KernelSnitch: Side-Channel Attacks on Kernel Data Structures

Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, Stefan Mangard
Graz University of Technology
{lukas.maar, jonas.juffinger, daniel.gruss, stefan.mangard}@tugraz.at,
thomas.steinbauer@student.tugraz.at

***Abstract*—The sharing of hardware elements, such as caches, is known to introduce microarchitectural side-channel leakage. One approach to eliminate this leakage is to not share hardware elements across security domains. However, even under the assumption of leakage-free hardware, it is unclear whether other critical system components, like the operating system, introduce software-caused side-channel leakage.**

In this paper, we present a novel generic software side-channel attack, KernelSnitch, targeting kernel data structures such as hash tables and trees. These structures are commonly used to store both kernel and user information, e.g., metadata for user-space locks. KernelSnitch exploits that these data structures are variable in size, ranging from an empty state to a theoretically arbitrary amount of elements. Accessing these structures requires a variable amount of time depending on the number of elements, i.e., the occupancy level. This variance constitutes a timing side channel, observable from user space by an unprivileged, isolated attacker. While the timing differences are very low compared to the syscall runtime, we demonstrate and evaluate methods to amplify these timing differences reliably. In three case studies, we show that KernelSnitch allows unprivileged and isolated attackers to leak sensitive information from the kernel and activities in other processes. First, we demonstrate covert channels with transmission rates up to 580 kbit/s. Second, we perform a kernel heap pointer leak in less than 65 s by exploiting the specific indexing that Linux is using in hash tables. Third, we demonstrate a website fingerprinting attack, achieving an F_1 score of more than 89 %, showing that activity in other user programs can be observed using KernelSnitch. Finally, we discuss mitigations for our hardware-agnostic attacks.

I. Introduction

The performance of modern computer systems crucially depends on the efficiency of hardware and software. On the hardware level, numerous optimizations, such as caching, contribute significantly to hardware performance. Instead of always taking the slow path to the main memory, caches offer a shortcut by providing a local copy of the data. Inherently, this introduces a timing difference. Side-channel attacks exploit specifically such timing differences [37], allowing an attacker to infer secret information and, e.g., covertly transmit data [45], break Address Space Layout Randomization (ASLR) [28], leak cryptographic keys [68], or spy on user input [24]. Besides caches, numerous other optimizations have been discovered to leak information through timing, e.g., contention of execution ports [2] or execution unit schedulers [18], [19]. An intuitive approach to eliminate all microarchitectural side-channel attacks, commonly considered a last resort, is to not share hardware elements across security domains anymore.

However, even under the assumption of leakage-free hardware, the software can also introduce timing side channels for the same reason: improving efficiency. The exploitation of timing differences has been studied on algorithms designed for security contexts, e.g., in weak cryptographic implementations [48], as well as on algorithms that were not primarily designed for security contexts but used in them [6], [21], [33], [53], [55]. Timing differences can also be introduced generically at the system level: For instance, the software also has different types of caches that leak information [16], [22], [63], in-kernel allocators or synchronization primitives [41], [43], [56], interrupts [15], [59] or variation in the instruction or memory access sequence [62], [68]. Although the concept of constant-time code [37] is well-understood, its general-purpose application is impractical [54]. In particular, leakage introduced on the operating system level [22], [41], [43], [56], [59] is critical, as this leakage is not visible in the user-level source code and is not mitigated by constant-time code in the user application. Hence, it is unclear whether, despite leakage-free hardware and hardened security-critical algorithms, other critical system components, like the operating system, introduce generic software-observable side-channel leakage.

Recent research highlights leakage from system components: Gruss et al. [22] showed that the operating system page cache can be exploited like hardware caches. Patel et al. [49] demonstrated a performance-degrading attack exploiting intra-kernel resource contention. Lee et al. [41] and Maar et al. [43] presented timing side channels in the Linux slab allocator to infer when a new slab is created. Shen et al. [56] presented a covert channel based on mutual exclusion primitives. These works motivate further research on side channels introduced by the operating system architecture to understand whether more security- and privacy-critical attacks are possible.

In this paper, we present a novel generic side-channel attack, KernelSnitch, that targets data container structures inside the kernel. We present attacks on four types of data structures in the kernel: fixed-size hash tables, dynamically resizable hash

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240223
www.ndss-symposium.org

tables, radix trees, and red-black trees, all commonly used data structures in the Linux kernel. KernelSnitch exploits that these data structures have a variable size, ranging from an empty state to a theoretically arbitrary amount of elements. Accessing these data structures requires operations that depend on the amount of elements in the data structure, i.e., the occupancy level. This variance in the operations, which depends on the occupancy level, constitutes a timing side channel an unprivileged user can observe to leak information about other processes or privileged information from the kernel.

One challenge for KernelSnitch is amplification: Timing differences between occupancy levels of the victim workload can be very low, e.g., 8 additional instructions. Compared to the syscall overhead, distinguishing the timing of these extra instructions is challenging. We demonstrate two approaches to address this challenge by increasing the timing difference and, thereby, amplifying the leakage: First, we degrade the performance of the system with memory thrashing. Second, we manipulate the data structures with additional elements. We evaluate the KernelSnitch's leakage to distinguish occupancy levels with and without amplification and with different noise floors. We mainly evaluate on an Intel i7-1260P, with other processors and another architecture yielding similar results for the same evaluation programs, demonstrating hardware independence. We show that we can distinguish the occupancy level with an accuracy of better than 99.9 % on an idle system and better than 98.5 % on a noisy system up to $3/4$ full load.

We evaluate KernelSnitch in three case studies, showing that KernelSnitch can leak sensitive information from the kernel and activity in other user processes. Our evaluation works from unprivileged, isolated processes on the same system, e.g., within a sandbox. In our first case study, we measure the capacity of the side channel in a covert-channel scenario, achieving a true capacity of 580 kbit/s at an error rate of 2.8 %. In our second case study, we exploit the specific indexing of Linux for hash tables. To randomize the indices, the kernel generates the index by combining user-controlled information and kernel-controlled secret information. Using KernelSnitch, an attacker can infer the secret information that the kernel uses as input. This allows us to leak the locations of targeted kernel objects (i.e., `mm_struct` and `msg_msg`) in less than 65 s. We refer to this as a kernel heap pointer leak. While prior side-channel research [9], [23], [28], [29], [36] leaked Kernel ASLR (KASLR), e.g., the start of the text section or physical mapping, we are the first to perform a kernel heap pointer leak using a side channel[1]. In our third case study, we show the activity leakage of other user programs, e.g., Firefox. In particular, we perform a website fingerprinting attack on the Ahrefs Top 100 [1], achieving an F_1 score of 89.5 %.

Finally, we discuss defenses against the hardware-agnostic KernelSnitch attack. We identify challenges for efficient mitigation, such as the theoretically unbounded worst-case execution time and eliminating constant time as a viable solution.

Contributions. The main contributions of our work are:

1) **Identification of Critical Timing Side Channels in the Kernel:** We analyze the security properties of kernel data container structures, presenting a new side channel, KernelSnitch, exploiting the kernel's internal architecture to leak sensitive information to unprivileged users.
2) **Leakage Amplification and Evaluation:** We demonstrate information leakage amplification approaches and evaluate the leakage for multiple data container structures.
3) **Side-Channel Attacks:** We demonstrate three attack case studies: A covert channel with up to 580 kbit/s, a kernel heap pointer leak in less than 65 s, and a website fingerprinting attack with an F_1 score of more than 89 %.

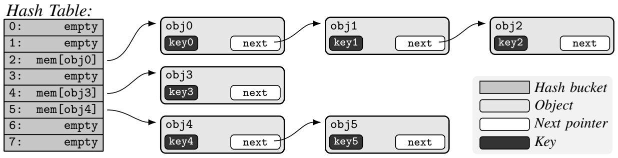


Fig. 1: Visual representation of a Linux kernel's hash table.

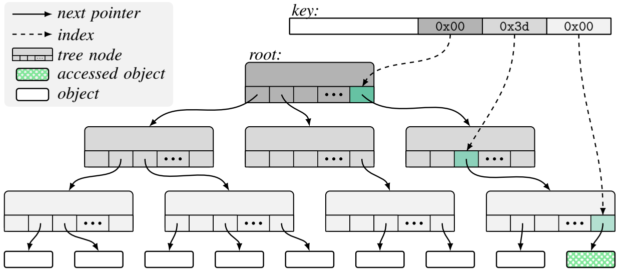


Fig. 2: Visual representation of a three-level radix tree, with *key* referencing this tree to index the hatched object.

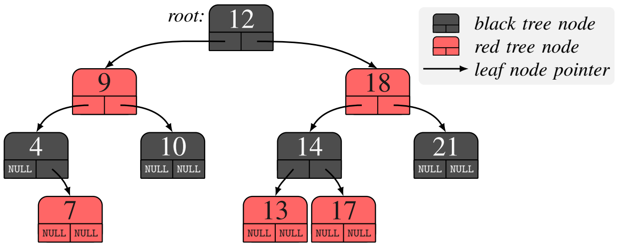


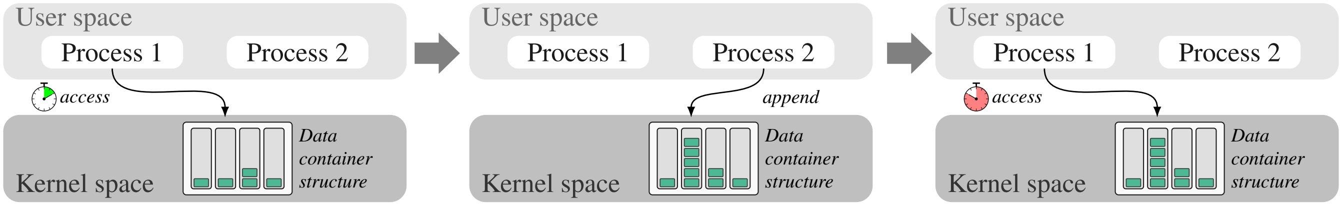
Fig. 3: Visual representation of a red-black tree, which stores items with an increasing order.

Disclosure. We have disclosed our KernelSnitch attack to the Linux kernel security team.

Opensource. We provide open-source implementations of our timing side-channel attack, which leaks the occupancy level of data structures discussed in the paper. The code is available at https://doi.org/10.5281/zenodo.14249716.

Outline. Section II provides background information. Section III presents an overview of KernelSnitch. Section IV presents a root cause analysis. Section V details how we amplify the leakage. Section VI presents three attack case

[1] As Linus Torvalds and Kees Cook noted during our disclosure, KASLR is broken against local attackers, but leaking kernel heap pointers is not.



(a) Process 1 accesses (fast) a kernel *data container structure* via the syscall interface.

(b) Process 2 appends data to the *data container structure* via the syscall interface.

(c) Processes 1 re-accesses the *data container structure* which is now slower.

Fig. 4: High-level of exploiting a kernel data container structure for a side channel.

studies. Section VII discusses related work, and Section VIII discusses mitigations. Section IX concludes our work.

II. Background

In this section, we provide background on Linux kernel data container structures, including hash tables and trees.

A. Container Structures in the Linux Kernel

The C language used in Linux has no container structures like C++ (e.g., `vector`). Hence, several containers are explicitly written for Linux and optimized for their use in the kernel.

Double-Linked Lists. Linux provides a generic double-linked list, i.e., `list_head`, which is extensively used throughout the kernel. The `list_head` struct consists of `next` and `prev` pointers and is commonly used to organize lists, e.g., for the `mm_struct` or `task_struct` lists. Insertion and deletion work by re-linking pointers correspondingly, with a constant runtime. In the worst case, an object lookup requires iterating through the entire list, with a runtime linear in length.

Hash Tables. In the Linux kernel, hash tables use a hash function to compute the index in an array of buckets. In each bucket, objects are typically stored in a linked list (see Figure 1). To access the object with key `key1`, the kernel computes its hash value to determine the hash bucket, resulting in bucket 2. It then iterates through the linked list within bucket 2, comparing the keys of stored objects until it finds a match with the key from `obj1`. While all hash tables in the Linux kernel use this approach to access objects by their keys, there are variations between *fixed-size* and *dynamically resizable* hash tables: Fixed-size hash tables have an array with a predetermined number of buckets, e.g., the `plist_head` object for the buckets, using `list_head` internally. Dynamically resizable hash tables, e.g., `rhashtable`, adjust their bucket sizes based on the occupancy level of the buckets.

Trees. Linux supports multiple trees, including two widely-used ones, i.e., *radix tree* [11] and *red-black tree* [12].

