LeakyTree: Cross-VM Bit-by-Bit Deduplicated Page Leakage via Linux Kernel Samepage Merging Internals

Ernesto Martínez García
Graz University of Technology
Graz, Austria
ernesto.martinezgarcia@tugraz.at

Lukas Maar
Graz University of Technology
Graz, Austria
Calif
Los Altos Hills, CA, United States
lukas.maar@tugraz.at

Martin Unterguggenberger
Graz University of Technology
Graz, Austria
martin.unterguggenberger@tugraz.at

Mathias Oberhuber
Graz University of Technology
Graz, Austria
mathias.oberhuber@tugraz.at

Stefan Mangard
Graz University of Technology
Graz, Austria
stefan.mangard@tugraz.at

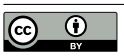
Abstract

Memory page deduplication in modern systems is known to introduce side-channel leakages. An attacker can maliciously craft pages and probe, via CPU cache side channels, whether the malicious pages got deduplicated with others in the system, implicitly leaking the contents of memory the attacker does not have access to. However, these attacks require crafting an exact replica of the victim's page in order to initiate deduplication. Effort-wise, they thus have a complexity of $O(2^n)$, which makes them infeasible for leaking data in high-entropy scenarios, such as when brute-forcing cryptographic keys.

In this paper, we present LeakyTree, a novel side-channel attack on the Linux Kernel Samepage Merging (KSM) memory-deduplication mechanism that exploits cache-observable traversals of KSM's stable tree. LeakyTree reduces the overall leakage complexity from $O(2^n)$ to $O(n)$, enabling efficient bit-by-bit information leakage from a victim page regardless of its entropy. The attack proceeds in three stages: first, it massages internal KSM data structures (i.e., red-black trees) by forcing the deduplication of attacker-controlled pages; second, it exploits tree-ordering properties, combined with a CPU cache side channel, to infer layout information crucial to the attack; third, it uses KSM's page comparison process during tree traversal to leak bit by bit from victim memory pages. We demonstrate LeakyTree's efficacy by leaking secret data cross-process and cross-VM from KSM-deduplicated pages on Linux kernel versions 5.15 to 6.18. In particular, we show that LeakyTree can leak high-entropy pages, such as ECC and RSA private keys as well as API tokens, at rates of up to 30 bits per second. For realistic key sizes, the extraction time is within minutes, which has been infeasible in this scenario with prior deduplication attacks.

CCS Concepts

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



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846575

Keywords

Linux Kernel, Memory Deduplication, KSM

ACM Reference Format:
Ernesto Martínez García, Lukas Maar, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. 2026. LeakyTree: Cross-VM Bit-by-Bit Deduplicated Page Leakage via Linux Kernel Samepage Merging Internals . In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3830454.3846575

1 Introduction

Modern operating systems increasingly employ resource overcommitment mechanisms to improve utilization and efficiency. Techniques such as swapping, filesystem compression, and Copy-on-Write (CoW) provide applications with the abstraction of larger available physical resources. Among these, transparent memory deduplication techniques scan system memory for identical pages across processes and merge them into a single shared physical frame. This page-sharing typically marks the merged pages as read-only, triggering CoW on memory writes. The Linux kernel implements this functionality through the Kernel Samepage Merging (KSM) [2, 23, 55] mechanism since Linux v2.6 [40] through the `madvise` and `prctl` system call. Modern open-source virtualization platforms, such as Proxmox VE, make use of KSM transparently by default.

From the user's perspective, KSM is transparent; that is, applications behave as if they still own private pages, while the system reclaims redundant memory in the background. However, the kernel is a shared resource, and prior research has demonstrated attacks on transparent memory deduplication. The traditional side channel on KSM [35, 61] works as follows: the adversary creates a payload to guess memory content and marks it eligible for memory deduplication. The side channel is based on observing whether a subsequent write causes CoW, as indicated by the slower timing behavior of merged pages. Thereby, an adversary can learn sensitive data from the victim's process memory.

This concept has been extended to cross-VM covert channels [29] and cross-VM fingerprinting attacks [26, 44], because these systems still share the same host kernel. Two works are particularly relevant here. Bosman et al. [5] and, subsequently, Schwarzl et al. [56]

showed that an attacker can leak data byte by byte when having strong control over the alignment and padding of victim data. Bosman et al. [5] demonstrated Address Space Layout Randomization (ASLR) and password leakage in settings where such control is available, e.g., in JavaScript and nginx-based server scenarios. Schwarzl et al. [56] extended this approach to remote leakage of database records and Kernel ASLR (KASLR) [12] within VMs by exploiting a similar control primitive in database systems such as InnoDB. However, because these attacks depend on such a strong alignment and padding primitive on the victim side, their applicability to leaking high-entropy data, such as cryptographic key material, remains limited to such controlled scenarios.

All these prior KSM side channels reveal only whether an identical page already exists in the victim's memory. For high-entropy targets such as cryptographic keys, this severely limits the attacker's capabilities, as the attacker must guess the *exact page* contents to trigger deduplication. Since secret keys consist of random bits, constructing such an exact matching page is infeasible in practice without this strong alignment and padding primitive. Recovering an n-bit secret therefore requires exhaustive search over a space of a time complexity $O(2^n)$, where n is the page size in bits.

In this paper, we present LeakyTree, a novel exploitation technique that enables bit-by-bit leakage from deduplicated pages by leveraging Linux KSM internals. LeakyTree does not require this control over victim-memory alignment or padding. It reduces the leakage complexity from $O(2^n)$ to $O(n)$ for an n-bit target, and allows leakage in cross-process and cross-VM settings. The key idea is to shift from a page equality oracle to a content-dependent greater-or-lesser traversal oracle inside KSM's stable tree, a *red-black tree* [4, 6, 38] that tracks already deduplicated user pages.

At its core, LeakyTree relies on four observations: (1) KSM's stable-tree ordering is keyed by sensitive page contents (i.e., deduplicated user pages), (2) KSM traversal operations induce cache traces that depend on those contents, (3) these traces can be correlated with sensitive content, and (4) by strategically triggering unprivileged KSM operations, an attacker can leak the contents of deduplicated victim pages.

With these observations, LeakyTree proceeds in three stages. First, we shape the layout of KSM's stable tree around the victim page. We do so by forcing deduplication of attacker-controlled pages through user-space operations, which insert corresponding nodes into the tree. These pages approximate lower and upper bounds of plausible victim-page contents. Hence, they are placed in the victim node's left and right subtrees and moved closer to the victim node through iterative red-black-tree rebalancing. Second, we infer the layout information of the red-black stable tree using a CPU cache side channel, such as Flush+Reload [69] or Flush+Flush [18]. From the inserted attacker-controlled pages, we select suitable deterministic nodes as traversal oracles. The observed cache activity determines whether the left or right subtree is accessed to traverse the stable tree, including the victim node. Third, we exploit KSM's comparison operations and tree-ordering properties to leak victim-page contents bit by bit. We craft and measure candidate queries that force a lookup directly through the victim node. By observing which oracle is accessed, we determine whether the queried value is smaller or larger than the victim value bit by bit.

To show the practicality and applicability, we provide a detailed attack evaluation and analysis. We further demonstrate end-to-end leakage of cryptographic key material across process-isolation and VM-isolation boundaries. Our attack is applicable to perform host-to-host, guest-to-guest, and guest-to-host leakage. In particular, we are able to leak high-entropy pages bit-by-bit, e.g., ECC and RSA private keys as well as API tokens, with a throughput of up to 30 bits per second. Additionally, we demonstrate that our attack works for KSM-deduplicated pages on systems with both synthetic (i.e., `stress-ng`) and realistic noise (i.e., Phoronix Test Suite [48]) as well as on Linux kernel versions 5.15 to 6.18.

Contributions. The main contributions of this work are:

(1) KSM Primitives & Synchronization: We introduce primitives that enable unprivileged control over KSM's data structures and synchronization with its scanning behavior. These primitives allow precise interaction with KSM internals, forming the basis for our leakage attack.

(2) A Novel KSM Leakage Attack: Using our massaging and layout leakage primitives, we present a novel attack capable of leaking KSM deduplicated pages bit by bit in $O(n)$, where previous attacks on similar conditions required an infeasible complexity of $O(2^n)$ or specific victim memory alignment properties.

(3) Evaluation and End-to-End Case Studies: We evaluate our attack on an up-to-date Ubuntu 24.04 machine, where we successfully leaked high-entropy deduplicated pages at rates of up to 30 bits per second. We evaluate our attack within synthetic and a realistic victim scenarios, analyzing the behavior under different KSM workloads and settings.

Outline. Section 2 provides the background. Section 3 gives a high-level overview of the attack. Section 4 presents our new attack, LeakyTree. Section 5 evaluates the attack, and Section 6 presents two case studies in end-to-end attack scenarios. Section 7 discusses the results, and Section 8 concludes the work.

Responsible Disclosure. We disclosed LeakyTree to the Linux Kernel Security Team, the Proxmox Security Team, and Andrea Arcangeli. We evaluated downstream mitigation patches from Andrea Arcangeli and planned a Proxmox VE security advisory.

2 Background

This section provides the background on memory deduplication, red-black trees, Linux Kernel Samepage Merging (KSM), and CPU caches.

2.1 Memory Deduplication

Memory is a crucial resource in modern computer systems. As users require increasingly more complex tasks to run, memory consumption scales accordingly. Memory is a physical hard limit, as a system cannot use more memory than is equipped. Depending on the required resources, it might be unfeasible to scale. Operating systems implement different techniques to reduce the memory footprint of workloads running on the system. For example, techniques such as Copy On Write (CoW) deduplicate memory when a process forks, as both processes share the memory transparently with a shallow copy until a write to it triggers a deep copy [56].

Memory deduplication is another approach to reduce the memory footprint. At its core, the operating system scans the entire memory of the system, deduplicating identical chunks into a single one and marking it as CoW. When needed, the operating system reverses the process by duplicating the memory transparently. Multiple operating systems [23, 71] and hypervisors [46, 55, 57] implemented transparent content-based memory deduplication: Linux KSM [2, 23, 55], Windows Memory Combining [71], and VMware ESXi Transparent Page Sharing (TPS).

