling of this list, any subsequent slab allocation from the vulnerable cache is likely to reuse the same page. After the same page is reused, the attacker can allocate the full slab with target objects to reuse the object slot controlled by the dangling pointer. Notably, previous work [9, 12], which implemented a similar cross-cache attack approach, use page tables as the target object, focusing on a single target cache.

Thus, while both attack scenarios exploit traits of the underlying Page Frame Allocator, they fundamentally differ: cross-cache overflow attacks (OOB) rely on spatial adjacency between caches, whereas cross-cache temporal attacks (UAF or double-free (DF)) rely on the controlled recycling of pages via spraying.

Covert timing measurements can help also in cross-cache attacks. In particular, SLUBStick [9] introduces a novel technique to reduce noise when establishing a timing side-channel. Instead of timing system calls indiscriminately, SLUBStick distinguishes two categories of objects: (i) objects used solely for measuring the system call delay (referred to as "timing objects" in this work), and (ii) objects used to modify the state of the SLUB allocator, which remain allocated (referred to as "persistent objects"). Timing objects are allocated by system calls that can enter an error state, leading to their immediate deallocation upon function return. Therefore, the allocation workflow involves a timing object allocation, which is used to measure the time required to allocate the object while minimizing non-allocation tasks at the kernel level, followed by a persistent object allocation to make the change effective from the SLUB side. This dual-object strategy significantly reduces noise in timing measurements. SLUBStick uses this technique to reveal the SLUB state, recycle the vulnerable slab page and mount a cross-cache attack for temporal vulnerabilities, focusing on page tables as target objects [9, 12].

Therefore, previous work on cross-cache massaging either focuses on temporal vulnerabilities [9, 12], or establishes a contiguous layout for out-of-bounds exploitation but without inferring the internal state of the allocator [13, 34, 45–47], yielding an unreliable solution [27–29]. None of the existing work explores or makes use of the inner interaction between PCP lists and Page Frame Allocator, nor provides a reliable solution to cross-cache overflows.

B. SLUB Mitigations

Mainline mitigations for SLUB vulnerabilities primarily address in-cache feng-shui and corruption. Among notable mitigations, `SLAB_FREELIST_RANDOM` [23] randomizes the order of freed slots returned upon allocation, while `SLAB_FREELIST_HARDENED` (free-list pointer obfuscation) [22] manipulates free-list next pointers to prevent trivial pointer leaks and corruption. Other mitigations include `RANDOM_KMALLOC_CACHES` [21] and `SLAB_BUCKETS` [24] which probabilistically address heap spray and grooming attacks [14], targeting fixed-sized and variable-sized heap objects, respectively. In particular, `SLAB_BUCKETS` requires explicit use by the programmer through a specific API, while `RANDOM_KMALLOC_CACHES`, when enabled, affects all `kmalloc-*` caches transparently. Finally, *heap zeroing* [48] ensures that object slots are zero-initialized upon allocation and deallocation, while *structure layout randomization* [49], implemented as a GCC plugin [50], randomizes the order of fields in C structs after boot, making it difficult for attackers to predict offsets between fields and the start of a corruption target. These measures generally impose negligible performance overhead and have been integrated into the upstream Linux kernel.

For cross-cache memory massaging, the only defense mechanism proposed for contribution to the upstream kernel is `SLAB_VIRTUAL` [26]. The goal of `SLAB_VIRTUAL` is to deterministically mitigate cross-cache attacks based on page recycling by ensuring that virtual addresses used by one `kmem_cache` are never reassigned to a different `kmem_cache`. However, this mitigation strategy remains under consideration and has not been merged upstream due to concerns raised by Linus Torvalds and Ingo Molnár regarding performance and the lack of DMA support [51].

We must note that the previously cited `RANDOM_KMALLOC_CACHES` [21] and `SLAB_BUCKETS` [24] potentially make cross-cache attacks more difficult in practice: attackers need to find suitable vulnerable and spray objects sharing the same slab cache to successfully mount the page spraying phase of the attack. However, as of kernel version 6.8 (the version used in our experiments) `SLAB_BUCKETS` was not yet merged as a mainline solution; in subsequent kernel versions, the mitigation applies only to a small subset of objects. Instead, the `RANDOM_KMALLOC_CACHES` mitigation is not enabled by default in most Linux distributions.

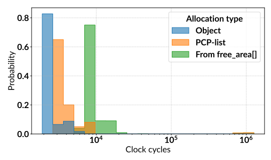
AUTOSLAB [52] is a proprietary solution developed by grsecurity. To prevent in-cache corruption, AUTOSLAB separates object allocations by type, ensuring that different object types do not share the same cache and providing each generic allocation site calling `k*alloc*` with its own dedicated memory cache. Additionally, AUTOSLAB randomizes the starting offset of the first object in each slab to reduce the likelihood of successful cross-cache corruption. However, separating kernel objects per slab also makes it easier for attackers to recycle memory from the Buddy Allocator side, potentially reusing the same pages [53].

To highlight the critical implications of cross-cache attacks on kernel security, we evaluate the efficacy of our memory massaging using Linux kernel version 6.8 (see Section V) by selectively activating `SLAB_FREELIST_RANDOM`, `SLAB_VIRTUAL` and `SLAB_FREELIST_HARDENED`. In particular, we demonstrate a bypass for both `SLAB_VIRTUAL` and `SLAB_FREELIST_RANDOM` mitigations (Section V-C). Then, in all CVEs evaluated in Section V-D, none of the implemented attacks rely on free-list pointer leaks, providing effectiveness against `SLAB_FREELIST_HARDENED` as well.

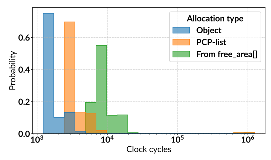
IV. Approach

The goal of PCPLost is to assist an attacker in forcing adjacency between two objects (*vulnerable* and *target*) that belong to different caches. To achieve this, we utilize a timing side-channel to determine whether a page allocation was served from a PCP list or the zoned `free_area` (Section II-B), assuming that these operations exhibit significant timing differences, as discussed in Section IV-A.

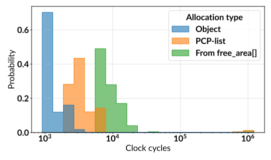
We provide two strategies based on whether the target and vulnerable object caches use same-order or differing-order pages, referred to as "same-order cross-cache" and "cross-order cross-cache" attacks, respectively. Each strategy leverages distinct timing side-channels and incorporates additional *spray* objects, which can either match or differ in type from the vulnerable and target objects. These spray objects serve a crucial purpose: they are utilized to force the SLUB allocator to exhaust its available slots when requesting new pages.



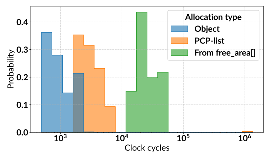
(a) `kmalloc-4k`; slabs are order-3 pages.



(b) `kmalloc-2k`; slabs are order-3 pages.



(c) `kmalloc-1k`; slabs are order-3 pages.



(d) `kmalloc-256`; slabs are order-1 pages.

Fig. 3: Timing values for "timing objects" allocations in `kmalloc-*` caches. Allocation types "PCP-list" and "From `free_area[]`" refer to, respectively, allocations done through `rmqueue_pcplist()` and `rmqueue_bulk()`.

Both strategies leverage *timing* objects [9] and *persistent* objects. Timing objects serve solely as probes and are immediately freed; they are utilized to infer the underlying SLUB state from the execution time of the system call responsible for their allocation. Persistent objects are employed to change the SLUB state permanently. More specifically, PCPLost operates through a loop where, in each iteration, it first collects a timing sample using timing objects. After that, it allocates a persistent object (either vulnerable, target, or spray), which effectively drains the free space; we refer to this sequence as *probe and drain*.

In subsequent sections, we denote N as the number of objects per slab under investigation and refer to n_v and n_t as the page orders requested by the vulnerable and target caches, respectively.

A. Timing side-channels

The proposed massaging technique is guided by hypotheses regarding the potential state of SLUB. These hypotheses are refined over time by observing internal SLUB events, which are inferred from the timing of specific system calls (side-channels) due to the impossibility of direct observation. Specifically, we focus on events related to invocations of `rmqueue_pcplist()` and `rmqueue_bulk()` following the allocation of a timing object.

To recap from Section II, these invocations occur when the SLUB exhausts available slabs and needs another slab to store a new object. The kernel attempts to allocate a new slab by retrieving a free page from the PCP list via `rmqueue_pcplist()`. If a suitable page is found, it is removed and returned. Otherwise, `rmqueue_bulk()` is used, reserving multiple free pages from the zone's `free_area` and moving them to the PCP list. When necessary, a page is split into two buddies.

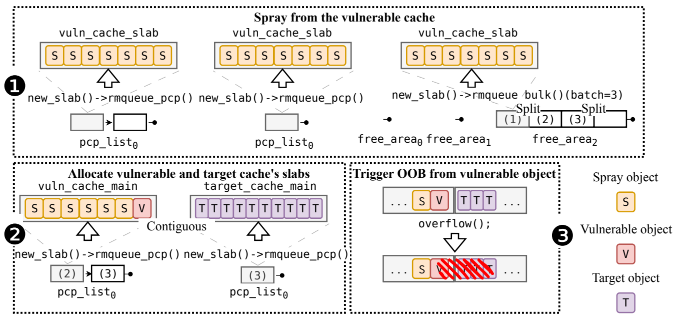

Spray from the vulnerable cache
vuln_cache_slab
new_slab()->rmqueue_pcp()
pcp_list0
new_slab()->rmqueue_bulk()(batch=3)
Split
free_area0 free_area1 free_area2
Allocate vulnerable and target cache's slabs
vuln_cache_main
target_cache_main
Contiguous
Trigger OOB from vulnerable object
overflow();
Spray object
Vulnerable object
Target object

Fig. 4: Overview of the same-order cross-cache massaging. We use `pcp_list`$_i$ to denote the PCP list of order i and `free_area`$_j$ to indicate the `free_area` list of order j. Here, vulnerable and target cache both request order-0 pages. The lower section of each stage illustrates the states of relevant PCP lists and `free_area` lists. Stage ❶ shows the initial spray with vulnerable (spray) objects splitting pages at higher orders, while stages ❷ and ❸ show, respectively, vulnerable and target slabs allocation (contiguously) and successful target object corruption.

This leads to three timings measured from the allocated timing objects. (i) "fast" allocation from SLUB without accessing the PCP list; (ii) "slower" allocation requesting a new page via `rmqueue_pcplist()`; (iii) "slowest" allocation, triggering `rmqueue_bulk()` from the zone `free_area`, potentially involving a page split.

As shown in Figure 3, the values of these timing profiles allows to discriminate between these events. While previous literature has exploited the difference between "fast" and "slower/slowest" [9, 14], we are the first to utilize the difference between "slower" and "slowest" for cross-cache massaging. This three-events distinction is pivotal to comprehensively support page-adjacency attacks, for same-order and cross-order caches. By optimizing to eliminate false positives, as detailed in Section IV-D, we achieve a highly reliable massaging strategy in most cases.

The following sections demonstrate how these timing side-channels can be used to manipulate the SLUB memory layout and execute a successful page-adjacency cross-cache attack.

B. Same-order cross-cache massaging

In same-order attacks, both the vulnerable and target caches use same-order pages ($n_v = n_t = k$). The strategy proceeds as follows. First, we probe and drain the vulnerable cache with persistent objects to trigger a refill of the order-k PCP list from the `free_area` (since the pages inserted in the PCP list will be the result of a split, they will be contiguous). Next, we coordinate a sequence of allocations to ensure that both the vulnerable and target caches acquire one of these two contiguous pages. This guarantees that a linear out-of-bounds (OOB) read or write operation on the vulnerable object affects the contents of the target object.

Let us consider a detailed example with $k = 0$ (see Figure 4). The attack begins with a *probe and drain* iteration to drain both the main and partial slabs of the vulnerable cache. This forces the kernel to repeatedly trigger `new_slab()`, which, in turn, activates `rmqueue_pcplist()` until the order-k PCP list becomes fully depleted (stage ❶, first two `rmqueue_pcplist()` in Figure 4).

When a subsequent allocation occurs, `rmqueue_bulk()` engages with the `free_area` to reserve `batch` pages (stage ❶, the last step in Figure 4). The kernel prioritizes fulfilling this request through either immediate next-order ($k+1$) pages or higher-order alternatives, depending on current availability. By repeating this process, attackers can increase the likelihood of obtaining adjacent page batches, a critical requirement for successful cross-cache attacks via page adjacency, as detailed in Section IV-B1. In our specific scenario, `rmqueue_bulk()` acquires three pages (`(2)` and `(3)`) from both order-0 and order-1 `free_area` lists. These pages constitute two buddies, indicating that they are physically contiguous in memory.