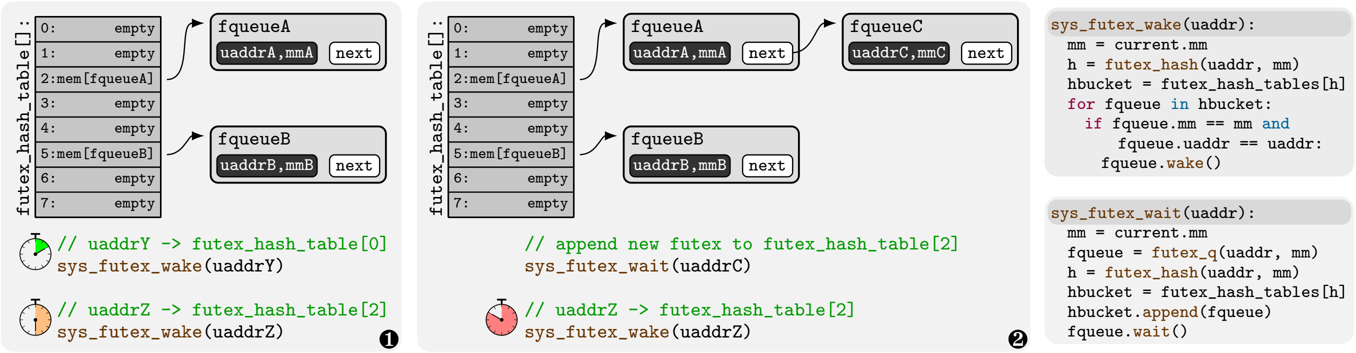
The *radix tree* associates a pointer value with an integer key, offering efficient memory usage and quick lookups [11]. Figure 2 illustrates a three-level radix tree example. In general, each tree node contains multiple slots (typically 6 bit), each pointing to `NULL`, a child tree node, or a stored object. These slots are indexed by parts of the integer key. During a key lookup, the kernel uses the most significant bit block to find the corresponding slot in the root node, followed by subsequent bit blocks for lower-level tree nodes. In the example of Figure 2, with the key having three bit blocks, a single tree lookup uses the most significant bit block (i.e., `0x00`) for the third level (i.e., root node), the middle bit block (i.e., `0x3d`) for the second level, and the least significant bit block (i.e., `0x00`) for the first level, referencing the accessed object pointer.

The *red-black tree* is a variant of a semi-balanced binary tree. Each tree node contains a value and up to two references to child nodes. All child nodes on the left branch are smaller, while all child nodes on the right branch are greater than the current tree node. Hence, nodes are ordered from lowest to highest (see Figure 3 with values 4 to 21). Each tree node is colored either red or black, with the root node being black. Insertion and deletion operations involve re-coloring tree nodes to ensure an approximation of the tree balance [12]. Consequently, due to this re-balancing, the red-black tree has a logarithmic worst-case execution time for lookups [12].

III. High-level Overview

This section provides an overview of KernelSnitch, a timing attack on kernel container structures to leak sensitive information. At its core, KernelSnitch observes the occupancy level of data container structures in the kernel and deduces sensitive information from it through the following process, as shown in Figure 4: KernelSnitch measures the timing of accesses to data container structures shared between processes (see Figure 4a), by timing the syscall that accesses the kernel structure. Depending on the occupancy level, the timing of this access syscall varies. KernelSnitch then deduces the occupancy level from the obtained timing. The measured timing of accessing the data structure by process 1 indicates a fast operation, letting KernelSnitch infer a low occupancy (see Figure 4a). The occupancy level increases when additional data is appended to the structure (see Figure 4b). For example, appending data to a list increases its size, requiring additional iterations to access all elements. When KernelSnitch re-accesses the structure via a syscall, e.g., iterates through the entire list, it observes a slower syscall timing than on the initial access (see Figure 4c).

We perform three case study attacks by setting and observing the data structure occupancy level. First, we perform a *covert channel* attack by controlling two processes, such as processes 1 and 2 from Figure 4. One process sets the



```
// uaddrY -> futex_hash_table[0]
sys_futex_wake(uaddrY)

// uaddrZ -> futex_hash_table[2]
sys_futex_wake(uaddrZ)
```

```
// append new futex to futex_hash_table[2]
sys_futex_wait(uaddrC)

// uaddrZ -> futex_hash_table[2]
sys_futex_wake(uaddrZ)
```

```
sys_futex_wake(uaddr):
  mm = current.mm
  h = futex_hash(uaddr, mm)
  hbucket = futex_hash_tables[h]
  for fqueue in hbucket:
    if fqueue.mm == mm and
       fqueue.uaddr == uaddr:
      fqueue.wake()
```

```
sys_futex_wait(uaddr):
  mm = current.mm
  fqueue = futex_q(uaddr, mm)
  h = futex_hash(uaddr, mm)
  hbucket = futex_hash_tables[h]
  hbucket.append(fqueue)
  fqueue.wait()
```

Fig. 5: Representation of `futex_hash_table` as a fixed-size array of hash bucket, each containing a linked list of futex queues (i.e., `fqueue`). Initially ❶, buckets 2/5 store queues `fqueueA/B`. Accessing bucket 0 with `uaddrY` is fast, while accessing bucket 2 with `uaddrZ` is moderate. After adding `fqueueC` ❷ to bucket 2, accessing this bucket with `uaddrZ` becomes slow.

occupancy level of a data structure to either low or high, while the other process measures its occupancy level. Second, using hash tables as data structures that use kernel heap addresses as part of the keys, KernelSnitch deduces hash collisions from setting and observing the occupancy level of hash buckets. This allows us to reconstruct kernel heap addresses from user space, i.e., *leaking kernel heap pointers*. Third, if an attacker controls process 1 while process 2 is a victim, KernelSnitch can deduce the activity of the victim, e.g., Firefox, from the occupancy level of data container structures. This activity deduction allows us to perform a *website fingerprinting* attack.

Several technical challenges have to be addressed to perform these three attacks. The following briefly describes these challenges, while subsequent sections discuss our solutions.

Occupancy Level Leakage of Data Structures. We measure the timing of syscalls that access kernel data container structures. While this timing depends on the occupancy level of these structures, we need to study this dependency in close detail, taking the numerous operations performed as part of a syscall into account. In Section IV, we address this and successfully determine the occupancy level for various data structures via timing measurements. We demonstrate the side channel in particular on fixed-size hash tables, dynamically resizable hash tables, radix trees, and red-black trees.

Amplification of the Information Leakage. Distinguishing lower and higher-level occupancy from user space is challenging. In some cases, the difference is only a few additional executed instructions, which we require to distinguish from user space. To overcome this, we demonstrate information leakage amplification methods in Section V. These methods are classified as structure-agnostic and hardware-agnostic. We demonstrate that with these methods, we can reliably distinguish the occupancy level of data structures in idle and noisy systems. We also demonstrate that occupancy leakage and amplification are independent of the structures' allocation addresses and are consistent between reboots.

Attack Specifics. We perform three case studies of KernelSnitch attacks: covert channel, kernel heap pointer leak, and website fingerprinting, each of which has its sub-challenges. For instance, the covert channel relies on identifying a channel of the structure for communication. Consider a hash table, this involves identifying a shared bucket known to both processes. In Section VI, we detail solutions for overcoming these sub-challenges and show how we leverage occupancy-level leakage to execute each attack. We demonstrate a covert channel with up to 580 kbit/s, a kernel heap pointer leak in less than 65 s, and website fingerprinting with an F_1 of score more than 89 %.

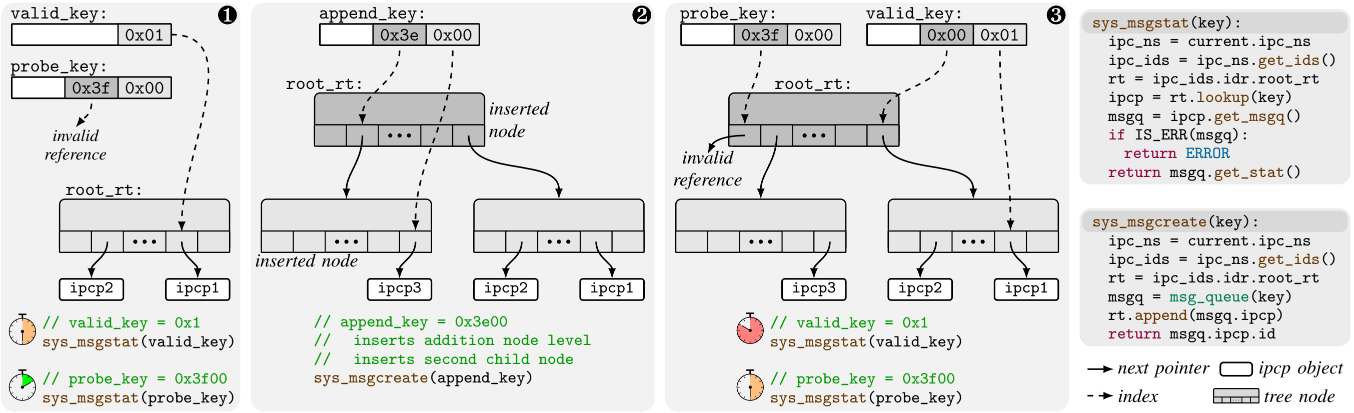
IV. Root Cause Analysis

We analyze the security properties of data container structures, revealing a novel timing side channel, KernelSnitch, in the kernel that leaks sensitive information to unprivileged and isolated users. We showcase leakage from fixed-size hash tables (i.e., `hlist_head[]` and `plist_head[]`), a dynamically resizable hash table (i.e., `rhashtable`), a radix tree (i.e., `radix_tree_root`), and a red-black tree (i.e., `rb_root`). KernelSnitch deduces the occupancy level by measuring the timing of syscalls that access these structures from user space.

A. Leaking Occupancy Levels of Hash Tables

Hash tables in Linux consist of an array of key-indexed buckets, each containing a linked list of objects (see Section II-A). When performing a hash table lookup, the kernel computes the bucket index by applying a hash function to the key. It then iterates through the linked list to find the corresponding object. As this iteration through the linked list takes time, it leaks the occupancy level of the iterated hash bucket through the required access timing. While this approach is generically applicable, we describe the occupancy leakage using the `futex_hash_table` (or `__futex_data.queues`) as an illustrative example (see Figure 5).

Futex Hash Table. Linux supports futexes [34] as fast user-space locking mechanisms, which mainly operates in user space and invokes syscalls for sleeping and waking otherwise. We exploit `sys_futex_wait/wake` as primitives to probe and alter the occupancy level of hash buckets within `futex_hash_table`. The wait operation (i.e., `futex` syscall with `FUTEX_WAIT_PRIVATE`), or `sys_futex_wait` in Figure 5, is a syscall, during which a local futex queue object (i.e., `futex_q`)



```
// valid_key = 0x1
sys_msgstat(valid_key)

// probe_key = 0x3f00
sys_msgstat(probe_key)
```

```
// append_key = 0x3e00
//   inserts addition node level
//   inserts second child node
sys_msgcreate(append_key)
```

```
// valid_key = 0x1
sys_msgstat(valid_key)

// probe_key = 0x3f00
sys_msgstat(probe_key)
```

```
sys_msgstat(key):
  ipc_ns = current.ipc_ns
  ipc_ids = ipc_ns.get_ids()
  rt = ipc_ids.idr.root_rt
  ipcp = rt.lookup(key)
  msgq = ipcp.get_msgq()
  if IS_ERR(msgq):
    return ERROR
  return msgq.get_stat()
```

```
sys_msgcreate(key):
  ipc_ns = current.ipc_ns
  ipc_ids = ipc_ns.get_ids()
  rt = ipc_ids.idr.root_rt
  msgq = msg_queue(key)
  rt.append(msgq.ipcp)
  return msgq.ipcp.id
```

Fig. 6: The representation of `ipc_ids.ipcs_idr.root_rt` initially consists of a one-level radix tree ❶. Probing `valid_key` with `sys_msgstat` results in a moderate access time due to a single node lookup, while probing `probe_key` results in a fast response due to no lookup. When adding `append_key` with `sys_msgcreate` ❷, the kernel inserts a second node level. Now, when probing with `probe_key` and `valid_key` ❸, an additional lookup is required, leading in increased access times.

is created and placed in the futex hash table (i.e., `futex_hash_table`). Storing in the hash table involves computing the hash using `futex_hash` with the current `mm_struct`'s kernel address and user-space address `uaddr`, which holds the user-space address of its futex structure.

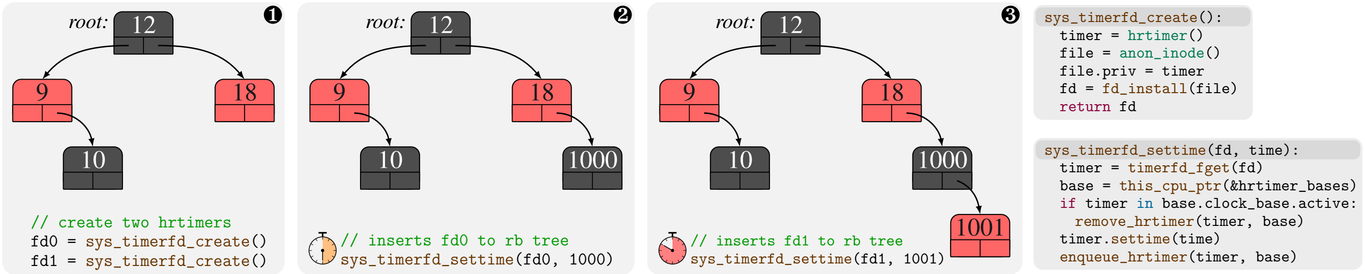
We use the wait operation to increase the occupancy level of specific hash buckets in the futex hash table. In particular, we create a thread that subsequently executes `sys_futex_wait`, increasing the occupancy level. There are two ways to fill hash buckets. First, using the same `uaddr` within the same process (same `mm_struct`), KernelSnitch increases the occupancy of the same hash bucket. Second, using a different `uaddr` or a different process, KernelSnitch increases the occupancy of (most likely) different hash buckets. We use the wake operation with a mismatched identifier to probe the occupancy level of hash buckets. This syscall, simplified as `sys_futex_wake` in Figure 5, first computes the hash of the current `mm_struct` and the input `uaddr`. It then iterates through all futex queues linked to the corresponding hash bucket. Since we provide an `uaddr` that does not match any futex queue, the kernel iterates through the entire list, leaking the occupancy level of the hash bucket with the hash index `futex_hash(uaddr, mm)` through the wake syscall's execution time. To remove a futex queue from the `futex_hash_table`, we perform the wake operation on the sleeping thread.