2.2 Red-Black Trees

A red-black tree is a self-balancing *binary search tree* that guarantees logarithmic time complexity for queries, insertions, and deletions [4, 6, 38], with a maximum height of $2\log_2(n+1)$[8]. Compared to other binary search trees, red-black trees include a color field for nodes—red or black—, and establish a set of properties that the tree must follow. Basic operations are carried out the same way as in standard binary search trees, with the exception that the properties of the red-black tree are verified afterwards. In case an operation causes a violation of the properties, the tree applies self-balancing techniques to restore them.

Red-black trees must always satisfy the following properties: (1) every node is either red or black, (2) the root node is always black, (3) all *nil* leaves are black, (4) no red node can have a red child, (5) for each node, all simple paths from the node to the descendant leaves must contain the same number of black nodes [8].

To preserve red-black tree properties, the tree utilizes two operations: rotations and recoloring. Rotations are operations on nodes that cause changes to the layout of the tree; there are two types of rotations: *right-rotate* and *left-rotate*. Another complementary operation, recoloring, is used to change the color property of nodes.

2.3 Kernel Samepage Merging

KSM is the Linux transparent page deduplication mechanism, introduced in Linux `v2.6.32` [40] and bundled in common distributions, such as Ubuntu 24.04 and Fedora 43 [42]. Kernels with KSM support spawn a kernel thread named `ksmd`, which scans memory regions that were marked for deduplication. It iterates over processes, then over their Virtual Memory Area (VMA)s, and finally over pages within each VMA in increasing address order. During this scan, KSM skips pages that are not eligible for deduplication. To keep track of processed and deduplicated pages, KSM maintains two red-black trees ordered by `memcmp`: the *stable tree* and *unstable tree*.

The KSM *stable tree* is a red-black tree whose nodes represent pages that have been deduplicated and are managed by KSM. KSM enforces that the pages stored in this stable tree are read-only at the OS level, utilizing special page fault routines to capture user writes and revert the deduplication on the written page. Each node represents one physically-backed KSM page that can have multiple references—from different deduplications—up to a configurable upper limit, in which case a linked list of page nodes is created.

The KSM *unstable tree* is a red-black tree keeping processed pages for which no duplicate has yet been found. Unlike the stable tree, its nodes refer to user-controlled pages rather than KSM-managed deduplicated pages. Because these pages may change under normal user writes, the tree ordering is not guaranteed to remain valid. Hence, when KSM finishes a full scan of the system, this tree is discarded as a best-effort to tackle tree corruptions caused by users.

KSM processes each opted-in page for deduplication in turn. First, it queries the stable tree to locate an already deduplicated page. If a match is found, the queried page is deduplicated with the KSM page from the stable tree, discarding the processed page. Otherwise, KSM calculates a non-cryptographic hash (i.e., `xxHash`) of the page and compares it against the previously stored value as a heuristic to avoid processing rapidly changing pages. Third, KSM queries the unstable tree. If a matching page is found there, the pages are deduplicated and the processed page is promoted into the stable tree. Otherwise, the page is inserted into the unstable tree.

Once a page has been deduplicated, the resulting shared KSM page is marked read-only. A later write triggers a write-protection fault, causing the page to be duplicated again through CoW. The write is then applied to the newly created private copy.

KSM's scans operate in configurable bursts: `pages_to_scan` determines how many pages `ksmd` processes per pass, after which the process is put to sleep for `sleep_millisecs` milliseconds.

Deployment. Despite the security concerns raised by prior memory-deduplication attacks [3, 5, 16, 29, 56], KSM remains deployed in production-oriented environments. Proxmox Virtual Environment (Proxmox VE), with over two million hosts globally [50] in mission-critical environments [49], enables and utilizes KSM by default through the `ksmtuned` service and marks QEMU guest memory as mergeable. KSM has also been used to mitigate memory pressure in large-scale application deployments, such as Instagram workloads at Meta [58]. Moreover, continued upstream development from corporations and developers indicates that KSM remains an actively maintained and relevant subsystem.

2.4 CPU Caches

CPU caches are hardware structures (constructed from a fast memory technology like SRAM [36]) that temporarily store copies of frequently used data between the processor and the main memory (i.e., the DRAM [20]) to improve system performance [20, 47]. Cache access either results in a cache hit, i.e., when the requested data is found in cache memory, or a cache miss, i.e., when the data does not reside within the cache and must be fetched from main memory. Therefore, caches exploit the principle of locality [9, 10], that is, computer programs have the tendency to repeatedly access the same set of memory resources.

The system's memory subsystem connects the CPU and memory components via a hierarchical structure, i.e., multiple levels of cache memory [11, 13]. In particular, the memory subsystem typically consists of a private L1 (instruction and data) cache and L2 cache, alongside a shared L3 cache serving as Last-Level Cache (LLC) [60], while maintaining cache coherence across multiple cores [39]. Modern processors commonly use a set-associative cache architecture, where the L1 data cache is typically Virtually-Indexed and Physically-Tagged (VIPT), allowing a parallel indexing of the L1 cache lookup [27, 31, 72]. Additionally, the L2 and L3 cache is often Physically-Indexed and Physically-Tagged (PIPT) [34, 66].

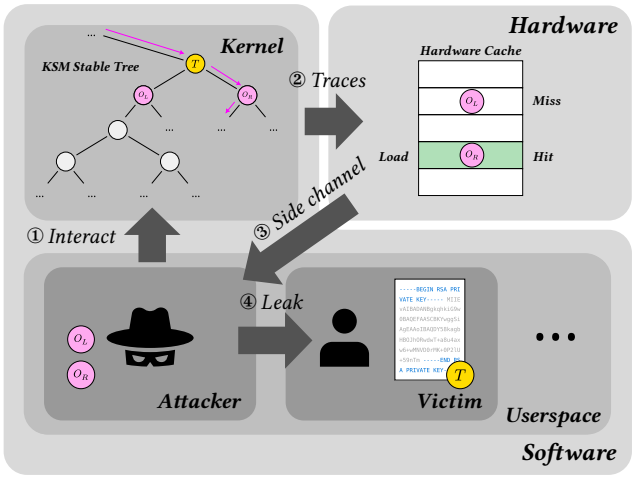


Figure 1: Interaction between unprivileged attacker code, privileged KSM operations, and hardware-cache observations in LeakyTree.

3 High-Level Overview

This section provides a high-level overview of LeakyTree. LeakyTree makes memory-deduplication attacks practical for high-entropy pages by exploiting observable software internals of Linux KSM. In contrast, prior attacks primarily relied on CoW behavior and required brute-force page matching for high-entropy leakage.

Figure 1 highlights how LeakyTree combines software behavior in KSM with hardware-cache observations: ① the attacker triggers KSM operations from user space using crafted pages marked for deduplication; ② KSM performs content-dependent stable-tree traversals over deduplicated pages (i.e., sensitive user data), leaving hardware-cache traces; ③ the attacker observes these traces through a hardware-cache side channel and correlates them with traversal decisions; and ④ by repeating this process with specifically crafted KSM operations, the attacker leaks the contents of deduplicated victim pages. By combining these observations, LeakyTree recovers deduplicated page contents *bit by bit* with complexity $O(n)$ across multiple attack scenarios. This substantially improves practicality over prior work, which either required $O(2^n)$ effort (i.e., impractical for large high-entropy pages) or requires a strong alignment and padding primitive for victim memory.

The following part first introduces the key KSM primitives used by LeakyTree (see Section 3.1) and then presents an overview of LeakyTree's multi-stage attack (see Section 3.2).

3.1 Attack Primitives

LeakyTree relies on three essential attack primitives that are used to carry out the attack. The primitives operate on KSM tree nodes, which represent memory pages of the system.

(P1) Insertion. This primitive inserts a new node into the KSM stable tree, which is a KSM-managed red-black tree and tracks already deduplicated pages. To achieve this, the attacker obtains two memory pages and initializes them with the same random contents. The contents are chosen such that no other page in the system is likely to contain the same data, for example, by using randomized values on the least significant byte (LSB) of the page. The attacker also marks both memory pages as candidates for KSM deduplication. As a result, KSM will process both pages and merge them into a single shared physical page, inserting a corresponding node into the stable tree representing the deduplicated page.

(P2) Query. This primitive queries the KSM stable tree for a node. To achieve this, the attacker obtains one memory page and initializes it with the contents to be queried in the tree. It is important that the queried content does not fully match an already existing page in the system, e.g., by randomizing the LSB of the page. During a subsequent scan, KSM processes the page and attempts to locate a matching entry in the stable tree via a tree lookup. As no matching node is found, KSM will continue performing the lookup during subsequent cycles.

(P3) Measure. This primitive determines whether KSM has accessed a node in the stable tree during a query or a non-deduplicated page during a KSM scan. To achieve this, the attacker must have the reference to the page, i.e., its user-space virtual address. The attacker then applies a CPU cache side-channel, such as Flush+Reload [69], to that address. Because stable-tree queries compare page contents using `memcmp` (see Section 2.3), KSM accesses the corresponding page contents during tree traversal and deduplication processing. These accesses bring the page into the CPU cache and therefore leave a measurable trace that can be observed from user space.

The measurement is possible because the attacker and KSM access the same physical cache line of the deduplicated page. Combined with the fact that we do not assume core colocation between the attacker and the *ksmd* thread, this requires a side channel that exposes accesses through the shared LLC. On modern systems, this effect is observable through the LLC, which is typically PIPT [34, 66]. Although we use Flush+Reload [69] in this paper, LeakyTree is not tied to a specific cache side channel, as discussed in Section 7.1.

3.2 Overview

LeakyTree leaks the contents of a deduplicated victim page bit by bit with overall complexity $O(n)$. The attack targets KSM's stable tree, which stores deduplicated pages and is ordered by page contents. We assume that the target victim page has previously been deduplicated and that its node is therefore present in this tree. This requires at least two identical instances of the page to occur in KSM-eligible memory, a condition that can arise naturally (see Section 7.3). We briefly introduce the attack in three stages, while Section 4 presents each stage in detail.