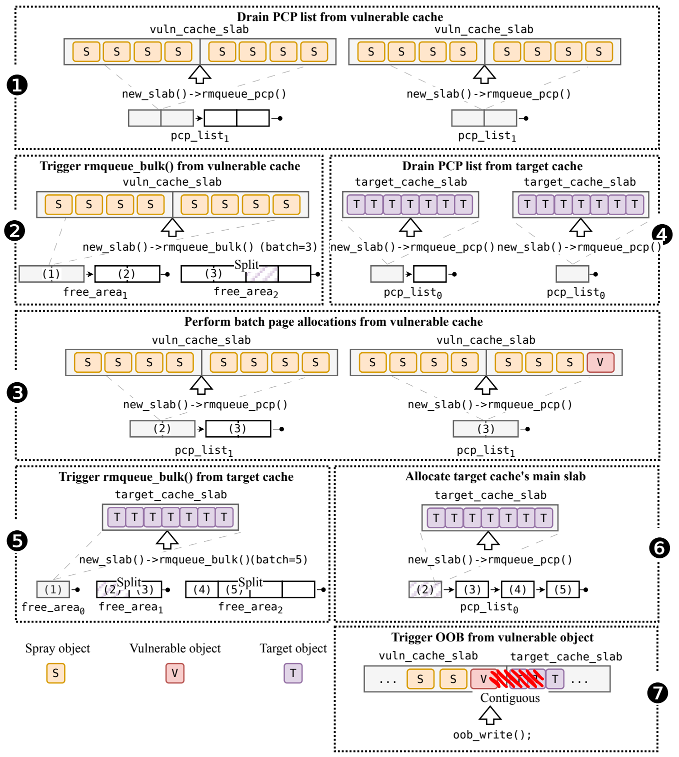
After completing the spraying session, we *probe and drain* the vulnerable cache again (stage ❷ in Figure 4) until a new order-k page allocation from the PCP list is detected. This page is acquired by the vulnerable cache as its main slab. We proceed to fill it with $N - 1$ objects, which can be either vulnerable or spray objects, followed by a single allocation of a vulnerable object located at the end of the slab.

If the spraying phase at stage ❶ was successful, this object will be contiguous to the first order-k page in the PCP list, which has not yet been allocated (page `(3)` in Figure 4). We thus *probe and drain* the target cache until that page is allocated and fill it with target objects. Crucially, the first target object allocated will be adjacent to the vulnerable object that was allocated earlier, allowing an OOB vulnerability to be triggered (stage ❸ in Figure 4).

1) The preliminary "probe and drain" phase: The first stage of same-order massaging involves "probe and drain" (page spraying). This round of allocations aims to drain pages from the PCP list and split them at higher orders. We empirically validate that using a base value of 512, adjusted based on the page order via `base_value >> page_order`, allows us to determine the number of page requests – inferred through the side-channel (Section IV-A) – that consistently achieve high reliability across all same-order massaging scenarios in generic caches, as shown by our experiments (see Table II and Table IX in Appendix C-B). Intuitively, the number of pages to drain must decrease as the page order increases: the higher the page order, the smaller the `batch` value (see Table VI), which results in a smaller average number of pages to drain in the PCP list. Consequently, performing this quantity of page allocations almost always results in splits at higher orders. This process ensures that batches of contiguous pages are moved to the corresponding PCP list, which is crucial for both cross-cache adjacency layout and facilitating same-order cross-cache attacks.

C. Cross-order cross-cache massaging

In the cross-order attack scenario, vulnerable and target caches request different page orders to initialize new slabs



Drain PCP list from vulnerable cache
vuln_cache_slab
new_slab()->rmqueue_pcp()
pcp_list$_1$
Trigger rmqueue_bulk() from vulnerable cache
new_slab()->rmqueue_bulk() (batch=3)
Split
free_area$_1$
free_area$_2$
Drain PCP list from target cache
target_cache_slab
pcp_list$_0$
Perform batch page allocations from vulnerable cache
Trigger rmqueue_bulk() from target cache
new_slab()->rmqueue_bulk()(batch=5)
free_area$_0$
Allocate target cache's main slab
Trigger OOB from vulnerable object
Contiguous
oob_write();
Spray object
Vulnerable object
Target object

Fig. 5: Overview of the cross-order cross-cache massaging. Here, the vulnerable cache requests order-1 pages, while the target cache requests order-0 pages, thereby showing the $n_v > n_t$ case, where n_v and n_t are the vulnerable and target cache orders. The lower section of each stage illustrates the states of relevant PCP lists and `free_area` lists. Stages ❶, ❷ and ❸ show vulnerable cache massaging, while stages ❹, ❺ and ❻ show target cache massaging, placing the two contiguously in memory. Stage ❼ shows successful corruption.

($n_v \neq n_t$). Therefore, within this massaging strategy, two possible scenarios may arise: (i) the vulnerable cache requests a page order greater than the target cache ($n_v > n_t$); (ii) the vulnerable cache requests a smaller page order compared to the target cache ($n_v < n_t$).

In both cases, we exhaust all slots in the vulnerable cache and all PCP list pages of order n_v through allocations. This process forces the zoned Buddy Allocator (via `rmqueue_bulk()`) to execute a page split while reserving `batch` pages of order n_v. As a consequence, the last page in the PCP list from the batch allocation is typically an L-buddy of the higher-order split, while the corresponding R-buddy is located in the `free_area`. Next, we deplete the PCP list through vulnerable (or spray) object allocations until we reach the L-buddy, at which point we allocate a target object, ensuring that the R-buddy is utilized by the target cache. This ensures that an OOB in the vulnerable object (L-buddy) will affect the target object (R-buddy).

While the high-level attack strategy is similar in both cases, there are several subtleties that we will address in the following sections.

1) The $n_v > n_t$ case: In this case, we *probe and drain* the vulnerable cache (stage ❶) until a single call to `rmqueue_bulk()` is detected (stage ❷ in Figure 5 with $n_v = 1$ and $n_t = 0$). We then perform $N \times$ `batch` $- 1$ (spray) object allocations to consume those pages, finally using the last slot for the vulnerable object (stage ❸, lower section of each stage illustrates the states of relevant PCP lists and `free_area` lists). Given the sparse use of `free_area` lists of orders supported by the PCP optimization, the last page in the batch allocation was likely involved in a high-order page split. Hence, it is the L-buddy of another contiguous page that got moved in the `free_area` and is now ready to be allocated by the next call to `rmqueue_bulk()`; for example in Figure 5, page `(3)` – split in stage ❷ and allocated as the main slab of the vulnerable cache in stage ❸ – is the L-buddy of the page identified by the hatched rectangle in stage ❷.

We now *probe and drain* the target cache (stage ❹ in Figure 5) until a call to `rmqueue_bulk()` is detected (stage ❺). Such an invocation reserves `batch` pages starting from the order-n_t `free_area` list. For the reasons previously discussed, the reservation of `batch` pages by `rmqueue_bulk()` likely taps into a `free_area` list whose order is $\geq n_v$, including the R-buddy page that is contiguous to the previously allocated vulnerable cache's main slab (stage ❻).

If this step succeeds, vulnerable and target slabs are contiguous and allow the desired OOB (stage ❼).

While in Figure 5, $\Delta = \|n_v - n_t\| = 1$, the massaging strategy presented above generalizes to $\Delta > 1$. The only modification to the strategy pertains to the massaging of the target cache: PCPLOST needs to cause allocation requests from the lower target cache order n_t to cause page splits at higher order n_v. This can be reliably accomplished via multiple invocations of `rmqueue_bulk()`, as we shall see in Section V.

2) The $n_v < n_t$ case: To describe this case, let us consider the case $n_v = 0$ and $n_t = 1$ as shown in Figure 6. As before we start with *probing and draining* the vulnerable cache until detecting a call to `rmqueue_bulk()` (stage ❷). Also in this case, the `rmqueue_bulk()` function likely triggers page splits at orders $> n_v$. For instance, in Figure 6, the page split at stage ❷ involves order-2 pages, while the vulnerable cache requests order-0 pages. These *probe and drain* operations can be continued until `rmqueue_bulk()` taps into the order-($n_t + 1$) lists in the `free_area`. At this point, any split at an order $> n_t$ produces an L-buddy page (recursively split afterwards until reaching order n_v) and an R-buddy page that gets moved into the `free_area`. At this point we drain all pages involved in the batch allocation in the vulnerable cache (stage ❸) through $N \times$ `batch` $- 1$ spray object allocations followed by one vulnerable object allocation.

We then *probe and drain* the target cache (stage ❹) up until detecting an `rmqueue_bulk()` (stage ❺). This results in `batch` pages from `free_area` being reserved, and with the first one being returned to the caller. The latter likely corresponds to the R-buddy page previously split by vulnerable cache allocations (stage ❷). As a result, vulnerable and target

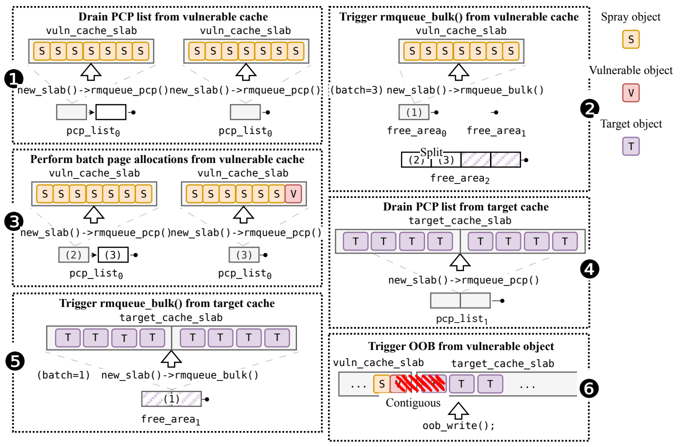

Drain PCP list from vulnerable cache
vuln_cache_slab
new_slab()->rmqueue_pcp()
pcp_list$_0$
Trigger rmqueue_bulk() from vulnerable cache
(batch=3) new_slab()->rmqueue_bulk()
free_area$_0$
free_area$_1$
Split
free_area$_2$
Perform batch page allocations from vulnerable cache
Drain PCP list from target cache
target_cache_slab
pcp_list$_1$
Trigger rmqueue_bulk() from target cache
(batch=1) new_slab()->rmqueue_bulk()
Trigger OOB from vulnerable object
Contiguous
oob_write();
Spray object
Vulnerable object
Target object

Fig. 6: Overview of the cross-order cross-cache massaging. In this example, the vulnerable cache requests order-0 pages, while the target cache requests order-1 pages, thereby showing the $n_v < n_t$ case, where n_v and n_t are, respectively, the vulnerable and target cache orders. Stages ❶, ❷ and ❸ show the vulnerable cache massaging, while stages ❹ and ❺ show the target cache massaging, placing the two contiguously in memory. Stage ❻ shows the successful corruption.

slabs would now be contiguous: any OOB write would corrupt the attacker-chosen target object (stage ❻).

In Figure 6, $\Delta = \|n_v - n_t\| = 1$. However, the massaging strategy presented above generalizes to $\Delta > 1$, showing varying success rates. Indeed, when $\Delta > 1$, an attacker can spray page allocations to trigger calls to `rmqueue_bulk()` to eventually reach the order-n_t `free_area` list. Although an attacker can detect calls to `rmqueue_bulk()`, the number of such calls they should force the kernel to perform in order to reach the order-n_t list is not known. To overcome this uncertainty, the attacker could spray vulnerable slabs allocations multiple times, knowing that they would eventually reach order-n_t pages in the `free_area`. The imprecision resulting from spraying vulnerable objects compels attackers to trigger the vulnerability multiple times. While object spraying was effective for target objects, it proves less so for vulnerable ones. Indeed, repeatedly triggering the vulnerability substantially increases the likelihood of causing a kernel crash.

However, this scenario where $n_v < n_t$ and $\Delta = \|n_v - n_t\| > 1$ represents an edge case. Attackers can select target objects to either match the vulnerable cache's order, reducing it to the $n_v = n_t$ situation described in Section IV-B, or opt for a more favorable configuration.

D. Minimizing false positives

To enhance the accuracy of the detection of the two events (`rmqueue_pcplist()` and `rmqueue_bulk()`) by our side-channel and minimize false positives, we adopt the following strategy: instead of relying on a single timing sample within a given range as evidence of an allocation event, we test for recurrence after spraying sufficient allocations to refill the area supposedly provided by the original event. If subsequent timing samples match expectations, we infer with high probability that the allocation event (from either the PCP list or `free_area`) occurred. By utilizing this strategy, we minimize false positives, but we may discard some viable hits. Apart from potentially allocating more than strictly necessary, discarding good hits does not pose a significant issue from the PCPLOST massaging perspective. Inducing a slightly higher memory pressure can be considered an acceptable cost, given a higher overall success rate. In Table VII, we provide an evaluation of false positive and negative rates.

E. Exploiting temporal vulnerabilities

PCPLOST produces a contiguous layout between vulnerable and target caches in the SLUB allocator. This contiguous layout is favorable for OOB vulnerabilities, where the attacker can corrupt the target object (stored in the target cache) through a linear overflow from the vulnerable one. However, by means of *pivoting* techniques, PCPLOST supports temporal as well, broadening its applicability. The overall strategy is to establish page-adjacency between vulnerable (to UAF or DF) and target caches, to then pivot from temporal bug to an OOB primitive, and exploit the contiguous layout provided by PCPLOST to corrupt the target object.

We define *pivot object* as a specific target object that enables a pivot from a temporal vulnerability (UAF or DF) to a spatial vulnerability (OOB) by corrupting one of its fields. The pivot object may be either from the same cache as the vulnerable object (*in-cache pivoting*) or from a different cache (*cross-cache pivoting*).

a) UAF vulnerabilities: To pivot from a UAF vulnerability to an OOB vulnerability, we can manipulate the length field of suitable pivot objects [54] during the "use" phase of the UAF (`struct msg_msg` is a widely used example of pivot object). Intuitively, the pivot is obtained by (i) using PCPLOST to prepare cross-cache adjacency between UAF-vulnerable and target cache pages (ii) allocating the vulnerable object and immediately freeing it, holding a dangling pointer to it. After, (iii) allocate a pivot object in the same slot as the vulnerable object. Note that depending on the type of pivot (in-cache or cross-cache), this step may require releasing the page to the buddy allocator, and later reclaiming the page for a cache of pivot objects. Then, (iv) corrupt the length field of the pivot object through the UAF and trigger an OOB write. If the pivot object occupies the last position in the cache, the OOB operation will affect objects in the target cache, finalising the attack.

b) DF vulnerabilities: Pivoting from DF to OOB involves transforming the DF vulnerability into a UAF vulnerability [9] and subsequently applying the previous strategy. Specifically, the attacker allocates the DF object to obtain pointer p_1, frees it, and then immediately reallocates it to acquire p_2 (which points to the same address). Freeing p_1 a second time leaves p_2 dangling, allowing the application of the previous strategy from step (iii).

