HEAP LOCALIZATION: Cache Side-Channel based Linux Kernel Heap Exploit Techniques

Yoochan Lee*, Sihyun Roh†, Hyuk Kwon‡, Byoungyoung Lee†, and Thorsten Holz*
*Max Planck Institute for Security and Privacy (MPI-SP)
†Seoul National University
‡Theori, Inc.
{yoochan.lee, thorsten.holz}@mpi-sp.org, {sihyeonroh, byoungyoung}@snu.ac.kr, pwn3r@theori.io

***Abstract*—As kernel mitigations that reduce exploit success rates continue to be deployed, exploitation techniques have become increasingly sophisticated to maintain high reliability under such constrained environments. These techniques, however, fundamentally rely on precise knowledge of the *locations* of kernel heap objects—information that is not available to unprivileged users and forces attackers to depend on coarse and speculative inferences about allocator behavior. As a result, existing exploit techniques inevitably exhibit structural failure cases when vulnerable or target objects occupy unexpected intra-page positions.**

To address this limitation, we present HEAP LOCALIZATION, the first primitive that enables object-level heap layout inference in the Linux kernel. HEAP LOCALIZATION recovers the precise intra-page offset of kernel heap objects by exploiting deterministic VIPT L1 cache behavior, enabling deterministic object placement without requiring memory disclosure. By providing exact object-location information, HEAP LOCALIZATION eliminates layout-induced failure cases and transforms several previously probabilistic heap exploitation techniques into deterministic ones. Our evaluation demonstrates that HEAP LOCALIZATION consistently localizes and reliably positions objects, achieving average success rates of 99.3% in the idle state and 95.7% under heavy load. We further demonstrate its practicality by applying HEAP LOCALIZATION to real-world kernel vulnerabilities, where it significantly increases exploit reliability.

1. Introduction

As attacks targeting the Linux kernel continue to increase, numerous kernel mitigations have been introduced to strengthen system security. Broadly, these defenses can be categorized into two classes: (i) those that prevent vulnerabilities from occurring in the first place, and (ii) those that hinder exploitation techniques. The first class [1, 11, 19, 24, 25, 51, 58] focuses on eliminating the root causes of a specific type of vulnerabilities. For example, to prevent uninitialized-use vulnerabilities, mitigations such as [11, 25] enforce explicit initialization of heap or stack objects upon allocation. The second class [4, 10, 12, 14, 15, 23, 26, 34, 56] aims to complicate exploitation by altering kernel mechanisms that existing exploitation techniques depend on. For instance, slab freelist randomization [23] modifies the SLUB allocator's object allocation policy to invalidate the assumptions made by traditional heap object layout manipulation techniques [20, 22, 54].

Despite these defenses, attackers continue to develop increasingly sophisticated exploitation techniques to achieve successful kernel compromises. These state-of-the-art exploitation techniques can be broadly categorized into two classes: (i) techniques that bypass existing mitigations, and (ii) techniques that exploit newly introduced kernel features that current mitigations do not cover. The first class [6, 7, 32, 39, 40, 53, 55] identifies weaknesses in mitigation designs and leverages those weaknesses to achieve the same attack outcomes that the mitigations intended to block. For example, Pspray [39] exploits a design flaw in the slab freelist randomization, effectively restoring the exploit success rate to the level observed before that mitigation was introduced. The second class [3, 30, 33, 42, 62] finds a new kernel feature, which is not utilized before, and suggests a methodology to achieve privilege escalation. For instance, the temporal cross-cache attack [62] shows that the page management mechanism in the SLUB allocator can be abused for exploitation. This technique operates beyond the scope of mitigations related to the allocation pool [4, 12, 26].

However, existing exploitation techniques are non-deterministic because they rely on fragile assumptions. Deterministic exploitation typically requires precise information about kernel memory (e.g., the exact location of objects), which is not available to unprivileged users. Consequently, attackers attempt to infer kernel object layouts from observable kernel mechanisms and manipulate allocations to approximate a desired layout. These inferences are inherently probabilistic, resulting in layouts different from the attacker's expectations, which eventually leads to exploitation failure because of overwriting unintended data.

To clearly illustrate the limitations of object layout inference, we present three representative exploit techniques as examples: (1) an out-of-bounds exploit using Pspray, (2) a temporal cross-cache attack, and (3) a spatial cross-cache attack. First, Pspray [39] attempts to co-locate the vulnerable and target objects within the same page to increase the

likelihood of adjacency. However, Pspray only guarantees that both objects reside on the same page and cannot ensure their adjacency. Specifically, if the vulnerable object happens to occupy the final slot of a page, it becomes adjacent to an object on the next page rather than the intended target, breaking the exploit's assumptions and causing it to fail. Second, temporal cross-cache attacks require precise alignment between the vulnerable and target objects for successful exploitation. This is because each cache serves objects of different sizes, and misalignment shifts the overwrite offset, corrupting unintended fields within the target object. Finally, spatial cross-cache attacks require the vulnerable object to be placed in the final slot of a page so that its overflow extends into the next page, where the sprayed target objects may reside. Achieving such boundary placement (i.e., where the vulnerable object resides at the end of a page) is challenging since page boundaries are allocator-controlled and influenced by fragmentation and randomization. These challenges reveal a fundamental limitation: attackers lack a primitive that exposes the exact intra-page slot of a kernel heap object. Existing methods only infer coarse relationships (e.g., same page or same cache), without determining the precise slot a vulnerable object occupies.

In this paper, we present HEAP LOCALIZATION, the first primitive that enables object-level heap layout inference in the Linux kernel. HEAP LOCALIZATION recovers the exact intra-page offset of a kernel heap object using deterministic VIPT L1 cache indexing, without requiring memory disclosure. With this capability, multiple exploitation techniques shift from probabilistic placement to deterministic, attacker-guided manipulation. The cache side-channel serves only as an implementation mechanism for realizing this primitive. To the best of our knowledge, although cache side-channel attacks [29, 47, 63] have been used to infer memory-access patterns, HEAP LOCALIZATION is the first to employ them for systematic localization of kernel heap objects. With this capability, HEAP LOCALIZATION removes the need for complex heap manipulation in prior work (e.g., retries, alignment, and spraying).

Using the location information provided by HEAP LOCALIZATION, we develop more reliable versions of several heap exploitation techniques. For out-of-bounds exploits, we trigger the vulnerability only after verifying that the vulnerable object is not placed in a page's final slot. Similarly, to reliably align objects for a temporal cross-cache attack, we place the vulnerable object in a slot that yields the required offset (e.g., the first slot or another offset that is a common multiple of both object sizes). Finally, after arranging the vulnerable and target pages to be adjacent, we trigger an out-of-bounds vulnerability only when the vulnerable object occupies the final slot of its page, ensuring adjacency with the target object on the neighboring page.

To demonstrate the effectiveness of HEAP LOCALIZATION, we conducted two sets of experiments on Linux kernel v6.16.9. Using a synthetic object, we show that HEAP LOCALIZATION's localization precision is practical in both the idle state (99.3% on average) and busy state (95.7% on average). We then test HEAP LOCALIZATION's applicability by developing six end-to-end privilege-escalation exploits based on approach approach, using four real-world kernel vulnerabilities.

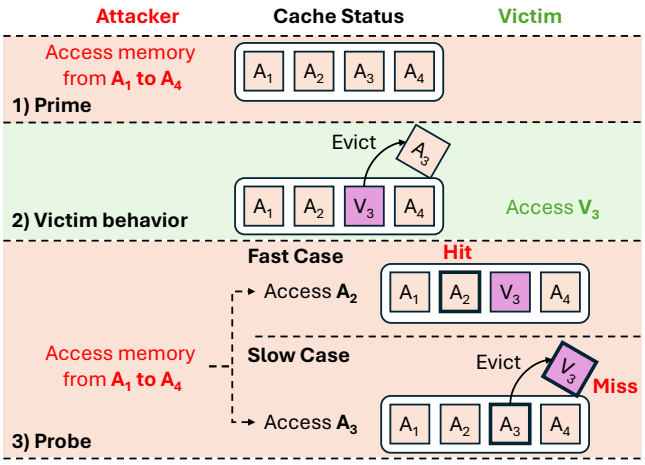

Figure 1: Overview of the Prime+Probe attack: The attacker determines which cache line the victim accessed by measuring access times during the probe phase.

To summarize, we make the following key contributions:

- We identify a fundamental gap in current kernel exploitation techniques: the absence of a primitive for determining the exact intra-page slot of a kernel heap object, which is the root cause of widespread structural failures.
- We introduce HEAP LOCALIZATION, the first primitive that enables object-level heap layout inference in the Linux kernel. HEAP LOCALIZATION recovers the precise intra-page offset of a kernel heap object using deterministic VIPT L1 behavior, enabling deterministic heap placement.
- We validate HEAP LOCALIZATION through controlled experiments using synthetic objects, and demonstrate its practicality by building six end-to-end exploits against real-world kernel vulnerabilities.

We release the proof-of-concept code of our approach at https://github.com/MPI-SysSec/Heap-Localization to foster open science.

2. Background

In this section, we provide brief background information on cache side-channel attacks (§2.1), heap management in the Linux kernel (§2.2), and our threat model (§2.3).

2.1. Cache Side-Channel

Cache side-channel attacks [5, 29, 47, 63] exploit timing differences between cache hits and misses to infer a victim's memory-access patterns. Modern CPUs keep recently used data in small, fast caches; by observing how the cache is filled and evicted, an attacker can learn which regions of memory a co-resident program accesses. In practice, repeatedly timing accesses to carefully chosen addresses reveals whether data

was served from the cache (low latency) or from main memory (high latency), thereby leaking information about the victim's execution.

Prime+Probe. In this work, we focus on *Prime+Probe* [47], a cache side-channel attack that does not require shared memory between attacker and victim. As shown in Figure 1, the attacker (in our setting, the user process) first **primes** selected cache *sets* by filling all their ways with an *eviction set* of addresses. When the victim (the kernel thread) executes, its accesses to addresses mapping to those sets may evict the attacker's cache lines. The attacker then **probes** by measuring the access latency to the eviction-set addresses: slower accesses indicate evictions. Thus, this attack enables an adversary to infer which cache lines the victim used.