First, in the **stable tree massaging** stage, we shape the layout of KSM's stable tree around the victim node using only unprivileged user-space operations. Using primitive **(P1)**, we insert attacker-controlled nodes to manipulate the local tree layout around the victim node. Specifically, based on the known victim's page prefix, we generate pages with contents close to the lower and upper bounds of plausible victim-page values. For example, for an RSA private key with known header `−−BEGIN RSA PRI`, we define bounds `−−BEGIN RSA PRI00...0` and `−−BEGIN RSA PRIFF...F`, which set the victim search range. Due to the tree ordering, these pages are inserted into the victim node's left and right subtrees. Inserting

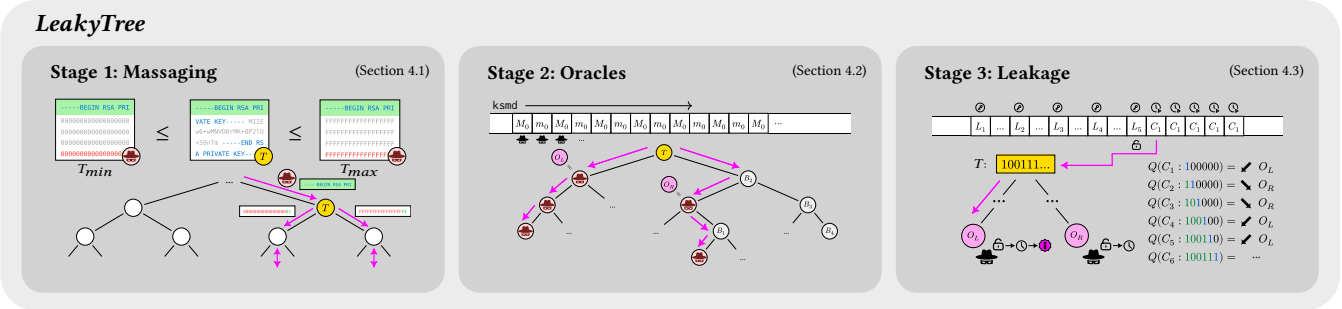


Figure 2: Three-stage LeakyTree. First, an attacker inserts pages under T's subtree by bounding its value range with a prefix. Second, the attacker selects the closest controlled left and right page to T such that any query within the defined T has a path crossing a page. Third, the attacker finds T with a binary search using the comparison oracle constructed with O_L and O_R.

many attacker-controlled pages near these range boundaries triggers red-black-tree rebalancing, which moves attacker-controlled nodes closer to the victim node. As a result, both the left and right subtrees of the victim node contain attacker-controlled nodes.

Second, in the **finding oracle nodes** stage, we identify two attacker-controlled nodes in the victim node's left and right subtrees, which we denote as the left oracle O_L and the right oracle O_R. Due to the ordering property of the stable tree, any traversal within the victim's search range that passes through the victim node must continue into either its left or right subtree. We therefore select one node from each subtree such that these traversals also access either O_L or O_R. To identify suitable candidates, we partially infer the local subtree structure around the victim node. This is achieved by combining primitives **(P2)** and **(P3)**: we issue boundary-value queries for the victim's search range while monitoring cache accesses to attacker-controlled nodes. From the observed accesses, we select the highest-valued left-side candidate as O_L and the lowest-valued right-side candidate as O_R.

Third, in the **oracle-assisted key space search** stage, we exploit KSM's comparison operations to leak the victim-page content bit by bit. Because the stable tree is ordered by page contents, each lookup compares page values along the traversal path. Using **(P2)**, we issue crafted candidate queries that cause the traversal to pass through the victim node. Depending on the comparison result, the traversal continues into either the left or the right subtree of the victim node. As established in the second stage, accesses to oracle nodes O_L and O_R leave observable cache traces that the attacker can detect using **(P3)**. By observing which oracle is accessed, the attacker learns whether the queried value is smaller or larger than the victim-page value. By repeating this process and refining the queries accordingly, the attacker recovers the victim page contents bit by bit, with overall complexity $O(n)$.

4 LeakyTree

This section presents the detailed three stages of our LeakyTree attack while Figure 2 shows the internals of the stages.

4.1 Stable Tree Massaging

The attack begins by restricting the search to a finite key space that contains the unknown target page T. We define this key space as

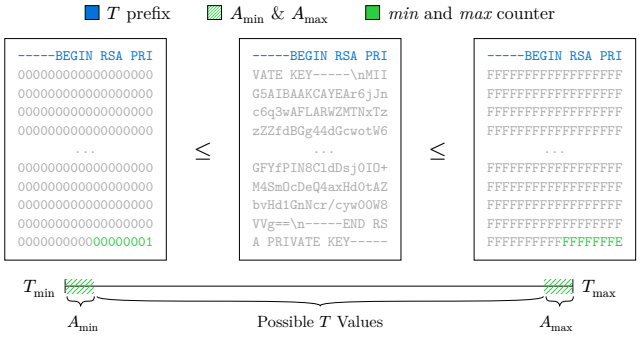


Figure 3: *min* and *max* pages.

an interval $[T_{min}, T_{max}]$ that bounds the contents of T. The interval is constructed by fixing a prefix of the page to a known value—e.g., headers—while leaving the remaining bytes unconstrained. This defines the class of victim pages that LeakyTree targets for leakage. Figure 3 illustrates this process for an ASCII-encoded RSA private key. In this example, the interval is defined by fixing part of the known RSA header, while all remaining bytes span values between `0x00` and `0xFF`. Although this constraint reduces the search space, it remains infeasible for naive enumeration (e.g., an $O(2^n)$ key-space search), as it would require testing all possible keys. As LeakyTree reduces the leakage complexity to $O(n)$, the key space becomes tractable.

After defining the range $[T_{min}, T_{max}]$, the attacker allocates pages whose contents correspond to the lower and upper bounds of the defined interval. These pages are organized into two classes, referred to as *min* pages and *max* pages, denoted A_{min} and A_{max}. A_{min} contains pages with values close to T_{min}, whereas A_{max} contain pages with values close to T_{max}. Within each class, pages are ordered: A_{min} is arranged in increasing page-value order, while A_{max} is arranged in decreasing order. Practically, we increase or decrease the least significant bytes of T_{min} and T_{max} to generate pages from both classes.

With both classes—A_{min} and A_{max}—allocated, we iterate over them and utilize the insertion primitive **(P1)** to insert the pages onto the stable tree. As the inserted page is relatively close to T in value—due to the same known header—its insertion will cause a

path that traverses T, meaning that the page gets inserted on the subtree formed by T. Due to the sorted property of the red-black stable tree, pages belonging to A_{min} are inserted on the left branch of T, while pages from A_{max} are inserted on the right branch of T. These attacker-controlled pages are initially inserted on leaf nodes of the subtree from T.

With these crafted insertions, we seek to force structural changes in the red-black subtree formed by T via forced red-black tree rebalancements. As pages are inserted in the *worst-case* scenario—ordered in content—nodes are inserted as the direct children of the previously inserted one, leading to red-black tree property violations. This forces the red-black tree to rebalance itself, applying rotations and recolorings that lift the attacker-inserted pages further up in the subtree formed by T. With enough insertions from A_{min} and A_{max}, we force enough rebalancements to have attacker-inserted pages as direct children of T, or close to T. Formally, we require that for every page value v in the interval $[T_{min}, T_{max}]$, the search path for v through the tree is guaranteed to traverse T and at least one attacker-controlled node from A_{min} or A_{max}.

$$\forall v \in [T_{\min}, T_{\max}] : \text{path}(root, v) \supseteq \{T\} \cup \{a \mid a \in A_{\min} \cup A_{\max}\} \quad (1)$$

The number of attacker-controlled insertions required to achieve the state in Equation (1) depends on both the total size of the tree and the depth of the target node T in the tree. As we lack direct introspection into the fine-grain tree structure, we must rely on an estimate. We can conservatively under-approximate the required number of insertions, and increase it if the attack does not proceed or stops at a certain point. Most of our evaluations required only $1 - 16$ node insertions per subtree, with none requiring more than 256 nodes. Inserting a low number of pages carries no impact and can further be adjusted.

Target Page. Crucially, establishing the interval range does not imply the knowledge of T. Defining the interval allows us to target the leakage of a page in the system with specific data, such as keys or images. This gives us a more granular leakage attack, although we can also define the interval range as the minimum and maximum page value in order to leak a generic system page.

4.2 Finding Oracle Nodes

In this stage, we use the structural manipulation in Section 4.1 to build reliable primitives for a practical side channel that, for any query $v \in [T_{min}, T_{max}]$, reveals whether T proceeds to the left or right subtree. This side channel can be interpreted as a branch side channel because it leaks information about the branch being taken.

After the massaging stage, the target page node T contains attacker-inserted pages in both its left and right subtrees. Now, we aim to find the most favorable attacker-inserted page to build the branch side channel. We denote the most favorable page as O_L for the left branch and O_R for the right branch; both are referred to as oracle nodes. For any queried value within our defined range, the path must contain at least O_L or O_R (see Equation (1)). As an illustrative example, an optimal pick for O_L and O_R nodes is the direct children of T. In this case, any query branching on T will always result in a path containing one of the selected oracle nodes, following our defined rule (see Equation (1)).