The manipulation and observation of occupancy levels are detailed in Figure 5. The initial `futex_hash_table` ❶ contains `fqueueA/B` for hash buckets 2 and 5, respectively. We perform `sys_futex_wake` with a mismatched address (i.e., `uaddrY`), which, in combination with the current `mm_struct`, maps to bucket 0. Thus, KernelSnitch observes a fast time measurement corresponding to a low occupancy level. Similarly, repeating this process with the mismatched `uaddrZ` address associated with bucket 2 reveals a moderate occupancy level, indicating the presence of a single queue element. Furthermore, the execution of `sys_futex_wait` ❷ with the address `uaddrC` associated with bucket 2 allows KernelSnitch to increase the occupancy level of bucket 2. Consequently, performing `sys_futex_wake` with the mismatched address `uaddrZ`, corresponding to bucket 2, will have an even slower access time, leaking an increase in its occupancy level.

Vulnerable Hash Tables. We extend our analysis, showing that KernelSnitch is a generic attack, also leaking from other hash table implementations. These include `hlist_head[]` (i.e., `posix_timers_hashtable`) and `rhashtable` (i.e., `ipc_ids.key_ht`), which consist of fixed-size hash buckets and dynamically resizable hash tables, respectively.

The `posix_timers_hashtable` serves as a hash table to store POSIX interval timers, i.e., `k_itimer`. Linux has various timer-related syscalls, including `sys_timer_create` and `sys_clock_gettime` (see Listings 3 and 4 in Appendix A), designed for creating timers and retrieving timer information. We demonstrate that these syscalls can be exploited to alter and probe the occupancy level of hash buckets within the timer hash table, which is similar to the futex hash table. In this implementation, the hash value is computed using `timer_hash` with the current `signal_struct`'s kernel address and the unique timer identifier `id`. Based on this hash value, these syscalls access the corresponding hash bucket, adding a new timer or iterating through existing timers to retrieve information about the matching timer. To remove a timer from the hash table, a close syscall can be performed.

The `ipc_ids.key_ht` is a dynamically resizable hash table which stores `kern_ipc_perm` (short `ipcp`) objects that contain metadata used for user-space Inter-Process Communication (IPC). Specifically, `ipcp` objects are inherited by objects such as the `msg_queue` struct, which are intended for `msg` communication. The same applies to other IPC mechanisms, such as `shm` and `sem`. Linux provides syscalls for interacting with these parent objects, e.g., `sys_msgcreate/msgget` for the `msg` IPC mechanism (see Listings 5 and 6). KernelSnitch similarly exploits these syscalls as with the previous instances.



```
// create two hrtimers
fd0 = sys_timerfd_create()
fd1 = sys_timerfd_create()
```

```
// inserts fd0 to rb tree
sys_timerfd_settime(fd0, 1000)
```

```
// inserts fd1 to rb tree
sys_timerfd_settime(fd1, 1001)
```

```
sys_timerfd_create():
  timer = hrtimer()
  file = anon_inode()
  file.priv = timer
  fd = fd_install(file)
  return fd
```

```
sys_timerfd_settime(fd, time):
  timer = timerfd_fget(fd)
  base = this_cpu_ptr(&hrtimer_bases)
  if timer in base.clock_base.active:
    remove_hrtimer(timer, base)
  timer.settime(time)
  enqueue_hrtimer(timer, base)
```

Fig. 7: `hrtimer_bases.clock_base.active` includes four timers arranged by time values ❶, ranging from 9 to 18. When the new timer identified by `fd0` is enqueued to the tree ❷, the insertion process takes moderate time since it involves accessing two tree nodes. However, when the `fd1` timer is enqueued ❸, insertion time increases as it now requires accessing three nodes.

B. Leaking Occupancy Levels of Trees

This section demonstrates that trees are also vulnerable.

Radix Tree. This tree associates a pointer value with an integer key [11]. Each tree node contains multiple slots (typically 6 bit in size) pointing to `NULL`, child nodes, or stored objects. These slots are indexed by bit blocks of the key. During a key lookup, the kernel uses the most significant index to locate the corresponding slot in the root node, followed by subsequent indexes for lower-level nodes. The timing of a lookup depends on the tree's level, allowing for the leakage of its internal occupancy level through timing measurements.

One instance is the `ipc_ids.ipcs_idr.root_rt` radix tree, storing `ipcp` objects identified by unique key identifiers. Linux provides various syscalls to interact with this tree, such as `sys_msgcreate/msgstat`, used for appending `ipcp` objects to the tree or obtaining information from its parent object `msg_queue`. Similar to our approach with hash tables, we exploit these syscalls to alter and leak the occupancy level of the radix tree. The scenario depicted in Figure 6 illustrates how KernelSnitch exploits the `ipc_ids.ipcs_idr.root_rt` to leak its occupancy level. The radix tree initially consists of one level ❶, with the tree node having 64 slots. In the lookup of `valid_key` within `sys_msgstat`, the kernel uses the least significant 6 bits to access the `0x1` slot of the root node, retrieving the `ipcp1` object. Since only one tree node is accessed during this lookup, the syscall runtime is moderate. By performing `sys_msgstat(probe_key)`, the kernel does not access any tree node as the second 6-bit index `0x3f` requires a second tree level which is not present. Thus, the access time is fast as no tree node is accessed. When adding `append_key` with `sys_msgcreate` ❷, the kernel inserts a second level and another first-level node, replacing the root node with the newly inserted second level. When re-accessing the radix tree with `valid_key` and `probe_key` using `sys_msgstat` ❸, their lookup and, consequently, execution time change. For `valid_key`, the kernel first fetches the `0x00` slot from the new root node, followed by fetching the `0x01` slot. Since a lookup for `valid_key` now accesses two tree nodes, the runtime is increased. For `probe_key`, the kernel initially fetches the `0x3f` slot similarly the previous lookup. However, since this slot contains no valid next child node, the lookup yields an invalid reference. The lookup time, now accessing one node, increases compared to no node access.

Red-Black Tree. Linux uses red-black trees as key-sorted data structures [12], e.g., `hrtimer_bases.clock_base.active` which manages active high-resolution timers, sorting timer events by how close they are to their firing time.

Linux supports syscalls to interact with high-resolution timers, two of which are: `sys_timerfd_create` to create a timer and `sys_timerfd_settime` to activate it. Upon activation, the timer is enqueued to the `hrtimer_bases.clock_base.active` red-black tree. Considering the scenario in Figure 7, initially, the tree ❶ includes 4 timers sorted by time values ranging from 9 to 18. By calling `sys_timerfd_create`, the kernel creates two timers identified as `fd0/1`, which are not yet enqueued to the tree. Upon activating the timer identified with `fd0` using `sys_timerfd_settime` ❷, the corresponding `hrtimer` is inserted at the tail of the tree since its value is 1000 larger than 18. This insertion requires two tree node accesses, resulting in a moderate enqueue time and indicating a moderate occupancy level. Activating `fd1` using `sys_timerfd_settime` ❸ involves three node accesses, resulting in higher execution time and suggesting a higher occupancy level compared to the previous enqueuing. Although enqueuing `fd1` triggers a tree rebalancing, this does not impact the structure's exploitability (see Section V-B).

V. Amplification and Evaluation

KernelSnitch distinguishes lower and higher-level occupancy from user space. For instance, for the POSIX timer hash table, KernelSnitch needs to distinguish as few as 8 extra instructions executed based on time measurements from user space. We demonstrate in Section V-A how to amplify the information leakage in KernelSnitch attacks to a degree where these few extra instructions can be distinguished. We then evaluate the leakage without and with our amplification methods in Section V-B, as well as with different noise floors.

A. Leakage Amplification

The amplification methods make the difference in occupancy of data container structures between lower and upper levels distinguishable from user space. We categorize these methods into *structure-agnostic* and *hardware-agnostic*.

Structure-Agnostic Amplification. Our structure-agnostic amplification mechanism extends the execution time of the instructions executed for each additional element. For example, the hash table `posix_timers_hashtable` iterates over the linked list of a hash bucket (see Listing 4 in Appendix A). For each iteration, 8 additional instructions are executed (see Listings 1 and 2). They are two memory loads, three compares, and three jumps. Flushing targets of memory loads is a known technique [3], [24], [51], exploiting that cache hits are faster than misses [68]. As we cannot use Flush+Reload due to the lack of shared memory, we use cache eviction from user space. Since we do not assume to know the allocation addresses of the container structures nor any low-level information about the hardware caches, our eviction set consists of an array equal or larger to the Last Level Cache (LLC). Eviction is performed by accessing the entire array, essentially thrashing the LLC. This approach ensures that the targeted memory loads cause cache misses, thereby increasing the timing difference between low and high occupancy. This amplification is agnostic to specific container structures and can be applied to any structures.

Hardware-Agnostic Amplification. We can also modify the state of the data container structure to increase the access time to a particular hash bucket by appending additional elements. For example, for the `futex_hash_table`, instead of appending one futex queue object to a specific hash bucket, we append multiple futex queues. We do this via the `sys_futex_wait` syscall, using the same user-space address `uaddr` within the same process, and, therefore, the same `mm_struct`. Since both the `uaddr` and `mm` are identical, their hash value `h0 = futex_hash(uaddr,mm)` also matches. Consequently, these futex queues are appended to the same hash bucket's linked list. Next, we invoke the probe syscall `sys_futex_wake` with a different user-space futex address `uaddr'` but the same `mm`. The syscall iterates through the futex queues within the hash bucket matching hash `h1 = futex_hash(uaddr',mm)`. If `h0` and `h1` match, the measured time becomes a function of the futex queues appended in the first stage. With more objects in the linked list of the hash bucket, the lookup time increases, significantly improving the detection of hash collisions. This generic approach works for all data containers that can contain linked data structures, i.e., other hash tables and the red-black tree.

For the radix tree, the amplification process works differently. Considering `ipc_ids.ipcs_idr.root_rt`, we aim to maximize the timing difference of the `sys_msgstat` syscall between scenario ❶ and ❸ in Figure 6. To achieve this, we append a specific `ipcp` object to the radix tree, introducing a new tree level. This operation is exemplified by the `sys_msgcreate(append_key)` action in scenario ❷, while we use `sys_msgstat` with `probe_key` to probe the internal state of the tree. However, to insert the new tree level, we need to occupy all slots in the first tree level beforehand. Hence, we initially append 64 `ipcp` objects. With the first level occupied, we alternate between insertion and removal of `append_key` to append or remove the second tree node, respectively. Hence, we obtain a notable timing difference between scenario ❶ and ❸, i.e., the insertion or removal of just one key.

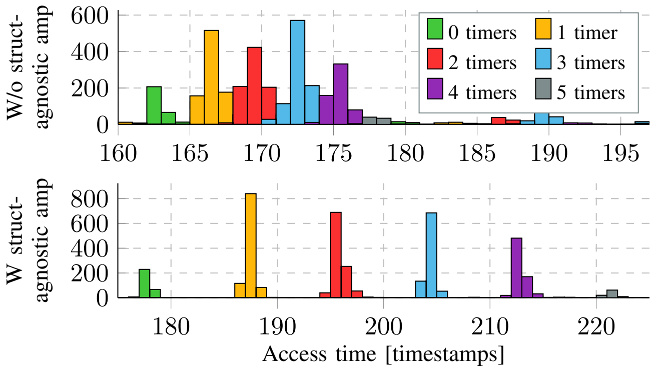


Fig. 8: Information leakage of `posix_timers_hashtable` with and without amplification. We can see that the timings spread over a wider range with the amplification.

B. Evaluation

We evaluate the KernelSnitch leakage with and without amplification and its noise resilience on each container structure. We then show the hardware independence of our evaluation by running it on 4 different systems with the same source code. Crucially, the results do not depend on the allocation addresses of the structures and remain consistent between reboots.

We developed a helper kernel module to obtain the ground truth, e.g., the occupancy of a specific hash bucket. We then fill the data container with objects, modifying its occupancy level. We measure time using `rdtsc` before and after the syscall, storing the difference between these two timestamps. Using ground truth and KernelSnitch-deduced occupancies, we determine the False-Positive Rate (FPR) and False-Negative Rate (FNR). We run the evaluation with and without our amplifications for comparison (see Table I). For consistent timing results, we filter outliers and focus on the ones not influenced by noise. Noise can only increase the timing and, hence, we average the lowest 8 values over 512 measurements. We evaluate on an Intel i7-1260P, with an Ubuntu 22.04.4 and kernel v6.5. We obtain similar results on 3 other processors with the same evaluation code (see Table I); one even runs with kernel v5.15. We also evaluated on AArch64 (i.e., Raspberry Pi 4) with the same code except for using `clock_gettime` instead of `rdtsc`, showing similar results.

All operations performed, including syscalls and instructions, are available to unprivileged users and require no extra capabilities. Our side channel also does not rely on CPU frequency pining, as it is a privileged operation. In fact, as we show in our stress evaluation, the most dominant noise factor is the frequency fluctuation as we do not pin the frequency.