In Section V-B, we discuss the experimental results and the primary challenges associated with pivoting UAF and DF into OOB.

V. Evaluation

In this section, we discuss the evaluation criteria to establish the effectiveness of PCPLost and present experiments that address four research questions (RQs):

- **RQ1**: How reliable is PCPLost in achieving the desired cross-cache memory layout for OOB vulnerabilities?
- **RQ2**: Can PCPLost be broadened to support other classes of vulnerabilities like UAF and DF?
- **RQ3**: Can PCPLost bypass cross-cache defenses proposed, or already accepted, as mainline solutions?
- **RQ4**: What is the applicability of the PCPLost technique to real-world vulnerabilities?

Experiments were conducted on a 64-bit x86 QEMU virtual machine with 12 CPU cores and 32 GB of RAM, running a Debian installation (trixie/sid) and Linux kernel version 6.8.

The host machine is a 64-bit x86 architecture equipped with a Xeon Gold 6210U CPU, featuring 20 cores per socket and 2 threads per core, and is configured with 394 GB of RAM.

A. *Cross-cache massaging*

RQ1: How reliable is PCPLost in achieving the desired cross-cache memory layout for OOB vulnerabilities?

To address this question, we evaluate the success rates of both same-order and cross-order cross-cache memory massaging on generic SLUB caches under two assumptions: "adjacency-only" and "adjacency + object alignment".

In the "adjacency-only" scenario, memory massaging is deemed successful if a simple page adjacency is achieved between the vulnerable and target caches. In the "adjacency + object alignment" scenario, a successful memory massaging requires not only contiguous caches but also the precise placement of the vulnerable object at the end of its corresponding cache's main slab (with `SLAB_FREELIST_RANDOM` disabled). These two massaging settings are evaluated under three operating conditions: "Idle with CPU pinning", "Idle without CPU pinning", and "Noise with CPU pinning". The first two cases correspond to an Idle CPU where the process is or isn't pinned to a CPU core to prevent context switching. "Noise with CPU pinning" corresponds to a system under a heavy workload generated by `stress-ng` in the background, creating pressure on SLUB caches, while the massaging process is pinned to a CPU core. We do not consider the scenario of background noise without CPU pinning, as an unprivileged user can always pin a process to a specific CPU core when the system is noisy.

To conduct our experiments, we use 40 distinct QEMU sessions, performing 20 attack runs per session, resulting in a total of 800 runs. We record the success or failure of each run and model our sample distribution as a binomial distribution. Finally, we compute the 95% confidence interval (95%-CI) using the Wilson interval score for each scenario described above.

Table II presents the reliability results for same-order ($n_v = n_t$) and cross-order massaging for the case ($n_v > n_t$). Appendix C provides reliability results for the opposite case ($n_v < n_t$) and offers a more extensive evaluation of the same-order case ($n_v = n_t$) across all generic kmem-caches, as Table II includes only a subset of these caches.

In the following three paragraphs, we discuss the experimental results for the cases mentioned above.

a) $n_v = n_t$: The same-order case is the most common scenario in exploitation, as attackers can often choose both the target object and cache, selecting a target with the same page order as the vulnerable cache. According to Table II (same-order case), the success rate consistently exceeds 90% across all scenarios, regardless of SLUB cache characteristics, such as the number of objects per slab or page order (see Table V). The results in Table IX from Appendix C further demonstrate that generic caches exhibit high reliability. We note that background noise has a minimal impact on the success rate due to the reduced complexity of the memory massaging process, as the only requirement is to obtain two contiguous pages from the PCP list and allocate them sequentially with vulnerable and target caches.

b) $n_v > n_t$: Table II shows that the success rate for cross-order massaging remains consistently high with CPU pinning under "Idle" and "Noise" operating conditions (around 90%) while it decreases up to 41% without CPU pinning. This suggests that descheduling can affect the success rate, as other processes may interact with the `free_area` while the attack process is sleeping, thereby interferring with key massaging operations.

c) $n_v < n_t$: The overall success rate of the massaging, as shown in Table Xc in Appendix C, decreases significantly. This cross-order scenario exhibits increased complexity and inherent nondeterminism due to the challenges attackers face in predicting when to stop allocating from the vulnerable cache (see Section IV-C). Furthermore, the greater the distance between the two orders $\Delta = \|n_v - n_t\|$, the higher the uncertainty and the smaller the success rate, reaching as low as 20%. While this remains the least reliable case, the attackers can avoid it by selecting a target object in the $n_v \geq n_t$ cases.

B. *Cross-cache temporal to spatial pivoting*

RQ2: Can PCPLost be broadened to support other classes of vulnerabilities like UAF and DF?

We demonstrate that starting from a UAF or DF vulnerability, by selecting an appropriate pivot object, this type of vulnerability can be transformed into an OOB vulnerability, exploitable with PCPLost, a technique known as pivoting. To test this, we employ a custom kernel module to introduce a temporal vulnerability and apply PCPLost on top. For an evaluation on real-world vulnerabilities, we refer to RQ4.

a) UAF to OOB pivoting: To validate the effectiveness of the UAF to OOB pivoting, we use a custom kernel module to create a synthetic UAF vulnerability and use it as the vulnerable object. As pivot object we use the `struct msg_msg` kernel object in `kmalloc-cg-*` caches.

Vulnerable cache	Target cache	Success rate											
		Adjacency only						Adjacency + object alignment					
		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)	
		Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI
$(n_v = n_t)$													
`kmalloc-16`	`kmalloc-64`	98.75	$(97.98, 99.23)$	99.14	$(98.47, 99.52)$	97.26	$(96.22, 98.02)$	98.83	$(98.07, 99.29)$	98.51	$(97.69, 99.05)$	97.42	$(96.40, 98.16)$
`kmalloc-16`	`kmalloc-128`	98.59	$(97.79, 99.10)$	98.83	$(98.07, 99.29)$	96.48	$(95.33, 97.36)$	98.35	$(97.50, 98.92)$	98.91	$(98.17, 99.35)$	96.80	$(95.68, 97.63)$
`kmalloc-64`	`kmalloc-16`	98.59	$(97.79, 99.11)$	94.60	$(93.23, 95.72)$	92.50	$(90.92, 93.82)$	95.78	$(94.54, 96.75)$	96.09	$(94.89, 97.02)$	92.26	$(90.67, 93.60)$
`kmalloc-64`	`kmalloc-128`	97.73	$(96.76, 98.42)$	96.09	$(94.89, 97.02)$	91.33	$(89.66, 92.75)$	96.80	$(95.68, 97.63)$	95.62	$(94.36, 96.61)$	90.94	$(89.24, 92.39)$
`kmalloc-128`	`kmalloc-16`	99.68	$(99.20, 99.88)$	99.69	$(99.20, 99.88)$	98.51	$(97.69, 99.04)$	99.84	$(99.43, 99.96)$	99.53	$(98.98, 99.78)$	98.28	$(97.41, 98.86)$
`kmalloc-128`	`kmalloc-64`	99.37	$(98.77, 99.68)$	99.21	$(98.57, 99.57)$	98.75	$(97.98, 99.22)$	99.61	$(99.10, 99.83)$	99.37	$(98.77, 99.68)$	98.28	$(97.41, 98.86)$
$(n_v > n_t)$													
`kmalloc-2k`	`kmalloc-16`	95.15	$(93.84, 96.20)$	47.66	$(44.93, 50.39)$	87.34	$(85.41, 89.05)$	94.30	$(92.89, 95.43)$	47.96	$(45.24, 50.71)$	87.58	$(85.66, 89.27)$
`kmalloc-2k`	`kmalloc-64`	90.55	$(88.82, 92.03)$	43.91	$(41.21, 46.64)$	91.87	$(90.25, 93.25)$	91.88	$(90.25, 93.25)$	42.50	$(39.81, 45.23)$	91.48	$(89.83, 92.89)$
`kmalloc-2k`	`kmalloc-128`	96.48	$(95.33, 97.36)$	52.81	$(50.07, 55.53)$	87.58	$(85.66, 89.27)$	95.47	$(94.19, 96.48)$	48.98	$(46.25, 51.72)$	88.12	$(86.23, 89.78)$
`kmalloc-4k`	`kmalloc-16`	96.01	$(94.90, 96.96)$	55.86	$(53.12, 58.56)$	87.89	$(85.99, 89.56)$	95.94	$(94.71, 96.89)$	50.62	$(47.89, 53.36)$	87.97	$(86.07, 89.64)$
`kmalloc-4k`	`kmalloc-64`	93.20	$(91.69, 94.46)$	48.91	$(46.17, 51.64)$	91.48	$(89.83, 92.89)$	93.20	$(91.69, 94.46)$	49.45	$(46.72, 52.19)$	92.42	$(90.84, 93.75)$
`kmalloc-4k`	`kmalloc-128`	91.09	$(89.41, 92.53)$	41.33	$(38.66, 44.05)$	92.42	$(90.84, 93.75)$	90.31	$(88.57, 91.81)$	44.22	$(41.52, 46.95)$	90.08	$(88.32, 91.60)$

TABLE II: Success rate for cross-cache massaging. We evaluate the success rate of the same-order ($n_v = n_t$) and cross-order massaging ($n_v > n_t$) in three cases: system in IDLE (no external stress workloads) with and without pinning to a CPU-core, and under heavy workload generated by `stress-ng` while pinning.

In this preliminary experiment, we perform the UAF to OOB pivoting by corrupting the `m_ts` field (message text size) of `struct msg_msg`. After obtaining an out-of-bounds primitive with `struct msg_msg`, we mount a reliable cross-cache attack, targeting `struct shm_file_data`, allocated in `kmalloc-32`.

The challenging aspect of the pivoting phase revolves around identifying a suitable pivot object for the vulnerability at hand. From our experiments, an ideal type of object for this phase is one that has a "length" field used in the primitive method provided by the object, which is later used in a `memcpy` operation. The UAF vulnerability must allow an attacker to overwrite with attacker-controlled data the memory location containing the length field. If these requirements are met, due to the broad applicability of timing objects [9], and given the high success rate of PCPLost, the reliability of the pivoting phase depends only on the specific vulnerability at hand.

b) DF to OOB pivoting: For the DF to OOB pivoting, we perform an evaluation analogous to the one of the UAF to OOB case, targeting the same objects and caches, with the additional pivot from DF to UAF using our custom kernel module. We rely on the fact that most DF vulnerabilities can be turned into UAFs: by allocating an object after the first free, the subsequent free creates a dangling pointer to that object. Following the preliminary pivoting to UAF, we apply the UAF-OOB pivoting. Since the DF-OOB pivoting is a double pivot from DF to UAF and finally OOB, the same challenges for the UAF-OOB pivoting apply. In particular, due to the broad applicability of timing objects and the high success rate of PCPLost, the reliability of the DF-OOB pivoting phase depends only on the specific vulnerability at hand, as the UAF-OOB pivoting case.

C. SLUB defenses bypass

RQ3: Can PCPLost bypass cross-cache defenses proposed, or already accepted, as mainline solutions?

We analyze the effectiveness of PCPLost against `SLAB_VIRTUAL` (proposed mainline mitigation against cross-cache attacks, Section V-C1) and `SLAB_FREELIST_RANDOM` (which randomizes the order of allocations within a slab, Section V-C2).

1) Bypassing `SLAB_VIRTUAL`: `SLAB_VIRTUAL` is a hardening solution proposed in 2023 on the Linux Kernel Mailing List [26]. Its goal is to mitigate cross-cache attacks arising from page recycling by ensuring that virtual addresses are never reassigned to slabs from different `kmem_cache`s. When this mitigation is enabled, kernel logical addresses are no longer directly mapped.

a) Spatial vulnerabilities: Our experiments show that PCPLost exhibits high reliability (over 90%) in producing favorable OOB layout both in same-order and cross-order massaging when `SLAB_VIRTUAL` is enabled (see Table Xa, Appendix C).

The success rate is even higher in some cases (compared to a vanilla kernel): `SLAB_VIRTUAL` has a highly predictable allocation strategy, where slabs gets reserved from a virtually contiguous region of memory. When the free slab pool is exhausted and a new slab is allocated, the addresses involved remain virtually contiguous. Allocating a vulnerable slab followed immediately by a target slab therefore creates a contiguous virtual layout with very high probability.

b) Temporal vulnerabilities : It is also possible to employ the pivoting technique described in Section IV-E to exploit temporal vulnerabilities: this shows how PCPLost is effective against `SLAB_VIRTUAL`. To bypass `SLAB_VIRTUAL`, only the in-cache pivoting strategy can be employed: cross-cache pivoting is thwarted by the fact that `SLAB_VIRTUAL` prevents page reuse across caches by design. For such in-cache variant, vulnerable and pivot object must reside in the same SLUB cache, but the eventual target object is cross-cache.