Practically, we do not require the oracle nodes to be the direct children of T, as the condition is more relaxed. Hence, we can find suitable oracle nodes deeper in the subtree, which are easier to find than the optimal ones. Finding nodes that can be candidates for being an oracle node involves the partial leakage of the subtree structure. Specifically, we partially leak the structure to infer which of the inserted nodes from A_{min} and A_{max} are the closest to T within its subtree. In order to obtain this information, we use the measure primitive **(P3)** on a limited, tractable subset of the attacker-inserted pages, while using the query primitive **(P2)** for the extreme values of the range (e.g., T_{min} and T_{max}). The query for the aforementioned extreme values causes an *always-left* and *always-right* access pattern over the attacker-inserted pages, as they are the minimum and maximum value of the range, respectively. By using the measure primitive **(P3)**, we observe the resulting access pattern of the query on A_{min} and A_{max} page nodes. Afterwards, we group all pages where we measure an access on the sets H_{min} and H_{max}, containing accessed *min* and *max* pages, respectively. Finally, considering the caused path follows an *always-left* and *always-right* for branches on nodes H_{min} and H_{max}, and the red-black tree ordering, we define O_L and O_R by Equation (2).

$$O_L = \max(H_{min}), \qquad O_R = \min(H_{max}) \quad (2)$$

The selection rule in Equation (2) follows directly from the tree ordering invariant. For H_{min}, the node that is placed at the highest level from the set within the subtree formed by T is the one with the highest value from the set. As we queried for the minimum value of the range, the branch taken is always the left, meaning the one with a page node of smaller value than the current one. This causes the query to traverse nodes in decreasing order always, meaning that the highest node from the resulting path is the one of higher value. The opposite occurs for H_{max}, based on the same principle, the query searches the maximum value of the branch, always taking the right branch, where nodes are larger in value. We end up with a path of nodes in increasing value, meaning that the highest node in the subtree for H_{max} is the minimum page in value, as all the rest were taken after a right branch.

Query & Measure Primitives. For this stage of the attack, we combined the query **(P2)** and measure **(P3)** primitives in order to partially leak the structure of the subtree formed by T. The combination of these primitives into an operation that lets an attacker measure the access pattern of the executed query is not straightforward and requires additional considerations, as we will later explain in Section 4.3.1.

4.3 Oracle-Assisted Key Space Search

The last stage of LeakyTree leverages the previously obtained nodes O_L and O_R to construct a *greater-or-lesser* oracle for any query $q \in [T_{min}, T_{max}]$ against T. We then leverage this oracle to perform a binary search of T on the range $[T_{min}, T_{max}]$. Effectively, every usage of the *greater-or-lesser* oracle leaks one bit of the target page at a time.

The subtree formed by node T contains the attacker-controlled node O_L on the subtree formed by the left child, and O_R on the subtree formed by the right child. As O_L and O_R follow Equation (1),

we model T as a node with O_L and O_R as left and right children nodes.

An attacker queries a value in the range $[T_{min}, T_{max}]$ via **(P2)**. KSM queries the stable tree for the value, arriving at target node T as part of the query path. Then, branches left or right depending on the result of the *memcmp* between the queried page and T, i.e., depending if the value is lesser or bigger. This constitutes a branch over secret data contained by T. Based on the result of the branch, KSM continues traversing the subtree formed by T on the left or right side of the tree. As per Equation (1), O_L or O_R will form part of the traversed path for the query afterwards. Whether O_L or O_R form part of the path depends on the branch taken on T, as the left side of the subtree contains *min* pages—close to T_{min} in value—, and the right side contains *max* pages—close to T_{max} in value. When either of the oracle nodes gets processed by KSM as part of the query, the page gets accessed in order to perform the memory comparison to decide on which branch to take. This memory comparison causes a memory load, leaving an observable trace by the attacker via the primitive **(P3)**. Based on this measurement, the attacker constructs the *greater-or-lesser* oracle shown in Equation (3), where hit represents an observed memory access.

$$\text{hit}(O_L) \implies q < T, \qquad \text{hit}(O_R) \implies q > T. \tag{3}$$

With the constructed oracle, an attacker performs a binary search of T over the range $[T_{min}, T_{max}]$. First, it picks a value v in the middle of the range, and queries the constructed oracle, determining if v is greater or lesser than T. Based on the obtained result, the attacker picks the next middle value of the upper or lower half of the range, repeating the query. With each query, the attacker halves the range by two, effectively searching in $O(log_2(2^n))$ which is $O(n)$. The attacker continues until it finds T in the range.

Practically, the attacker sets the first unknown bit after the fixed header to one, and utilizes the oracle to compare against T. If the value is *lower*, it unsets the bit to zero. If the value is *higher*, it keeps the bit set. Afterwards, it sets the next bit and proceeds again. Each query leaks one bit of the contents of the secret page T, from the Most Significant Bit (MSB) to the Least Significant Bit (LSB).

4.3.1 Precise Measurements. In Section 4.2 and Section 4.3 we utilized the combination of the query **(P2)** and measure **(P3)** primitives. When combining both primitives, an attacker can leak the path caused by an attacker-controlled query. Combining the query primitive **(P2)** and the measurement primitive **(P3)** poses a synchronization problem: the attacker must ensure that the short measurement window of **(P3)** overlaps the exact instant when *ksmd* performs the stable tree lookup caused by the attacker's **(P2)** query. As *ksmd* is a separate privileged kernel thread, standard synchronization techniques are not available.

We synchronized the attacker and *ksmd*'s kernel thread by making use of the predictable scan order of KSM. KSM processes the candidate system memory in a stable, predictable order. For each candidate VMA within a `struct mm`, all contained anonymous pages are processed in an increasing order within its range. We utilize this property as a distinguisher to detect KSM processing within a VMA.

Synchronization Primitive. To synchronize a query Q of a page v with measurement primitives, we allocate N contiguous pages via *mmap*. *First*, we fill the contents of all allocated pages with the contents of v, randomizing the least significant integer of the page to avoid deduplication between them. *Second*, we mark the allocated page for deduplication via `madvise` [2], as we do in the query primitive. *Third*, we utilize the measure primitive over a set of L pages from the range, we repeat the measurements in a loop until we detect an increasing sequential access pattern, ignoring individual hits. *Finally*, we utilize the measure primitive on selected nodes—such as O_L or O_R—, meanwhile KSM processes the end of our allocated range, causing tree lookups. This technique serves as a synchronization primitive between the unprivileged attacker thread and the privileged *ksmd* kernel thread. In Figure 2, stage 3 visualizes the mentioned page range, showing the pages used for synchronization as L_x, while measurement pages are shown as C_i.

4.3.2 Multioracles. Performing a combined query and a measure primitive requires to wait for KSM to arrive to the attacker VMA and process its pages. Effectively, an attacker has to wait for a full processing cycle of KSM to complete in order to leak one single bit of T. On systems with conservative configurations or heavy load, cycles can take longer and thus, decrease the leakage range of T. An attacker can overcome this limitation by making use of the property of KSM where it processes VMAs and pages in order.

An attacker first allocates a large memory range on a VMA (e.g., via `mmap`), and subdivides it in oracle regions, consisting of N pages each. Each oracle region is separated from the next one by G dummy pages containing randomized contents. When KSM arrives to the attacker VMA, it will process each oracle region in order, and waste time between regions by processing the dummy pages. The attacker can leak one bit from the target page T on the first oracle region, and adjust the second oracle region with the leaked bit in order to leak the second bit. Meanwhile, KSM is processing the dummy pages, when it finishes it continues processing the second region, leaking another bit for the attacker. This continues for each oracle region contained in the VMA, leaking one bit per oracle region. Afterwards, KSM leaves the attacker VMA region and continues its normal processing, taking longer to leak the next bit based on system configuration and workload.

By employing this technique, an attacker is able to leak multiple bits per KSM cycle, increasing the leakage rate on all conditions. On unfavorable conditions, such as high KSM workload or conservative KSM configurations, this technique can make the attack suitable where previously the leakage rate would have not been feasible.

5 Evaluation

This section evaluates LeakyTree in a controlled synthetic setting that emulates different victim conditions, allowing us to systematically analyze the resulting attack metrics. The goal is to provide an isolated evaluation of LeakyTree, while realistic attack scenarios are explored separately in Section 6.

Environment. We conduct all experiments on an Ubuntu 24.04 desktop system running an upstream-compiled Linux `6.6.133` *longterm* kernel, released the 6th of April 2026. We also evaluate on other, recent *longterm* kernels between `5.15.202` and `6.18.21`.

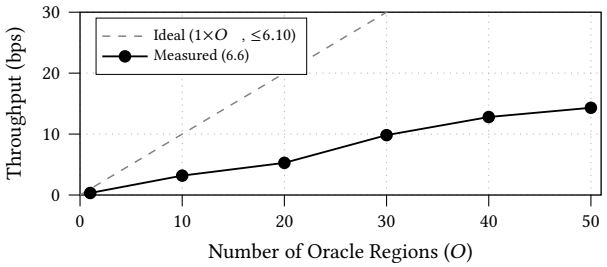


Figure 4: Multioracle speedup on KSM with $3 \cdot 10^6$ shared and unshared pages. Dashed line: ideal linear scaling per oracle region. Measured throughput scales linearly at a lower rate.

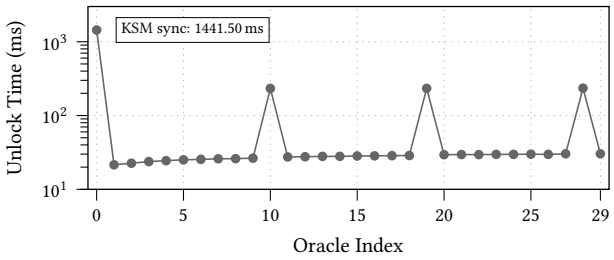


Figure 5: Time to leak bit per oracle index. Oracle 0 waits for a full KSM cycle, while subsequent oracles unlock quicker, yielding the higher throughput observed in Figure 4.

Table 1: LeakyTree Performance Evaluation.