POSIX Timer Hash Tables. We populate the `posix_timers_hashtable` with 4096 timers using `sys_timer_create` to increase the occupancy of one randomly selected hash bucket out of 512. After appending each timer, we measure the time of the `sys_clock_gettime` syscall with a randomly selected, invalid `id`. We then compare the KernelSnitch-

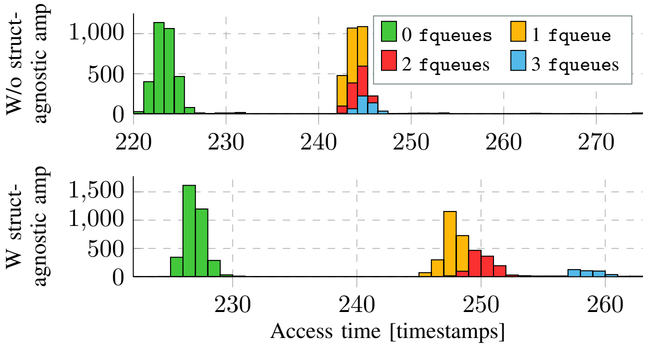


Fig. 9: Information leakage of `futex_hash_table`.

deduced occupancy of specific hash buckets with the ground truth. The measured timing is shown in Figure 8 without (distinguishing 1 and 0 timers, i.e., 1 to 0) and with (i.e., 3 to 0) hardware-agnostic amplification as well as without and with our structure-agnostic amplification (i.e., cache flushing). We compute the FPR and FNR using a threshold value between the medians of both histograms, e.g., 164 to distinguish 1 and 0 timers. We obtain 1.8 % (FPR) and 10.0 % (FNR) for distinguishing 1 and 0 timers. For distinguishing 3 and 0 timers, we obtain 0 % (FPR) and 9.4 % (FNR), showing the efficacy of the hardware-agnostic amplification. With structure-agnostic amplification, the FPR and FNR decrease to 0 %. The elimination of FNR and FPR yields an accuracy of 100 %.

Futex Hash Tables. We conduct a similar evaluation with `futex_hash_table`, using `sys_futex_wait` to append 8192 futex queue objects to the hash table, which consists of 4096 hash buckets on our default system[2]. The access timing was measured using `sys_futex_wake`. The evaluation results are illustrated in Figure 9. In contrast to the POSIX timer hash table results, the distinction between no elements within the hash buckets and one is more significant, while distinguishing multiple elements becomes less significant. One possible reason for this disparity lies in the differences in the lookup loop and the object's structure, i.e., `futex_hash_table` exhibits an early exit on lookup if no element is present. This characteristic renders the time difference between no elements and any element significant, shown in both with and without structure-agnostic amplification. When distinguishing between multiple elements, we suspect only one cache miss occurs in each iteration, where with `k_itimer` two misses occur. This results in a less significant timing difference for multiple elements.

IPC Hash Tables. For the `ipc_ids.key_ht`, we exploit `sys_timer_create` to populate data objects into pseudo-random hash buckets. We then exploit `sys_clock_gettime` to probe pseudo-randomly selected buckets. In total, we populate 4096 objects, probing the hash buckets after each insertion, and obtained the ground truth using our helper module. Figure 10 illustrates the results of this evaluation, with Table I summarizing the FPR and FNR, along with the

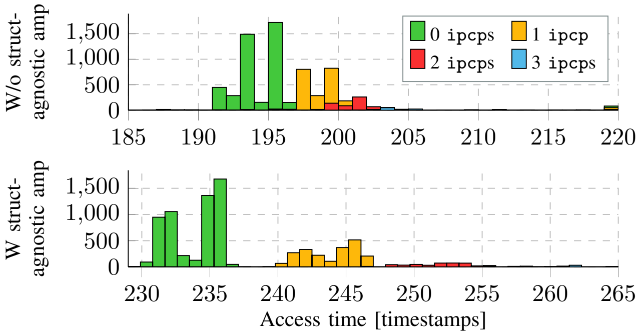


Fig. 10: Information leakage of `ipc_ids.key_ht`.

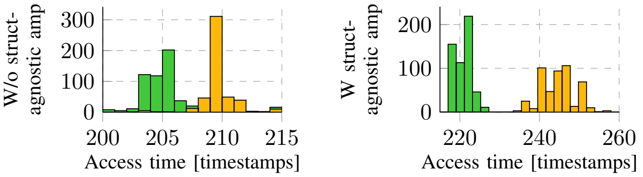


Fig. 11: `ipc_ids.ipcs_idr.root_rt` information leakage.

improvements achieved in both leakage amplifications. Both figures show that this hash table primarily comprises hash buckets with low occupancy levels. This characteristic stems from the hash table's dynamically resizable property. If the occupancy level of multiple buckets becomes too high, the hash table automatically resizes its bucket array and restructures objects within the buckets. Importantly, as demonstrated in Section VI, although this dynamic resizing property may complicate exploitation, KernelSnitch still leaks information from these hash tables.

Radix Tree. For `ipc_ids.ipcs_idr.root_rt`, our evaluation was as follows: We begin by inserting `ipcp` objects to the radix tree using `sys_msgcreate` until the entire first tree level is filled with valid slots (i.e., 64 slots). We then alternatively insert and remove an `ipcp`. This prompts the kernel to either append a new tree level, as depicted in scenario ❷ of Figure 6, or remove the just-appended level. After every insertion and deletion, we perform a radix tree lookup with the probe syscall `sys_msgstat` using an invalid key, where we do 1024 in total. Depending on whether the radix tree consists of one level ❶ or two levels ❸, the lookup timing varies. Figure 11 illustrates the histograms of access times with no structure-agnostic amplification, depending on whether the radix tree has one or two levels, as well as with amplification, increasing the distinguishable between these histograms. While KernelSnitch without the amplification resulted in FPR and FNR of 1.7 % and 3.9 %, the amplification eliminated all of them.

Red-Black Tree. Contrary to the prior evaluations, we conduct a slightly different one for the `hrtimer_bases.clock_base.active` red-black tree. In this evaluation, we aim to demonstrate how the enqueuing time depends on the occupancy level. As the tree maintains self-balancing, we anticipate that the enqueuing time follows a logarithmic function

[2] Its size is computed as $256 \cdot$ `nr_cores`, where our system has 16 cores.

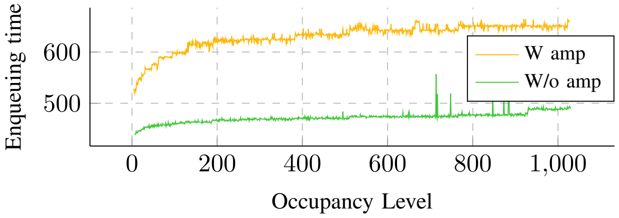


Fig. 12: Leakage of `hrtimer_bases.clock_base.active`.

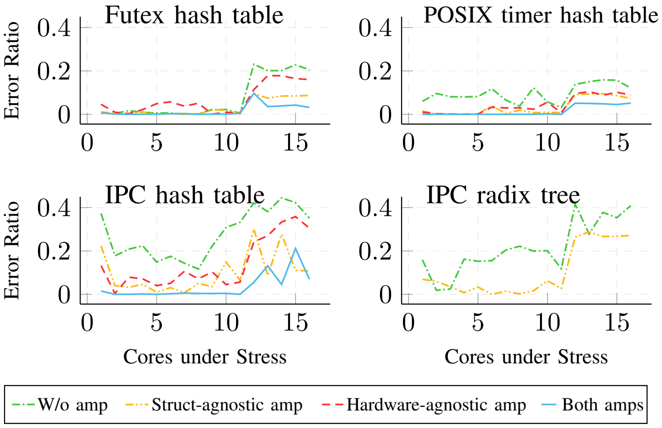


Fig. 13: Noise evaluation results without and with amplification methods as a function of the stress cores (i.e., 1 to 16).

depending on the occupancy level. We enqueue 1024 high-resolution timers and simultaneously measure the enqueuing time and obtain the ground truth using our helper kernel module after each enqueue operation. Figure 12 depicts the enqueuing time relative to the tree's occupancy level with and without structure-agnostic amplification. Both functions exhibit a logarithmic dependency on the actual occupancy level. Our amplification increases the enqueuing time by over 347 %. The occupancy level does not start at 0 timers, as the system always has default timers enqueued, e.g., `tick_sched_timer`.

External Noise. We introduce noise either through *stress evaluation* or *directly into data structures*. We show that the most dominant noise factor is the CPU frequency fluctuation and the noise resilience of our KernelSnitch side channel.

For the *stress evaluation*, we vary the number of workload threads of `stress-ng` [41], [43], [52], ranging from 1 to 16, i.e., the number of logical cores for the evaluated Intel i7-1260P. These workloads stress the CPU cores on which the workload is running. We separately evaluate structure- (i.e., distinguishing between 3 and 0 elements) and hardware-agnostic amplification (i.e., flushing CPU caches), as well as their combination. Figure 13 illustrates the error ratio observed in the stress evaluation, relative to the number of CPU cores experiencing stress. The error ratio represents the proportion of incorrectly deduced elements compared to the total evaluated (i.e., FPR+FNR). We note a rise in the error ratio when stress is introduced to an equal or greater number of physical cores (i.e., 12 cores). However, with both amplifications applied, we observe a negligible error ratio below 1.5 % if stress is applied to fewer than 12 cores. This yields an accuracy of more than 98.5 % until 3/4 of the full load. Concurrent measurement of the CPU frequency shows that frequency fluctuation caused by adaptive power management are the dominant noise factor for the stress evaluation. These fluctuations result in varying syscall execution times, perceived as noise for the attacker.

We also stress the Intel Xeon Gold 6530 (i.e., 64 cores) with workload threads ranging from 16 to 63. Specifically, we evaluate the POSIX timer hash table, observing no error ratio across all tests with both amplifications applied. By simultaneously measuring the CPU frequency, we observed it to be almost constant throughout the evaluation, as this is a desktop CPU with powerful cooling. This underscores the finding that the dominant noise factor is frequency fluctuation, most prevalent in laptop CPUs, e.g., Intel i7-1260P.

For introducing noise *directly into the data structure*, we found that Phoronix's Apache benchmark with 1 000 concurrent requests introduces the most noise into the futex hash table compared to other benchmarks. It introduces noise in the form of 55 000 (up to 100 000) hash bucket changes per second. We applied both amplifications and evaluated on an Intel i7-1260P, resulting in an error ratio less than 1 %. The simultaneous measurement of the CPU frequency shows that noise due to frequency fluctuations is predominant.

VI. Attack Case Studies

In this section, we demonstrate the practicality of KernelSnitch attacks in three side-channel case studies: a covert channel, a kernel heap pointer leak, and a website fingerprinting attack. We run the experiments on an Intel i7-1260P with Ubuntu 22.04.4 and a Linux kernel v6.5. Our attacks have an automated calibration phase directly at the start of the exploit to determine the threshold between a low and high occupancy levels. These thresholds remain consistent for the specific data container structures and do not depend on the allocation addresses of the structures as shown in Section V-B.

A. Covert Channel

We demonstrate that all container structures analyzed can be used to establish a covert channel. Our covert channel uses time slicing on the occupancy side channel to transmit data.

Threat Model. We assume that the sender and receiver run co-located on the same system. The sender has access to sensitive data but is strictly isolated and has no network access, e.g., within a sandbox. The receiver has no access to sensitive data but network-access permission, e.g., to a remote server to exfiltrate data. The sender and receiver have no shared memory or other resources shared besides the kernel itself.

Overview. We transmit data as a binary signal, e.g., in Figure 14 a bit sequence of '0101'. We transmit a '1' by increasing the occupancy level by appending one or more objects and a '0' by reducing the occupancy level by removing one or more objects. This results in a higher or lower probe syscall time, which is what we build our channel on. We synchronize our covert channel with a shared timer, e.g., `rdtsc` on x86_64.

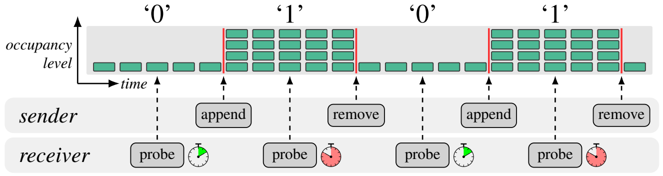


Fig. 14: KernelSnitch covert channel's overview, where the sender alters and the receiver probes the occupancy level of a data structure in fixed time slices.

The transmission starts at a coarse-grained predetermined time offset (e.g., the last 38 bits wrap around at a full minute), while bits are transmitted in short predetermined time slices. For each time slice, KernelSnitch adjusts the occupancy based on the data being transmitted. The receiver process continuously probes the occupancy throughout the time slice, using the minimum probe value as a result, minimizing noise.

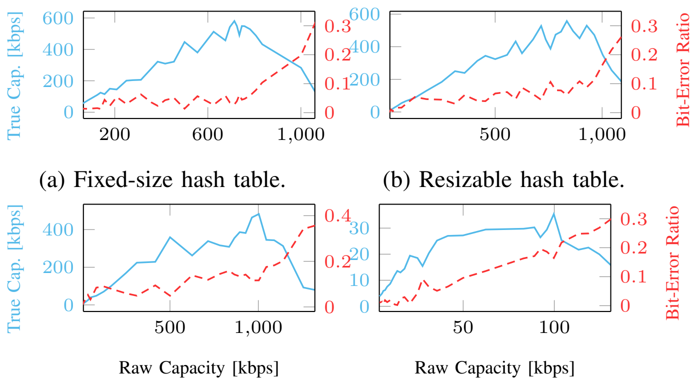
Design for Hash Tables. We initially identify a hash bucket for communication. The sender process uses hardware-agnostic amplification and populates one hash bucket with many elements. Subsequently, the receiver process iterates through the hash table, probing each bucket to determine if it contains a substantial number of elements. When it identifies the bucket, the receiver knows the bucket that will serve as the shared channel. With this stage complete, the transfer starts.