2.2. Heap Management in Linux Kernel

The Linux kernel heap is managed by two allocators: the Buddy and the SLUB allocators [13]. The Buddy allocator manages page-level memory regions and provides pages to higher-level allocators. The SLUB allocator then handles object allocation within those pages. When the existing heap space becomes insufficient, the kernel requests additional, non-contiguous pages from the Buddy allocator.

To reduce memory fragmentation and improve security, the SLUB allocator organizes pages into logical units called *caches*. The kernel maintains more than a hundred caches, broadly divided into two categories: *general caches* and *dedicated caches*. General caches are organized by object size (e.g., `kmalloc-32`, `kmalloc-64`) and are used for most kernel objects. Dedicated caches are assigned to specific object types (e.g., `cred` and `task_struct`) to isolate security-critical object from others. Each page belonging to a cache is further divided into multiple *slots*, each slot representing a single allocation unit for that cache. The slot size corresponds to the object size managed by the cache (e.g., 32 bytes for `kmalloc-32`), and thus determines how many objects can reside within a single page.

2.3. Threat Model

We assume an unprivileged local attacker capable of executing arbitrary code, i.e., the typical starting point for full-chain kernel exploit scenarios [2], along with the presence of a kernel heap memory-corruption vulnerability. The kernel is assumed to run a recent upstream Linux version (v6.16 at the time of writing) and may enable common mitigations such as KPTI [15], freelist randomization [23], SMAP [14], SMEP [34], and ASLR [10], as well as cache-related defenses including memory cgroups [4], randomized kmalloc [26], and slab buckets [12]. These cache-related mitigations may change which cache simultaneously contains the vulnerable and localization objects, but they do not affect our core technique: once such a cache is identified, our object-level localization operates entirely within that cache and relies solely on VIPT L1 hardware indexing—a hardware property independent of SLUB internals or allocation diversification.

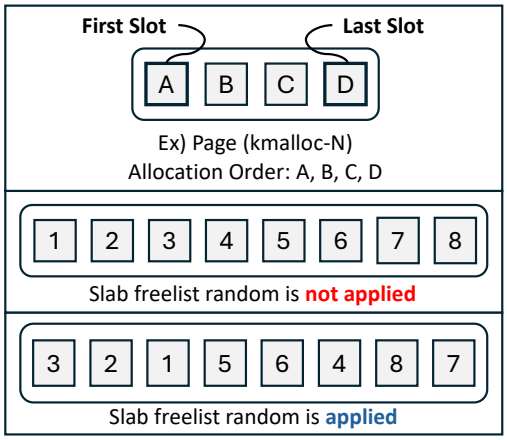

Figure 2: The key mechanism of slab freelist random mitigation.

3. Previous Heap Exploit Techniques and Limitations

In the following, we summarize several representative heap exploitation techniques that have been widely used in both prior and ongoing research. While some have been successfully adopted in real-world attacks, others remain largely theoretical. We then discuss their limitations and the challenges they face in the context of modern kernel exploitation.

3.1. Out-Of-Bounds Exploit

To thwart traditional heap placement techniques, kernel developers introduced slab freelist randomization [23], which breaks the sequential allocation assumption by randomizing the order of free-slot selection, as shown in Figure 2. Previously, attackers had relied on placement techniques such as heap spray [20] and heap feng shui [54] to place a vulnerable object adjacent to a target object, enabling out-of-bounds corruption. These techniques exploited the largely sequential nature of object allocation within a cache, allowing attackers to predict and manipulate heap layout. As a result, slab freelist randomization significantly reduces the effectiveness of such placement-based exploits.

Pspray. Pspray [39] was proposed to bypass slab freelist randomization. Pspray exploits a design weakness in SLUB's object allocation mechanism to infer the occupancy status of object slots on an in-use page. More specifically, Pspray leverages a timing side channel to detect when a new page is assigned. A newly assigned page contains no allocated objects; therefore, the attacker can predict the object layout on that page. This technique increases the success rate of OOB exploits even when slab freelist randomization is enabled.

Limitation of Pspray. Pspray's layout prediction is not perfect and has identifiable failure cases. A primary failure case occurs when the vulnerable object is allocated into the last slot of a page; in that case, the adjacent slot lies on the next page and is not the intended target. This issue becomes

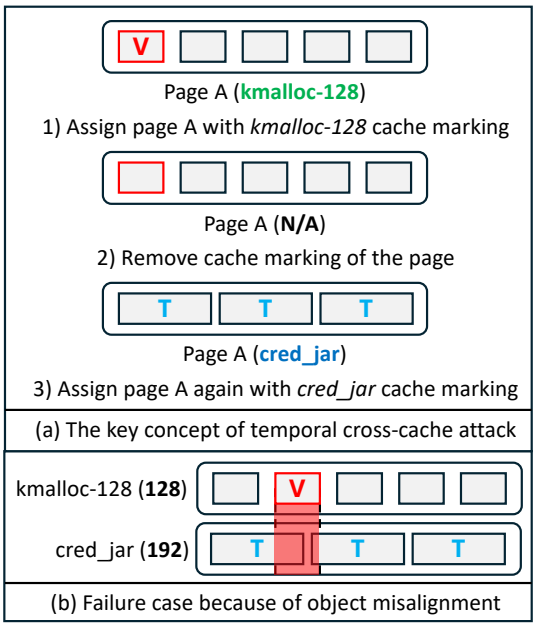

Figure 3: Key mechanism of temporal cross-cache attack and failure case because of misalignment.

more significant when only a few objects can be allocated per page. For example, in `kmalloc-8192`, which accommodates four objects per page, the final-slot case results in a 25% theoretical failure rate. Therefore, Pspray does not guarantee deterministic adjacency under this limitation.

3.2. Use-After-Free Exploit in Uncommon Caches

Use-after-free (UAF) exploitation requires that a freed vulnerable object be reallocated (aliased) by a target object of a different type in the same allocation slot. Concretely, the target object must satisfy two conditions: (i) it must belong to the same cache as the vulnerable object, and (ii) it must be of a different type to enable type confusion. These conditions are often met in common caches (e.g., `kmalloc-64`, which hosts many object types). By contrast, they are rarely satisfied in uncommon or dedicated caches, and thus many UAFs in those caches have typically been regarded as unexploitable.

Temporal Cross-Cache Attack. The temporal cross-cache attack [3, 30, 42, 45, 62] was introduced to relax the constraints on target-object selection. The key idea of the temporal cross-cache attack is changing the cache marking of a page, as described in Figure 3-(a). To this end, it exploits SLUB's page management behavior: when certain conditions are satisfied, SLUB returns the corresponding pages to the buddy allocator. As a result, any cache-specific markings on those pages are cleared. Subsequently, if the attacker reallocates the same page through a different slab cache, the page retains the same memory address but is now associated with the new cache. This allows the attacker to select a target object from the newly assigned cache, effectively removing the prior constraints on target-object selection.

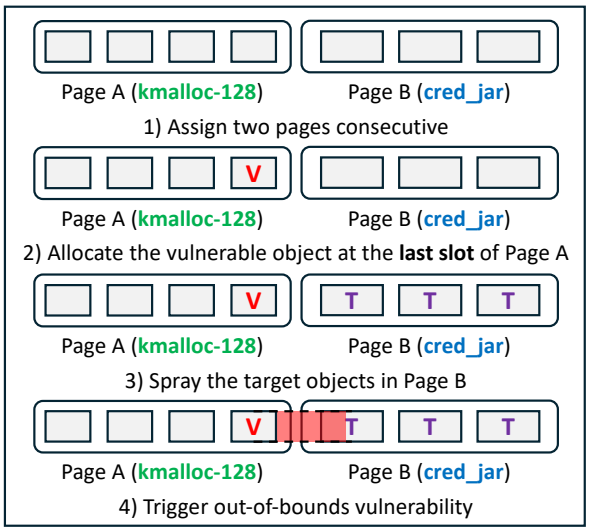

Figure 4: Theoretical concept of spatial cross-cache attack.

Limitation of Temporal Cross-Cache Attack. Since the temporal cross-cache attack does not preserve memory alignment, it is difficult to precisely target specific data within the target object, as illustrated in Figure 3-(b). The vulnerable and target objects involved in the temporal cross-cache attack typically have different sizes—for example, an object in `kmalloc-128` is 128 bytes, whereas an object in `cred_jar` is 192 bytes. As a result, these objects are generally misaligned, except at their common multiples (e.g., 1920). In other words, except for such special cases, the objects are misaligned, causing the attacker to overwrite unintended data and ultimately leading to exploitation failure.

3.3. Out-Of-Bounds Exploit in Uncommon Caches

Similar to UAF exploits in uncommon caches, exploiting an OOB vulnerability also becomes challenging in such environments. An OOB exploit requires that the vulnerable and target objects reside in the same cache so that they can be placed adjacently. However, uncommon or dedicated caches contain only a limited number of object types, which makes it difficult to find suitable targets that meet this requirement. As a result, many OOB vulnerabilities in these caches have likewise been regarded as unexploitable.

Spatial Cross-Cache Attack. To overcome this limitation, attackers proposed the concept of the spatial cross-cache attack, illustrated in Figure 4. Using a specialized technique, they arrange two pages associated with different caches to be adjacent. On the preceding page, the vulnerable object is allocated in the last slot, while the attackers spray the target object on the following page. In this configuration, the vulnerable and target objects become adjacent despite belonging to different pages. Consequently, an OOB primitive can corrupt the target object under this arrangement.

Challenge in Spatial Cross-Cache Attack. The technique remains largely theoretical because there is no reliable

method to force the vulnerable object into the last slot of a page. An attacker cannot predict which object slot will be used next, so placement at the desired slot cannot be guaranteed. One possible mitigation is a brute-force strategy that repeatedly triggers the vulnerability until the vulnerable object happens to occupy the last slot and thus corrupts the target on the following page. However, repeated exploitation attempts increase the probability of kernel panics and produce observable anomalies that defenders can detect; therefore, attackers must avoid such behaviour. Accordingly, to make the spatial cross-cache attack practical, a deterministic allocation technique that reliably places the vulnerable object in the page's final slot is required.