Scenario		LeakyTree						Results				
K_{sh}	K_{un}	I	M	L	W	O	G	$O_t(1)$ (ms)	O_m	O_s	*Rollb.*	**Throughp.** (bps)
Baseline												
10^5	10^5	0	10^3	3	3	50	10^3	1020.42	4	0	3	31.315
10^5	10^5	0	10^3	3	1	50	10^3	1033.18	3	0	1	30.067
10^5	10^5	0	10^3	5	1	50	10^3	810.16	7	0	2	15.258
10^5	10^5	0	10^3	5	1	50	500	1246.54	12	0	1	0.826
Different KSM Workloads												
1	0	0	10^3	3	3	50	10^3	274.00	0	11	2	4.684
10^3	10^3	0	10^3	3	3	50	10^3	198.12	0	1	2	18.683
10^4	10^4	0	10^3	3	3	50	10^3	224.71	0	1	2	61.293
10^5	10^5	0	10^3	3	3	50	10^3	1189.26	0	0	2	31.216
10^6	10^6	0	10^3	3	3	50	10^3	9799.00	0	1	0	4.208
Multioracle Speedup												
10^5	10^5	0	10^3	3	3	1	10^3	1158.73	0	0	0	0.846
10^5	10^5	0	10^3	3	3	10	10^3	1202.64	0	0	1	6.766
10^5	10^5	0	10^3	3	3	20	10^3	1093.55	1	0	2	15.389
10^5	10^5	0	10^3	3	3	30	10^3	1093.37	0	0	4	21.638
10^5	10^5	0	10^3	3	3	40	10^3	1095.64	5	0	2	28.852
10^5	10^5	0	10^3	3	3	50	10^3	1050.28	4	1	4	30.147

K_{sh}: stable tree size; K_{un}: unstable tree size; I: inserted A_{min} and A_{max} pages as 2^I; M: pages per oracle region; L: number of locks; W: required hits on oracle; O: number of oracle regions; G: garbage (pad) between oracles; $O_t(1)$: leak time of first oracle (ms); O_m: total oracle misses; O_s: total oracle skips; *Rollb.*: rollbacks needed; Throughp.: throughput (bps).

The used CPU is an Intel Core i7-13700 (13th Gen) paired with 32GB of RAM.

Synthetic Victim. To precisely emulate different KSM workload scenarios, we implement the victim target as a tweakable unprivileged userspace program. The victim program induces controllable KSM activity and embeds a target page for leakage. It allocates and initializes duplicate memory pages to force their insertion into the stable tree, enabling control over its size. At the same time, it allocates randomized, non-duplicate pages to regulate the size of the unstable tree. Finally, a randomly selected page pair from the duplicated set is initialized with an ASCII-encoded RSA private key, ensuring its insertion into the stable tree as the target of the attack.

5.1 Performance Evaluation

This experiment evaluates the core performance of LeakyTree under different conditions and attack parameters. We prepare different KSM workload scenarios with our synthetic setup and run LeakyTree, varying the attack parameters and observing the leakage result. To ensure consistent evaluation, we set `pages_to_scan` to 100,000 on all experiments, while showing the effect of this parameter on leakage rates in Section 5.4.

Table 1 summarizes the observed performance of LeakyTree across a range of workload scenarios. Overall, our results demonstrate that the attack is feasible, both in small-scale settings with stable and unstable trees on the order of 10^3 nodes, and in larger settings with millions of nodes (10^6). We observe that the optimal parameterization of LeakyTree depends strongly on the underlying KSM workload. For instance, configurations with $O = 50$ oracle regions performed well for tree sizes around 10^4, whereas in smaller trees the leakage rate decreased. This effect is primarily due to faster scan cycles, which may skip oracle regions, causing a desync between `ksmd` and the attacker process. In such scenarios, parameters such as the gap size G between oracle regions must be adjusted accordingly. Finally, Figure 4 illustrates the near-linear performance improvement achieved through the presented multioracle technique.

In practice, this parameter dependency does not preclude exploitation, as an attacker can adapt these parameters during runtime to the observed KSM behavior. By monitoring whether oracle regions are processed reliably and whether synchronization with `ksmd` is maintained, the attacker can tune parameters such as the number of oracle regions O and the gap size G to better suit the current scenario.

Multioracle Unlock Time. LeakyTree benefits from the use of multiple oracle regions, particularly on systems with high KSM workloads, where scan cycles take longer before attacker-controlled pages are processed again. To observe this behavior, we evaluate LeakyTree with 30 oracle regions (O) on a system containing $3 \cdot 10^5$ nodes across the stable and unstable trees, and measure the time required for each region to leak a single bit. Figure 5 illustrates the results. The first oracle region exhibits the highest latency, as it reflects the time elapsed from the last attacker-controlled region until KSM completes a full scan cycle and revisits the first attacker-controlled region. In contrast, subsequent oracle regions yield results almost immediately, as they are processed sequentially within the same scan pass. We further observe periodic latency spikes of approximately 200 ms, which we attribute to the KSM configuration parameter `sleep_millisecs`, whose default value is 200 ms.

Table 2: Evaluation results for synthetic noise using `stress-ng` and realistic noise using the Phoronix Test Suite.

Workload	Results				
	$O_t(1)$ (ms)	O_m	O_s	*Rollb.*	**Throughp.** (bps)
Baseline					
Idle	1176.32	0	0	4	6.382
Synthetic Noise (stress-ng)					
50% CPU load	22562.82	1	0	4	0.220
70% CPU load	19472.23	0	0	4	0.348
90% CPU load	24724.46	1	0	6	0.098
Realistic Noise (Phoronix Test Suite)					
`pts/nginx-3.0.1` (c=100)	11495.09	2	0	1	0.479
`pts/build-imagemagick-1.7.2`	1882.84	0	0	3	3.179
`pts/build-linux-kernel-1.16.0`	3645.35	13	1	6	1.512

Table 3: LeakyTree evaluation results across different Linux kernel versions from `5.15` to `6.18`.

Kernel	Scenario			LeakyTree		Leakage				
Version	K_{sh}	K_{un}	`smart`	O	Variant	$O_t(1)$ (ms)	O_m	O_s	*Rollb.*	**bps**
`5.15.202`	10^5	10^5	–	1	base	1062.90	0	0	0	0.924
`5.15.202`	10^5	10^5	–	4	base	1082.07	0	1	2	2.985
`6.1.167`	10^5	10^5	–	1	base	1077.40	0	0	0	0.906
`6.1.167`	10^5	10^5	–	4	base	1404.92	0	1	0	2.652
`6.6.133`	10^5	10^5	–	1	base	1083.59	0	0	0	0.902
`6.6.133`	10^5	10^5	–	4	base	1057.86	0	0	1	3.296
`6.12.80`	10^5	10^5	×	1	base	1890.98	0	0	0	0.518
`6.12.80`	10^5	10^5	✓	1	base	10755.27	0	0	0	0.009
`6.12.80`	10^5	10^5	✓	1	base+F1	1456.54	0	2	0	0.648
`6.12.80`	10^5	10^5	✓	4	base+F1	6472.00	7	1	0	0.147
`6.12.80`	10^5	10^5	✓	4	base+F1+F2	1441.00	0	0	0	1.300
`6.18.21`	10^5	10^5	✓	1	base+F1+F2	1279.22	1	0	0	0.642
`6.18.21`	10^5	10^5	✓	4	base+F1+F2	1529.11	0	0	0	1.147

5.2 Noise Evaluation

To assess the robustness of LeakyTree under varying system workloads, we evaluate its performance in the presence of both synthetic noise and noise induced by real workloads. We reuse the experiment setup from Section 5.1 and introduce controlled synthetic noise using `stress-ng`, as well as realistic workload noise using the Phoronix Test Suite [48]. The results are summarized in Table 2.

Overall, we observe that LeakyTree's progress primarily depends on two conditions: whether the attacker remains synchronized with `ksmd`, and whether the attacker is scheduled long enough to carry out the corresponding measurement and update steps. Consequently, system load mainly affects throughput by disturbing this synchronization or by reducing the runtime available to the attacker, rather than changing the underlying attack mechanism.

Synthetic Noise. We introduce synthetic noise using `stress-ng`, executing `stress-ng –cpu 0 –cpu-method fft –cpu-load N –cache 4` with varying values of N, which controls overall CPU utilization. As shown in Table 2, system load has a significant impact on the performance of LeakyTree, although the leakage rate is still feasible. At 50 % CPU utilization, throughput drops to 0.220 bps, from the baseline 6.382 bps. The primary source of slowdown is the time to reach the first oracle region $O_t(1)$, which drastically increases from 1176.32 ms, to 22562.82 ms.

Phoronix Test Suite Noise. We further evaluate LeakyTree under realistic system workloads using benchmarks from the Phoronix Test Suite. The results in Table 2 show that the slowdown is less pronounced compared to synthetic noise. This behavior is primarily due to the dynamic nature of real workloads, which involve frequent context switches and fluctuating resource utilization. For example, during workloads such as the Linux kernel compilation (`pts/build-linux-kernel`), we observe short periods of reduced system load during which LeakyTree achieves temporary higher leakage rates. Overall, we observe that LeakyTree adapts to varying system conditions and leaks in short bursts when pressure is reduced—even for very short slices—and both the attacker process and `ksmd` obtain CPU time.

5.3 Multiple Kernels

We evaluate the behavior of LeakyTree across five Linux kernel *long-term* versions, ranging from `5.15` to `6.18`. For each kernel, we replicate the performance evaluation setup described in Section 5.1 and execute the attack with $O = 1$ and $O = 4$. We further evaluate the impact of KSM's `smart_scan` feature on newer kernels.

Table 3 presents the obtained results. Kernels `5.15`, `6.1`, and `6.6` present consistent behavior with past evaluations. The baseline configuration ($O = 1$) yields stable leakage rates around 0.924 bps, while increasing the number of oracle regions to $O = 4$ improves throughput approximately by the rate of O. Starting from kernel `6.12`, we observe a substantial degradation in performance, bisected to the commits `5e924ff54d08` [59] and `d58a361b0350` [7]. These changes are not intended as defenses against LeakyTree. However, they introduce artifacts in KSM's behavior that affect how LeakyTree interacts with the subsystem and therefore require minor adaptations of the attack. Therefore, we present the following tailored attack variants:

Variant F1. Commit `5e924ff54d08` [59], introduced in Linux `6.7`, introduces the default `smart_scan` heuristic in KSM, which skips pages that repeatedly fail to merge. As shown in Table 3, enabling `smart_scan` on the base LeakyTree significantly degrades the throughput, as attacker VMAs are skipped after some scan cycles. To address this, we designed a lightweight variation of LeakyTree (F1), which bypasses the `smart_scan` heuristic. We maintain two copies of the attacker-controlled VMA of LeakyTree's third stage (see Section 4.3.1). While one VMA is actively used for leakage on the current KSM scan, the other gets marked as `MADV_UNMERGEABLE`, which removes the associated KSM metadata and resets the corresponding `smart_scan` counters. By alternating between the two VMAs, we partly regain the speed lost by `smart_scan` as attacker VMAs are not skipped due to the counter reset, recovering the original throughput, as shown in Table 3.