The strategy proceeds as follows: (i) PCPLost ensures a contiguous layout between the vulnerable and target slabs. (ii) We allocate $N-1$ spray objects and one vulnerable object, where N is the number of objects in the vulnerable slab. Afterwards, the vulnerable object is freed while maintaining a dangling pointer to it. (iii) We then allocate the pivot object, using the recently freed slot in the vulnerable cache due to the LIFO nature of SLUB free-lists. (iv) Through the dangling pointer, the length field of the pivot object is corrupted to

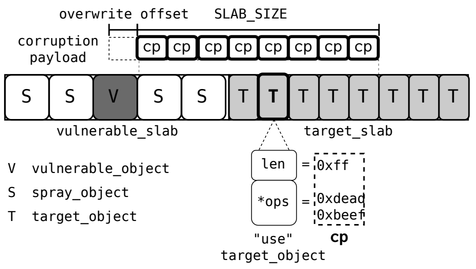


Fig. 7: Overview of the `SLAB_FREELIST_RANDOM` bypass. The payload is structured in a way that the corruption pattern (`cp`) overlaps with target objects.

transition from temporal to spatial vulnerability. (v) Due to the contiguous layout established in (i), OOB writes from the vulnerable cache can corrupt the target object.

We evaluate the effectiveness of the `SLAB_VIRTUAL` bypass strategy using both synthetic and real-world out-of-bounds (temporal and spatial) vulnerabilities. In Table III, we provide our real-world experiments, where we explicitly highlight the scenarios where `SLAB_VIRTUAL` is successfully bypassed.

2) Bypassing `SLAB_FREELIST_RANDOM`: The PCPLOST technique achieves a cross-cache layout with the target slab placed right after the vulnerable slab. Once this layout is achieved, it is in general very easy to place to vulnerable object adjacent to the target object, by ensuring that the vulnerable object is the last allocation within the vulnerable slab, and the target object is the first allocation of the adjacent vulnerable slab. However, this is more challenging to achieve when the `SLAB_FREELIST_RANDOM` mitigation [23] is in use. The `SLAB_FREELIST_RANDOM` mitigation randomizes the order of allocations within a slab, thereby making it difficult for the attacker to be able to infer the first (or last) allocation. PSPRAY bypasses this mitigation via a side-channel [14]. Unfortunately this in-cache bypass is not applicable for our cross-cache attack. We refer to Section VI-E for an explanation.

The natural approach for an attacker to bypass `SLAB_FREELIST_RANDOM` would be to spray vulnerable objects into the vulnerable slab, and target objects into the target slab. However, because the attacker does not know which vulnerable object is placed last, it would have to try multiple times. If the vulnerability is an out-of-bounds read this is an acceptable solution. However, in the case of an out-of-bounds write, this overwrites other vulnerable objects, meaning that a subsequent trigger of such an overwritten object will likely lead to a kernel panic. We propose here a generic solution to avoid this issue.

If the out-of-bounds write vulnerability allows the attacker to write at least `SLAB_SIZE+overwrite_offset` bytes beyond the bounds of the vulnerable object, as shown in Figure 7, whereby `SLAB_SIZE` is the size of the vulnerable slab in bytes, the attacker can obtain a successful cross-cache attack even when `SLAB_FREELIST_RANDOM` is active. We notice in the simple spraying scenario above that, although the attacker does not know which vulnerable object will be triggered, they are guaranteed that there will be a target object placed at offset `SLAB_SIZE` starting from the end of any vulnerable object, as long as the slab size of the target slab is more than `SLAB_SIZE`: this is guaranteed in both the same-order case and the cross-order case when $n_t > n_v$. Therefore, a strategy for the attacker could consist in writing at least `SLAB_SIZE` bytes and the overwriting the target object with the desired contents. Finally, the attacker can trigger all sprayed target objects, one after the other, testing for success in between. To guarantee that no panic occurs when making use of the sprayed target objects, the attacker needs to repeat the corruption pattern such that any corrupted target object will be a valid trigger. With this method, only one of the sprayed vulnerable objects needs to be triggered, therefore the corruption of the other vulnerable objects does not lead to a panic.

If $n_t < n_v$, the same method applies, with the caveat that the corruption may overwrite beyond the bounds of the target slab, potentially leading to a panic if the object is not under the attacker's usage. A simple fix is to reduce in this case the size of the write to the size of the target slab instead. This may lead to a case where no target object is overwritten, therefore resulting in a case where triggering all sprayed target objects would not lead to the exploit succeeding. In this case, the attacker could start over the entire attack until they eventually succeed.

This observation leads to a solution in the generic case where the attacker may only be able to write a limited number of bytes, but still desires an attack that would not trigger a panic, at the cost of a longer exploit runtime: by repeating PCPLOST with the limited overwrite and the above method, the attacker will eventually achieve its goal, as long as PCPLOST can reliably establish the cross-cache layout each time.

We evaluate the effectiveness of this bypass strategy using both synthetic and real-world out-of-bounds vulnerabilities. In Table III, we explicitly highlight the scenarios where `SLAB_FREELIST_RANDOM` is enabled an bypassed in our real-world experiments. In general, we notice that for some target objects, a desirable feature of the vulnerability is its ability to write zeroes. This nullifies pointers and it is useful in many cases to avoid kernel crashes and corrupt the target object(s) reliably.

D. Real-world vulnerabilities

RQ4: What is the applicability of the PCPLOST technique to real-world vulnerabilities?

To assess the effectiveness of PCPLOST in real-world scenarios, we evaluate its applicability using 6 publicly available CVEs. In Table III, alongside the name and category (temporal, spatial) of the CVEs, we highlight the step in the exploitation chain where we employ PCPLOST. In addition, we also specify the SLUB mitigations employed in each scenario. Enabling mitigations often adds complexity to the exploitation strategy, and requires additional work for the exploit writer, but PCPLOST remains effective. This is a key difference with previous work on cross-cache attacks:

CVE	Category	Caches		Objects		Attack type	Mitigations bypassed
		Vulnerable cache	Target cache	Vulnerable object	Target object(s)	(R/W)	(FR/SV/FH)
CVE-2024-53141	OOB	kmalloc-cg-1k	kmalloc-cg-2k	struct bitmap_ip	struct msg_msgseg	W	SV,FH
CVE-2021-22555	OOB/UAF ❖	kmalloc-cg-4k	kmalloc-cg-1k	struct xt_table_info	struct pipe_buffer	R	FR,SV,FH
CVE-2022-0185	OOB	kmalloc-4k	kmalloc-cg-4k	ctx->legacy_data	struct msg_msg	W	SV,FH
CVE-2023-0461	UAF ❖	kmalloc-512	kmalloc-cg-1k	struct tls_context	struct pipe_buffer	R	FR,SV,FH
CVE-2021-3715	UAF ◆	kmalloc-192	kmalloc-32	struct route4_filter	struct shm_file_data	R	FR,FH
CVE-2022-27666	OOB	–	kmalloc-4k	pfrag->page	struct user_key_payload	W	FR,FH

TABLE III: PCPLOST applicability on real-world CVEs. Under "Attack type", we use `W` to denote a cross-cache (write) corruption and `R` for a cross-cache leak (read). For employed mitigations, `FR`, `SV` and `FH` indicates, respectively, `SLAB_FREELIST_RANDOM`, `SLAB_VIRTUAL` and `SLAB_FREELIST_HARDENED`. The ❖ sign for temporal vulnerabilities indicates an in-cache pivot from UAF to OOB, while ◆ indicates a cross-cache pivot.

Generic cache type	Objects
kmalloc-⋆ caches	poll_list, shm_file_data
kmalloc-cg-⋆ caches	simple_xattr, sk_buff, msg_msg, pipe_inode_info

TABLE IV: Objects used in the PCPLOST spraying phase during evaluation with real-world vulnerabilities.

`SLAB_VIRTUAL` [26] renders the manipulation technique sustaining the SLUBStick [9] attack unfeasible.

We use PCPLOST at different steps of the exploitation chain, even multiple times in the same exploit, where it makes sense. For instance, with CVE-2024-53141, a heap overflow vulnerability in `bitmap_ip`, we use PCPLOST to establish a cross-cache adjacency between `kmalloc-cg-1k` (vulnerable cache) and `kmalloc-cg-2k`, where we allocate `struct msg_msgseg` (target object). This enables a cross-cache overflow from the vulnerable to the target cache. In the same exploit, we leverage PCPLOST to create a predictable (contiguous) layout between `kmalloc-192` and `kmalloc-cg-192`: we leverage the read primitive provided by `msg_msgseg` to leak a `kmalloc-192` heap address and, provided the contiguous layout, we can arbitrarily free (using `msg_msgseg` again) objects in `kmalloc-cg-192`, including `struct cred`. This allows to create fake credentials and escalate privileges.

For temporal vulnerabilities, we can additionally perform a vulnerability pivot (c.f., Section IV-E), exploiting PCPLOST for cross-cache attacks, or use our massaging technique to create predictable and contiguous memory layouts, useful in many primitives. For instance, with CVE-2021-22555, a heap OOB vulnerability in `netfilter` that allows writing a few bytes (all zeroes) out-of-bounds, we employ PCPLOST to prepare a predictable cross-cache layout between `kmalloc-cg-4k` and `kmalloc-cg-1k`, which is used later on in the exploit. From the out-of-bounds primitive, we pivot to a use-after-free, and from there to an out-of-bounds leak (`kmalloc-cg-4k` addresses). Then, leveraging the temporal vulnerability, we counterfeit `struct msg_msg` and perform a KASLR leak in `kmalloc-cg-1k` (with `struct pipe_buffer` as victim), exploiting the predictable and contiguous layout established by PCPLOST. The free primitive provided by `struct msg_msg` can be further used to counterfeit `struct pipe_buffer` and obtain code execution. As before, even in the presence of a hypothetical future mitigation segregating massage-sensitive objects in a different cache (in this case, `struct pipe_buffer`), PCPLOST allows attackers to employ any target object allocated in a different cache from the vulnerable one.

VI. DISCUSSION

In this section, we expand on the PCPLOST's effects on existing mitigations, its applicability with other techniques in related work, as well as suggesting potential hardening solutions to reduce the effectiveness of our proposed massaging strategy.

A. Kernel objects segregation

Modern kernel heap mitigations are shifting towards segregating object allocations based on their type [21, 44, 53, 55]. While this method is generally effective in countering in-cache spray attacks [11, 35], its efficacy diminishes when faced with page-level manipulation strategies such as PCPLOST or page-recycling [9, 12]. Specifically, hardening measures that do not account for allocation and reclaiming processes at the Page Allocator level, such as `RANDOM_KMALLOC_CACHES` [21] or `SLAB_BUCKETS` [24], are insufficient in mitigating these advanced manipulation techniques.

In particular, two considerations are noteworthy as far as cross-cache massaging techniques are concerned: (i) for page-adjacency, the kernel allocates pages (with `new_slab()`) from the same memory area for all object types, ensuring that PCPLOST remains applicable; (ii) instead, for page-recycling, the attacker can always induce memory pressure and force the kernel to reclaim pages at the Page Frame Allocator level. This in turn ensures that the same memory page can eventually be reused for both vulnerable and target cache, to which the attacker has access through the dangling pointer.

Mitigations implemented at the Page Frame Allocator level [53, 55] effectively prevent the two identified scenarios above from occurring. More specifically, these solutions define an *allocation context*, typically based on the object type, and create distinct memory areas for each context. This ensures that both page allocation and reclaim activities are confined within their respective areas. Such hardening measures significantly diminish the effectiveness of PCPLOST. However, their performance impact is potentially high, as they may prevent physical memory from being re-used across segregated memory areas. Moreover, no mitigation of this sort has appeared as mainline solution for the Linux kernel.

B. Guard pages in SLAB_VIRTUAL

In its publicly available patch [26], SLAB_VIRTUAL prevents page-recycling based cross-cache attacks, without sacrificing physical memory usage, by ensuring that only virtual addresses are used for slabs (and are not re-used). However, this does not prevent page-adjacency attack as we show. This can be fixed by adding guard pages around each allocated slab. We have discussed this issue with one of its authors, and they pointed out that the new version of SLAB_VIRTUAL deployed in Google's kernelCTF [56] platform uses guard pages. We refer to this implementation as SLAB_VIRTUAL_GP. We assessed the performance impact of SLAB_VIRTUAL_GP using the LMBench benchmarking suite. The results, summarized in Table XI (Appendix C-C), quantify the associated overhead. SLAB_VIRTUAL_GP has demonstrated robust effectiveness in mitigating cross-cache linear overflows with PCPLOST, reliably and consistently preventing our attack.

However, while such a solution effectively mitigates linear OOB vulnerabilities, it fails to mitigate non-linear overflows (e.g., those allowing to write some bytes at an offset from the vulnerable object). Given distance_to_slab_end as the distance between the vulnerable object and the boundary of its slab, an attacker can employ PCPLOST to establish a cross-cache page-adjacency layout and write after the guard page (assuming that the offset is greater than distance_to_slab_end+PAGE_SIZE). While this solution constraints the exploitation strategy further, it still leaves a residual attack surface.

C. Specialized slabs

PCPLOST performs a cross-cache massaging between generic caches (accounted and non-accounted). We apply our technique on generic caches because of the timing primitive we use [9]. For specialized slabs, PCPLOST also applies if a suitable timing primitive is available. The timing primitive needs to satisfy the following constraint: non-allocation tasks performed by the system call allocating the SLUB object should be as minimal as possible. Provided a similar primitive that allocates objects in specialized caches, or a similarly effective side channel, the PCPLOST technique can exploit it to detect PCP list allocations and interactions with the zoned free_area and mount the attack reliably.

D. Higher order slabs

In the Linux kernel, the Per-CPU-Pageset optimization was originally used for order-0 pages only. However, a 2021 patch by Mel Gorman enabled their use also for higher order pages [57]. At the time of writing, every page whose order is less than PAGE_ALLOC_COSTLY_ORDER = 3 allocates from PCP lists. SLUB caches mainly request page orders ranging from 0 to 3, thereby often using PCP lists. In our experiments, only one cache, namely 9p-fcall-cache-1, requires pages with an order greater than 3. In this edge case, PCPLOST should remain an effective approach after a few necessary adjustments. The major challenge is related to the side channel. On code paths bypassing PCP lists, attackers are interested in two different kernel events: (i) Simple allocations from the free_area and (ii) Higher-order page splits from the free_area. Distinguishing these two events would allow implementing an attack similar to PCPLOST. However, the two events occur in the same kernel function, and differ only by few machine instructions. Nevertheless, an attacker may employ techniques to amplify the timing disparity between these two events [20, 58, 59] to achieve a robust side channel in this context.