3.4. Insight from the Analysis

The limitations above share a common root cause: knowledge of slot occupancy. For OOB exploitation, ensuring that the vulnerable object is not allocated in the page's final slot prevents the primitive from spilling into the next page, thereby enabling deterministic corruption within the same page. For temporal cross-cache attacks, aliasing the vulnerable and target objects with the expected alignment increases the attack's success probability. For spatial cross-cache attacks, reliably placing the vulnerable object in the page's final slot enables adjacent placement across page boundaries and thus makes the attack practical. In short, a deterministic (or high-confidence) slot-occupancy technique would substantially increase the success rates of these three exploit techniques.

4. Heap Localization

Heap Localization leverages a cache side-channel attack to localize objects within a page. Concretely, it exploits the fact that the L1 data cache is virtually indexed and physically tagged (VIPT): the cache set (index) is determined by the low 12 bits of the virtual address.

4.1. Core Mechanisms of Heap Localization

To achieve precise heap object localization, Heap Localization builds upon several core mechanisms.

4.1.1. How Memory Addresses Map to Cache Lines. The memory address inherently determines the cache line it maps to. Figure 5-(a) illustrates the characteristics of the CPU cache. Each cache line is 64 bytes, and thus a 4 KB range contains 64 cache-line slots (64 lines × 64 bytes = 4 KB). When accessing a specific memory address, the cache line used is determined by the address itself, as illustrated in Figure 5-(b). Naturally, not all bits of a memory address influence which cache line is used, since cache-line indexing repeats every 4 KB range. More precisely, dividing the lower 12 bits of a memory address by 64 indicates which cache line within a 4 KB range the address maps to. For example, if a user accesses a specific memory address

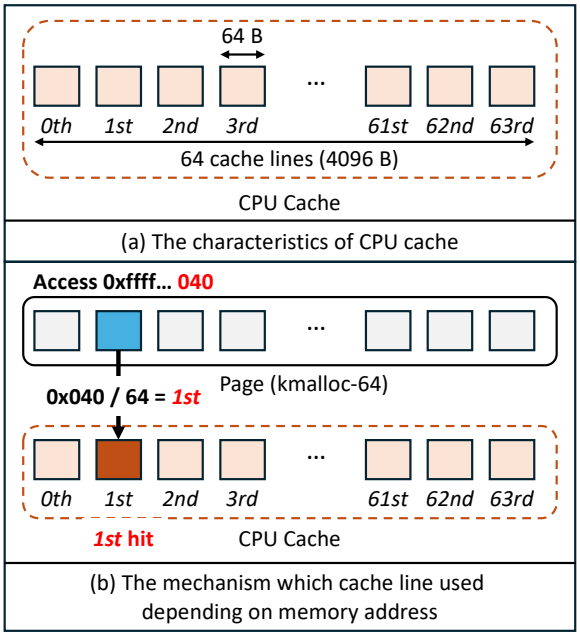

Figure 5: How memory addresses map to cache lines.

(e.g., *0xffff...040*), the upper 52 bits do not affect cache-line selection. In contrast, the lower 12 bits (i.e., 0x040) determine which cache line is accessed — in this case, the first cache line. As a result, if we know the memory address, we can predict which cache line to be used.

4.1.2. Inferring Memory Addresses from Cache-Line Accesses. By reversing this mapping mechanism, we can infer the partial memory address from the accessed cache line. Recall that, the cache line is determined by the lower 12 bits of the memory address. Therefore, although the full 64-bit memory address cannot be recovered, the lower 12 bits can be inferred, which correspond to the offset within a page.

Even with only a partial memory address, the leakage has significant implications for kernel-heap exploitation. Heap exploits commonly rely on placing the vulnerable and target objects within the same page; various grooming techniques (e.g., Pspray [39]) are used to enforce this condition. Under this same-page assumption, the upper bits only identify the page frame and are often irrelevant for exploitation, whereas the lower 12 bits – the intra-page offset – are decisive. Hence, the ability to recover the page offset (i.e., lower 12 bits of memory address) from cache-line accesses substantially facilitates exploit development.

4.1.3. Cache-Line Verification through Prime+Probe. To determine which cache lines are accessed, we employ Prime+Probe [47] (see §2.1). Prime+Probe monitors a victim's memory-access pattern by priming targeted cache sets and later probing them for evictions. In our setup, the kernel process that accesses the *localization object* is treated as the victim. Consequently, Prime+Probe can identify which

cache lines are accessed by the localization object, enabling inference of that object's partial memory address.

4.2. Localization Object

In HEAP LOCALIZATION, the *localization object* is used for Prime+Probe rather than directly targeting the vulnerable object. This is because directly targeting the vulnerable object inherently involves uncertainty. Prime+Probe detects cache-line state changes caused by all memory accesses during the execution of the victim process. However, because the vulnerable object exhibits diverse and noisy memory-access patterns across different executions, it is unsuitable for reliable object localization.

Requirement. For HEAP LOCALIZATION to operate reliably, the *localization object* must satisfy several conditions. First, there must exist three distinct functions that allow the user to allocate, deallocate, and access the localization object. Second, the function responsible for accessing the localization object should minimize additional memory accesses unrelated to it. The fewer non-localization object accesses occur, the higher the localization success rate becomes.

Note that we manually identified several candidates and selected the `msg_msg` object as the *localization object* because it incurs less memory-access activity than the others.

4.3. Process of HEAP LOCALIZATION

Figure 6 shows the overall process and the flow between phases in HEAP LOCALIZATION.

4.3.1. Phase 1: Empty Page Preparation. First, HEAP LOCALIZATION starts with preparing an empty page; no object is allocated. This is done to ensure the vulnerable and target objects are allocated within the same page, thereby amplifying HEAP LOCALIZATION's partial memory address inference. If this phase would be omitted, the vulnerable and target objects could be placed on different pages. In such a case, the inferred address information would no longer be meaningful, as the page offset alone cannot reveal their relative distance or adjacency across different pages.

To obtain an empty page, we adopt the Pspray technique [39]. Pspray detects the allocation of a new page by observing a characteristic slowdown when the SLUB allocator requests a page from the buddy allocator. A freshly allocated page is not empty: because the allocator obtained it to satisfy a pending allocation, the requested object is placed on the new page immediately. Note that the presence of unrelated objects on the same memory page as the vulnerable and target objects can degrade exploit reliability. Therefore, we prefill the page by allocating $N-1$ dummy objects, where N denotes the number of slots per page, thereby occupying all available slots. With no free slots remaining, subsequent allocations must obtain a new page from the allocator. Leveraging this behavior, we then allocate the vulnerable and target objects so that the newly assigned page is populated exclusively by them.

4.3.2. Phase 2: Iterative Localization. Second, we perform localization iteratively until a localization object is allocated at the attacker-desired offset. Due to slab freelist randomization, a localization object is not guaranteed to be placed at the desired location on the first attempt; thus, this phase must be repeated. In each iteration, we allocate one localization object and use Prime+Probe to infer its location, repeating the process until it is placed at the desired offset. If the inferred offset does not match the target location, the iteration continues. On the other hand, if the inferred offset matches the desired position, we move to the next phase.

4.3.3. Phase 3: Placement of Vulnerable Object. In the final phase, we ensure that the vulnerable object occupies the attacker-desired slot previously held by a target localization object. This placement relies on the allocator's Last-In-First-Out (LIFO) allocation behavior: by freeing the target localization object and requesting an allocation shortly thereafter, the allocator is likely to reuse the most recently freed slot for the next allocation, thereby placing the vulnerable object at the same offset. Note that, it is possible that another allocation occurs between the deallocation and subsequent allocation, which may lead to placement failure. However, because the time window between these two operations is extremely short, such interleaving is highly unlikely even under heavy background activity.

After placement, we perform a page cleanup to improve exploit reliability. Phase 2's iterative allocations may leave localization objects at offsets other than the intended slot. These residual objects can interfere with exploitation—for example, a localization object adjacent to the vulnerable object may affect out-of-bounds exploit. Therefore, we deallocate all remaining localization objects on the same page, leaving only the vulnerable object in the intended slot.

5. Application of HEAP LOCALIZATION

In this section, we introduce several applications of HEAP LOCALIZATION to avoid exploitation failures described in §3.

5.1. Advanced Out-Of-Bounds Exploit

As previously stated in §3.1, out-of-bounds exploits fail when the vulnerable object is placed in the page's final slot. This is because the intended adjacency cannot be established across the page boundary. Apart from this edge case, the attack is theoretically successful. In other words, the reliability of out-of-bounds depends on avoiding last-slot allocations. Accordingly, it is sufficient to verify, via HEAP LOCALIZATION, whether the vulnerable object occupies the page's final slot prior to exploitation.

Figure 7 illustrates how HEAP LOCALIZATION is applied to avoid the failure case. First, HEAP LOCALIZATION is used to place the vulnerable object at the desired location. Since the only condition we need to check is whether the object occupies the last slot, we simply verify that it is not located there. As described in §4.3.2, Phase 2 normally

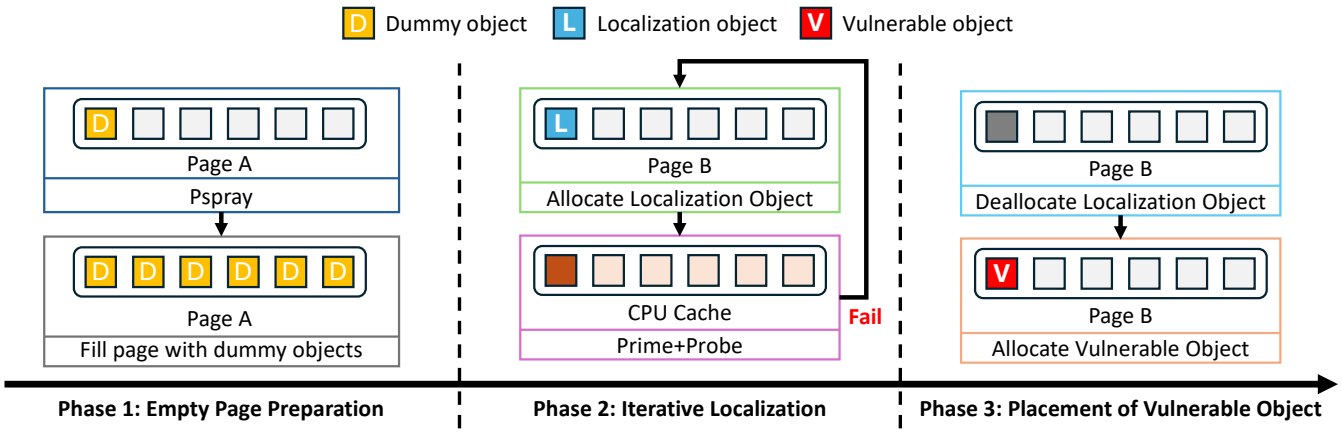

Figure 6: Overview of HEAP LOCALIZATION's three phases for placing the vulnerable object at an attacker-controlled location.