Variant F2. Commit `d58a361b0350` [7], introduced in Linux `6.12`, modifies the KSM scan behavior to recompute and check the page hash value before performing a stable tree lookup. This change breaks LeakyTree configurations with $O > 1$, as the result obtained

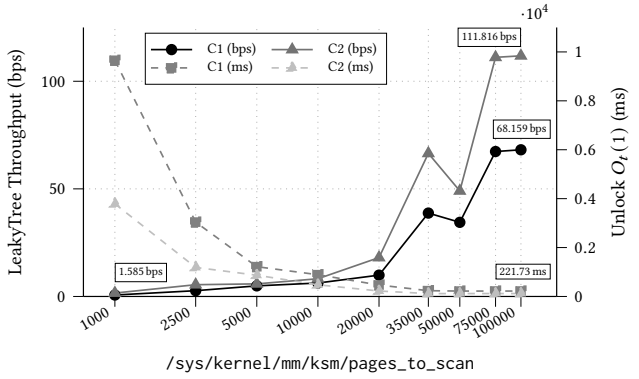


Figure 6: Impact of `pages_to_scan` on LeakyTree throughput.

from oracle region N must be written into oracle region $N+1$. Writing the required value alters the page contents and consequently its hash, preventing the modified page from being queried against the stable tree. To address this change, we designed a variant of LeakyTree (F2), which avoids runtime modifications of oracle regions. Instead, F2 precomputes combinations of candidate bits for each oracle region in advance. As a result, no dynamic writes are required during the attack, preserving the hash value and the stable tree query. This approach reduces the theoretical speedup of the multioracle technique from linear in O to $\log_2(O)$. Our evaluations show how $O = 4$ with patches F1 and F2 on Linux `6.18` approximately doubled the speedup from the $O = 1$, expected with the new $\log_2$ speedup.

5.4 KSM Configuration

We evaluate how KSM's scan configuration influences LeakyTree by varying `pages_to_scan` under two configurations: C1, using the default `sleep_milliseconds` of 200 ms and C2, using a reduced sleep interval of 100 ms. For the environment and workload simulation, we use the same setup as in Section 5.1, with K_{sh} and K_{un} both set to 10^4. Figure 6 visualizes the results. As `pages_to_scan` increases, throughput improves, while the first oracle unlock time $O_t(1)$ decreases accordingly. This highlights $O_t(1)$ as the primary bottleneck of LeakyTree under different KSM configurations. In particular, lower scan budgets cause *ksmd* to spend more time processing unrelated pages, increasing $O_t(1)$ and reducing throughput. Reducing `sleep_millisecs` (C2) consistently improves throughput by decreasing the wait time between `pages_to_scan` bursts. We observe occasional throughput drops, which are caused by temporal desynchronization between the attacker and the *ksmd* thread. These effects can be easily mitigated by tuning LeakyTree with parameters correspondent to the current workload—i.e., faster scans, lower scans. However, we intentionally keep LeakyTree parameters fixed to ensure comparability across configurations.

6 Case Studies

In this section, we present two case studies with threat models that arise in realistic workloads, where we apply LeakyTree from a fully unprivileged attack end-to-end scenario. First, we show how LeakyTree is used to leak pages between two processes. Second, we show how we can alter the primitives and use LeakyTree to leak pages between two different virtual machine guests.

Table 4: Case Studies Evaluation.

Scenario			LeakyTree		Results
Scenario	K_{sh}	K_{un}	I	Target / Leaked	**Throughp.** (bps)
Process-to-Process (Section 6.1.3)					
S1 - firefox	7782	8183	4	`sourceURL=https://en.wikipedia`	2.737
S2 - codex	8857	9701	8	`You communicate reasoning expl`	7.011
S3 - zsh/docker	186	1041	2	`export STRIPE_SECRET_KEY="sk_l`	10.671
Guest-to-Guest (Section 6.2.3)					
S4 - Ubuntu	398854	911811	0	`root:x:0:0:root:/root:/bin/bas`	4.282
S4 - Ubuntu	364494	1070436	4	`-----BEGIN CERTIFICATE-----\nM`	0.858
S4 - Ubuntu	343325	1084096	6	`-----BEGIN EC PRIVATE KEY-----`	1.452
S5 - Windows	689360	1340033	3	`C:\Windows\Microsoft.NET\Frame`	1.423
S5 - Windows	802844	1077969	0	`MicrosoftTelemetryAssertTrigge`	1.864
Guest-to-Host (Section 6.2.3)					
S6 - Synthetic	153651	1552667	0	`-----BEGIN RSA PRIVATE KEY----`	1.147

Environment. We perform all case study evaluations on the same hardware and software configurations described in Section 5. To validate the correctness of the leaked page with LeakyTree, we employ a privileged kernel module that allows dumping the selected pages from the KSM stable tree on demand. The module requires a minimal kernel patch to export the necessary symbols for linking. Importantly, this instrumentation is solely used for validation and does not influence or interfere with the execution of LeakyTree or its evaluation.

6.1 Cross-Process Leakage

6.1.1 Threat Model. We consider a Linux system with KSM enabled, where multiple privileged and unprivileged processes execute concurrently on the same host. The attacker controls an unprivileged process with no permissions beyond standard user-level access and can interact with the host kernel through regular *syscalls*. The objective is to recover the contents of memory pages belonging to other processes on the system, including both privileged and unprivileged targets, without any direct interaction or communication. Victim pages are assumed to be subject to KSM deduplication, either explicitly or implicitly through common mechanisms. These include invoking `madvise` with `MADV_MERGEABLE` (e.g., in applications such as QEMU or Tensorflow [62]); enabling KSM via `prctl` with `PR_SET_MEMORY_MERGE`; inheriting mergeable properties through process forking; being part of a `systemd` service configured with `MemoryKSM=y` [14]; or system-wide configurations enforced by the administrator (e.g., via `LD_PRELOAD`) [58].

6.1.2 Primitive Implementation. In the cross-process setting, all primitives of LeakyTree can be implemented directly from an unprivileged process using standard system interfaces. LeakyTree requires two main capabilities: interaction with KSM and the allocation of virtually contiguous memory. Interaction with KSM is necessary to mark pages as candidates for deduplication, while contiguous memory is required to ensure a predictable scanning order by KSM. Since the attacker can directly interact with the KSM-enabled kernel, marking pages for deduplication is straightforward and can be achieved through existing interfaces such as

`madvise` or `prctl`. Allocating virtually contiguous memory is likewise a standard operation, which can be achieved using common interfaces (e.g., with `mmap`).

6.1.3 Evaluation. We evaluate the cross-process case study on a KSM-enabled Linux system using the same hardware and software environment as described in Section 5.1. The attacker and victim run as independent processes on the same host, without any direct interaction. We consider multiple scenarios (**S1-S3**), each targeting a different victim process under memory deduplication, reflecting diverse scenarios. Table 4 presents the results. In scenario **(S1)**, we target a *firefox* process with Wikipedia open in two instances, manually enabling memory deduplication via `prctl`. From a separate unprivileged process on the same host running LeakyTree, we successfully leak parts of the loaded webpage, recovering the visited URL fragment. In scenarios **(S2,S3)**, we enable `MemoryKSM=y` on the *sshd* system service, activating deduplication for *sshd* and *sshd*-spawned processes (i.e., *ssh* clients). In **(S2)**, we establish multiple *ssh* sessions running *Codex*, triggering deduplication of prompt data. Using LeakyTree, we recover fragments of the internal Codex `gpt5.4` prompt from deduplicated pages. In **(S3)**, we embed API secrets as environment variables in the shell configuration file and initiate multiple *ssh* sessions. We then execute LeakyTree from an unprivileged Docker container without capabilities and successfully recover an API key from the host. Overall, we show that LeakyTree is able to leak both structured application data and bounded high-entropy secrets without alignment primitives over the victim process, and within restricted cross-process scenarios (i.e., Docker).

6.2 Cross-VM Leakage

6.2.1 Threat Model. We consider a cloud environment in which a Linux system acts as a hypervisor and enables memory deduplication via KSM to reduce overall memory consumption. As a hypervisor, multiple Virtual Machine (VM)s managed by QEMU share resources on the same physical host. The attacker has root privileges within its provisioned VM but has no code execution capabilities outside of it, and no direct interaction with other VMs. The goal of the attacker is to recover the contents of a high-entropy, non-guessable page residing in a separate victim VM. This threat model closely reflects realistic cloud environments and aligns with the threat model of prior KSM attacks from the VM [29, 61].

6.2.2 Primitive Implementation. In the cross-VM case study, attackers face two additional challenges. *First*, they cannot directly interact with the host kernel and thus cannot explicitly mark pages as mergeable using the `madvise` *syscall*. *Second*, they cannot guarantee that memory allocated within the guest VM is laid out contiguously in the host, which is required for predictable KSM scanning behavior. To address these limitations, we rely on two mechanisms. We leverage QEMU's default behavior to implicitly mark guest memory as mergeable, and we use guest privileges to allocate physically contiguous memory via the Contiguous Memory Allocator (CMA).

For the first limitation, although the attacker cannot invoke `madvise` on the host, QEMU defaults to marking all guest memory as `MADV_MERGEABLE` [52] by default. As a result, pages within the guest VM are automatically considered for deduplication by the host's KSM subsystem. This provides an implicit implementation of the insertion primitive **(P1)**: by carefully crafting page contents, the attacker can influence the host's stable tree without requiring direct kernel interaction. Pages without a duplicate are processed each cycle as normally, enabling for a implicit query primitive **(P2)**. For the second limitation, ensuring predictable scanning order requires that attacker-controlled pages appear contiguous from the host's perspective, where KSM operates. Standard user space allocations inside the guest do not guarantee this property, as memory that is contiguous in the guest's virtual address space may be fragmented in the host. In QEMU/KVM, guest memory is managed as RAMBlocks backed by `mmap` allocation in the host process [53]. As a result, contiguity in the guest physical address space translates to contiguity in the host virtual address space within a given RAMBlock. Leveraging full control over the guest kernel, we allocate memory via the Contiguous Memory Allocator (CMA) [63], reserving a sufficiently large physically contiguous region at boot time. This ensures that attacker-controlled pages are virtually contiguous from the host's perspective. Together, these techniques allow us to recover the required primitives for LeakyTree in the cross-VM setting despite the lack of direct interaction with the host kernel.