E. PSPRAY massaging applicability

The PSPRAY [14] massaging technique is an in-cache memory massaging designed to improve the overall reliability of kernel exploits and to bypass SLAB_FREELIST_RANDOM [23]. This approach targets both spatial and temporal vulnerabilities. Let us use N_t as the number of objects per slab in the target cache. For OOB vulnerabilities, PSPRAY bypasses SLAB_FREELIST_RANDOM by detecting a new page allocation via a timing side-channel and allocating $N_t - 1$ target objects. This leaves a free slot for the vulnerable object allocation, knowing that it will be surrounded by target objects. For UAF and DF vulnerabilities, PSPRAY bypasses SLAB_FREELIST_RANDOM by detecting a page allocation and allocating the vulnerable object (alongside additional objects), which is then immediately freed. Subsequently, it allocates the target object, ensuring its placement in the slot previously occupied by the vulnerable object, where control is maintained through a dangling pointer.

While the PSPRAY technique is effective for in-cache massaging against SLAB_FREELIST_RANDOM, it is not applicable to cross-cache OOB attacks. Specifically, corruption occurs in a separate slab within a different cache. Consequently, target objects are allocated independently, and no guarantee exists that a vulnerable object will precede a target object, which is essential for the effective exploitation. For this reason, we propose a different SLAB_FREELIST_RANDOM bypass method (Section V-C2).

VII. CONCLUSION

This work introduces PCPLOST, a cross-cache memory massaging technique within the Linux kernel. By strategically exploiting the page split feature of the Page Frame Allocator, PCPLOST establishes a contiguous cross-cache layout favorable for exploiting out-of-bounds (OOB) vulnerabilities in the SLUB allocator. This massaging is made practical by using a timing side-channel that allows to detect interactions between SLUB, PCP lists and free_area, revealing their state. By employing a "pivoting" strategy, we expand the attack surface to include temporal vulnerabilities, significantly broadening the applicability of our technique. We evaluate PCPLOST success rate over the frequently used generic caches (kmalloc-*), reaching a high reliability (over 90% in most cases), and apply our technique on real-world CVEs. The significant efficacy of our approach suggests that current mitigation strategies are inadequate to provide comprehensive protection against cross-cache attacks within the Linux kernel.

Ethics Considerations

PCPLost is a heap massaging technique that can only be used by attackers together with a memory safety vulnerability targeting the kernel heap. We do not disclose any such new memory safety vulnerability in this paper, therefore systems are not immediately put at risk by this work. Nevertheless, we are in discussion with Linux kernel developers regarding PCPLost, including regarding possible mitigations. As part of the NDSS Artifact evaluation, we commit to making available artifacts developed for this paper upon acceptance, including tooling developed for measuring PCPLost reliability, our detailed benchmark data, and the corresponding kernel images.

Acknowledgments

The authors would like to thank Pietro Moretto, Damian Pfammatter, and Bernhard Tellenbach for discussions on the attacks and mitigations in this paper. This work has been partially funded by the Swiss National Science Foundation (SNSF Grant no. 10004669), as well as by armasuisse Science and Technology.

References

[1] T. Koczka, "Learnings from kctf vrp's 42 linux kernel exploits submissions," 2023, [Accessed 2025-03-31]. [Online]. Available: https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html

[2] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 414–425.

[3] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS*, 2016.

[4] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 781–797. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei

[5] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1707–1722.

[6] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 563–577.

[7] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC conference on computer and communications security*, 2022, pp. 1963–1976.

[8] D. Liu, P. Wang, X. Zhou, W. Xie, G. Zhang, Z. Luo, T. Yue, and B. Wang, "From release to rebirth: Exploiting thanos objects in linux kernel," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 533–548, 2022.

[9] L. Maar, S. Gast, M. Unterguggenberger, M. Oberhuber, and S. Mangard, "SLUBStick: Arbitrary memory writes through practical software Cross-Cache attacks within the linux kernel," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4051–4068. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/maar-slubstick

[10] N. Wu, "Dirty pagetable: A novel exploitation technique to rule linux kernel," 2023, [Accessed 2025-03-26]. [Online]. Available: https://web.archive.org/web/20250226150503/https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html

[11] A. Popov, "Four bytes of power: Exploiting cve-2021-26708 in the linux kernel," 2021, [Accessed 2025-02-24]. [Online]. Available: https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html

[12] J. Horn, "How a simple linux kernel memory corruption bug can lead to complete system compromise," 2021, [Accessed 2025-02-25]. [Online]. Available: https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html

[13] arttnba3, "arttnba3/d3ctf2023_d3kcache: attachment and write up for d^3ctf," 2023, [Accessed 2025-02-25]. [Online]. Available: https://github.com/arttnba3/D3CTF2023_d3kcache

[14] Y. Lee, J. Kwak, J. Kang, Y. Jeon, and B. Lee, "Pspray: Timing Side-Channel based linux kernel heap exploitation technique," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6825–6842. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/lee-yoochan

[15] V. Nikolenko and M. S, "Linux kernel heap feng shui in 2022," 2022, [Accessed 2025-03-30]. [Online]. Available: https://duasynt.com/blog/linux-kernel-heap-feng-shui-2022

[16] Z. Guo, D. K. Le, Z. Lin, K. Zeng, R. Wang, T. Bao, Y. Shoshitaishvili, A. Doupé, and X. Xing, "Take a step further: Understanding page spray in linux kernel exploitation," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1189–1206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/guo-ziyi

[17] M. Momeu, F. Kilger, C. Roemheld, S. Schnückel, S. Proskurin, M. Polychronakis, and V. P. Kemerlis, "ISLAB: Immutable memory management metadata for commodity operating system kernels," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1159–1172.

[18] M. Momeu, S. Schnückel, K. Angnis, M. Polychronakis, and V. P. Kemerlis, "Safeslab: Mitigating use-after-free vulnerabilities via memory protection keys," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1345–1359.

[19] L. Maar, L. Giner, D. Gruss, and S. Mangard, "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, 2025.

[20] L. Maar, J. Juffinger, T. Steinbauer, D. Gruss, and S. Mangard, "KernelSnitch: Side-channel attacks on kernel data structures," 2025.

[21] V. Babka and R. Gong, "Randomized slab caches for kmalloc() - kernel/git/torvalds/linux.git - linux kernel source tree," 2023, [Accessed 2024-12-06]. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3c6152940584290668b35fa0800026f6a1ae05fe

[22] K. Cook, "[patch v3] mm: Add slub free list pointer obfuscation - kees cook," 2017, [Accessed 2025-02-24]. [Online]. Available: https://lore.kernel.org/all/20170706002718.GA102852@beast/

[23] T. Garnier, "[patch v2] mm: Slab freelist randomization," 2016, [Accessed 2025-02-27]. [Online]. Available: https://lore.kernel.org/all/1461616763-60246-1-git-send-email-thgarnie@google.com/

[24] K. Cook, "[patch v3 0/6] slab: Introduce dedicated bucket allocator," 2024, [Accessed 2025-02-25]. [Online]. Available: https://lore.kernel.org/all/20240424213019.make.366-kees@kernel.org/

[25] W. Liu, "Reviving exploits against cred structs - six byte cross cache overflow to leakless data-oriented kernel pwnage," 2022, [Accessed 2025-02-25]. [Online]. Available: https://www.willsroot.io/2022/08/reviving-exploits-against-cred-struct.html

[26] J. Horn and M. Rizzo, "Prevent cross-cache attacks in the slub allocator [lwn.net]," 2023, [Accessed 2024-12-06]. [Online]. Available: https://lwn.net/Articles/944647/

[27] J. Zhou, J. Hu, W. Shen, and Z. Qian, "A novel page-uaf exploit strategy for privilege escalation in linux systems," 2024, [Accessed 2025-07-04]. [Online]. Available: https://phrack.org/issues/71/13

[28] S. S. D. T. Team, "Linux kernel hfsplus slab-out-of-bounds write," 2025, [Accessed 2025-07-14]. [Online]. Available: https://ssd-disclosure.com/ssd-advisory-linux-kernel-hfsplus-slab-out-of-bounds-write/

[29] D. Xie, D. He, W. You, J. Huang, B. Liang, S. Gan, and W. Shi, "BridgeRouter: Automated capability upgrading of out-of-bounds write vulnerabilities to arbitrary memory write primitives in the linux kernel," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 772–790. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00132

[30] W. Long, "[patch v3 2/2] mm: memcg/slab: Create a new set of kmalloc-cg-n caches," [Accessed 2024-12-06]. [Online]. Available: https:

//lore.kernel.org/all/20210505154613.17214-3-longman@redhat.com/

[31] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, Inc. [Online]. Available: https://openlibrary.org//books/OL7581061M

[32] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR. [Online]. Available: https://openlibrary.org//books/OL9291158M

[33] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "MAZE: Towards automated heap feng shui," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1647–1664. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-yan

[34] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 71–88. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zeng

[35] A. Guerrero, "Cve-2022-0185 - linux kernel slab out-of-bounds write: exploit and writeup," 2022, [Accessed 2025-03-30]. [Online]. Available: https://www.openwall.com/lists/oss-security/2022/01/25/14

[36] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap taichi: exploiting memory allocation granularity in heap-spraying attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 327–336. [Online]. Available: https://doi.org/10.1145/1920261.1920310

[37] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: a defense against heap-spraying code injection attacks," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, p. 169–186.

[38] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 573–584. [Online]. Available: https://doi.org/10.1145/1866307.1866371

[39] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. Lee, "Last-level cache side-channel attacks are practical," vol. 2015-July, pp. 605–622, ISSN: 1081-6011.

[40] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Springer International Publishing, vol. 9721, pp. 279–299, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-319-40667-1_14

[41] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on AES to practice," in *2011 IEEE Symposium on Security and Privacy*, pp. 490–505, ISSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/document/5958048

[42] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *2013 IEEE Symposium on Security and Privacy*. IEEE, pp. 191–205. [Online]. Available: http://ieeexplore.ieee.org/document/6547110/

[43] V. Babka, "[v3,2/2] mm: memcg/slab: Create a new set of kmalloc-cg- caches," 2021, [Accessed 2024-12-06]. [Online]. Available: https://patchwork.kernel.org/project/linux-mm/patch/20210505154613.17214-3-longman@redhat.com/

[44] K. Cook, "[rfc][patch 0/5] slab: Allocate and use per-call-site caches," 2024, [Accessed 2025-07-18]. [Online]. Available: https://lore.kernel.org/all/20240809072532.work.266-kees@kernel.org/

[45] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, 2020, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng

[46] M. A. Ramdhan and B. Jheng, "All roads leads to gke's host : 4+ ways to escape," 2022, [Accessed 2024-05-07]. [Online]. Available: https://media.defcon.org/DEF%20CON%2030/DEF%20CON%2030%20presentations/Billy%20Jheng%20%20%20Muhammad%20Alifa%20Ramdhan%20-%20All%20Roads%20leads%20to%20GKEs%20Host%20%204%2B%20Ways%20to%20Escape.pdf

[47] X. Zou and Z. Qian, "Cve-2022-27666 - exploit esp6 modules in linux kernel," 2022, [Accessed 2025-06-08]. [Online]. Available: https://etenal.me/archives/1825

[48] A. Potapenko, "[patch v8 0/3] add init_on_alloc/init_on_free boot options," 2019, [Accessed 2025-02-25]. [Online]. Available: https://lore.kernel.org/all/20190626121943.131390-1-glider@google.com/

[49] M. Leibowitz, "[patch] add the randstruct gcc plugin - michael leibowitz," 2016, [Accessed 2025-02-25]. [Online]. Available: https://lore.kernel.org/all/1477071466-19256-1-git-send-email-michael.leibowitz@intel.com/

[50] (n.d.) Gcc, the gnu compiler collection - gnu project. [Accessed 2025-02-27]. [Online]. Available: https://gcc.gnu.org/

[51] L. Torvalds, "Re: [rfc patch 00/14] prevent cross-cache attacks in the slub allocator - linus torvalds," 2023, [Accessed 2024-12-06]. [Online]. Available: https://lore.kernel.org/lkml/CAHk-=wgGzB4u-WZsDpdgjwX1w5=9CLE0gorhaNFD09P1FUGeuQ@mail.gmail.com/

[52] Z. Lin, "How autoslab changes the memory unsafety game," 2021, [Accessed 2025-02-27]. [Online]. Available: https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game

[53] Z. Wang, Y. Guang, Y. Chen, Z. Lin, M. Le, D. K. Le, D. Williams, X. Xing, Z. Gu, and H. Jamjoom, "SeaK: Rethinking the design of a secure allocator for OS kernel," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1171–1188. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/wang-zicheng

[54] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1165–1184. [Online]. Available: https://doi.org/10.1145/3372297.3423353

[55] A. S. Research, "Towards the next generation of xnu memory safety," 2022, [Accessed 2025-03-31]. [Online]. Available: https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/

[56] Google, "kernelctf rules," n.d., [Accessed 2025-08-03]. [Online]. Available: https://google.github.io/security-research/kernelctf/rules.html

[57] M. Gorman, "[patch 2/2] mm/page_alloc: Allow high-order pages to be stored on the per-cpu lists," 2021, [Accessed 2025-03-31]. [Online]. Available: https://lore.kernel.org/lkml/20210611103827.4b78b776@canb.auug.org.au/T/

[58] Y. Lee, C. Min, and B. Lee, "ExpRace: Exploiting kernel races through raising interrupts," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2363–2380.