We build two covert channels. The first covert channel is based on fixed-size hash tables, leveraging `futex_hash_table`. A similar principle can be applied to other hash tables with a fixed size, e.g., `posix_timers_hashtable`. The second covert channel is based on dynamically-sized hash tables, leveraging `ipc_ids.key_ht`. For the fixed-size futex hash table, the sender initially populates the hash bucket with 64 futex queues. Subsequently, the receiver finds this hash bucket, as described above. For the transmission, a single appended futex queue is enough to transmit a '1' bit, while the absence of this futex queue transmits a '0' bit. For the dynamically-sized hash table, `ipc_ids.key_ht`, we initially populate it with 16 keys to find the shared bucket and then transmit data with occupancy differences of one key.

Design for a Radix Tree. The sender transmits '1' with the tree occupancy level 2 and a '0' with tree level 1. The tree level is manipulated by appending a specific key requiring an additional level (see `append_key` in Figure 6) or removing this key again. The receiver probes the radix tree occupancy, where higher probe times indicate a '1' and lower times a '0'.

Design for a Red-Black Tree. Appending 16 timers is sufficient to create a distinct probe timing difference (see Section V). Hence, the sender appends 16 timers at the tree's tail by setting a very high initial value, causing the red-black tree to insert multiple levels. The sender and receiver encode the data in timings: a lower timing corresponds to fewer levels, i.e., a '0' bit; a higher timing indicates a '1' bit.

Evaluation. We evaluate all four data container structures (i.e., fixed-size and resizable hash tables, radix tree, and red-black tree). For each structure, we evaluate different time slice lengths and record the channel's raw capacity – the maximum potential data rate – and bit-error ratio. Shorter time slices yield a higher transmission rate but reduce the receiver's ability to probe the occupancy level reliably, e.g., the measurement may be more noisy. Consequently, while the raw capacity increases with shorter time slice lengths, the true channel capacity might decrease due to a higher bit-error ratio. To represent our channels' effectiveness, we compute the true capacity[3] based on the raw capacity and the bit-error ratio.



(a) Fixed-size hash table.

(b) Resizable hash table.

(c) Radix tree.

(d) Red-black tree.

Fig. 15: KernelSnitch covert channel's raw capacity, bit-error ratio, and true capacity, ranging between 35 kbit/s to 580 kbit/s.

Figure 15 shows the true capacity as a function of the raw capacity and bit-error ratio for all four structures. For the fixed-size `futex_hash_table` (see Figure 15a), the bit-error ratio is below 7 % until the raw capacity reaches 781 kbit/s. Beyond this point, the slice length becomes so short that the receiver can only execute a maximum of 3 probing syscalls to deduce the occupancy level. We observe that with at most 3 probing syscall, there is a steady increase in the bit-error ratio, reducing the actual capacity. For the futex table, the optimal true capacity of 580 kbit/s is achieved at a raw capacity of 714 kbit/s and a bit-error ratio of 2.8 %. We observe a similar behavior for both the dynamically-sized hash table (see Figure 15b) and the radix tree (see Figure 15c). The point of a steady increase in the bit-error ratio occurs at a raw capacity of 963 kbit/s and 1 003 kbit/s, respectively. Their optimal true capacity is reached with 528 kbit/s and 483 kbit/s. As both data structures are used for IPC communication, Linux isolates them within the IPC namespace. Hence, this channel is restricted to the sender and receiver sharing the same IPC namespace, prevented by sandboxes, e.g., Docker or browsers. For the red-black tree (see Figure 15d), we observe that the optimal true capacity is 35 kbit/s, achieved at a raw capacity of 100 kbit/s with a 16.5 % bit-error ratio. This capacity is the lowest of the four structures, primarily due to the extended time required for appending and probing operations.

[3] We use Shannon's theorem: $T = C \cdot (1 + ((1-p) \cdot \texttt{ld}(1-p) + p \cdot \texttt{ld}(p)))$.

B. Kernel Heap Pointer Leak

Linux uses kernel heap addresses of objects such as mm_struct in the indices for hash table lookups. We demonstrate that we can leak these kernel heap addresses used to index hash table lookups using KernelSnitch to observe hash collisions from user space. We then demonstrate that by performing a cross-cache reuse [43], [65], [67], we can place other objects, such as the security-critical msg_msg [14], [30], [71], at this leaked address, thereby obtaining the location of other objects.

Threat Model. We assume an attacker has no privilege to access information about kernel addresses, and the attacker has an exploit that only works if a targeted kernel heap pointer is known, e.g., for multiple exploits [5], [14], [25], [30], [71].

Design. Our kernel heap pointer leak consists of two steps: First, we *detect hash collisions* of hash table entries with the same kernel address but different user identifiers. Second, we *enumerate all possible kernel addresses* to match the detected collisions for the user identifiers used, resulting in the kernel address used for indexing being leaked. Using the futex hash table as an example, we aim to leak the mm_struct address, which is used with the user identifier uaddr for indexing.

To *detect hash collisions*, we exploit the KernelSnitch side channel as follows: Initially, we append one futex queue to a hash bucket, such as fqueueA with uaddrA and mmA to bucket 2 as futex_hash(uaddrA,mmA) $= 2$. This state is depicted with ❶ of Figure 5, where fqueueB represents a queue in another bucket. We then apply structure-agnostic amplification, appending multiple queues to the hash bucket 2 using the same uaddrA and mmA. With hardware-agnostic amplification, we observe the occupancy level of the hash bucket using the same mm_struct (as the same user process) but with different and invalid user identifiers, i.e., uaddrs. A low occupancy level, such as for the user identifier uaddrY, indicates a different hash bucket. Conversely, a higher occupancy level, e.g., uaddrZ, means that the values futex_hash(uaddrA,mmA) and futex_hash(uaddrZ,mmA) match. We denote this as a hash collision of the user identifiers uaddrA and uaddrZ using the same mm_struct. We repeat this process until a sufficient number of hash collisions are found.

We now have a list of known user identifiers (i.e., uaddrs) that, combined with one unknown kernel heap address (i.e., mm_struct), results in the same hash value. With the hash function known (i.e., jhash2 for the futex hash table), we *enumerate all possible kernel addresses* together with the known user identifiers in an offline phase to determine hash collisions. As described in the next paragraph, the search space for all possible heap addresses of a specific mm_struct can be reduced to $\approx 2^{35.5}$. If we find the address that, combined with all user identifiers, results in the same hash, we leak the mm_struct heap address. In the simplified example of Figure 5, we leak mmA as it results in the same hash value when combined with the identifiers uaddrA and uaddrZ.

Since the kernel heap is directly accessed via the Direct Physical Mapping (DPM) [44] (see Figure 16), the search space is the DPM offset by the randomized page_offset_base. The DPM serves as a virtual memory mapping of, typically, the physical memory range and spans over a significant part of the kernel address space. For instance, on x86_64, it ranges between 0xffff888000000000 and 0xffffc87fffffffff, representing a search space of 2^{46} (when considering 8 B kernel heap alignment, it results in an entropy of 43 bit). To reduce the search space, we consider the alignment constraints of mm_struct, enforced by the page and slab allocator[4]. The page allocator guarantees the outer alignment, ensuring that the memory chunk (also called slab) from which mm_struct objects are allocated is aligned to 8 pages. The slab allocator sits on top of the page allocator and ensures that objects within these slabs are aligned to object size (and usually also to the cache line size). Using these insights allows us to substantially reduce the search space for possible mm_struct addresses as follows: On our experimental system using Linux v6.5 x86_64 with the default, generic configuration, the mm_struct has a size of 1 360 B. Considering the alignment of the cache lines (rounded up to 1 408 B), 23 locations are possible within the 8 slab page. Hence, the search space is $2^{46-12-3} \cdot 23 \approx 2^{35.5}$, with 12 bits representing the page size and 3 bits representing the mm_struct slab size. Given a complexity of $\approx 2^{35.5}$, we iteratively examine all possible addresses and try to match them with previously leaked hash collisions produced with different user identifiers. Subsequently, we reconstruct the key corresponding to these hash collisions. As the kernel heap address is one of the key's inputs, we successfully obtain this address, consequently leaking heap pointers.

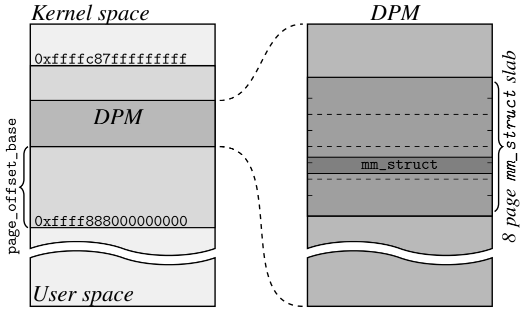


Fig. 16: The kernel memory layout on x86_64 illustrates the kernel heap is accessible directly via the DPM, showcasing how the heap-allocated mm_struct object is located.

Cross-Cache Reuse. We perform a cross-cache reuse [43], [65], [67], which frees the leaked mm_struct object (including all objects of its slab) and reallocates the freed (and leaked) memory chunk for other objects. This allows us to leak the location of objects other than those directly leaked via KernelSnitch. Below, we demonstrate that by using this approach, we can leak the address of the security-critical msg_msg, used in several kernel exploits [14], [30], [71]. While msg_msg is an example, we can also leak the address of other objects.

[4] These details can be obtained from /sys/kernel/slab/mm_struct and remain consistent across the same kernel binary.

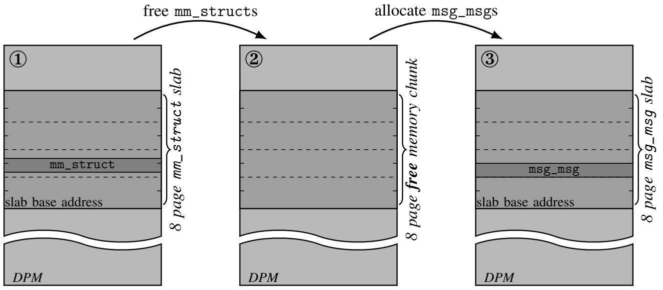


Fig. 17: Cross-cache reuse which frees the leaked `mm_struct` (and all of its slab) and reallocate its memory chunk as `msg_msgs`, thereby obtaining the location of these `msg_msgs`.

Figure 17 shows the high-level overview, where Figure 20 provides more details. The state ① represents the leaked `mm_struct` address within its slab. From this `mm_struct` address, we derive the base address of its slab, which is done by applying a bitmask of the 8 page memory chunk (i.e., `((1<<15)-1)`) to the address. Next, we deallocate all `mm_structs` within this slab ②, causing the kernel to recycle the leaked 8 page free memory chunk. We then allocate multiple objects ③ of the targeted type (i.e., `msg_msg` with size $4\,048$) to reclaim the 8 page memory chunk. Using the alignment information of the allocator cache of `msg_msg` with size $4\,048$ (i.e., `kmalloc-cg-4096`), we deduce all possible object locations within this chunk. This results in 8 locations of $\mathtt{n} \cdot 4\,096 + \mathtt{slab\_base}$ where `n` is between 0 and 7.

We reclaim the leaked slab previously used for `mm_structs` as `msg_msgs` with size $4\,048$, as both objects use the same size per-CPU page free list order of 3 (i.e., 2^3 page memory chunk). These per-CPU page free lists act as a first-level allocator cache of the page allocator. If we want to reclaim the leaked 8 page memory chunk as a different page size chunk, such as `kmalloc-cg-512` (which uses the per-CPU page free list of order 2), we must first drain the page free list of order 2. Prior work [65] has presented appropriate techniques to reliably perform this cross-page free-list reuse.

Evaluation. We implement our KernelSnitch kernel heap leak in an architecture-agnostic manner. Architecture-specific information is required to reconstruct the `mm_struct` address from hash collisions due to variations in the DPM across different architectures. We implement KernelSnitch to leak the kernel heap address for x86_64, AArch64, and RISC-V architectures. In our experiments, we successfully perform this attack natively with our x86_64 experimental setup as well as in QEMU for AArch64 and RISC-V architectures. For the native experiment, we repeat the hash-collision leaking attack 10 times, with a leak time between 1.7 s to 2.1 s. Using these collisions, we recover the correct `mm_struct` address in 2 s to 61.5 s (iterating through 1.8 % to 28.5 % of the possible kernel addresses) on a 24-core AMD EPYC 7443 processor. Consequently, KernelSnitch requires between 3.7 s to 63.6 s for a kernel heap leak. In addition to leaking the `mm_struct`, we implement the cross-cache reuse described above for our native x86_64 system. We successfully reclaimed the leaked `mm_struct` slab for the `kmalloc-cg-4096` slab cache, leaking the address of the 8 `msg_msg` objects it contains. Similar to the above, we performed 10 successful cross-cache reuses on our native system. They took between 0.847 s to 0.901 s, resulting in a total time of under 65 s for leaking a `msg_msg`.

We also exploit the POSIX timer hash table, reducing potential `k_itimer` addresses from 2^{38} to 2^9. While it only led to a partial kernel heap address leak, it advances understanding system vulnerabilities. The POSIX timer's hash function lacks uniform output distribution, preventing determination of the linearly mapped input-to-output part (i.e., 9 bits). In contrast, the futex hash table, using `jhash`, ensures uniform distribution, enabling of the leakage the entire kernel heap address.

C. Website Fingerprinting

This section presents a website fingerprinting attack, showing its capability to determine when a user accesses a website from the Ahrefs top 100 [1] with an F_1 score of 89.3 %.

Threat Model. We assume the attacker executes code on the same machine as the victim but is isolated by the web browser's sandbox. Since we assume a sandboxing isolation (e.g., for the mount and network namespace), approaches such as calling `netcat` to leak browser activity cannot be used.