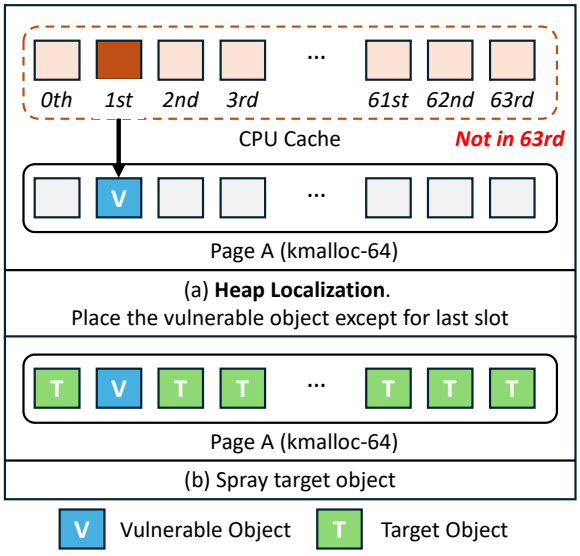

Figure 7: Exploiting Out-Of-Bounds with HEAP LOCALIZATION.

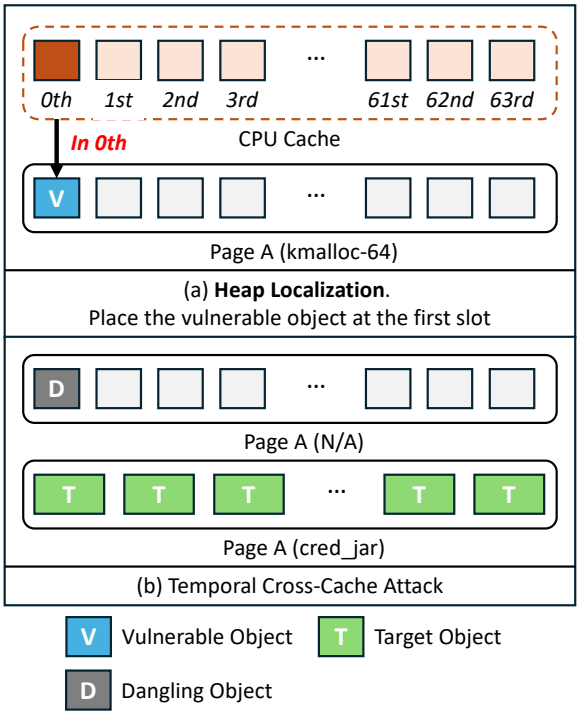

Figure 8: Exploiting Temporal Cross-Cache Attack with HEAP LOCALIZATION.

repeats the allocation when the localization object is not placed in the target cache line. In this case, however, we proceed to the next phase only when the localization object is not placed in the target cache line. Afterwards, we allocate $N-1$ target objects to completely fill the page. At this point, the vulnerable object and the target object are guaranteed to be adjacent, as HEAP LOCALIZATION ensures that the vulnerable object is not allocated in the last slot.

5.2. Advanced Temporal Cross-Cache Attack

The temporal cross-cache attack fails due to misalignment between the vulnerable and target objects, as described in §3.2. Since the vulnerable and target objects differ in size, their allocated slots may not align. Therefore, successful exploitation requires that the two objects be aligned. Concretely, the vulnerable object must be placed at the first slot in a page or at an offset that is a common multiple of the two object sizes. As the target object is sprayed across all slots, it is sufficient to verify only the vulnerable object's placement prior to exploitation.

Figure 8 shows our approach for applying HEAP LOCALIZATION to achieve a successful temporal cross-cache attack. First, we use HEAP LOCALIZATION to place the vulnerable object in one of the common multiple offsets of a page. The reason we target only one offset is that our design verifies a single cache line per iteration, thereby increasing speed. After HEAP LOCALIZATION, we trigger the temporal cross-cache attack by deallocating (to clear cache marking)

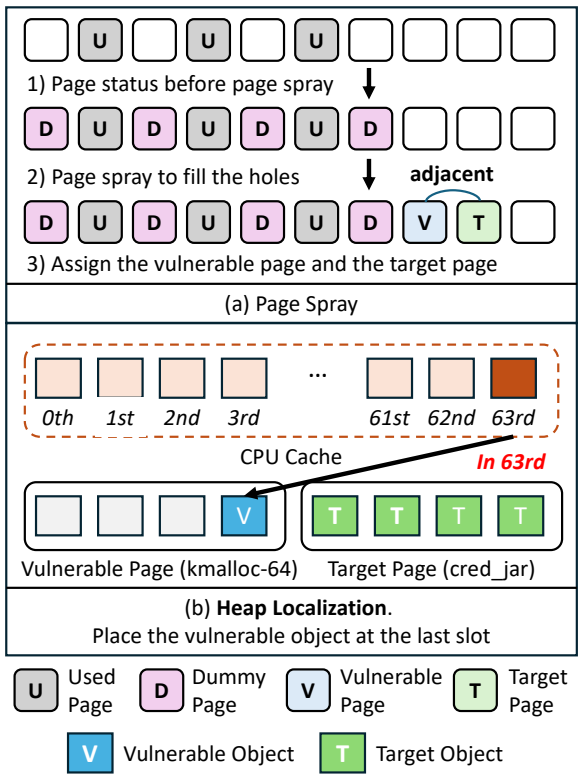

Figure 9: Exploiting Spatial Cross-Cache Attack with HEAP LOCALIZATION.

and then allocating a large number of target objects (to set new cache markings). As a result, HEAP LOCALIZATION ensures that the vulnerable object and the target object are perfectly aligned.

5.3. Practical Spatial Cross-Cache Attack

As discussed in §3.3, the spatial cross-cache attack is desirable but requires a strict placement condition. To corrupt a target object on a different page, the vulnerable object must reside in the page's last slot. Thus, meeting the attack prerequisite only requires verifying that the vulnerable object occupies the last slot. Note that we place two pages adjacent using the *page spray* technique. In short, page spray allocates the intermittently empty gaps so that available space becomes contiguous for sequential allocations.

Figure 9 illustrates the spatial cross-cache attack procedure using HEAP LOCALIZATION. To allocate two pages adjacently, we insert a page spray phase to fill intermittently empty pages after §4.3.1. Then, if the vulnerable and target pages are allocated consecutively, they become adjacent. Thereafter, we continue the remaining HEAP LOCALIZATION phases to place the vulnerable object in the page's final slot. As a result, an out-of-bounds write can corrupt the target object on the adjacent page.

Challenge and Solution. HEAP LOCALIZATION faces a specific challenge under certain conditions of spatial cross-cache attacks. In short, if the kernel cache uses higher-order page (i.e., order-1 or above), HEAP LOCALIZATION may produce false positives. In our environment, the CPU uses a 4 KB cache-line mapping granularity, so order-0 pages allow accurate localization without false positives. However, for order-1 (8 KB) pages and above, multiple objects can share the same cache-line index (e.g., accesses at offset 0 and offset 4096 may map to the same cache-line), which can induce false positives. Importantly, this ambiguity does not affect other applications: out-of-bounds exploitation can avoid the last slot, and temporal cross-cache attacks remain aligned when the object resides on the 0th cache-line.

To address this challenge, we introduce an attack extension that disambiguates candidates by carefully shaping the object layout. First, we use HEAP LOCALIZATION iteratively to place 2^N vulnerable objects at locations likely to occupy the last slot of an order-N page. Here, we illustrate the case of the order-1 page as an example. Second, we spray benign *bumper* objects (non-functional buffers) into the same page to control adjacency relationships. This arrangement aims to place one vulnerable object adjacent to a bumper while placing another vulnerable object adjacent to the intended target on the neighboring page. Finally, we trigger the vulnerability on both vulnerable objects; this corrupts the bumper object (which is safe to corrupt) and the real target object, respectively. Note that repeatedly triggering the vulnerability increases the likelihood of a kernel panic; therefore, our extension limits triggers to the minimum necessary to reduce this side effect. Consequently, the method remains practical for spatial cross-cache attacks on order-1 and higher-order pages.

6. Evaluation

In the following, we evaluate the effectiveness of HEAP LOCALIZATION. To this end, we conduct experiments to answer the following research questions:

- **RQ1.** How effective is HEAP LOCALIZATION at placing a vulnerable object at a desired offset (placement efficacy)?
- **RQ2.** What is the applicability of HEAP LOCALIZATION to real-world kernel vulnerabilities (end-to-end exploitability)?

Experimental Setup. The experiments were conducted on a desktop running Ubuntu 22.04.5 LTS, equipped with an Intel Core i7-14700K processor (28 cores) and 32 GB of RAM. We compiled Linux v6.16.9—the latest release at the time of writing—with all default kernel mitigations enabled, except for the kernel cache-related mitigations.

6.1. Reliability of HEAP LOCALIZATION

To accurately measure the reliability of HEAP LOCALIZATION, we performed experiments using the synthetic vulnerable object. The reason why we use the synthetic vulnerable object is that real-world vulnerable objects do not expose their precise locations, and obtaining them requires

TABLE 1: Reliability of HEAP LOCALIZATION using synthetic vulnerable object. `Prev` denotes the previous approaches for each exploit technique and `HL` denotes HEAP LOCALIZATION.