[59] H. Ragab, A. Mambretti, A. Kurmus, and C. Giuffrida, "GhostRace: Exploiting and mitigating speculative race conditions," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, aug 2024, pp. 6185–6202. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/ragab

[60] C. Migliorelli, A. Mambretti, A. Sorniotti, V. Zaccaria, and A. Kurmus, "Pcplost ae," Nov. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17640260

Appendix A
The SLUB allocator

This section provides further details on the SLUB Allocator's internal workings. In SLUB, generic caches differ based on the size of the objects they accommodate. Their object size also influences the number of pages used to allocate one slab to store such objects. The more we go towards higher object sizes, the smaller the number of objects per slab and the higher the page order. Table V shows the object size in bytes, the number of objects per slab and the page order for each generic cache (and accounted counterparts) in the Linux kernel.

When requesting pages of orders greater than 0, slabs allocate contiguous page-sized chunks. Indeed, slabs are sets of contiguous memory pages provided by the Page Frame Allocator. The strategy of allocating contiguous pages for slabs allows to exploit data locality on the hardware cache level.

Generic cache	Object size (bytes)	Objects per slab	Page order
kmalloc-(cg-)8	8	512	0
kmalloc-(cg-)16	16	256	0
kmalloc-(cg-)32	32	128	0
kmalloc-(cg-)64	64	64	0
kmalloc-(cg-)96	96	42	0
kmalloc-(cg-)128	128	32	0
kmalloc-(cg-)192	192	21	0
kmalloc-(cg-)256	256	32	1
kmalloc-(cg-)512	512	32	2
kmalloc-(cg-)1k	1024	32	3
kmalloc-(cg-)2k	2048	16	3
kmalloc-(cg-)4k	4096	8	3

TABLE V: The capacity of generic caches (accounted and non-accounted) varies with object size, resulting in different numbers of objects per slab and page orders. The data was obtained from experiments conducted on a 12-core machine.

PCP list order	**batch** value
0	$63 \gg 0 = 63$
1	$63 \gg 1 = 31$
2	$63 \gg 2 = 15$
3	$63 \gg 3 = 7$

TABLE VI: The value of batch when varying the page order. According to the table above, a base_batch value of 63 determines the computation of the final batch (computed as max(base_batch >> order, 2)). The data was obtained from experiments conducted on a 12-core machine.

APPENDIX B
THE PAGE FRAME ALLOCATOR

As discussed in Section II-B, the Page Frame Allocator has an optimization called Per-CPU-Pageset (PCP) to provide a per-CPU page allocation pool to serve requests locally to the CPU-core. These per-CPU lists store free pages for each order, up to PAGE_ALLOC_COSTLY_ORDER = 3. When exhausted, PCP lists allocate (reserve) pages from the zoned free_area, according to a value called batch (see Section II-B for further details). Table VI shows the value of batch for each page order supported by the Per-CPU-Pageset optimization.

APPENDIX C
ADDITIONAL DETAILS ON THE EVALUATION

A. *PCP list side-channel*

Cache	Order	False positives	False negatives
PCP list allocations			
kmalloc-256	1	6.60%	0.06%
kmalloc-2k	3	17.84%	0.22%
kmalloc-1k	3	18.34%	0.08%
kmalloc-4k	3	19.42%	0.48%
free_area allocations			
kmalloc-256	1	50.0%	0.01%
kmalloc-2k	3	25.50%	0.14%
kmalloc-1k	3	30.0%	0.01%
kmalloc-4k	3	23.90%	0.39%

TABLE VII: False positive and false negative rates for two events (PCP list and free_area allocations) across caches.

This section provides further details on the evaluation strategy employed to assess the feasibility of the timing side-channel in the context of the Page Frame Allocator's PCP lists and free_area. Figure 3 in Section IV-A illustrates the timing values of generic kmem-caches using histograms.

The timing distributions for "timing objects" allocations show the same trend suggested by previous work [9, 14]. However, we highlight that the same side-channel strategy can be used to infer two further page allocation events: interactions with PCP lists (fast page allocation path) and allocations from the free_area (slow page allocation path). Table VII shows the false positive/negative rates for our side-channel. Combining the two different thresholds to differentiate between the two events with the double measurement described in Section IV-D, our side-channel yields substantial massaging reliability in almost all cases (see Section V and Appendix C-B).

To collect the timings of objects and categorize their allocations, we utilize a custom kernel module to allocate kmalloc-* objects and obtain their addresses. Therefore, from a user-space program, we measure the clock cycles required to allocate timing objects (through rdtsc) and then perform a persistent object allocation, retrieving the allocated address. Given the nature of "timing objects" and their immediate freeing, the object slot allocated for the persistent object is the same as that used for the timing object, due to the LIFO nature of SLUB free-lists. We leverage ftrace to observe kernel allocations and cross-reference the allocation type (according to the object virtual address) with the time observed from the user-space program.

B. *Massaging success rate*

Extended experiments (Table VIII, Table IX, Table X) were conducted across both bare-metal and virtualized environments. For the virtualized setup, x86-based evaluations utilized a 64-bit QEMU virtual machine configured with 12 CPU cores and 32 GB of RAM, running Debian trixie/sid and Ubuntu jammy 22.04, each with Linux kernel version 6.8. ARM-based experiments were carried out on an ARM QEMU virtual machine provisioned with 8 CPU cores and 8 GB of RAM. To evaluate the feasibility of PCPLOST in a non-virtualized context, we additionally deployed our strategy on a bare-metal machine equipped with an AMD Ryzen 7 CPU (16 cores) and 16 GB of RAM.

C. *SLAB_VIRTUAL overhead evaluation*

This section presents the performance overhead introduced by SLAB_VIRTUAL with guard pages (SLAB_VIRTUAL_GP). While its publicly available patch [26] does not use guard pages, the new version of SLAB_VIRTUAL deployed in Google's kernelCTF [56] platform reserves a virtual page — without a corresponding physical frame — around each virtual slab, effectively acting as a guard page. The detailed overhead evaluation of SLAB_VIRTUAL_GP using the LMBench suite is summarized in Table XI.

Vulnerable cache	Target cache	Success rate											
		Adjacency only						Adjacency + object alignment					
		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)	
		Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI
`kmalloc-1k`	`kmalloc-2k`	99.5	(97.22,99.91)	100.0	(98.11,100)	77.0	(70.79,82.29)	100.0	(98.11,100)	99.5	(97.22,99.91)	81.0	(75,85.83)
`kmalloc-1k`	`kmalloc-4k`	99.5	(97.22,99.91)	100.0	(98.11,100)	75.0	(69.09,80.94)	100.0	(98.11,100)	100.0	(98.11,100)	77.0	(70.69,82.29)
`kmalloc-2k`	`kmalloc-1k`	99.5	(97.22,99.91)	93.5	(89.19,96.16)	76.0	(69.63,81.39)	97.0	(93.61,98.62)	97.0	(0.9361,98.61)	76.0	(69.62,81.39)
`kmalloc-2k`	`kmalloc-4k`	99.5	(97.22,99.91)	100.0	(98.11,100)	83.0	(77.18,87.57)	100.0	(98.11,100)	99.5	(97.22,99.91)	84.0	(78.28,88.43)
`kmalloc-4k`	`kmalloc-1k`	88.5	(83.33,92.21)	84.0	(88.43,88.43)	88.43	(88.43,88.43)	88.0	(88.43,88.43)	88.43	(88.43,88.43)	66.0	(88.43,88.43)
`kmalloc-4k`	`kmalloc-2k`	97.5	(94.28,98.92)	93.5	(88.43,88.43)	78.0	(88.43,88.43)	93.0	(88.43,88.43)	92.0	(88.43,88.43)	88.43	(88.43,88.43)

(a) Massaging evaluated on ARM64 QEMU machine. Experiments run for this evaluation involve less samples: 10 QEMU runs and 5 massaging executions per run, as opposed to Table II where we perform 40 QEMU runs and 20 massaging executions per run.

Vulnerable cache	Target cache	Success rate											
		Adjacency only						Adjacency + object alignment					
		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)	
		Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI
`kmalloc-1k`	`kmalloc-2k`	100.0	(96.30,100)	100.0	(96.30,100)	100	(96.30,100)	100	(96.30, 100)	100	(96.30,100)	100	(96.30,100)
`kmalloc-1k`	`kmalloc-4k`	100.0	(96.30,100)	100.0	(96.30,100)	100	(96.30,100)	100	(96.30, 100)	100	(96.30,100)	100	(96.30,100)
`kmalloc-2k`	`kmalloc-1k`	100.0	(96.30,100)	100.0	(96.30,100)	100	(96.30,100)	100	(96.30, 100)	100	(96.30,100)	100	(96.30,100)
`kmalloc-2k`	`kmalloc-4k`	100.0	(96.30,100)	100.0	(96.30,100)	100	(96.30,100)	100	(96.30, 100)	100	(96.30,100)	100	(96.30,100)
`kmalloc-4k`	`kmalloc-1k`	100.0	(96.30,100)	100.0	(96.30,100)	100	(96.30,100)	100	(96.30, 100)	100	(96.30,100)	100	(96.30,100)
`kmalloc-4k`	`kmalloc-2k`	100.0	(96.30,100)	100.0	(96.30,100)	100	(96.30,100)	100	(96.30, 100)	100	(96.30,100)	100	(96.30,100)

(b) Massaging evaluated on a AMD bare-metal machine with 16 GB of RAM. Experiments run for this evaluation involve less samples: 10 machine runs and 10 massaging executions per run, as opposed to Table II where we perform 40 QEMU runs and 20 massaging executions per run.

TABLE VIII: Same-order massaging ($n_v = n_t$) for order-3 caches, under different settings; `SLAB_VIRTUAL` disabled.

Vulnerable cache	Target cache	Success rate											
		Adjacency only						Adjacency + object alignment					
		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)	
		Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI
`kmalloc-1k`	`kmalloc-2k`	100.0	(99.05, 100)	99.50	(98.19, 99.86)	99.75	(98.60, 99.95)	99.75	(98.60, 99.95)	99.0	(97.46, 99.61)	99.50	(98.19, 99.86)
`kmalloc-1k`	`kmalloc-4k`	100.0	(99.05, 100)	100.0	(99.05, 100)	99.75	(98.60, 99.95)	100.0	(99.05, 100)	100.0	(99.05, 100)	100.0	(99.05, 100)
`kmalloc-1k`	`kmalloc-8k`	100.0	(99.05, 100)	100.0	(99.05, 100)	99.75	(98.60, 99.95)	100.0	(99.05, 100)	100.0	(99.05, 100)	99.0	(97.46, 99.61)
`kmalloc-2k`	`kmalloc-1k`	98.75	(97.11, 99.46)	95.25	(92.70, 96.94)	97.5	(95.46, 98.64)	99.75	(98.60, 99.95)	92.50	(89.49, 94.70)	98.5	(96.77, 99.31)
`kmalloc-2k`	`kmalloc-4k`	100.0	(99.05, 100)	100.0	(99.05, 100)	99.0	(97.46, 99.61)	100.0	(99.05, 100)	100.0	(99.05, 100)	99.50	(98.19, 99.86)
`kmalloc-2k`	`kmalloc-8k`	100.0	(99.05, 100)	100.0	(99.05, 100)	99.0	(97.46, 99.61)	100.0	(99.05, 100)	100.0	(99.05, 100)	99.50	(98.19, 99.86)
`kmalloc-4k`	`kmalloc-1k`	92.50	(89.49, 94.70)	79.25	(75.01, 82.94)	91.25	(88.07, 93.64)	94.25	(91.52, 96.14)	85.0	(81.17, 88.16)	92.5	(89.49, 94.70)
`kmalloc-4k`	`kmalloc-2k`	98.0	(96.10, 98.98)	91.0	(87.79, 93.43)	97.50	(95.46, 98.64)	98.0	(96.10, 98.98)	92.75	(89.78, 94.90)	96.75	(94.52, 98.09)
`kmalloc-4k`	`kmalloc-8k`	100.0	(99.05, 100)	100.0	(99.05, 100)	99.0	(97.46, 99.61)	100.0	(99.05, 100)	100.0	(99.05, 100)	99.25	(97.82, 99.74)

(a) Order-3 caches.