Design. Web browsers like Firefox rely on user-space locks, i.e., futexes, to handle transmitted and received data. During website access, the browser acquires and releases these locks and interacts with the futex hash table. This behavior creates a unique occupancy level fingerprint in the futex hash table. We leverage KernelSnitch to leak the futex hash table's occupancy level in this side-channel attack. We use a Convolutional Neural Network (CNN) to classify these occupancy level fingerprints of the Ahrefs top 100 websites [1].

We use the occupancy-level side channel of the futex hash table to obtain website traces in two stages: First, we find for each hash bucket a unique user-space address, allowing us to leak the occupancy level of each bucket with `sys_futex_wake`, as represented in Figure 5. To achieve this, we leverage the hardware-agnostic leakage amplification by appending a significant number of futex queues to a hash bucket. Using structure-agnostic leakage amplification, we probe all buckets with incremental user-space addresses to identify a bucket with a significant number of queues appended. Upon discovering a significant hash bucket, we validate that this user-space address does not cause a collision with a previously found user-space address. This appending and probing routine is repeated for all buckets. As a result of this initial stage, we have a set of user-space addresses indexing all hash buckets, which we refer to as our probe set. Second, we use our probe set to determine the access times of each hash bucket with 20 samples per second. For probing, we first iterate over all hash buckets and then repeat this process for the entire sample timeframe (i.e., 50 ms). This way, we have structure-agnostic amplification without explicitly flushing the CPU caches. Subsequently, we take the minimum probe value for each hash bucket within a timeframe. This is repeated during

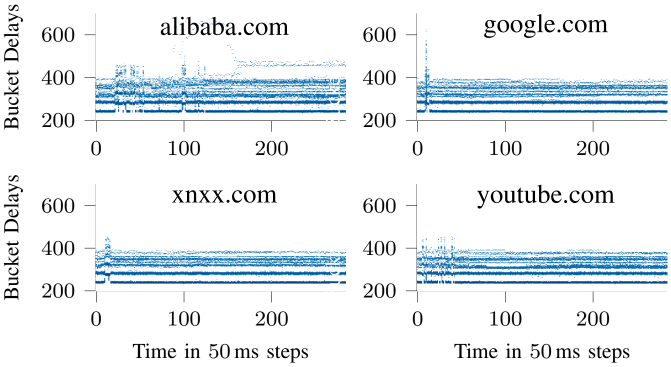


Fig. 18: Traces of 4 famous websites, showing the delay of all bucket measurements on the y axis while website loading.

loading a website (i.e., 15 s), creating a two-dimensional trace, consisting of the access times of all buckets at 300 timestamps. Finally, we create a histogram, rounding all bucket delays to the nearest integer and summing all up. Figure 18 shows four website traces. The x-axis is the time axis, and the y-axis shows the bucket delays, with the color darkness representing the number of buckets with each delay.

Our attack consists of an online and offline phase for data collection and evaluation of website traces. In the online phase, a user-space process runs KernelSnitch on the system within a sandbox, probing all hash buckets of the futex hash table, resulting in a website accessing trace. The offline phase consists of analyzing and classifying the collected traces. To classify the traces, we use a CNN with nine convolutional layers in three different sizes. The two-dimensional histograms, as shown in Figure 18, are the inputs to the CNN.

Evaluation. We record 100 traces for each of the 100 websites on Firefox (Chrome results in similar traces). We split the traces into 80 % training and 20 % test sets. Of the training set, we use 10 % for the validation used set while training. We perform a 5-fold cross validation by training the CNN with five randomly selected sets. Over the five validations, we achieve an average F_1 score of 89.3 %. Figure 21 shows the confusion matrix with an F_1 score of 89.5 %.

VII. Related Work

Numerous physical properties carry an information signal that is often tied to the specific implementation of a system or algorithm. These include power, radiation, temperature, sound, light emission, and time, with time being the most commonly used. More recent software side channels also extract information through timing differences induced by other physical properties [46], [64]. Lampson already reported in 1973 that timing differences could be exploited to transmit or extract information covertly [40]. In 1996, Kocher [37] presented the first timing side-channel attack on a cryptographic algorithm. Following the numerous timing-based attacks on cryptographic algorithms [7], [60], Osvik et al. [48] generalized the approaches into two generic techniques, Evict+Time and Prime+Probe. A decade later, Yarom et al. [68] presented Flush+Reload, monitoring cache-lines' state by removing them from the cache and timing a reload to the corresponding memory location. The timing depends on if the victim has accessed the cache line in between. Flush+Reload has virtually no false results and is frequently used by other attacks [38], [42].

Software-observable timing differences can be induced by caches and any behavioral difference on the software or instruction level, e.g., software caches [16], [22], [43], [63], page-fault interrupts [59], other interrupts [15], compression algorithms [6], [21], [33], [53], [55], differences in instruction or memory lookup sequences [62], [68], and many others. While the concept of constant-time [37] implementations has found wide adoption for cryptographic algorithms, the situation is much more difficult for general-purpose code [54]. Gao et al. [17] presented information leakage of files not fully namespaced allowing for covert channels. Gruss et al. [22] found that the operating system page cache can be exploited similarly to hardware caches. Their insights show that microarchitectural buffers and caches similarly exist in operating systems, again with caches, indicating an architectural interface is also applicable to the operating system. More recently, Patel et al. [49] presented a novel performance-degrading attack that exploits intra-kernel contention of locked kernel resources. While not a side channel, their result indicates that more architectural elements in the kernel may be exploited. Jiang et al. [31] showed that file system sync operations affect each other's timing and can be used to build a covert channel, achieving transmissions of up to 20 kbit/s with an error rate 0.4 %. Chen et al. [10] showed that a similar timing influence also exists with write buffers for shared files. By filling or not filling the write buffer, they can covertly transmit up to 10 kbit/s with a 0.004 % error rate. Lee et al. [41] and Maar et al. [43] discovered timing side channels in the Linux slab allocator, inferring whether a new slab is created. This leakage increases the success rate for heap spraying [41] or cross-cache attacks [43]. Shen et al. [56] presented a covert channel based on mutual exclusion primitives [70], achieving transmission rates of up to 13.1 kbit/s with a 0.65 % error rate.

The concept of software-induced timing side channels is also related to the research problem of algorithmic complexity attacks [13]. Algorithmic complexity attacks try to provide systems with input that triggers the algorithmic worst case, e.g., a bucket with a long linked list instead of a flat hash table. The goal in these attacks is often denial of service [13], [57], deteriorating the runtime. Several works, therefore, discuss mitigations against denial-of-service algorithmic complexity attacks [4], [35]. Petsios et al. [50] presented a fuzzer to find algorithmic worst cases, including compression algorithms, for algorithmic complexity attacks. Schwarzl et al. [55] similarly built a fuzzer to find algorithmic worst cases in compression algorithms to build a new side-channel attack, showing the close relation between these two strands of research. Sun et al. [58] showed that algorithmic complexity attacks can also be used to build covert channels. Cai et al. [8] exploited algorithmic complexity attacks when exploiting race conditions on Unix file systems.

VIII. Mitigations

Our work extends prior research on software-induced timing side channels, showing that varying access timings of any shared kernel resource can also introduce a timing side channel, which, given the kernel interfaces, can potentially be exploitable from user space. Thus, the operating system level introduces exploitable leakage even when eliminating all hardware side channels and following all best practices for user software. Mitigating KernelSnitch has to be evaluated with performance and user experience, similar to other side channels [26], [47]. The key factors that enable KernelSnitch are the sharing of data container structures, the combination of privileged and unprivileged information in the same shared element, the runtime variance depending on a secret state, and the possibility to measure the runtime. Eliminating any of these can fully or partially mitigate KernelSnitch.

Measuring Time. Prior work studied the removal of precise timing measurements as a defense against side-channel attacks [27], [39]. However, other works also showed how attackers can resort to alternative timing methods or mount attacks entirely without timing [51], [63], [66], [69]. In our threat model, multiple timing sources are available for legitimate reasons, and removing them would be a highly disruptive change for software developers and require hardware changes.

Combining Privileged and Unprivileged Information. The aggregation of privileged and unprivileged information in shared elements has already been identified to introduce security issues on the hardware level [23], [61]. For KernelSnitch, we also exploit that unprivileged (i.e., user identifiers) are combined with privileged information (i.e., kernel addresses). However, eliminating this only prevents the kernel heap leak.

Runtime Variance. A constant-time approach [7], [37] is not directly possible against KernelSnitch as the container structures are, in principle, only bounded by the available memory. However, KernelSnitch could be mitigated by using a watermark constant-time approach: Instead of always resorting to a (hypothetical) worst-case execution time, it may be a viable approach to maintain a watermark level of the maximum number of elements to be iterated through. Syscalls iterating through structures will consequently wait until the watermark execution time is reached, eliminating the secret-dependent runtime variance. Further security can be gained by increasing the watermark level in a coarse step size.

Sharing of Container Structures. Another mitigation is to eliminate the sharing of the kernel data container structures, e.g., isolating kernel data container structures within namespaces. However, this involves significant reworking and notable performance and memory overheads for the kernel. Future work needs to investigate the practicality of such isolation. For instance, the red-black tree that organizes global timers by their firing order poses a challenge.

IX. Conclusion

In this paper, we presented KernelSnitch, a novel generic side-channel attack targeting kernel data container structures. We demonstrated and evaluated leakage amplification to make this side channel exploitable from user space. We performed three case study side-channel attacks: covert channel, kernel heap pointer leak, and website fingerprinting. Finally, we discussed potential mitigations and highlighted challenges.

Acknowledgment

We thank Mathias Oberhuber and the anonymous reviewers for their valuable feedback. This research is supported by the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087), the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided from Red Hat and Google. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Ahrefs. Top Websites, 2024. URL: https://ahrefs.com/top.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *S&P*, 2019.

[3] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying Side Channels Through Performance Degradation. In *ACSAC*, 2016.

[4] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. Surgeprotector: Mitigating Temporal Algorithmic Complexity Attacks Using Adversarial Scheduling. In *ACM SIGCOMM*, 2022.

[5] Awarau and pql. CVE-2022-29582 An io_uring vulnerability, 2022. URL: https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/.

[6] Tal Be'ery and Amichai Shulman. A Perfect CRIME? Only TIME Will Tell. In *Black Hat Europe*, 2013.

[7] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[8] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting UNIX file-system races via algorithmic complexity attacks. In *S&P*, 2009.

[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*, 2020.

[10] Congcong Chen, Jinhua Cui, Gang Qu, and Jiliang Zhang. Write+Sync: Software Cache Write Covert Channels Exploiting Memory-disk Synchronization. *TIFS*, 2024.

[11] Jonathan Corbet. Trees I: Radix trees, 2006. URL: https://lwn.net/Articles/175432/.

[12] Jonathan Corbet. Trees II: red-black trees, 2006. URL: https://lwn.net/Articles/184495/.

[13] Scott A Crosby and Dan S Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security*, 2003.

[14] Devil. CoRJail: From Null Byte Overflow To Docker Escape Exploiting poll_list Objects In The Linux Kernel, 2022. URL: https://syst3mfailure.io/corjail/.

[15] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *S&P*, 2016.

[16] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In *CCS*, 2000.

[17] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *DSN*, 2017.

[18] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote Scheduler Contention Attacks. In *FC*, 2024.

[19] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In *S&P*, 2023.

[20] Luke Gix. FUSE for Linux Exploitation 101, 2022. URL: https://exploiter.dev/blog/2022/FUSE-exploit.html.

[21] Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: reviving the CRIME attack. *Unpublished manuscript*, 2013.

[22] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In *CCS*, 2019.

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*, 2016.

[24] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*, 2015.

[25] h0mbre. Escaping the Google kCTF Container with a Data-Only Exploit, 2023. URL: https://h0mbre.github.io/kCTF_Data_Only_Exploit/#.

[26] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level. In *EuroSys*, 2021.

[27] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. *Journal of Computer Security*, 1992.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[29] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*, 2016.

[30] javierprtd. No CVE for this bug which has never been in the official kernel, 2023. URL: https://soez.github.io/posts/no-cve-for-this.-It-has-never-been-in-the-official-kernel/.

[31] Qisheng Jiang and Chundong Wang. Sync+Sync: A Covert Channel Built on fsync with Storage. In *USENIX Security*, 2024.

[32] Choo Yi Kai. A new method for container escape using file-based DirtyCred, 2023. URL: https://starlabs.sg/blog/2023/07-a-new-method-for-container-escape-using-file-based-dirtycred/.

[33] John Kelsey. Compression and Information Leakage of Plaintext. In *Fast Software Encryption*, 2002.

[34] Michael Kerrisk. *futex(2) — Linux manual page*, 2023. https://man7.org/linux/man-pages/man2/futex.2.html.

[35] Suraiya Khan and Issa Traore. A Prevention Model for Algorithmic Complexity Attacks. In *DIMVA*, 2005.

[36] Amit Klein and Benny Pinkas. From IP ID to Device ID and KASLR Bypass. In *USENIX Security*, 2019.

[37] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[38] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[39] David Kohlbrenner and Hovav Shacham. Trusted Browsers for Uncertain Times. In *USENIX Security*, 2016.

[40] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[41] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique. In *USENIX Security*, 2023.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[43] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In *USENIX Security*, 2024.

[44] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In *ACSAC*, 2023.

[45] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, 2017.

[46] Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, and Stefan Mangard. Power-Related Side-Channel Attacks using the Android Sensor Framework. In *NDSS*, 2025.

[47] OpenSSL. Security Policy, 2024. URL: https://www.openssl.org/policies/general/security-policy.html.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[49] Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Using Trātṛ to tame Adversarial Synchronization. In *USENIX Security*, 2022.