Vulnerable Cache		Out-Of-Bounds				Temporal Cross-Cache				Spatial Cross-Cache			
		Theory		Real		Theory		Real		Theory		Real	
Name	# of objs	Prev	HL	Idle	Busy	Prev	HL	Idle	Busy	Prev	HL	Idle	Busy
kmalloc-64	64	98.4%	100.0%	100.0%	99.1%	34.4%	100.0%	98.9%	96.0%	1.6%	100.0%	98.9%	91.1%
kmalloc-96	42	97.6%	100.0%	100.0%	98.4%	50.0%	100.0%	98.3%	95.1%	2.4%	100.0%	98.2%	90.9%
kmalloc-128	32	96.9%	100.0%	100.0%	98.2%	34.4%	100.0%	98.8%	95.2%	3.1%	100.0%	98.6%	92.9%
kmalloc-192	21	95.2%	100.0%	99.8%	98.5%	100% (Same size with target cache)				4.8%	100.0%	97.9%	91.5%
kmalloc-256	16	93.8%	100.0%	99.9%	98.3%	31.3%	100.0%	98.2%	95.9%	6.3%	100.0%	99.1%	92.2%
kmalloc-512	8	87.5%	100.0%	99.8%	97.5%	25.0%	100.0%	98.1%	93.6%	12.5%	100.0%	99.0%	90.5%
kmalloc-1024	8	87.5%	100.0%	99.6%	97.0%	50.0%	100.0%	99.2%	97.1%	12.5%	50.0%	51.9% (96.8%)	42.3% (88.2%)
kmalloc-2048	8	87.5%	100.0%	99.3%	96.9%	25.0%	100.0%	99.0%	97.8%	12.5%	25.0%	24.5% (95.5%)	18.4% (81.9%)

intrusive methods such as kernel debug messages, which hinder controlled experimentation. Therefore, we implemented three dedicated system calls that allocate and deallocate the synthetic object and transfer the location information to userspace.

Experimental Design. Using HEAP LOCALIZATION, we evaluate whether the synthetic vulnerable object is placed at the intended location under two different system conditions: idle and busy (with `stress-ng`). We evaluated HEAP LOCALIZATION across multiple kmalloc size classes but excluded `kmalloc-4096` and `kmalloc-8192` because the size of objects is equal to or greater than the size of the CPU cache. That is, every object in those kernel caches accesses the same cache line (i.e., the 0th cache line). However, given that only one of the 31 kernel CTF [27] vulnerabilities disclosed in the past two years involved these caches, omitting them does not materially affect the practical relevance of our results. Finally, each experiment consists of ten rounds, and each round attempts HEAP LOCALIZATION 100 times. Note that, we reboot the system after every round to refresh the system.

6.1.1. Out-of-Bounds Results. As shown in Table 1, HEAP LOCALIZATION achieves a very high success rate in out-of-bounds exploitation. After HEAP LOCALIZATION, we verify whether the synthetic object was not placed in the last slot, as the limitation of prior techniques arises when the vulnerable object occupies this position.

Theory. If we theoretically estimate the success rate of prior techniques, failure occurs only when the vulnerable object is placed in the last slot; thus, their success rate is $1 - 1/N$ where N is the number of objects in the page. Accordingly, `kmalloc-32` with 128 objects per page achieves an almost perfect theoretical success rate, while larger caches (e.g., `kmalloc-1024`) with fewer objects show only about 87.5%. In contrast, HEAP LOCALIZATION achieves a theoretical success rate of 100%.

Result. In both environments, HEAP LOCALIZATION demonstrated high success rates. Under the idle condition, our technique achieved 100% or near-perfect success rates across all caches, which aligns with our expectation. On the other hand, under the busy condition, the success rate slightly decreased (averaging 98%) due to increased noise, yet remained consistently high overall. We attribute this robustness to the nature of the out-of-bounds exploitation: even if the synthetic object fails to occupy the exact intended position because of noise, it is still more likely not to be placed in the last slot, thus leading to a successful outcome. However, caches with fewer objects showed a marginally lower success rate, indicating that a smaller number of allocation slots amplifies the impact of noise. In conclusion, our results confirm that HEAP LOCALIZATION provides a highly reliable and effective approach for out-of-bounds exploitation.

6.1.2. Temporal Cross-Cache Attack Results. HEAP LOCALIZATION achieved high success rates across all caches. The target cache used in our experiment was `cred_jar`, whose object size is 192 bytes. Since the common multiple between `cred_jar` and each kmalloc cache differs, we applied different target offsets for HEAP LOCALIZATION in each cache. For measuring the success rate, we verify whether the synthetic object is placed at offset 0 or one of the common multiple offsets. Note that we exclude `kmalloc-192` because it has the same object size as the target cache, so that all slots are common multiples, which means there is no misalignment (no failure case).

Theory. Since no existing technique can bypass misalignment, we calculated the probability of misalignment when a vulnerable object is randomly placed within a page. Accordingly, the theoretical success rate can be formulated as $(1 + C)/N$, where C represents the number of common multiples and N denotes the total number of objects within the page. Thus, caches with more common multiples or fewer objects tend to have higher theoretical success rates. In contrast, HEAP LOCALIZATION achieves a theoretical success rate of 100%, as it enforces object placement at the aligned position.

Result. HEAP LOCALIZATION achieved practical success rates under both environments. Under the idle condition, all caches showed results closely matching their theoretical success rates. In contrast, under the busy condition, the success rate slightly decreased (averaging 95.8%). In our analysis, the noise interferes with the replacement between

the localization object and the synthetic vulnerable object. Nevertheless, the drop was not significant, as multiple slot positions can still lead to a successful exploit. In conclusion, these results demonstrate that HEAP LOCALIZATION effectively overcomes the misalignment limitation of temporal cross-cache attacks.

6.1.3. Spatial Cross-Cache Attack Results. HEAP LOCALIZATION achieved high success rates across most caches. To determine the success, we verify whether the synthetic vulnerable object is placed at the last slot, since spatial cross-cache attacks can only be performed under this condition.

Theory. As discussed in §3.3, spatial cross-cache attacks have been considered impractical mainly because no existing technique can reliably place an object at the last slot of a page. To evaluate this theoretically, we assume the object is allocated in a random position in page, which results in 7% on average. On the other hand, the theoretical success rate of HEAP LOCALIZATION depends on the page order used by the corresponding cache. For caches using order-0 pages, where the page size is 4096 bytes, the cache size is perfectly aligned, resulting in a 100% theoretical success rate. In contrast, the page sizes of the other caches exceed 4096 bytes, causing multiple objects in the same page to occupy the last cache lines.

Result. Regarding kernel caches using order-0 page, HEAP LOCALIZATION achieved practical success rates in both idle and busy environments. Under the idle condition, all caches performed as expected, showing little deviation from theoretical results due to minimal noise. Under the busy condition, HEAP LOCALIZATION was more affected by noise compared to other techniques. This is because other techniques can succeed even if noise alters the placement, as multiple slot positions may still lead to success, whereas spatial cross-cache attacks have only a single valid slot, resulting in failure when disturbed. In conclusion, for caches using order-0 pages, HEAP LOCALIZATION enables spatial cross-cache attacks that were previously considered purely theoretical to be realized in real-world settings. However, HEAP LOCALIZATION alone is not effective at placing a synthetic object reliably in a single attempt on order-N pages.

To validate whether the technique presented in §5.3 effectively addresses this issue, we conducted additional experiments. To this end, we measured success rates in exploit scenarios. For each experiment, we used HEAP LOCALIZATION to place 2^N candidate synthetic objects and required that each candidate be adjacent to either a bumper or the target; otherwise, the trial was considered a failure. Under idle conditions, the procedure achieved high success rates across the caches. In the busy state, however, success rates dropped notably due to noise. The effect is more pronounced as the number of candidates increases (for example, two candidates in `kmalloc-1024` and four candidates in `kmalloc-2048`), because the probability that noise disrupts at least one required placement increases. Nevertheless, we believe that the technique yields a success rate exceeding 80%, making it usable for real-world exploitation.

6.2. Real-World Vulnerabilities

To assess the real-world applicability of our approach, we evaluated it on four publicly disclosed CVEs and developed end-to-end privilege-escalation exploits based on HEAP LOCALIZATION. Among them, we use three representative cases, namely `CVE-2022-34918`, `CVE-2024-53141`, and `CVE-2024-26581`, to illustrate how HEAP LOCALIZATION applies to different exploitation scenarios. To this end, we backported all tested vulnerabilities to our evaluation environment (i.e., v6.16.9).

6.2.1. Description of CVEs. We introduce the details of four CVEs to understand their exploitability more easily.

CVE-2024-53141. CVE-2024-53141 [18] is a heap out-of-bounds write vulnerability in `struct bitmap_ip`. This vulnerability arises from missing boundary checks when processing crafted network configuration data, resulting in repeated 8- or 16-byte heap overwrites. Since `bitmap_ip` is a dynamically sized object, it can be allocated in `kmalloc-96` or larger caches depending on its size.

CVE-2022-34918. CVE-2022-34918 [16] is a heap out-of-bounds write vulnerability in `struct nft_set_ext`. This vulnerability arises from a missing length validation when the element data type is `NFT_DATA_VERDICT`. Under this condition, the vulnerable object is allocated with a fixed-size extension area (i.e., 16 bytes), but the subsequent copy uses `set->dlen`, resulting in a heap overwrite of up to 48 bytes. Since the vulnerable object is a dynamically sized object, it can be allocated in `kmalloc-64` or larger caches depending on its key length and extensions.

CVE-2024-26581. CVE-2024-26581 [17] is a use-after-free (or double-free) vulnerability in `struct nft_rbtree_elem`. This vulnerability arises from improper validation in the rbtree garbage collection mechanism. Specifically, the code fails to verify whether end-interval elements are active before collecting them, causing an `struct nft_rbtree_elem` object to be freed twice. Since `struct nft_rbtree_elem` is also a dynamically sized object, it can be allocated in `kmalloc-128` or larger caches depending on its size.

CVE-2023-5345. CVE-2023-5345 [49] is a double-free vulnerability in `struct smb3_fs_context.password`. This vulnerability occurred because of a lack of pointer nullification after freeing the vulnerable object. Since the attacker can control the size of the vulnerable object, it can be allocated in `kmalloc-8` to `kmalloc-256`.

6.2.2. Applications. We leverage HEAP LOCALIZATION to construct end-to-end privilege-escalation exploits for four publicly disclosed CVEs, demonstrating its applicability across multiple exploit scenarios. Below, we describe representative cases that illustrate how HEAP LOCALIZATION applies to three different exploit techniques.