Vulnerable cache	Target cache	Success rate											
		Adjacency only						Adjacency + object alignment					
		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)	
		Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI
`kmalloc-8`	`kmalloc-16`	98.75	(97.11, 99.46)	99.25	(97.82, 99.74)	92.5	(89.49, 94.70)	97.25	(95.14, 98.46)	98.5	(96.77, 99.31)	91.0	(87.79, 93.43)
`kmalloc-8`	`kmalloc-32`	98.25	(96.43, 99.15)	97.25	(95.14, 98.46)	91.0	(87.79, 93.43)	96.75	(94.52, 98.09)	99.50	(98.19, 99.86)	93.5	(90.65, 95.52)
`kmalloc-8`	`kmalloc-64`	98.25	(96.43, 99.15)	98.5	(96.77, 99.31)	91.25	(88.07, 93.64)	97.75	(95.78, 98.81)	98.0	(96.10, 98.98)	90.0	(86.67, 92.57)
`kmalloc-8`	`kmalloc-96`	98.0	(96.10, 98.98)	99.25	(97.82, 99.74)	92.75	(89.78, 94.90)	98.0	(96.10, 98.98)	98.25	(96.43, 99.15)	91.5	(88.36, 93.85)
`kmalloc-8`	`kmalloc-128`	97.5	(95.46, 98.64)	99.0	(97.46, 99.61)	91.25	(88.07, 93.64)	98.25	(96.43, 99.15)	99.25	(97.82, 99.74)	92.75	(89.78, 94.90)
`kmalloc-8`	`kmalloc-192`	98.25	(96.43, 99.15)	99.50	(98.19, 99.86)	93.5	(90.64, 95.52)	98.50	(96.77, 99.31)	98.25	(96.43, 99.15)	92.25	(89.21, 94.49)
`kmalloc-16`	`kmalloc-8`	99.25	(97.82, 99.74)	99.25	(97.82, 99.74)	95.75	(93.30, 97.33)	98.75	(97.11, 99.46)	98.5	(96.77, 99.31)	94.75	(92.11, 96.54)
`kmalloc-16`	`kmalloc-32`	98.5	(96.77, 99.31)	98.75	(97.11, 99.46)	94.5	(91.81, 96.34)	99.50	(98.19, 99.86)	98.0	(96.10, 98.98)	97.0	(94.83, 98.27)
`kmalloc-16`	`kmalloc-64`	99.50	(98.19, 99.86)	99.50	(98.19, 99.86)	95.75	(93.30, 97.33)	99.25	(97.81, 99.74)	99.50	(98.19, 99.86)	97.25	(95.14, 98.46)
`kmalloc-16`	`kmalloc-96`	99.0	(97.46, 99.61)	98.75	(97.11, 99.46)	94.25	(91.52, 96.13)	98.25	(96.43, 99.15)	99.25	(97.82, 99.74)	94.25	(91.52, 96.14)
`kmalloc-16`	`kmalloc-128`	98.75	(97.11, 99.46)	98.0	(96.10, 98.98)	97.5	(95.46, 98.64)	98.75	(97.11, 99.46)	98.75	(97.11, 99.46)	95.25	(92.70, 96.94)
`kmalloc-16`	`kmalloc-192`	98.0	(96.10, 98.98)	98.25	(96.43, 99.15)	97.25	(95.14, 98.46)	99.50	(98.19, 99.86)	98.0	(96.10, 98.98)	94.0	(91.23, 95.93)
`kmalloc-32`	`kmalloc-8`	99.25	(97.82, 99.74)	99.75	(98.60, 99.95)	98.25	(96.43, 99.15)	99.25	(97.82, 99.74)	99.75	(98.60, 99.95)	99.0	(97.46, 99.61)
`kmalloc-32`	`kmalloc-16`	99.75	(98.60, 99.95)	99.25	(97.82, 99.74)	98.5	(96.77, 99.31)	99.25	(97.82, 99.74)	99.75	(98.60, 99.95)	98.75	(97.11, 99.46)
`kmalloc-32`	`kmalloc-64`	99.50	(98.19, 99.86)	99.25	(97.82, 99.74)	98.0	(96.10, 98.98)	99.75	(98.60, 99.95)	99.25	(97.82, 99.74)	98.75	(97.11, 99.46)
`kmalloc-32`	`kmalloc-96`	99.75	(98.60, 99.95)	99.50	(98.19, 99.86)	98.75	(97.11, 99.46)	99.50	(98.19, 99.86)	99.75	(98.60, 99.95)	99.0	(97.46, 99.61)
`kmalloc-32`	`kmalloc-128`	99.75	(98.60, 99.95)	99.50	(98.19, 99.86)	99.50	(98.19, 99.86)	99.0	(97.46, 99.61)	100.0	(99.05, 100)	98.25	(96.43, 99.15)
`kmalloc-32`	`kmalloc-192`	99.75	(98.60, 99.95)	99.25	(97.82, 99.74)	98.25	(96.43, 99.15)	99.50	(98.19, 99.86)	99.0	(97.46, 99.61)	99.0	(97.46, 99.61)
`kmalloc-64`	`kmalloc-8`	99.25	(97.82, 99.74)	95.75	(93.30, 97.33)	98.0	(96.10, 98.98)	99.0	(97.46, 99.61)	97.0	(94.83, 98.27)	96.25	(93.90, 97.71)
`kmalloc-64`	`kmalloc-16`	97.25	(95.14, 98.46)	97.5	(95.46, 98.64)	97.0	(94.83, 98.27)	99.50	(98.19, 99.86)	97.25	(95.14, 98.46)	97.5	(95.46, 98.64)
`kmalloc-64`	`kmalloc-32`	96.5	(94.21, 97.90)	97.75	(95.78, 98.81)	97.0	(94.83, 98.27)	99.25	(97.82, 99.74)	98.0	(96.10, 98.98)	96.25	(93.90, 97.71)
`kmalloc-64`	`kmalloc-96`	99.25	(97.82, 99.74)	98.25	(96.43, 99.15)	96.50	(94.21, 97.90)	100.0	(99.05, 100)	98.25	(96.43, 99.15)	97.25	(95.14, 98.46)
`kmalloc-64`	`kmalloc-128`	97.25	(95.14, 98.46)	96.0	(93.60, 97.52)	96.0	(93.60, 97.52)	100.0	(99.05, 100)	96.50	(94.21, 97.90)	97.0	(94.83, 98.27)
`kmalloc-64`	`kmalloc-192`	97.50	(95.46, 98.64)	97.75	(95.78, 98.81)	97.0	(94.83, 98.27)	99.50	(98.19, 99.86)	98.50	(96.77, 99.31)	96.50	(94.21, 97.90)
`kmalloc-96`	`kmalloc-8`	98.50	(96.77, 99.31)	98.0	(96.10, 98.98)	93.50	(90.64, 95.52)	98.0	(96.10, 98.98)	99.0	(97.46, 99.61)	88.5	(85.0, 91.27)
`kmalloc-96`	`kmalloc-16`	98.75	(97.11, 99.46)	97.50	(95.46, 98.64)	92.25	(89.21, 94.49)	99.50	(98.19, 99.86)	98.0	(96.10, 98.98)	93.50	(90.64, 95.52)
`kmalloc-96`	`kmalloc-32`	98.25	(96.43, 99.15)	98.0	(96.10, 98.98)	92.0	(88.92, 94.28)	98.75	(97.11, 99.46)	97.5	(95.46, 98.64)	94.25	(91.52, 96.14)
`kmalloc-96`	`kmalloc-64`	99.25	(97.82, 99.74)	98.0	(96.10, 98.98)	91.75	(88.64, 94.06)	99.25	(97.82, 99.74)	97.75	(95.78, 98.81)	91.75	(88.64, 94.06)
`kmalloc-96`	`kmalloc-128`	99.0	(97.46, 99.61)	98.75	(97.11, 99.46)	91.5	(88.36, 93.85)	97.75	(95.78, 98.82)	98.75	(97.11, 99.46)	91.0	(87.79, 93.43)
`kmalloc-96`	`kmalloc-192`	98.75	(97.11, 99.46)	98.75	(97.11, 99.46)	92.75	(89.78, 94.90)	99.25	(97.82, 99.74)	97.75	(95.78, 98.82)	91.75	(88.64, 94.06)
`kmalloc-128`	`kmalloc-8`	100.0	(99.05, 100)	100.0	(99.05, 100)	99.75	(98.60, 99.95)	100.0	(99.05, 100)	100.0	(99.05, 100)	98.5	(96.77, 99.31)
`kmalloc-128`	`kmalloc-16`	99.75	(98.60, 99.95)	99.50	(98.19, 99.86)	99.75	(98.60, 99.95)	99.50	(98.19, 99.86)	99.75	(98.60, 99.95)	99.50	(98.19, 99.86)
`kmalloc-128`	`kmalloc-32`	100.0	(99.05, 100)	99.0	(97.46, 99.61)	99.25	(97.82, 99.74)	99.75	(98.60, 99.95)	98.75	(97.11, 99.46)	99.0	(97.46, 99.61)
`kmalloc-128`	`kmalloc-64`	99.75	(98.60, 99.95)	99.0	(97.46, 99.61)	98.75	(97.11, 99.46)	99.75	(98.60, 99.95)	98.0	(96.10, 98.98)	99.25	(97.82, 99.74)
`kmalloc-128`	`kmalloc-96`	100.0	(99.05, 100)	97.0	(94.83, 98.27)	98.25	(96.43, 99.15)	100.0	(99.05, 100)	97.75	(95.78, 98.81)	97.75	(95.78, 98.81)
`kmalloc-128`	`kmalloc-192`	100.0	(99.05, 100)	99.25	(97.82, 99.74)	98.25	(96.43, 99.15)	99.50	(98.19, 99.86)	98.75	(97.11, 99.46)	98.0	(96.10, 98.98)
`kmalloc-192`	`kmalloc-8`	97.25	(95.14, 98.46)	98.75	(97.11, 99.46)	82.0	(77.94, 85.45)	100.0	(99.05, 100)	99.0	(97.46, 99.61)	91.0	(87.79, 93.43)
`kmalloc-192`	`kmalloc-16`	95.5	(92.0, 97.13)	99.0	(97.46, 99.61)	90.25	(86.95, 92.78)	97.25	(95.14, 98.46)	99.50	(98.19, 99.86)	95.75	(93.30, 97.33)
`kmalloc-192`	`kmalloc-32`	96.25	(93.90, 97.71)	98.25	(96.43, 99.15)	89.0	(85.55, 91.70)	96.25	(93.90, 97.71)	98.25	(96.43, 99.15)	94.5	(91.81, 96.34)
`kmalloc-192`	`kmalloc-64`	95.25	(92.70, 96.94)	98.25	(96.43, 99.15)	89.0	(85.55, 91.70)	95.25	(92.70, 96.94)	98.25	(96.43, 99.15)	94.25	(91.52, 96.14)
`kmalloc-192`	`kmalloc-96`	95.25	(92.70, 96.94)	97.25	(95.14, 98.46)	89.5	(86.11, 92.14)	97.5	(95.46, 98.64)	97.0	(94.83, 98.27)	94.0	(91.23, 95.93)
`kmalloc-192`	`kmalloc-128`	98.75	(97.11, 99.46)	98.25	(96.43, 99.15)	86.5	(82.80, 89.50)	92.75	(89.78, 94.90)	98.5	(96.77, 99.31)	90.75	(87.51, 93.21)

(b) Order-0 caches.

TABLE IX: Same-order massaging ($n_v = n_t$); `SLAB_VIRTUAL` disabled. In this case, we consider all kmem-caches, yielding a more comprehensive evaluation. Table IXa presents the success rate for order-3 caches, while Table IXb presents the success rate for order-0 caches. Experiments run for this more extensive evaluation involve less samples: 20 QEMU runs and 20 massaging executions per run, as opposed to Table II where we perform 40 QEMU runs and 20 massaging executions per run.

Vulnerable cache	Target cache	Success rate											
		Adjacency only						Adjacency + object alignment					
		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)	
		Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI
`kmalloc-16`	`kmalloc-64`	99.87	(99.29, 99.98)	100.0	(99.52, 100)	99.87	(99.29, 99.98)	99.75	(99.09, 99.93)	99.87	(99.29, 99.98)	100.0	(99.52, 100)
`kmalloc-16`	`kmalloc-128`	100.0	(99.52, 100)	99.87	(99.29, 99.98)	100.0	(99.52, 100)	100.0	(99.52, 100)	100.0	(99.52, 100)	100.0	(99.52, 100)
`kmalloc-64`	`kmalloc-16`	100.0	(99.52, 100)	99.87	(99.29, 99.98)	99.87	(99.29, 99.98)	99.87	(99.29, 99.98)	99.62	(98.90, 99.87)	99.87	(99.29, 99.98)
`kmalloc-64`	`kmalloc-128`	100.0	(99.52, 100)	99.875	(99.29, 99.98)	99.75	(99.09, 99.93)	100.0	(99.52, 100)	100.0	(99.52, 100)	100.0	(99.52, 100)
`kmalloc-128`	`kmalloc-16`	100.0	(99.52, 100)	99.75	(99.09, 99.93)	98.87	(97.87, 99.41)	100.0	(99.52, 100)	99.87	(99.29, 99.98)	99.125	(98.20, 99.57)
`kmalloc-128`	`kmalloc-64`	99.87	(99.29, 99.98)	98.75	(97.71, 99.32)	99.75	(99.09, 99.93)	100.0	(99.52, 100)	99.25	(98.37, 99.65)	99.25	(98.37, 99.65)
`kmalloc-2k`	`kmalloc-4k`	100.0	(99.52, 100)	100.0	(99.52, 100)	98.75	(97.71, 99.32)	100.0	(99.52, 100)	100.0	(99.52, 100)	98.0	(96.77, 98.76)
`kmalloc-4k`	`kmalloc-2k`	98.37	(97.24, 99.05)	89.87	(87.59, 91.78)	97.25	(95.87, 98.18)	98.87	(97.87, 99.41)	90.75	(88.54, 92.57)	96.75	(95.28, 97.77)

(a) Same-order massaging ($n_v = n_t$); `SLAB_VIRTUAL` enabled. The experimental results show that the massaging reliability remains comparable to Table II (`SLAB_VIRTUAL` disabled) in Section V.