[50] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *CCS*, 2017.

[51] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In *AsiaCCS*, 2023.

[52] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In *NDSS*, 2024.

[53] Juliano Rizzo and Thai Duong. The CRIME attack. In *ekoparty security conference*, volume 2012, 2012.

[54] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*, 2018.

[55] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical Timing Side Channel Attacks on Memory Compression. In *S&P*, 2023.

[56] Chaoqun Shen, Jiliang Zhang, and Gang Qu. MES-attacks: Software-controlled covert channels based on mutual exclusion and synchronization. In *DAC*, 2023.

[57] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking Algorithmic Complexity Attacks Against a NIDS. In *ACSAC*, 2006.

[58] Xiaoshan Sun, Liang Cheng, and Yang Zhang. A Covert Timing Channel via Algorithmic Complexity Attacks: Design and Analysis. In *IEEE International Conference on Communications (ICC)*, 2011.

[59] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication as a Threat to the Guest OS. In *EuroSys*, 2011.

[60] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In *CHES*, 2003.

[61] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[62] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*, 2018.

[63] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS*, 2015.

[64] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *USENIX Security*, 2022.

[65] Le Wu and Qi Zhang. Game of Cross Cache: Let's win it in a more effective way!, 2024. URL: https://i.blackhat.com/Asia-24/Presentations/Asia-24-Wu-Game-of-Cross-Cache.pdf.

[66] Haocheng Xiao and Sam Ainsworth. Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers. In *ASPLOS*, 2023.

[67] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *CCS*, 2015.

[68] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[69] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W Fletcher. Synchronization Storage Channels (S^2C): Timerless Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In *USENIX Security*, 2023.

[70] Jiliang Zhang, Chaoqun Shen, and Gang Qu. Mex+Sync: Software Covert Channels Exploiting Mutual Exclusion and Synchronization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[71] Xiaochen Zou and Zhiyun Qian. Exploit esp6 modules in Linux kernel, 2022. URL: https://etenal.me/archives/1825.

Appendix

```
struct signal_struct;
struct list_head {
  struct list_head *next, *prev;
};
struct hlist_head {
  struct hlist_node *first;
};
struct hlist_node {
  struct hlist_node *next, **pprev;
};
struct k_itimer {
  ...
  u32 it_id;
  struct hlist_node t_hash;
  struct signal_struct *it_signal;
  ...
};
DEFINE_HASHTABLE(posix_timers_hashtable, 9);

// Calculates hash for hash table
int hash(struct signal_struct *sig, unsigned int nr) {
  return hash_32(hash32_ptr(sig) ^ nr, 9);
}

// Iterates throught the bucket's linked list to find
// k_timer matching sig and id
struct k_itimer *__posix_timers_find(
              struct hlist_head *head,
              struct signal_struct *sig,
              u32 id) {
  struct k_itimer *tim;

  hlist_for_each_entry(tim, head, t_hash) {
    if ((tim->it_signal == sig) && (tim->it_id == id))
      return tim;
  }
  return NULL;
}

// k_timer lookup with id
struct k_itimer *posix_timer_by_id(u32 id) {
  struct hlist_head *head;
  struct signal_struct *sig = current->signal;

  head = &posix_timers_hashtable[hash(sig, id)];
  return __posix_timers_find(head, sig, id);
}
```

Listing 1: Simplified C equivalent for a timer lookup.

```
posix_timer_by_id:
  push  rbp
  push  r13
  push  rbx
  // sign = current->signal
  mov   gs:0x32880, rax
  mov   0xbf0(rax), rdx
  mov   rdx, rax
  // h = hash(sign, id)
  shr   $0x20, rax
  xor   rdx, rax
  xor   r13d, eax
  imul  $0x61c88647, eax, eax
  shr   $0x17, eax
  // head = &posix_timers_hashtable[h]
  mov   posix_timers_hashtable(, rax, 8), rbx
  // node = (hlist_node *)head
0x50:
  // tim = (k_itimer *)container_of(node,k_itimer,t_hash)
  sub   $0x10, rbx
  test  rbx, rbx
  je    <posix_timer_by_id+0x6b>
  // (tim->it_signal == sig)
  mov   0x60(rbx), rax
  cmp   rax, rdx
  je    <posix_timer_by_id+0x86>
0x62:
  // tim = (k_itimer *)tim->t_hash.next
  mov   0x10(rbx), rbx
  test  rbx, rbx
  jne   <posix_timer_by_id+0x50>
0x6b:
  // return NULL
  xor   ebx, ebx
  mov   rbx, rax
  pop   rbx
  pop   r13
  pop   rbp
  jmp   __x86_return_thunk
0x86:
  // (tim->it_id == id)
  cmp   0x34(rbx), r13d
  jne   <posix_timer_by_id+0x62>
  // return tim
  mov   rbx, rax
  pop   rbx
  pop   r13
  pop   rbp
  jmp   <__x86_return_thunk>
```

Listing 2: ASM instructions executed for the lookup.

Fig. 19: POSIX timer lookup from the timer's hash table using the function `posix_timer_by_id`. The instructions in blue represent the instructions executed for each element contained in the linked list of the hash bucket.

```
sys_timer_create():
  sign = current.signal
  id = sign.next_id++
  tim = k_itimer(sign, id)
  h = timer_hash(id, sign)
  hbucket =
   posix_timers_hashtable[h]
  hbucket.append(tim)
  return id
```

Listing 3: Alter occupancy of POSIX timer's hash table.

```
sys_clock_gettime(id):
  sign = current.signal
  h = timer_hash(id, sign)
  hbucket =
   posix_timers_hashtable[h]
  for tim in hbucket:
    if tim.sign == sign and
        tim.id == id:
      return tim.get_time()
  return ERROR
```

Listing 4: Probe occupancy of POSIX timer's hash table.

```
sys_msgcreate(key):
  ipc_ns = current.ipc_ns
  ipc_ids = ipc_ns.get_ids()
  msgq = msg_queue(key)
  h = ipc_ids_hash(key)
  ipc_ids[h].append(
    msgq.ipcp)
  return msgq.ipcp.id
```

Listing 5: Alter occupancy of IPC key's hash table.

```
sys_msgget(key):
  ipc_ns = current.ipc_ns
  ipc_ids = ipc_ns.get_ids()
  h = ipc_ids_hash(key)
  hbucket = ipc_ids[h]
  for ipcp in hbucket:
    if ipcp.key == key:
      return ipcp.id
  return ERROR
```

Listing 6: Probe occupancy of IPC key's hash table.

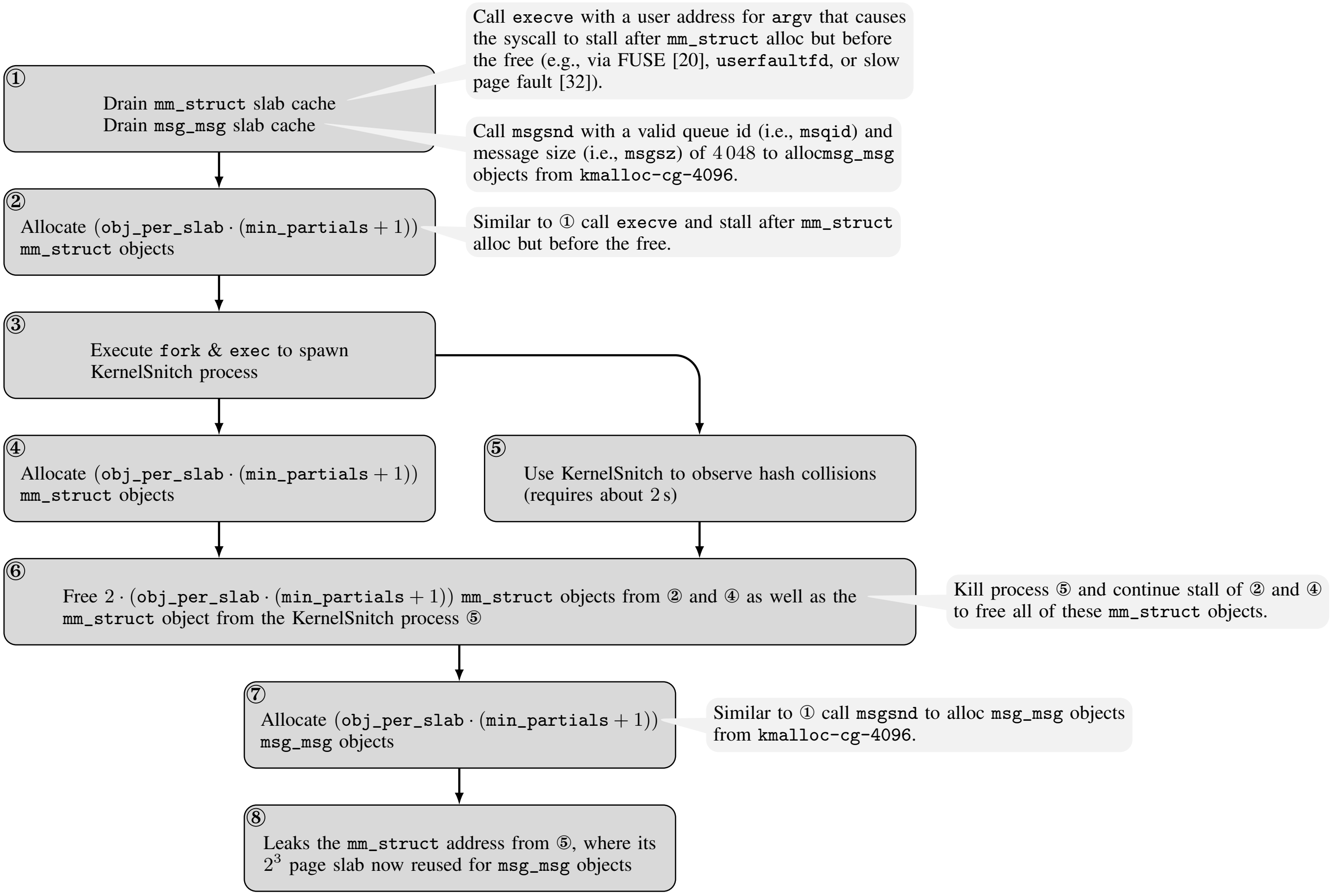

①
Drain mm_struct slab cache
Drain msg_msg slab cache
Call execve with a user address for argv that causes the syscall to stall after mm_struct alloc but before the free (e.g., via FUSE [20], userfaultfd, or slow page fault [32]).
Call msgsnd with a valid queue id (i.e., msqid) and message size (i.e., msgsz) of 4 048 to allocmsg_msg objects from kmalloc-cg-4096.
②
Allocate (obj_per_slab · (min_partials + 1)) mm_struct objects
Similar to ① call execve and stall after mm_struct alloc but before the free.
③
Execute fork & exec to spawn KernelSnitch process
④
Allocate (obj_per_slab · (min_partials + 1)) mm_struct objects
⑤
Use KernelSnitch to observe hash collisions (requires about 2 s)
⑥
Free 2 · (obj_per_slab · (min_partials + 1)) mm_struct objects from ② and ④ as well as the mm_struct object from the KernelSnitch process ⑤
Kill process ⑤ and continue stall of ② and ④ to free all of these mm_struct objects.
⑦
Allocate (obj_per_slab · (min_partials + 1)) msg_msg objects
Similar to ① call msgsnd to alloc msg_msg objects from kmalloc-cg-4096.
⑧
Leaks the mm_struct address from ⑤, where its 2^3 page slab now reused for msg_msg objects

Fig. 20: Detailed workflow of the cross-cache reuse.

TABLE I: Evaluation results for data container structures: **X to Y** denotes the difference from **X** compared to **Y** elements. **1 to 0** signifies no hardware-agnostic amplification, while **3 to 0** indicates hardware-agnostic amplification with 2 extra elements. ✶ denotes the decrease in FPR and FNR from no amplification (i.e., **1 to 0**) to structure- and hardware-agnostic amplification (i.e., **3 to 0**). ☆ denotes Linux kernel v5.15, where for the other three we used v6.5.