Out-Of-Bounds. For the out-of-bounds attack leveraging CVE-2022-34918, we demonstrate how HEAP LOCALIZATION enables reliable exploitation of this vulnerability. This attack requires avoiding last-slot

allocation to ensure that the out-of-bounds write reaches an adjacent object rather than overflowing into unused page padding. We first use HEAP LOCALIZATION to place the vulnerable set element in any slot except the last slot of the page within `kmalloc-64`. Next, we spray `user_key_payload` objects in the same cache to fill the remaining slots, ensuring controlled adjacency. When the overflow is triggered, it corrupts the `len` field of the adjacent `user_key_payload`, enabling an out-of-bounds read primitive via `keyctl` that leaks kernel heap and KASLR base addresses. We then spray `simple_xattr` objects and trigger a second overflow to corrupt the linked list pointers of an adjacent `simple_xattr`. Finally, we leverage the corrupted list pointers to obtain an arbitrary address write primitive, which we use to overwrite `modprobe_path` [55] with an attacker-controlled path, achieving privilege escalation.

Temporal Cross-Cache Attack. For the temporal cross-cache attack leveraging CVE-2024-26581, we show how HEAP LOCALIZATION facilitates reliable exploitation of the double-free condition. We first use HEAP LOCALIZATION to position the vulnerable object at a strategic offset within the `kmalloc-128` cache, aligning it with the boundary of a potential `cred` structure. Since `cred` objects are 192 bytes in size, we place the vulnerable object at offsets such as `0xX000` or their common multiples to facilitate alignment. Next, we use the vulnerability to create a controllable double-free condition by manipulating the `msg_msg` object, which allows us to obtain a reliable use-after-free primitive suitable for the temporal cross-cache scenario. This step is necessary because the original vulnerability frees the object twice within a single control flow, making it unsuitable for direct exploitation. We then deallocate several objects in the `kmalloc-128` cache and perform `cred` spraying to allocate numerous unprivileged `cred` objects for temporal cross-cache exploitation [62]. As a result, one of the sprayed `cred` objects becomes aligned with the previously freed vulnerable object. Finally, we trigger the double-free vulnerability again, which forcibly releases the aligned unprivileged `cred` object. We then apply the credential-swapping technique [41] to escalate privileges.

Spatial Cross-Cache Attack. For the spatial cross-cache attack leveraging CVE-2024-53141, we demonstrate how HEAP LOCALIZATION enables precise cross-page corruption using the same vulnerability. This attack requires placing the vulnerable object in the last slot of a page to achieve cross-page corruption. We first employ a page spray technique to construct contiguous memory regions, filling fragmented gaps to guarantee sequential page allocation. Using HEAP LOCALIZATION, we then place the vulnerable object at the last slot of a page. Simultaneously, we spray unprivileged `cred` objects so that one of them is allocated on the page immediately following the vulnerable page, making the vulnerable object adjacent to a `cred` object across the page boundary. When the vulnerability is triggered, it overwrites members of `struct cred` (e.g., `uid`, `euid`) with zeros. Since these fields represent privilege identifiers, zeroing them directly elevates the current process to root privileges.

7. Discussion

In this section, we discuss the technical details for reliability and outline potential directions for future work.

7.1. Technical Details

HEAP LOCALIZATION involves several practical considerations related to reliability and portability.

7.1.1. Noise. Even if we use HEAP LOCALIZATION, the noise stemming from the Linux kernel scheduler can hinder the exploitation. This is why the actual success rate is slightly lower than our expectation. The scheduler can introduce noise through two different mechanisms: i) CPU migration and ii) context-switch.

CPU Migration. CPU migration is a critical factor for HEAP LOCALIZATION, as it affects two essential components of our design. First, our cache side-channel relies on the L1 cache; if a migration occurs during the attack, the process moves to another core with a different L1 cache, invalidating previously primed cache states and resulting in inaccurate measurements. Second, the SLUB allocator maintains a separate freelist for each CPU core. For instance, if the CPU migration happens between phase 2 (iterative prime+probe) and phase 3 (vulnerable object allocation), the allocation of the vulnerable object uses the freelist from another CPU, leading to unpredictable placement.

To prevent such scenarios, we pin the process to a specific CPU core using the `sched_setaffinity()` function. This system call restricts the execution of a process to a designated CPU, ensuring that all localization, measurement, and exploitation steps occur on the same core. As a result, both cache-side measurements and heap allocations remain consistent throughout the entire attack phase.

Context-Switch. Context switches are a major obstacle for timing-based attacks. In cache side-channel measurements the timing gap between a hit and a miss is often small, so precise measurements are essential. If a context switch occurs during measurement, the timing becomes unreliable and can produce false positives. Acting on such false positives—i.e., attempting an exploit when the object actually resides in a different slot—may cause the exploit to fail and could even trigger a kernel panic.

To mitigate this, we apply two related countermeasures. First, we reduce the frequency of context switches during probing by calling `sched_yield()` at the end of each loop iteration. Under the Linux CFS scheduler, this lowers the likelihood that the attacker will be preempted during the critical probing window, thereby increasing the reliability of the cache side-channel measurements. Second, we increase measurement robustness by repeating the Prime+Probe on each cache line multiple times (e.g., 100 repetitions) and declaring the result (i.e., which cache line is used) only when the observed counts exceed a predefined threshold. Therefore, even if a context switch occurs during one repetition, the likelihood of repeated short-lived context switches across many

consecutive trials is low, so this repetition-and-threshold scheme substantially improves accuracy.

7.1.2. Interference. HEAP LOCALIZATION employs a specialized strategy to handle the issue arising from the gap between theory and implementation.

Problem. For HEAP LOCALIZATION, minimizing the number of additional memory accesses from the localization object is crucial to reducing noise in cache-side observations. In our experiments, we selected the `msg_msg` object as the localization object because it generates the fewest additional accesses among several candidates. However, the function `find_msg()`, which accesses `msg_msg`, still performs extra memory operations. As a result, during the *Prime+Probe* phase, it becomes difficult to distinguish whether a cache-line access originates from the localization object itself or from these incidental memory accesses.

Solution. We address this problem with two approaches, both requiring an additional verification step before starting phase 2. The first approach runs `find_msg()` twice with identical arguments except for the target `msg_msg` object. Since the same arguments cause identical additional memory accesses, we can infer that if the same cache lines appear in both runs, those lines correspond to incidental accesses rather than the localization object. The second approach leverages the fact that `find_msg()` can execute even when no `msg_msg` objects exist. By running `find_msg()` in this empty state, we can identify which cache lines are accessed solely due to additional memory operations. In our experiments, we adopted the second approach.

7.1.3. Portability. HEAP LOCALIZATION is applicable to any environment in which a cache side channel can be mounted, although some parameters must be adjusted for each target environment. In particular, porting HEAP LOCALIZATION requires calibrating several microarchitecture-dependent parameters: (i) the cache miss threshold, (ii) cache geometry, and (iii) the measurement criteria used to distinguish cache hits from misses. First, the cache miss threshold depends on the timing characteristics of the underlying CPU, since cache-hit and cache-miss latencies vary across microarchitectures. Second, cache geometry depends on cache organization (e.g., indexing and associativity), which is determined by the hardware design and varies across CPUs. Finally, the measurement criteria depend on system-level noise and microarchitectural behavior, both of which affect the stability of cache observations. We adjusted these parameters for a different CPU platform (E cores in Intel i5-13500T and Ultra7-265K) and confirmed that HEAP LOCALIZATION operates correctly after such calibration. We measured an L1 cache miss threshold of 40 cycles on the i5-13500T, and classified cache lines with a miss rate exceeding 70% as conflicted. For the Ultra7-265K, due to aggressive DVFS behavior, we fixed the clock frequency to 800 MHz. Under this configuration, we set the L1 cache miss threshold to 318 cycles and considered cache lines with a miss rate above 90% as conflicted.

7.2. Future Work

7.2.1. Mitigating Hardware Prefetcher Interference. Although we apply countermeasures against hardware prefetchers (see §A.3), adaptive prefetchers can still learn repeated access patterns and interfere with HEAP LOCALIZATION.

Hardware Prefetcher. Modern CPUs employ prefetching mechanisms that detect repetitive memory-access patterns and proactively load data into the cache. While this behavior improves general performance, it undermines cache-based side-channel measurements by changing cache state in ways that are not caused by the monitored program. Since HEAP LOCALIZATION relies on repeated, localized accesses for accurate object localization, aggressive prefetching can corrupt the measurement signal and lead to incorrect or missed observations.

Impact. Empirically, we observe that prefetchers tend to learn access patterns that arise during HEAP LOCALIZATION's second phase when allocated objects exhibit regular strides. Once the prefetcher begins issuing speculative loads for these patterns, HEAP LOCALIZATION experiences false negatives: the expected probe hits either disappear or become noisy, reducing phase-2 success rates. This behavior typically appears after roughly ten phase-2 iterations when the allocation pattern becomes learnable by the prefetcher. Consequently, such false negatives primarily affect out-of-bounds exploitation, as HEAP LOCALIZATION may identify a location that is not in the last slot of the page. However, since most cases complete within two iterations, this effect does not materially impact HEAP LOCALIZATION's overall success rate.

Our Approach. To mitigate the prefetcher interference observed in phase 2, we adopt a pragmatic strategy. We first bound the number of phase-2 iterations based on the maximum number of objects that can be allocated in a page, thereby preventing unbounded repetition when the prefetcher suppresses observable cache activity. If detection still fails within this bound, we treat the trial as a prefetcher-related failure, restart the probing process to reset the prefetcher state, and re-run HEAP LOCALIZATION from the beginning. This restart-based approach effectively suppresses occasional prefetcher learning, albeit at the cost of a longer runtime. Since our primary goal is to maximize exploit reliability, we consider runtime optimization to be out of scope for this work.

Future Directions. Our current countermeasure focuses solely on the attacker's access pattern and does not prevent the prefetcher from learning victim-driven memory activity. Future work should explore extending this countermeasure to also suppress prefetcher learning of victim memory access patterns, enabling HEAP LOCALIZATION to remain accurate even under more complex interference scenarios.