6.2.3 Evaluation. We evaluate the cross-VM case study using QEMU KVM virtual machines managed via *libvirt*, following the same hardware and host software setup described in Section 5.1. The attacker is confined to a single guest VM and has no direct interaction with the host or any co-located VMs. We consider multiple scenarios (**S4-S6**) covering guest-to-guest and guest-to-host leakage. Table 4 presents the results. In scenario **(S4)** we start two Ubuntu 24.04 VMs: one corresponding to the victim and the other to the attacker. Without manually constructing victim pages, we recover sensitive system data that was naturally deduplicated by KSM, including `/etc/passwd` entries, as well as found private key material and certificates. In scenario **(S5)**, we start two Windows 11 VMs acting as victims and one Ubuntu 24.04 VM as the attacker. We show that LeakyTree remains effective by leaking Windows-specific data such as `.NET`-related paths and internal telemetry-related logs. Finally, in scenario **(S6)**, we target the host (hypervisor) on which we allocate a page twice and initialize it to an RSA private key. From within the attacker VM, we successfully recover the host-resident RSA key with no interaction. Overall, LeakyTree reliably leaks high-entropy data across VMs and host boundaries, breaking the isolation guarantees in virtualized environments with KSM.

7 Discussion

7.1 Hardware Cache Side-Channel Primitive

LeakyTree relies on an LLC-based cache side-channel primitive to detect whether *ksmd* accessed selected deduplicated pages during stable tree queries. This side-channel primitive is used solely to probe the cache state and is not specific to the LeakyTree attack logic. Unlike prior cache attacks targeting shared memory, LeakyTree does not rely on pre-existing shared memory. Instead, *ksmd* merges two previously separate anonymous pages with identical contents such that both map to the same physical page, thereby creating a shared-memory mapping. This allows LeakyTree to use established shared-memory LLC side-channel techniques such as Flush+Reload [69] or Flush+Flush [18] to observe accesses to

attacker-selected oracle pages without assuming core co-location between the attacker and *ksmd*. Our evaluation focuses on a system with an inclusive LLC, where the victim's memory accesses are directly visible through the shared LLC. However, LeakyTree is not fundamentally restricted to inclusive LLCs. Specifically, prior work has shown that, even on systems with non-inclusive LLCs, core-private cache contents can be made observable through the shared LLC [51, 65, 68], suggesting LeakyTree could be adapted.

7.2 Related Work

Memory-deduplication Attacks. Suzaki et al. [61] were the first to demonstrate a practical attack on deduplicated CoW KSM pages by measuring write-access timing differences. Their attack crafts 4 KB pages identical to victim pages and uses the resulting CoW timing differences to infer whether the victim VM is running specific programs or has cached particular files. Xiao et al. [67] introduced the first cross-VM covert channel based on write-latency variations of deduplicated pages. Similarly, Lindemann et al. [29] presented cross-VM software- and operating system version fingerprinting [44], and Kim et al. [26] a KASLR break on virtual machines. Subsequently, Gruss et al. [16] showed that page deduplication attacks can also be performed from sandboxed JavaScript, enabling application and website fingerprinting. These page-deduplication attacks generally require the attacker to craft a page identical to the victim page in order to trigger deduplication, which makes them most practical for coarse-grained, low-entropy fingerprinting tasks and difficult to apply to high-entropy targets such as cryptographic material. Subsequent work presented the first attacks recovering high-entropy victim data. In particular, Bosman et al. [5] demonstrated that an attacker with strong control over data alignment and padding, can leak high-entropy data byte-by-byte. Without the strong alignment and padding ability, the attack falls back to a birthday-paradox primitive that relies on extensive spraying and probabilistic collisions. Subsequently, Schwarzl et al. [56] demonstrated the first fully remote memory-deduplication attack, where an attacker infers deduplication events by timing HTTP requests. Similarly, their attack requires the strong assumption of attacker-controlled data alignment and padding, which they achieved by inserting attacker-crafted queries into dedicated programs, such as a database system. Palfinger et al. [45] showed that CoW also introduces a side channel for modern file systems, thereby detecting files cross-VM in the cloud. Moreover, Bae et al. [3] proposed a KSM-based primitive that, unlike traditional CoW attacks [35, 61], does not require writing to the page. Their attack exploited KSM's limit on shared pages per node in the stable-tree, meaning that once the limit is exceeded, KSM has to create a linked list of duplicate nodes. This serves as a gadget to determine whether a specific page is already present in the system, i.e., by observing if the list creation was triggered.

Overall, LeakyTree improves over prior memory-deduplication attacks by removing the need for exact page matching, or strong victim-memory alignment and padding control, thereby enabling practical bit-by-bit leakage of high-entropy pages.

Cache Attacks. Cache attacks exploit observable effects caused by CPU caches [54], which temporarily store copies of data from main memory to reduce memory-access latency. A particularly important class of cache attacks observes timing differences caused by whether specific data is cached. A prominent subclass of these timing-based attacks relies on shared memory between attacker and victim. Flush+Reload [69] flushes target cache lines using a dedicated instruction such as `clflush` and detects subsequent victim memory accesses via timing differences. Evict+Reload [19] follows a similar concept but replaces the dedicated flush instruction with a cache-eviction strategy. Flush+Flush [18] exploits the timing variation of the flush instruction itself, depending on whether a victim memory is cached. Furthermore, techniques like Prime+Probe [22, 32, 43] do not require shared memory between the attacker and victim. However, cache attacks not relying on a shared-memory assumption often depend on reverse-engineering microarchitectural hash functions [15, 21, 33, 37, 70]. Although these attacks make weaker assumptions, they are more susceptible to noise and cache replacement policies [17]. Moreover, prior work has demonstrated cache attacks targeting Intel [69], ARM [30] and AMD [65] architectures.

7.3 Real-world Applicability

LeakyTree requires KSM to be enabled, target secrets to be deduplicated and boundable, and page-aligned memory. These conditions arise naturally, for example, cloned VMs such as workers in a build farm may load identical `sshd` server keys, or other secrets. File access can be done via `mmap`, which aligns the secrets to the page, satisfying LeakyTree's requirement. During our evaluation, we found the `sshd` scenario naturally and recovered `sshd`'s key material. Furthermore, KSM remains enabled in production platforms, for example, Proxmox Virtual Environment (Proxmox VE) activates KSM by default with `ksmtuned`. Hence, LeakyTree is not limtied to attacker-constructed secrets or synthetic victim workloads.

7.4 Limitations and Future Work

Broadening the Leakable Pages. This work focuses on the leakage of already deduplicated pages by the Linux KSM mechanism. This implies that the attack targets pages residing in the stable tree, the red-black tree holding deduplicated pages. However, the primitives constructed for the attack do not rely on any special feature or behavior from the stable tree. Used primitives could be constructed to also target the unstable tree, allowing the leakage of pages that have not been deduplicated yet. Due to the corruptible nature of the unstable tree, we expect this variant to be noisier, possibly slowing down overall performance, or requiring additional filtering.

Defenses. In the past, defenses against memory deduplication attacks have been proposed [1, 28, 41]. For instance, VUSion [41] protects against merge-based and unmerge-based attacks, and FakeDD [28] modifies KSM's logic to remove the unstable tree, eliminating the CoW side channel. However, we believe the most effective mitigation for LeakyTree would be to modify KSM's internals so that it hashes page contents rather than comparing each page's contents directly. A similar change was previously discussed within the Linux community [24, 25], but they ultimately chose to use the current approach, leveraging two separate red-black trees instead of a hash table. Note that this decision has been partly influenced by a VMWare patent [64]—now expired— covering an algorithm that uses hash values to identify identical pages in the system.

8 Conclusion

This paper presented LeakyTree, a novel side-channel attack on the Linux KSM [2, 23, 55] memory deduplication mechanism. LeakyTree enables practical bit-by-bit leakage of high-entropy deduplicated pages from an unprivileged context without memory alignment primitives over the victim memory. At its core, the attack massages KSM's red-black trees to trigger deduplication of attacker-controlled pages and subsequently exploits tree-ordering properties to infer layout information through a CPU cache side channel. This procedure allows LeakyTree to leak victim memory bit by bit, exploiting the page comparison of KSM upon tree traversal.

Compared with previous memory deduplication research, this paper presented an alternative attack that does not rely on the *copy-on-write* deduplication behavior. We presented an approach that does not require an identical copy of the page to trigger the leakage, reducing the leakage complexity from $O(2^n)$ to $O(n)$. Furthermore, we showcase LeakyTree's efficacy for cross-process and cross-VM leakage of secret (high-entropy) data, such as ECC and RSA private keys as well as API tokens, at rates of up to 30 bits per second.

We demonstrated the leakage of high-entropy pages from the Linux KSM mechanism in different scenarios, motivating the re-evaluation of KSM in shared computing environments.

Acknowledgments

We thank the anonymous reviewers and the Shepherd for their valuable feedback that improved this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the CREIS project (FFG grant number 926841), and from the Austrian Science Fund (FWF) via the SFB project SPyCoDe (10.55776/F85).