	Container instance	W/o struct-agnostic				W struct-agnostic				Reduction ✶	
		1 to 0		3 to 0		1 to 0		3 to 0			
		FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	in FPR	in FNR
		%	%	%	%	%	%	%	%	%	%
Intel i7-1260P	posix_timers_hashtable	1.8	10.0	0.0	9.4	0.0	0.0	0.0	0.0	100	100
	futex_hash_table	0.0	0.6	0.0	0.6	0.0	0.0	0.0	0.0	100	100
	ipc_ids.key_ht	3.1	2.8	1.7	2.7	0.1	0.1	0.0	0.1	100	97
	ipc_ids.ipcs_idr.root_rt	1.7	3.9	-	-	0.0	0.0	-	-	100	100
Intel i7-1165G7	posix_timers_hashtable	0.9	9.1	0.0	7.3	0.0	0.3	0.0	0.3	100	97
	futex_hash_table	0.0	0.5	0.0	0.5	0.0	0.2	0.0	0.1	100	76
	ipc_ids.key_ht	4.5	18.6	0.0	6.9	0.1	0.8	0.0	0.3	100	98
	ipc_ids.ipcs_idr.root_rt	0.0	32.9	-	-	0.0	1.7	-	-	100	95
☆ Intel i7-12700	posix_timers_hashtable	0.0	4.2	0.0	0.3	0.0	0.0	0.0	0.0	100	100
	futex_hash_table	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100	100
	ipc_ids.key_ht	1.7	0.8	0.0	0.0	0.2	0.0	0.0	0.0	100	100
	ipc_ids.ipcs_idr.root_rt	2.7	5.3	-	-	0.0	0.2	-	-	100	96
Intel Xeon Gold 6530	posix_timers_hashtable	0.2	0.3	0.0	0.0	0.0	0.0	0.0	0.0	100	100
	futex_hash_table	0.0	4.3	0.0	4.3	0.0	0.0	0.0	0.0	100	100
	ipc_ids.key_ht	0.7	0.8	0.0	0.0	0.1	0.0	0.0	0.0	100	100
	ipc_ids.ipcs_idr.root_rt	0.0	0.9	-	-	0.0	0.0	-	-	100	100

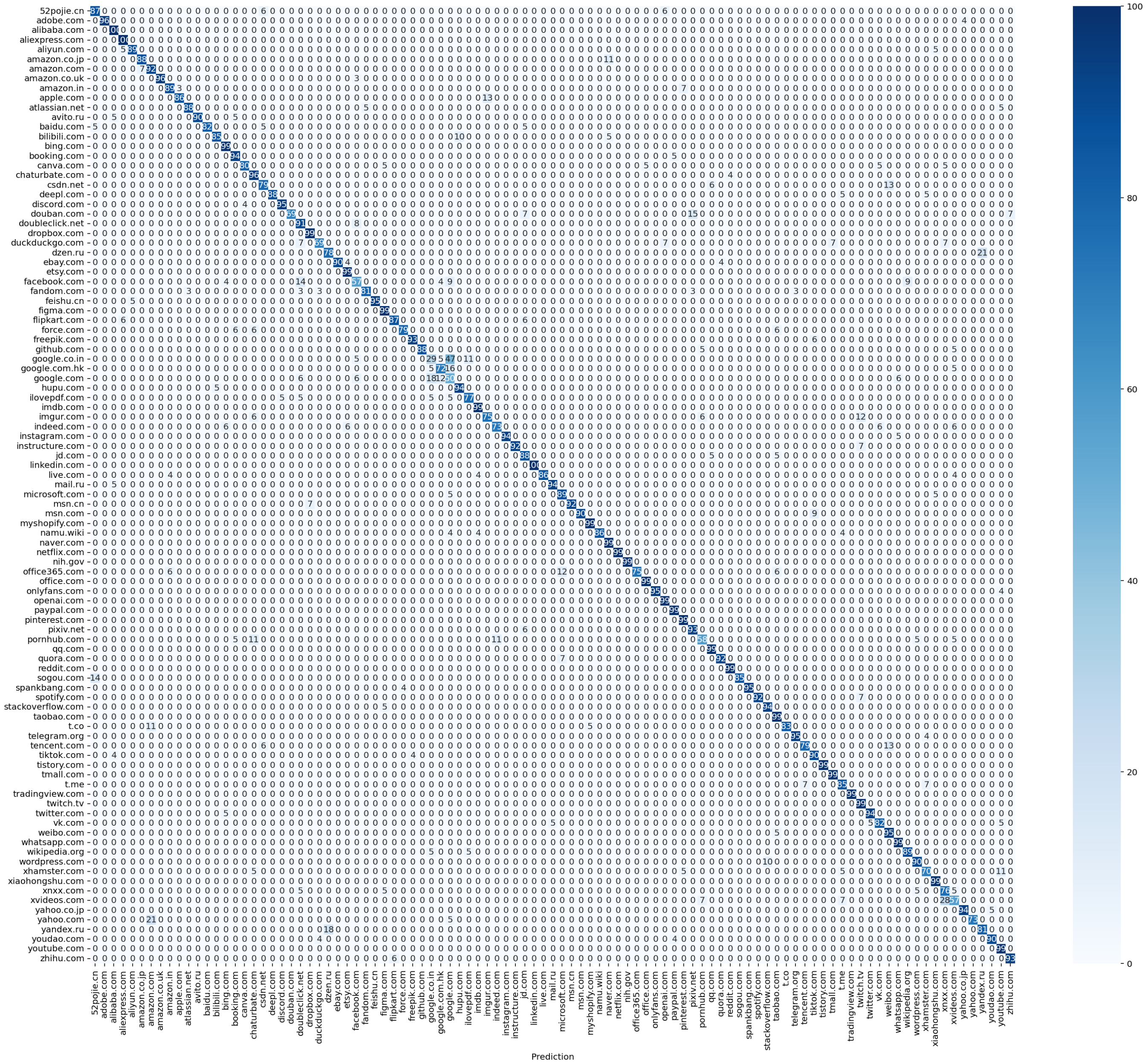


Fig. 21: Confusion matrix of our KernelSnitch website fingerprinting attack with an F_1 score of $89.5\,\%$.

Appendix A
Artifact Appendix

A. Abstract

KernelSnitch is a novel software-induced side-channel attack that targets kernel data container structures such as hash tables and trees. These structures vary in size and access time depending on the number of elements they hold, i.e., the occupancy level. KernelSnitch exploits this variability to constitute a timing side channel that is exploitable to an unprivileged, isolated attacker from user space. Despite the small timing differences relative to system call runtime, we demonstrate methods to reliably amplify these timing variations for successful exploitation.

The artifacts demonstrate the timing side channel and show the practicality of distinguishing between different occupancy levels. We provide a kernel module and execution scripts for evaluation. While our timing side channel is software induced, we recommend evaluation on hardware similar to ours (i.e., Intel i7-1260P, i7-1165G7, i7-12700, and Xeon Gold 6530) to reproduce similar results as in our paper. While the attacks should work generically on Linux kernels, we recommend to evaluate the artifacts on downstream Ubuntu Linux kernels v5.15, v6.5, or v6.8, as these are the versions we primarily evaluate. For the timing side channel, the evaluation shows that the occupancy level of data container structures can be leaked by measuring the timing of syscalls that access these structures.

B. Description & Requirements

1) Security, Privacy, and Ethical Concerns: The artifacts do not perform any destructive steps, as we only show the timing side channel to leak the occupancy level of data container structures and exclude the case studies, i.e., secretly transmitting data via a covert channel, leaking kernel heap pointers and monitoring user activity via website fingerprinting.

2) How to Access: We provide the source code (github) for performing the timing side channel.

3) Hardware Dependencies: While the timing side channel is software induced, one of the amplification methods depends on hardware buffers, i.e., the CPU caches. We have evaluated the Intel i7-1260P, i7-1165G7, i7-12700, and Xeon Gold 6530. We, therefore, expect similar results with similar processors.

4) Software Dependencies: While the attacks should generically work on Linux kernels, we recommend evaluating the artifacts on a downstream Ubuntu Linux kernel v5.15, v6.5, or v6.8. For reference, our primary evaluation system was Ubuntu 22.04 with generic kernels v5.15, v6.5, or v6.8.

One part of the artifact evaluation is to insert a kernel module that requires *root privileges*. This module is required to obtain the ground truth of the occupancy level of kernel data structures. We tested our kernel module with the downstream Ubuntu Linux kernels v5.15, v6.5, and v6.8. For other kernels that have different config files (or other downstream changes) our implemented module may not do what we intended. Specifically, in order to obtain the occupancy ground truth of the data structures, we redefined and reimplemented several structures and functions according to how they are implemented in the Ubuntu Linux kernel. We did this because several functions used to access kernel data structures are implemented as inline functions, e.g., `__rhashtable_lookup`, which prevented us from calling these functions directly. Another reason is that several structs, e.g., `msg_queue`, are defined in c files, which also prevents us from using these struct definitions. This module is required to obtain the ground truth of the occupancy level of kernel data structures.

5) Benchmarks: None.

C. Artifact Installation & Configuration

1) Installation: The installation required to perform the artifact evaluation works as following:

- Clone our github repository (github) to the `/repo/path` directory.
- Change directory to `/repo/path/modules`.
- Execute `make init` to build and insert the kernel module.
- Change directory to `/repo/path`.
- Select in `/repo/path/cacheutils.h` either the `INTEL` or `AMD` macro depending on your system.
- Execute `make` to build all experiment binaries.

2) Basic Tests: Testing the basic functionality works as following:

- Change directory to `/repo/path`.
- Execute `./basic_test.elf` should print `[+] basic test passed`.

D. Major Claims

We provide artifacts verifying the following claims:

(C1): KernelSnitch can distinguish different occupancy levels of the fixed-sized hash table `posix_timers_hashtable` by measuring the timing of `sys_clock_gettime`. This is proven by experiment (E1) whose approach is described in Section IV-A **Vulnerable Hash Tables** and results are illustrated in Section V-B **POSIX Timer Hash Tables** and Figure 8.

(C2): KernelSnitch can distinguish different occupancy levels of the fixed-sized hash table `futex_hash_table` by measuring the timing of `sys_futex`. This is proven by experiment (E2) whose approach is described in Section IV-A **Futex Hash Tables** and results are illustrated in Section V-B **Futex Hash Tables** and Figure 9.

(C3): KernelSnitch can distinguish different occupancy levels of the dynamically-sized hash table `ipc_ids.key_ht` by measuring the timing of `sys_msgget`. This is proven by experiment (E3) whose approach is described in Section IV-A **Vulnerable Hash Tables** and results are illustrated in Section V-B **IPC Hash Tables** and Figure 10.

(C4): KernelSnitch can distinguish two occupancy levels of the radix tree `ipc_ids.ipcs_idr.root_rt` by measuring the timing of `sys_msgget`. This is proven by experiment (E4) whose approach is described in Section

IV-B **Radix Tree** and results are illustrated in Section V-B **Radix Tree** and Figure 11.

(C5): KernelSnitch can distinguish occupancy levels of the red-black tree `hrtimer_bases.clock_base.active` by measuring the timing of `sys_timerfd_settime`. This is proven by experiment (E5) whose approach is described in Section IV-B **Red-Black Tree** and results are illustrated in Section V-B **Red-Black Tree** and Figure 12.

(C6): We demonstrate the improvements of using the struct-agnostic and hardware-agnostic amplification approaches with flushing the CPU caches and appending additional elements. This is proven by experiment (E6) whose approach is described in Section V-A and results are illustrated in Table I.

E. Evaluation

As described in Section V-B **External Noise**, the most dominant noise source is CPU frequency fluctuation. Therefore, perform the following experiments with as little background activity as possible to reproduce the figures from the paper. We even suggest to perform the experiments on an idle system with no other activity.

(E1): POSIX timer hash table experiment [10 human-seconds + 30 compute-seconds]:
[How to] Execute `./posix_timers_hashtable.elf <struct_agnostic_amp> <core> <file_name>`, with `<struct_agnostic_amp>` is a boolean which performs the experiment with/without structure-agnostic amplification, `<core>` pins the process to the specific core, and `<file_name>` stores the results in this file. For convenience, we provide the `eval_posix.sh` script which internally executes `posix_timers_hashtable.elf` with and without structure-agnostic amplification. To reproduce Figure 8, execute `./print_hist.py -f <file_name>`, where `<file_name>` is either `posix_ht_amp.csv` or `posix_ht_no_amp.csv`.
[Preparation] Do Section A-C1.
[Execution] `./eval_posix.sh` and `./print_hist.py -f <file_name>`, where `<file_name>` is either `posix_ht_amp.csv` or `posix_ht_no_amp.csv`.
[Results] `print_hist.py` should reproduce Figure 8.

(E2): Futex hash table experiment [10 human-seconds + 30 compute-seconds]:
[How to] Same as for (E1) but with `futex_hash_table.elf` and `eval_futex.sh`.
[Preparation] Do Section A-C1.
[Execution] `./eval_futex.sh` and `./print_hist.py -f <file_name>`, where `<file_name>` is either `futex_ht_amp.csv` or `futex_ht_no_amp.csv`.
[Results] `print_hist.py` should reproduce Figure 9.

(E3): IPC hash table experiment [10 human-seconds + 30 compute-seconds]:
[How to] Same as for (E1) but with `ipc_ids_key_ht.elf` and `eval_ipc_ht.sh`.
[Preparation] Do Section A-C1.
[Execution] `./eval_ipc_ht.sh` and `./print_hist.py -f <file_name>`, where `<file_name>` is either `ipc_ht_amp.csv` or `ipc_ht_no_amp.csv`.
[Results] `print_hist.py` should reproduce Figure 10.

(E4): IPC radix tree experiment [10 human-seconds + 30 compute-seconds]:
[How to] Same as for (E1) but with `ipc_ids_ipcs_idr_root_rt.elf` and `eval_ipc_rt.sh`.
[Preparation] Do Section A-C1.
[Execution] `./eval_ipc_rt.sh` and and `./print_hist.py -f <file_name>`, where `<file_name>` is either `ipc_rt_amp.csv` or `ipc_rt_no_amp.csv`.
[Results] `print_hist.py` should reproduce Figure 11.

(E5): Hrtimer red-black tree experiment [10 human-seconds + 30 compute-seconds]:
[How to] Similar as for (E1) but with `hrtimer_bases_clock_base_active.elf` and `eval_hrtimer_rbt.sh`, and `print_hrtimer.py`.
[Preparation] Do Section A-C1.
[Execution] `./eval_hrtimer_rbt.sh` and `./print_hrtimer.py hrtimer_rbt_no_amp.csv hrtimer_rbt_amp.csv`.
[Results] `print_hrtimer.py` should reproduce Figure 12.

(E6): Amplification improvement experiment [10 human-seconds + 10 compute-seconds]:
[How to] Execute `eval.py` prints similar results to Table I in ASCII form.
[Preparation] Do Section A-C1 and Experiments (E1-5).
[Execution] `./eval.py`.
[Results] `eval.py` should reproduce Table I.