7.2.2. Other Possible Applications. We believe HEAP LOCALIZATION can be applied in various areas.

Target Object. Most existing exploits assume the attacker can *spray* the target object, because spraying increases the chance that an attacker-controlled object is adjacent to or aliases the vulnerable object. For example, if a page contains $n = 10$ objects, the probability that a *target* object is placed immediately after the *vulnerable* object is $\frac{n-1}{n(n-1)} = 10\%$. On the other hand, if the target object can be sprayed, they succeed in every case except when the vulnerable object occupies the page's last slot; with $n = 10$ this yields 90%. Consequently, even if the object has strong primitives, it can be impractical when spraying is not possible. Our technique additionally controls the target object's placement, expanding the set of usable targets and enabling re-evaluation of vulnerabilities previously considered unexploitable due to poor target availability.

Other OSes. HEAP LOCALIZATION can also be applied to other operating systems such as macOS and FreeBSD. Many OSes manage the kernel heap in a page- and cache/zone-oriented manner similar to Linux, so the location-based limitations faced by previous exploit techniques are shared. For example, on macOS and FreeBSD, an attack can fail when the vulnerable object is placed in the last slot of a page, and cross-zone attacks that leverage zones analogous to Linux kernel caches suffer the same issues (i.e., misalignment or the vulnerable object not being adjacent to the target object on a different page). Therefore, implementing our technique on these OSes would remove failures caused by object placement and substantially improve exploit stability and reliability.

8. Related Work

In the following, we review related work on cache side-channel attacks and prior work on kernel exploitation techniques.

8.1. Cache Side-Channel

Methodology. Various methodologies [21, 29, 63] have been proposed to measure cache states—the core principle underlying cache side-channel attacks. Flush+Reload [63] leverages shared memory and explicit cache-line flushes to achieve high-resolution, low-noise timing measurements by observing whether a reloaded line remains cached. Flush+Flush [29] determines cache hits and misses by measuring the execution time of the `clflush` instruction itself, enabling very fast and stealthy measurements. Prime+Abort [21] exploits Intel TSX transaction abort timing—using abort latency as a measurement primitive—to achieve high-resolution cache observations without relying on external timers. These techniques produce the same observable outcome—the cache state—as the Prime+Probe [47] variant employed in our work and are therefore compatible with our approach.

Bypass Techniques. Techniques that circumvent mitigations for cache side channels have been actively studied [48, 52, 59]. PAPP [59] generates prefetcher-aware access patterns that avoid unwanted cache fills and preserve Prime+Probe accuracy under aggressive prefetching. FetchBench [52] provides empirical measurements of prefetcher behavior, enabling access sequences that minimize prefetch-induced interference. Prime+Scope [48] increases temporal resolution to suppress timing distortion from OS scheduling and prefetch activity. ShadowLoad [31] manipulates hardware prefetcher state to induce victim data to be loaded into the cache in advance, thereby amplifying cache-side signals. These works are orthogonal to our approach and can be adopted to further strengthen the cache-side measurements used by HEAP LOCALIZATION.

Applications. Cache side channels have been applied to several distinct goals. First, they can be used to recover cryptographic secrets [43, 57, 63, 64] by monitoring secret-dependent cache activity in crypto implementations. Second, they have been exploited to break isolation across co-located virtual machines [35, 50] by extracting sensitive data from neighboring VMs. These two application classes are orthogonal to our main objective of kernel-heap object localization and therefore not useful for HEAP LOCALIZATION. Third, cache channels have been used to defeat kernel address-space randomization by leaking address-dependent access patterns from privileged memory regions [28, 36–38, 44]. However, such KASLR-breaking techniques generally rely on narrowing down candidate addresses within predictable regions (e.g., kernel text or mapped modules). They are not suitable for heap regions whose base addresses and allocation offsets are not predictable, and thus cannot be directly applied to our object-localization problem.

8.2. Prior Heap Layout Manipulation Techniques

Exploiting heap vulnerabilities typically requires the vulnerable and target objects to be aliased or placed adjacent, so various techniques [8, 20, 54, 60, 65] have been developed to increase the probability of such placements. Heap spray [20] fills intermittent free slots by mass-allocating objects, thereby making it more likely that subsequent allocations occupy contiguous memory. Heap feng shui [54] extends this idea by combining spraying with selective frees and re-allocations to position a target object between controlled allocations, thereby increasing the probability of adjacency. However, these techniques implicitly assume allocator behavior and can therefore fail in real-world environments. Slake [8] achieves precise placement by using kernel debugging to determine exact object locations and then arranging allocations accordingly. Because kernel debugging requires root privileges, Slake is not applicable in typical unprivileged exploit scenarios. More broadly, prior heap manipulation techniques [8, 20, 39, 54, 60, 65] are limited by the non-deterministic placement of heap objects. As a result, they often impose restrictive constraints on target-object selection and require additional failure-tolerant logic to make exploitation reliable. Our work addresses the shortcomings by enabling high-probability placement of objects from an unprivileged context, allowing reliable layout control without privileged kernel introspection. In the following, we discuss the inevitable limitations of existing techniques in regards to

restrictive target-object choices and additional failure-tolerant handling.

Target Object Constraints. In prior techniques, selecting a target object requires satisfying several conditions to avoid exploitation failure. For example, to avoid misalignment in temporal cross-cache attacks, the target object is chosen such that its layout aligns with the vulnerable object. Moreover, even if misalignment occurs and unintended fields are overwritten, the target object must not immediately cause a failure (e.g., a kernel panic). In out-of-bounds settings, the target object must also be sprayable, as multiple instances must be allocated on the same page to increase the likelihood of adjacency. HEAP LOCALIZATION relaxes these constraints by avoiding misalignment and enabling precise placement of the target object without requiring spraying.

Failure-Tolerant Handling. Prior techniques must also tolerate failed placement attempts. When placement fails and the exploit cannot proceed to the next step, the corresponding step must be retried. To do so, the exploit must safely clean up the intermediate state created for that step, then reattempt the required heap manipulation. This requires careful analysis of which objects can be reclaimed without causing unintended side effects and how the exploit state can be restored for another attempt. As a result, prior techniques often require exploit-specific retry logic and recovery procedures. HEAP LOCALIZATION reduces this burden by making object placement more reliable and thereby reducing the need for such failure-tolerant handling.

8.3. Prior Kernel Exploitation Techniques

Beyond heap object layout manipulation techniques, there are also several other techniques related to our approach that we discuss next.

Exploitability Upgrade. When a kernel vulnerability exhibits only weak exploitability, numerous techniques [46, 61, 66] have been developed to upgrade its exploitability and turn it into a practical exploit. For example, BridgeRouter [61] leverages a bridge object to induce an out-of-bounds condition with attacker-controlled content and length, thereby increasing exploitability. Nguyen [46] forces a target object into a use-after-free state using an *arbitrary free* primitive, producing a highly exploitable condition. Hao et al. [66] show how abusing metadata fields can trigger additional vulnerabilities that further improve exploitability. These techniques are orthogonal to our approach, since they are employed only after HEAP LOCALIZATION has established the desired layout.

Target for Privilege Escalation. Researchers have studied various targets [3, 9, 33, 42, 45] that can lead to privilege escalation when exploited in the kernel. Elastic object [9] was proposed as a target to break KASLR—an essential step for ROP- and DOP-style kernel exploits. DirtyPageTable [3] targets page tables and presents methods to leverage page-table corruption for privilege escalation. SlubStick [45] uses a temporal cross-cache attack to corrupt the page upper directory (PUD) and accomplish privilege escalation. PageJack [33] targets objects that contain physical addresses, triggering page-level use-after-free conditions to mount exploits. These works are orthogonal to our technique since the proposed target objects can be used after HEAP LOCALIZATION positions the vulnerable object at the desired location.

9. Conclusion

Heap vulnerabilities in the kernel often require a specific object layout to be exploitable, even though object placement in the heap is inherently nondeterministic. To this end, existing kernel exploitation techniques rely on heuristics and kernel allocation behavior to guess layouts, which introduces unavoidable failure cases. In this paper, we presented HEAP LOCALIZATION, a cache side-channel technique that performs precise object localization. HEAP LOCALIZATION introduces a new primitive for object-level heap layout inference, enabling deterministic placement that was previously impossible. Using HEAP LOCALIZATION, we develop two widely used exploitation techniques by bypassing all failure cases and render a previously idealized technique practical. We validated the accuracy of HEAP LOCALIZATION with synthetic objects and demonstrated its applicability through end-to-end exploits against real-world vulnerabilities. A comprehensive mitigation would require addressing cache side channels at a more fundamental level. However, eliminating such side channels on commodity systems is widely regarded as an open research problem, likely requiring hardware-software co-design. Existing defenses in this space often involve substantial performance, compatibility, and deployment trade-offs, and it remains unclear to what extent these limitations can be overcome in practice in the future.

Ethical Considerations

The techniques described in this paper are intended exclusively for improving the security of the Linux kernel and are not designed to facilitate or encourage any form of malicious exploitation. All vulnerabilities used in our evaluation had already been reported and fully patched before our analysis. In addition, we performed all experiments in strictly isolated virtualized environments under our control to ensure that no real or production systems were impacted. In addition, potentially sensitive implementation details that could endanger the Linux kernel or its users are deliberately omitted. Note that our technique does not constitute a new memory-corruption vulnerability in the Linux kernel. Instead, it is a structural primitive that enables an unprivileged local attacker to infer the intra-page offset (i.e., the lower 12 bits) of kernel heap objects with high confidence, thereby increasing the reliability of existing heap exploitation techniques. Consistent with coordinated disclosure practices, we shared detailed technical information with the Linux security team and Intel PSIRT. Our discussion with the Linux kernel maintainers indicated that this method is not considered a distinct security bug in itself, such as a memory-corruption vulnerability. The disclosure process helped clarify

possible mitigation directions, e.g., partitioning kernel heap objects into smaller buckets may make such attacks more difficult. Lastly, we release our proof-of-concept code at https://github.com/MPI-SysSec/Heap-Localization to support reproducibility and responsible disclosure within the research community.

LLM Usage Considerations

LLMs were used solely for editorial assistance in preparing this manuscript, such as improving phrasing, grammar, and clarity of exposition. No technical ideas, system designs, experimental methodologies, or analysis results were generated by LLMs. All scientific content—including concepts, techniques, implementations, and evaluations—was independently developed and validated by the authors. All LLM outputs were manually reviewed to ensure accuracy and originality.