Vulnerable cache	Target cache	Success rate											
		Adjacency only						Adjacency + object alignment					
		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)	
		Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI
`kmalloc-2k`	`kmalloc-16`	100.0	(99.52, 100)	98.87	(97.87, 99.41)	98.62	(97.55, 99.23)	99.87	(99.29, 99.98)	98.87	(97.87, 99.41)	98.62	(97.55, 99.23)
`kmalloc-2k`	`kmalloc-64`	99.87	(99.29, 99.98)	99.12	(98.20, 99.57)	99.75	(99.09, 99.93)	99.87	(99.29, 99.98)	99.0	(98.04, 99.49)	99.75	(99.09, 99.93)
`kmalloc-2k`	`kmalloc-128`	99.87	(99.29, 99.98)	98.62	(97.55, 99.23)	99.37	(98.54, 99.73)	99.87	(99.29, 99.98)	99.12	(98.20, 99.57)	99.75	(99.09, 99.93)
`kmalloc-4k`	`kmalloc-16`	99.37	(98.54, 99.73)	97.12	(95.72, 98.08)	98.87	(97.87, 99.41)	99.87	(99.29, 99.98)	97.87	(96.62, 98.67)	98.87	(97.87, 99.41)
`kmalloc-4k`	`kmalloc-64`	99.37	(98.54, 99.73)	97.12	(95.72, 98.08)	99.37	(98.54, 99.73)	99.50	(98.72, 99.80)	97.12	(95.72, 98.08)	99.0	(98.04, 99.49)
`kmalloc-4k`	`kmalloc-128`	99.75	(99.09, 99.93)	96.87	(95.43, 97.87)	99.50	(98.72, 99.80)	99.50	(98.72, 99.80)	96.75	(95.28, 97.77)	99.37	(98.54, 99.73)

(b) Cross-order massaging ($n_v > n_t$); `SLAB_VIRTUAL` enabled. The experimental results show that the massaging reliability remains comparable to Table II (`SLAB_VIRTUAL` disabled) in Section V.

Vulnerable cache	Target cache	Success rate											
		Adjacency only						Adjacency + object alignment					
		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)		Idle (CPU pinning)		Idle (No CPU pinning)		Noise (CPU pinning)	
		Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI	Avg. (%)	95%-CI
`kmalloc-512`	`kmalloc-2k`	31.64	(29.15, 34.24)	8.28	(6.89, 9.92)	45.08	(42.37, 47.81)	33.36	(30.83, 35.99)	8.28	(6.89, 9.92)	44.37	(41.67, 47.11)
`kmalloc-512`	`kmalloc-4k`	28.51	(26.11, 31.05)	7.42	(6.11, 8.99)	42.73	(40.05, 45.46)	23.98	(21.72, 26.40)	7.89	(6.54, 9.50)	43.12	(40.44, 45.85)
`kmalloc-256`	`kmalloc-2k`	22.26	(20.07, 24.62)	8.83	(7.39, 10.51)	22.19	(20.00, 24.54)	21.87	(19.70, 24.22)	9.92	(8.40, 11.68)	22.58	(20.37, 24.95)
`kmalloc-256`	`kmalloc-4k`	21.33	(19.17, 23.66)	10.00	(8.47, 11.76)	23.90	(21.65, 26.32)	20.78	(18.65, 23.10)	9.84	(8.33, 11.60)	22.81	(20.60, 25.19)
`kmalloc-256`	`kmalloc-512`	45.62	(42.91, 48.36)	19.30	(17.23, 21.55)	47.03	(44.31, 49.77)	32.42	(29.91, 35.03)	19.92	(17.82, 22.20)	44.84	(42.14, 47.58)

(c) Cross-order massaging ($n_v < n_t$); `SLAB_VIRTUAL` disabled. The success rate drops significantly in this case for reasons described in Section IV-C. Vulnerable caches with $n_v = 0$ are omitted as their success rate is negligible.

TABLE X: Success rate for the cross-cache massaging. Table Xa and Table Xb show the PCPLOST success rate for, respectively, the same-order ($n_v = n_t$) and cross-order ($n_v > n_t$) cases with `SLAB_VIRTUAL` enabled. Table Xc, instead, presents the success rate for the cross-order massaging with $n_v < n_t$ and `SLAB_VIRTUAL` disabled.

Latency		Bandwidth	
Benchmark	Overhead	Benchmark	Speedup
Simple syscall	−1.01%	AF_UNIX sock stream	1.92%
Simple read	2.57%	Pipe	−0.92%
Simple write	0.17%	File /var/tmp/XXX write	−1.18%
Simple stat	0.40%	Socket using localhost	0.91%
Simple fstat	−0.32%	read	−15.83%
Simple open/close	2.36%	read open2close	−15.86%
Select on 10 fd's	−2.46%	Mmap read	0.49%
Select on 100 fd's	0.12%	Mmap read open2close	−8.89%
Select on 250 fd's	0.12%	libc bcopy unaligned	−1.60%
Select on 500 fd's	−2.80%	libc bcopy aligned	−1.59%
Select on 10 tcp fd's	0.11%	Memory bzero	−0.25%
Select on 100 tcp fd's	−2.50%	unrolled bcopy unaligned	−7.35%
Select on 250 tcp fd's	−0.29%	unrolled partial bcopy unaligned	−6.56%
Select on 500 tcp fd's	−1.04%	Memory read	0.18%
Signal handler installation	−1.11%	Memory partial read	−1.06%
Signal handler overhead	−0.42%	Memory write	−3.95%
Protection fault	7.79%	Memory partial write	−0.30%
Pipe latency	1.03%	Memory partial read/write	0.15%
AF_UNIX sock stream latency	3.27%		
Process fork+exit	3.51%		
Process fork+execve	0.99%		
Process fork+/bin/sh	2.48%		
TCP latency using localhost	0.72%		
UDP latency using localhost	0.27%		
TCP/IP connection cost to localhost	9.37%		
Geomean	0.90%	Geomean	−3.58%

TABLE XI: Overhead and speedup of `SLAB_VIRTUAL_GP` across, respectively, latency and bandwidth benchmarks from the LMBench suite. Benchmark results (30 runs) were obtained on a bare-metal AMD system equipped with 16 cores and 16 GB of RAM.

Appendix D
Artifact Appendix

This appendix provides a self-contained guide for setting up, executing, and validating the artifact accompanying our paper.

A. Description & Requirements

1) How to access: The artifact is publicly available on Zenodo [60].

2) Hardware dependencies: This section outlines the minimum hardware specifications required to successfully run the artifact and reproduce the experimental results.

- **Machine**: A standard commodity `x86_64` laptop or workstation.
- **CPU**: Minimum of **8 cores** to match the QEMU instance used in our experiments.
- **Memory**: At least **16 GB of RAM** to ensure stable execution of the virtual machines.
- **Storage**: A minimum of **5 GB of free disk space** is required to store the Docker image and temporary evaluation artifacts.

3) Software dependencies: This section outlines the software environment required to run the artifact.

- **Operating system**: A Linux-based host system is recommended. The artifact was tested on Ubuntu 22.04 (Jammy) with kernel version `5.15.0-140-generic`.
- **Programs**: To extract and build the containerized environment, the following dependencies are required for the host:
  - **GNU Make** — tested with version 4.3 (`x86_64-pc-linux-gnu`).
  - **Docker** — tested with version 27.5.1 (build `27.5.1-0ubuntu3~22.04.2`).
  - **GNU GCC** — tested with version 11.4.0.
  - **GNU tar** and **zstd** — tested with versions 1.34 and 1.4.8, respectively.

  Any reasonably recent version of these tools should work as well.

4) Benchmarks: None.

B. Artifact Installation & Configuration

To set up the environment for artifact evaluation, reviewers should select a target directory and place the provided archive file in it. Extract the contents using:

```
$ tar -xf ndss26ae-fall-paper66-
         submission_packaged_artifact.zst
```

After extraction, navigate to the resulting directory:

```
$ cd pcplost-artifact-evaluation
```

Once in the root directory, build the Docker container by running:

```
$ make
```

In addition to creating the container, the provided Makefile also compiles the user-space programs responsible for implementing the massaging technique (`pcplost_massaging.c`) and for collecting timing data for the side-channel (`pcplost_timing.c`).

From this point, the evaluation process can begin following the instructions provided in the accompanying documentation.

C. Experiment Workflow

All experiments can be executed from the main script directory located at `pcplost-artifact-evaluation/artifacts/scripts`, once inside the Docker container. Upon test execution completion, reviewers may occasionally encounter issues with terminal autocompletion. To mitigate this, we recommend the following workflow:

1) Run the desired experiment from within the container.
2) Exit the container using `exit` or `Ctrl+D`.
3) From the host system, navigate to the root directory (`pcplost-artifact-evaluation`) and reconnect to the container using `make connect`.
4) Once reconnected, results can be inspected in the `/tmp` directory inside the container.

D. Major Claims

- **(C1)**: The PCPLOST massaging technique achieves high reliability (exceeding 90% in most cases) against generic `kmem`-caches in the same-order scenario on a Linux kernel without experimental mitigations. This claim is substantiated by experiment (E1), with results reported in Table II, case $n_v = n_t$ (Section V-A).
- **(C2)**: The PCPLOST massaging technique maintains high reliability (exceeding 90% in most cases) against generic `kmem`-caches in the same-order scenario on a Linux kernel with the experimental `SLAB_VIRTUAL` mitigation. This claim is validated by experiment (E2), with results presented in Table Xa (Appendix C).
- **(C3)**: The PCPLOST massaging technique achieves high reliability (around 90% under idle load) against generic `kmem`-caches in the cross-order scenario on a Linux kernel without experimental mitigations. This claim is supported by experiment (E3), with results shown in Table II, case $n_v > n_t$ (Section V-A).
- **(C4)**: The side-channel used by PCPLOST can detect three kernel-level events: object allocations, page allocations from the PCP list and page allocations from the `free_area`. This claim is supported by experiment (E4), with results shown in Figure 3.

E. Evaluation

a) Experimental Scope and Scaling: To improve reproducibility and reduce runtime during artifact evaluation, we scaled down the number of QEMU and attack runs per session. Specifically, we use 10 QEMU runs and 5 attack runs per QEMU session, compared to 20 QEMU runs and 20 attack runs in the original experiments. While this reduction may slightly affect average values and confidence intervals, the results remain representative and statistically meaningful.

b) Core Experiments (E1–E3) [20 human-minutes + 12 compute-hours]: The following experiments are executed via the `massaging_evaluation` script located in `artifacts/scripts`. The entry point is the `start` script, located in the same directory. Running this script performs three core evaluations:

- **(E1):** Reliability of PCPLOST same-order massaging against generic `kmem`-caches (vanilla kernel). Results are reported in Table I, case $n_v = n_t$.
- **(E2):** Same as (E1), but with the `SLAB_VIRTUAL` mitigation enabled. Results are shown in Table VI.A (Appendix).
- **(E3):** Reliability of PCPLOST cross-order massaging ($n_v > n_t$) against generic `kmem`-caches (vanilla kernel). Results are reported in Table I, case $n_v > n_t$.

[Preparation]

- Ensure Docker is installed and the user is part of the `docker` group.
- From the `pcplost-artifact-evaluation` directory, run `make` to build the container environment.
- If KVM is not available on the host, comment out the `-cpu host` and `-accel kvm` lines in the QEMU scripts located in `artifacts/scripts`. This however will affect the final results.

[Execution]

- Enter the Docker container and execute:

```
$ ./start
```

- This command runs all three experiments (E1–E3) sequentially.

[Results]

- Results are stored in the `/tmp/csv_results` directory. Each file is timestamped and named according to the experiment:
  - (E1): `attack_eval_order0_vanilla_partial.csv`
  - (E2): `attack_eval_order0_slabvirt.csv`
  - (E3): `attack_eval_crossorder_vanilla.csv`
- Success rates are expressed as decimal values (e.g., 0.73). Refer to Tables I and VI.A for percentage representations.

c) Experiment E4 [25 human-minutes + 1 compute-hour]: This experiment evaluates the feasibility of a side-channel attack targeting SLUB-related kernel allocations, using system call timing as the observation vector. It is executed via the `side_channel_evaluation` script located in `artifacts/scripts`, with `measure` as the main entry point.

- **(E4):** Side-channel measurement of allocation behavior in the Linux kernel. The timing-based channel distinguishes between three types of allocation events:
  1) Object allocation from SLUB caches.
  2) Page allocation from the per-CPU page (PCP) lists.
  3) Page allocation from the `free_area` (buddy allocator).

[Preparation]

- Ensure Docker is installed and the current user belongs to the `docker` group.
- From the `pcplost-artifact-evaluation` directory, run `make` to build the container environment.
- If KVM is not available on the host, comment out the `-cpu host` and `-accel kvm` lines in the QEMU scripts located in `artifacts/scripts`. This however will affect the final results.

[Execution]

- Enter the Docker container and execute:

```
$ ./measure
```

- This command launches a fresh QEMU instance and collects timing measurements, using the execution time of system calls as observation vector. These timing measurements are then linked to data coming from `ftrace`, to later categorize the allocation type (with the observed timing).

[Results]

- Results are stored in the `/tmp/timing_results` directory. Each file is named according to the cache being evaluated. For example, `auto-plotting-kmalloc-2k.csv` contains timing data for the `kmalloc-2k` cache.
- Measurements are reported in clock cycles. To validate the side-channel, the average timing differences between allocation types should be significant:
  - Object vs. PCP list page: typically a few hundred cycles.
  - PCP list page vs. `free_area` page: similarly distinguishable.
- The `measure` script prints the median and average clock cycle values for each allocation category, facilitating direct comparison.
- An appropriately structured output would be a CSV file containing timing data for all three allocation types — *object*, *PCP list* and *buddy* — with each type's timing clearly distinguishable from the others based on differences in clock cycles.

F. Notes

E3 results can vary depending on hardware characteristics and may yield lower success rates (e.g., around 50%) for some cache pairs. To achieve higher reliability, the `free_area` threshold parameter used by the side-channel needs to be tuned for the specific configuration, using the timing measurement script. The resulting number can be used as `FREE_AREA_THR` in `artifacts/pcplost/lib/timing_utils.h`. On some machines, this number can require further manual modification.