References

[1] Abdullah Albalawi, Vassilios Vassilakis, and Radu Calinescu. 2021. Memory deduplication as a protective factor in virtualized systems. In *ACNS*.
[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Linux Symposium*.
[3] Seungyeon Bae, Taehun Kim, Woomin Lee, and Youngjoo Shin. 2023. Exploiting Memory Page Management in KSM for Remote Memory Deduplication Attack. In *International Conference on Information Security Applications*.
[4] Juan Besa and Yadran Eterovic. 2013. A concurrent red–black tree. *J. Parallel and Distrib. Comput.* 73, 4 (2013), 434–449.
[5] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*.
[6] Russell A Brown. 2025. Comparative Performance of the AVL Tree and Three Variants of the Red-Black Tree. *Software: Practice and Experience* 55, 9 (2025), 1607–1615.
[7] Chengming Zhou. 2024. mm/ksm: don't waste time searching stable tree for fast changing page. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit?id=d58a361b0350128bf5a5cf47773edaedbb6ea838. Accessed: 2026-04-16.
[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.
[9] Peter J. Denning. 2005. The Locality Principle. *Commun. ACM* 48, 7 (2005), 19–24.
[10] Peter J. Denning and Stuart C. Schwartz. 1972. Properties of the Working-Set Model. *Commun. ACM* 15, 3 (1972), 191–198.
[11] Ulrich Drepper. 2007. *What Every Programmer Should Know About Memory*. Technical Report.
[12] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/
[13] Agner Fog. 2022. *The Microarchitecture of Intel, AMD and via CPUs: An optimization guide for assembly programmers and compiler makers*. Technical Report.
[14] Freedesktop. 2026. systemd.exec manpage. https://www.freedesktop.org/software/systemd/man/latest/systemd.exec.html Accessed: 2026-04-29.
[15] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. 2024. Efficient and generic microarchitectural hash-function recovery. In *SP*.
[16] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *ESORICS*.
[17] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*.
[18] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*.
[20] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann.
[21] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES*.
[22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P*.
[23] Izik Eidus, Hugh Dickins. 2009. Kernel Samepage Merging. https://docs.kernel.org/admin-guide/mm/ksm.html. Accessed: 2026-03-31.
[24] Jonathan Corbet. 2009. KSM tries again. https://lwn.net/Articles/330589/. Accessed: 2026-03-31.
[25] Jonathan Corbet. 2023. Process-level kernel samepage merging control. https://lwn.net/Articles/928510/. Accessed: 2026-03-31.
[26] Taehun Kim, Taehyun Kim, and Youngjoo Shin. 2021. Breaking KASLR Using Memory Deduplication in Virtualized Environments. *Electronics* 10, 17 (2021).
[27] Jongmin Lee and Soontae Kim. 2012. Adopting TLB Index-based Tagging to Data Caches for Tag Energy Reduction. In *ISLPED*.
[28] Jens Lindemann. 2024. Faking deduplication to prevent timing side-channel attacks on memory deduplication. In *ACSAC*.
[29] Jens Lindemann and Mathias Fischer. 2018. A Memory-Deduplication Side-Channel Attack to Detect Applications in Co-Resident Virtual Machines. In *Symposium on Applied Computing*.
[30] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security*.
[31] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *ASIACCS*.
[32] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.
[33] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *DIMVA*.
[34] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID*.
[35] Grzegorz Miłós, Derek G Murray, Steven Hand, and Michael A Fetterman. 2009. Satori: Enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference*.
[36] Baker Mohammad. 2014. *Embedded Memory Design for Multi-Core and Systems on Chip*. Vol. 116. Springer.
[37] Bradley Morgan, Gal Horowitz, Sioli O'Connell, Stephan van Schaik, Chitchanok Chuengsatiansup, Daniel Genkin, Olaf Maennel, Paul Montague, Eyal Ronen, and Yuval Yarom. 2025. Slice+ slice baby: Generating last-level cache eviction sets in the blink of an eye. In *SP*. IEEE.
[38] Sourav Mukhopadhyay. 2020. Red-black Trees, Rotations, Insertions, Deletions. https://web.archive.org/web/20250527123427/http://www.facweb.iitkgp.ac.in/~sourav/Lecture-10.pdf
[39] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition*.
[40] Linux Newbies. 2009. Linux 2.6.32 Prominent Features. https://kernelnewbies.org/Linux_2_6_32.
[41] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2017. Secure Page Fusion with VUsion. In *Proceedings of the 26th Symposium on Operating Systems Principles*.
[42] Oracle. 2026. Distribution Kernel Configs. https://oracle.github.io/kconfigs/?config=UTS_RELEASE&config=KSM Accessed: 2026-04-27.
[43] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.
[44] Rodney Owens and Weichao Wang. 2011. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IPCCC*.
[45] Gerald Palfinger, Bernd Prünster, and Dominik Ziegler. 2019. Prying CoW: Inferring Secrets across Virtual Machine Boundaries. In *ICETE*.
[46] Luciano Patrão. 2024. *VMware vSphere essentials: A practical approach to vSphere deployment and management*. Springer Nature.
[47] David A. Patterson and John L. Hennessy. 2012. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. Academic Press.
[48] Phoronix. 2022. Phoronix Test Suite. https://www.phoronix-test-suite.com/

[49] Proxmox. 2026. Open Source in Hospital Operations: University Hospital Essen relies on Proxmox VE. https://www.proxmox.com/en/about/about-us/stories/story/ume Accessed: 2026-08-06.
[50] Proxmox. 2026. Proxmox Virtual Environment 9.2 with Dynamic Load Balancer released. https://www.proxmox.com/en/about/company-details/press-releases/proxmox-virtual-environment-9-2 Accessed: 2026-08-06.
[51] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2022. Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies. In *USENIX Security*.
[52] QEMU. 2026. QEMU Source Code - Merging. https://github.com/qemu/qemu/blob/45cad8beca3ad75ebe63a36fdc60c98395fd2df1/system/physmem.c#L2015 Accessed: 2026-04-14.
[53] QEMU. 2026. QEMU Source Code - mmap. https://github.com/qemu/qemu/blob/759c456b1d22fe4083c8b384da27d3f56fd53f82/util/oslib-posix.c#L215 Accessed: 2026-04-30.
[54] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. 2025. A Systematic Evaluation of Novel and Existing Cache Side Channels. In *NDSS*.
[55] Red Hat. 2017. *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*.
[56] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. 2022. Remote Page Deduplication Attacks. In *NDSS*.
[57] Furquan Shaikh, Fangzhou Yao, Indranil Gupta, and Roy H Campbell. 2014. Vmdedup: Memory de-duplication in hypervisor. In *2014 IEEE International Conference on Cloud Engineering*.
[58] Stefan Roesch. 2023. Kernel Samepage Merging (KSM) at Meta and Future Improvements to KSM. https://lpc.events/event/17/contributions/1625/attachments/1320/2649/KSM.pdf Accessed: 2026-08-06.
[59] Stefan Roesch. 2023. mm/ksm: add "smart" page scanning mode. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit?id=5e924ff54d088828794d9f1a4d5bf17808f7270e. Accessed: 2026-04-14.
[60] David Suggs, Mahesh Subramony, and Dan Bouvier. 2020. The AMD "Zen 2" Processor. *IEEE Micro* 40 (2020), 45–52.
[61] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Memory Deduplication as a Threat to the Guest OS. In *EuroSys*.
[62] Tensorflow. 2026. Tensorflow - OpData4Bit Struct. https://github.com/tensorflow/tensorflow/blob/e688ea02af5ba6bf08da1e27538a6802e8baa026/tensorflow/lite/kernels/internal/optimized/fully_connected_4bit.h#L78 Accessed: 2026-04-29.
[63] Toradex. 2026. Contiguous Memory Allocator - CMA (Linux). https://developer.toradex.com/software/linux-resources/linux-features/contiguous-memory-allocator-cma-linux/ Accessed: 2026-04-30.
[64] Carl A Waldspurger. 2004. Content-based, transparent sharing of memory units. US Patent 6,789,156.
[65] Han Wang, Ming Tang, Quancheng Wang, Ke Xu, and Yinqian Zhang. 2025. ZenLeak: Practical Last-Level Cache Side-Channel Attacks on AMD Zen Processors. In *DAC*.
[66] Jun Xiao, Andy D Pimentel, and Xu Liu. 2019. CPpf: A prefetch aware LLC partitioning approach. In *International Conference on Parallel Processing*.
[67] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2013. Security implications of memory deduplication in a virtualized environment. In *International Conference on Dependable Systems and Networks (DSN)*.
[68] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*.
[69] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.
[70] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. *Cryptology ePrint Archive, Report 2015/905* (2015).
[71] Pavel Yosifovich, Alex Ionescu, Mark E Russinovich, and David A Solomon. 2017. *Windows Internals: System architecture, processes, threads, memory management, and more, Part 1*. Microsoft Press.
[72] Tianhao Zheng, Haishan Zhu, and Mattan Erez. 2018. SIPT: Speculatively Indexed, Physically Tagged Caches. In *HPCA*.

A Ethical Considerations

We disclosed our findings to the Linux Kernel Security Team on May 8, 2026, including a copy of this article. On May 9, 2026, we received an initial response that interpreted our report as a proposal for a KSM defense rather than as the disclosure of an attack. We clarified the nature of our initial report on May 15, 2026. However, as of August 2026, we had not received a further response.

We reported LeakyTree to the Proxmox Security Team on July 3, 2026, due to their default use of KSM in Proxmox VE. Proxmox responded on July 4, 2026, indicating that it would reconsider enabling KSM by default in future Proxmox VE releases.

Finally, we contacted Andrea Arcangeli, who has prior experience developing downstream mitigations for KSM-related side-channel attacks. Arcangeli provided us with a KSM kernel patch and asked us to assess its interaction with LeakyTree. The patch introduces a new KSM operating mode that restricts deduplication to zero-filled pages only. We concluded the new operating mode mitigates LeakyTree and previous KSM attacks, as KSM cannot process secret or private pages anymore. All other default and current operation modes remain vulnerable.

B Open Science

We are committed to making all artifacts required to reproduce the experimental results of this work available upon acceptance. Additionally, we include all the artifacts related to our submission in https://anonymous.4open.science/r/342950843895273495. Specifically, our artifact package includes a visual demonstration, in the form of a video, of the bit-by-bit leakage of a high-entropy RSA key, an API key, and OS internal memory on different scenarios. Additionally, we include the synthetic evaluation and proofs of concept for the cross-process and cross-VM leakage experiments.