Acknowledgement

We thank all reviewers for their thoughtful feedback. This work was supported by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2092 CASA – 390781972). Moreover, this work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone).

References

[1] The core of apple is ppl: Breaking the xnu kernel's kernel. https://googleprojectzero.blogspot.com/2020/07/the-core-of-apple-is-ppl-breaking-xnu.html.

[2] The ups and downs of 0-days: A year in review of 0-days exploited in-the-wild in 2022, 2023. https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html.

[3] Dirty pagetable: A novel exploitation technique to rule linux kernel, 2024. https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html.

[4] S. Baghdasaryan. Memory allocation profiling, 2023. https://lwn.net/Articles/948695/.

[5] D. J. Bernstein. Cache-timing attacks on aes. 2005.

[6] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, 2014. IEEE.

[7] E. Bosman and H. Bos. Framing signals: A return to portable shellcode. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, 2014. IEEE.

[8] Y. Chen and X. Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

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

[10] K. Cook. Kernel address space layout randomization. *Linux Security Summit*, 2013.

[11] K. Cook. [patch v5 3/7] init_on_alloc: Unpessimize default-on builds, 2021. https://lore.kernel.org/kernel-hardening/CAAeHK+xog8-DP1o=1qqKgSP7Hii2Yjah6oyowNE3zSNVW5pRSw@mail.gmail.com/.

[12] K. Cook. slab: Introduce dedicated bucket allocator, 2024. https://lwn.net/Articles/976460/.

[13] Corbet. The slub allocator, 2007. https://lwn.net/Articles/229984/.

[14] J. Corbet. Supervisor mode access prevention, 2012. https://lwn.net/Articles/517475/.

[15] J. Corbet. The current state of kernel page-table isolation, 2017. https://lwn.net/Articles/741878/.

[16] CVE. Cve-2022-34918, 2024. https://www.cve.org/CVERecord?id=CVE-2022-34918.

[17] CVE. Cve-2024-26581, 2024. https://www.cve.org/CVERecord?id=CVE-2024-26581.

[18] CVE. Cve-2024-53141, 2024. https://www.cve.org/CVERecord?id=CVE-2024-53141.

[19] T. H. Y. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, 2015. ACM.

[20] 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*, pages 327–336, 2010.

[21] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[22] C. Evans. What is a "good" memory corruption vulnerability?, 2015. https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html.

[23] T. Garnier. mm: Slab freelist randomization, 2016. https://lwn.net/Articles/685047/.

[24] T. Garnier. Add slub free list pointer obfuscation, 2017. https://samsung.github.io/kspp-study/heap-ovfl.html#mitigating-heap-overflows.

[25] glider. [rfc] security: allow using clang's zero initialization for stack variables, 2020. https://lwn.net/Articles/823152/.

[26] R. Gong. Randomized slab caches for kmalloc(), 2023. https://lwn.net/Articles/938246/.

[27] Google. Google kernel ctf. https://github.com/google/security-research/tree/master/kernelctf.

[28] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (ACM CCS)*, 2016.

[29] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+ flush: A fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[30] 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 *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, Philadelphia, PA, 2024.

[31] L. Hetterich, F. Thomas, L. Gerlach, R. Zhang, N. Bernsdorf, E. Ebert, and M. Schwarz. Shadowload: Injecting state into hardware prefetchers. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Rotterdam, Netherlands, Mar. 2019.

[32] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2016.

[33] J. Hu, J. Zhou, Q. Tang, and W. Shen. Pagejack: A powerful exploit technique with page-level uaf, 2024. https://i.blackhat.com/BH-US-24/Presentations/US24-Qian-PageJack-A-Powerful-Exploit-Technique-With-Page-Level-UAF-Thursday.pdf.

[34] Intel. Supervisor mode execution prevention. https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/servers/platforms/intel-pentium-silver-and-intel-celeron-processors-datasheet-volume-1-of-2/005/intel-supervisor-mode-execution-protection-smep/.

[35] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application

to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, 2015.

[36] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security (ACM CCS)*, 2016.

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2019.

[38] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi. TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 309–321. IEEE, 2020.

[39] Y. Lee, J. Kwak, J. Kang, Y. Jeon, and B. Lee. Pspray: Timing side-channel based linux kernel heap exploitation technique. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[40] Y. Lee, H. Kwon, and T. Holz. DirtyFree: Simplified Data-Oriented Programming in the Linux Kernel. In *Proceedings of the 33rd Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2026.

[41] Z. Lin, Y. Wu, and X. Xing. Cautious! a new exploitation method! no pipe but as nasty as dirty pipe, 2022. https://i.blackhat.com/USA-22/Thursday/US-22-Lin-Cautious-A-New-Exploitation-Method.pdf.

[42] Z. Lin, Y. Wu, and X. Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2022.

[43] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P)*, 2015.

[44] W. Liu, J. Ravichandran, and M. Yan. Entrybleed: A Universal KASLR Bypass against KPTI on Linux. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10–18, 2023.

[45] 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 *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, 2024.

[46] Nguyen. Cve-2021-22555: Turning `\x00\x00` into $10000. https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html#vulnerability.

[47] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20. Springer, 2006.

[48] A. Purnal, F. Turan, and I. Verbauwhede. Prime+scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2021.

[49] quanggle97. Cve-2023-5345_lts_mitigation, 2023. https://github.com/google/security-research/tree/master/pocs/linux/kernelctf/CVE-2023-5345_lts_mitigation.

[50] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Nov. 2009.

[51] M. Rizzo. Prevent cross-cache attacks in the slub allocator, 2023. https://lwn.net/Articles/944647/.

[52] T. Schlüter, A. Choudhari, L. Hetterich, L. Trampert, H. Nemati, A. Ibrahim, M. Schwarz, C. Rossow, and N. O. Tippenhauer. Fetchbench: Systematic identification and characterization of proprietary prefetchers. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2023.

[53] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, Nov. 2007.

[54] A. Sotirov. Heap feng shui in Javascript. *Black Hat Europe*, 2007, 2007.

[55] Theori. Reviving the modprobe_path technique: Overcoming search_binary_handler() patch, 2025. https://theori.io/blog/reviving-the-modprobe-path-technique-overcoming-search-binary-handler-patch.

[56] S. Tolvanen. Kcfi support, 2022. https://lwn.net/Articles/893164/.

[57] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[58] P. Wagle, C. Cowan, et al. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–255. Citeseer, 2003.

[59] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy. Papp: Prefetcher-aware prime and probe side-channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[60] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou. MAZE: Towards Automated Heap Feng Shui. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Aug. 2021.

[61] 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 *Proceedings of the 46th IEEE Symposium on Security and Privacy (Oakland)*, May 2025.

[62] 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 the linux kernel. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (ACM CCS)*, 2015.

[63] Y. Yarom and K. Falkner. FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[64] Y. Yarom, D. Genkin, and N. Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[65] 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 *Proceedings of the 31st USENIX Security Symposium (Security)*, Aug. 2022.

[66] H. Zhang, J. Liu, J. Lu, S. Chen, T. Han, B. Zhang, and X. Gong. Reviving discarded vulnerabilities: Exploiting previously unexploitable linux kernel bugs through control metadata fields. Oct. 2025.

Appendix A. Enhancing Cache Side-Channel's Accuracy

We introduce improvements to cache side-channel methods that substantially increase measurement accuracy by reducing noise and minimizing micro-architectural interference.

A.1. Experimental Setup

All experiments were performed on an Intel Core i7-14700K and Ubuntu 22.04. Intel Core i7-14700K CPU has 8 performance (P) cores and 12 efficiency (E) cores, with a total of 28 threads supported. We targeted the L1 cache of E cores, which shows less noise and stable latency while running a cache side-channel attack.

A.2. Measurement Strategy

Rather than probing multiple cache lines simultaneously, our approach focuses on a single target cache line. This reduces the probe window and therefore can minimize the opportunity for noise to interfere with observations. We measured an L1 cache miss threshold of 70 cycles on this

platform, and treated any access above threshold (70 cycles) as a L1 cache miss, which indicates that the primed object was evicted by the kernel activity. For each target line, we perform 100 repetitions and record the cache miss rate. The empirical thresholds observed on our machine are as follows. When the object does not conflict with the probed line, the miss rate remains below 40%, while the miss rate exceeds 80% when the kernel object occupies the target line.

A.3. Challenges

Noise reduction. Probing many lines simultaneously extends the prime and probe time windows and increases exposure to scheduling interrupts, concurrent activity, and other noise sources. By restricting each measurement to a single cache line, the elapsed time per probe is substantially reduced, which in turn reduces the probability that unrelated events will affect the measurement. The exploit strategy was adapted to monitor the target cache line, and exploitation is performed when the cache line's state indicates the exploitation can be executed.

Hardware prefetcher interference. The L1 cache on modern Intel processors is subject to aggressive hardware prefetching. Within a single process, the prefetcher learns and follows recurring memory access patterns. If left unaddressed, this behavior can significantly distort cache side-channel observations and reduce reliability. To limit the prefetcher's ability to learn our probing pattern, we added a randomized memory access pattern between each probing. The randomized accesses are intended to obscure the probe access stream so the prefetcher cannot adapt to a stable, predictable pattern that would otherwise bias hit behavior.

Appendix B. Meta Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper presents a method to improve the reliability of Linux kernel heap exploitation. Specifically, the proposed approach leverages cache side-channel techniques to reliably localize heap objects relevant to exploitation. By enabling precise knowledge of object placement, it reduces failures caused by unpredictable allocations and consequently increases the reliability of exploits targeting heap memory safety vulnerabilities.

B.2. Scientific Contributions

3. Creates a New Tool to Enable Future Science
4. Identifies an Impactful Vulnerability
5. Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1. The paper proposes a novel application of cache side-channel techniques to support Linux kernel exploitation. In particular, the authors introduce a primitive that enables object-level heap layout inference in the Linux kernel, which achieves a high success rate in practice.
2. The proposed primitive can significantly improve the reliability of exploits targeting heap memory safety vulnerabilities. By allowing attackers to localize kernel heap objects more precisely, the work advances the state of the art in developing reliable exploits for modern Linux kernels.

B.4. Noteworthy Concerns

1. The paper does not provide an objective evaluation of how much the proposed technique simplifies exploit development.