On the Effectiveness of Intel's CAT as a Side-Channel Mitigation Technology

Pawel Wieczorkiewicz
Open Source Security Inc. USA
Email: wipawel@grsecurity.net

Rodrigo Branco and Ben Lee
Oregon State University
School of Electrical Engineering and Computer Science
Corvallis, OR, 97331, USA
Email: rodrigo@kernelhacking.com, benl@eecs.orst.edu

***Abstract*—Side-channels are created because contemporary systems share hardware components (e.g., caches) between different users and/or privilege domains. The paper by Yan *et al.* [1] introduced a novel side-channel attack against cache directory structures. They also discussed potential mitigations against their novel attack and suggested that a technique similar to Intel's Cache Allocation Technology (CAT) could be used to partition the directory structure, which hints that CAT as-is might not be enough to mitigate their proposed attack. The objective of this paper is to definitively answer the following questions: Is CAT partitioning an effective barrier against the cache directory attacks? And, are the Cache Directory Attacks possible in more constrained/realistic scenarios? To answer these questions, the work from [1] is reproduced, extended, and improved to run in a virtualized environment as a guest virtual machine (VM) attacking another guest (VM). This paper demonstrates that Intel's CAT is indeed not an effective mitigation against cache-based side-channels.**

***Index Terms*—Hardware Cache; Memory Layout; Cache-related Attacks**

I. Introduction

Cache side-channel attacks belong to a more general class of microarchitectural attacks [2]. The basic idea is to exploit the fact that the cache is a shared resource that can be used to leak information. This is typically done by measuring and profiling cache activity, i.e., cache hits and misses. By analyzing cache activity patterns, an attacker can infer secret-dependent code paths or sensitive access patterns of other processes. This information can then be used by an attacker to circumvent higher-level security mechanisms such as privilege separation. For example, cryptographic keys can be leaked, visited websites from sandboxed environments can be identified, Kernel Address Space Layout Randomization (KASLR) [3] can be broken, keyboard [4] and mouse activities can be spied on [5], and much more [3] [6] [7] [8] [9]. These attacks have been launched across many security boundaries, such as cross-cores, cross-VM, cross-users, and cross browser tabs. Therefore, cache side-channel attacks is a powerful threat.

In general, cache side-channel attacks work by measuring access timing information on a carefully manipulated/prepared cache. By analyzing and contextualizing these timing information, an attacker can reconstruct the secrets of a victim process. Mitigations to cache-based side-channel attacks oftentimes depend on code changes that adhere to specific constructs [10], but this has been proven to be quite hard to do even in security minded libraries [6] [11] [12] (such as cryptographic libraries). A more general mitigation is to remove the shared element, which can be performed in many different ways such as using separate physical machines, core isolation, and others. Intel provides a set of technologies called *L3 Cache Quality of Service (QoS) feature set* that includes the ability to partition the cache to avoid sharing it among different parts of the system. Such cache partitioning was proposed as a potential mitigation for cache-based side-channels [13]. Since then there has been no major announcements from cloud providers that includes the usage of this technology for the purpose of mitigating side-channels. This paper demonstrates that more recent attacks leverage the fact that many parts of a modern cache are shared and perform side-channel attacks on non-inclusive caches. These new attacks can also be used to bypass the partitioning provided by Intel's Cache Allocation Technology (CAT).

CAT was suggested as a potential cache-based side-channel mitigation in a paper by Intel researchers (with others) [13]. Meanwhile, the authors in [14] suggested that CAT is not an effective mitigation against some cache-based side-channels. They mentioned the work by Yan *et al.* [1] as the reason why CAT is not effective, which introduces a novel side-channel attack that leverages the directory design of Intel systems. Yan *et al.* also discussed mitigations against their novel attack and suggested that a technique similar to CAT could be used to partition the directory structure, which hints that CAT as-is might not be enough to mitigate their proposed attack (since CAT by itself does not partition the directory structure). However, another paper by the same authors, Yan *et al.* [15] included the proposal discussed in [13] (which is purely based on CAT) as a mitigation and pointed out that the main downside is that it requires a predefined split between protected/secured and non-secured domains. In another work, [7], the authors use other data structures (the TLB) as an attack vector (instead of the cache structure) because they claim that finding alternatives to the traditional caches is necessary because CAT is a mitigation to the traditional cache-based side-channels. That demonstrates that experts are, at least, unsure about the effectiveness of CAT.

Due to all the confusion and cross reference of previous research in regards of how effective the Intel's CAT is against the novel Cache Directory Attacks, this paper reproduces and

expands the previous cache directory attack in an environment with a guest virtual machine running on top of an open-source hypervisor (Xen). This setup is used to demonstrate that CAT is ineffective against such attacks, finally proving to finally prove that the technology should only be used to improve performance and not to prevent side-channels. Our findings have been shared with Intel and a confirmation of our conclusion has been received.

II. Background

This section presents the necessary cache concepts as well as Intel's CAT to better understand the rest of the paper.

A. Cache hierarchy

Caches can broadly be categorized by two properties: their relative distance from a given processing core and the type of data they cache. Current generation of CPUs typically implement a three-level cache hierarchy: Level 1 (separate data and instruction caches, L1I and L1D, respectively), Level 2 (L2), and Level 3 (L3, also called Last Level Cache or LLC for short).[1]

Each core has its own L1 cache structure, which on a typical Intel processor has a capacity of 64 KB and is split into L1I and L1D. The L1I feeds instructions to the CPU execution pipeline, whereas the L1D provides data operands for the instructions. The L1I/D caches are closest to the processing core and are also the fastest to access. For example, single digit latency (2-4 cycles) is expected for an L1 cache hit.

Each core typically has its own L2 cache, which is often referred to as a *unified cache* because it contains both data and instructions. When there is a cache miss on the L1 cache, the instruction/data is searched in the L2 cache. The size of L2 cache varies with CPU models. In the systems used for this research, each Intel Skylake CPU core has a 1 MB L2 cache. Cache hit latency for contemporary L2 caches is usually around one or two dozens of clock cycles.

The LLC is shared by all the CPU cores of a socket and has slightly different properties and organization than the L1 and L2 caches. The size of LLC usually depends on the number of CPU cores and is typically several times bigger than the capacity of all the L2 caches on a socket. The access latency of LLC is between 2x to 10x higher than for an L2 cache.

The LLC is organized into slices, where each *slice* can be viewed as a cache unit that can operate independently of other slices. The number of slices depends on the number of CPU cores.[2] The CPU uses proprietary hashing functions based on the physical address to calculate its target LLC slice. Even though the LLC is split into slices based on the number of cores, they are not private to the cores. This improves the distribution of traffic originating from different cores.

[1]There are differences in cache hierarchies among modern desktop, server, and mobile CPUs. The discussions in this paper are generalized as much as possible but are based on the server CPUs.

[2]In some cases, the number of cores plus one for integrated graphics.

B. PRIME+PROBE Attack

The *Prime+Probe* primitive allows an attacker to determine whether or not data is cached by simply measuring their access times. By comparing the cache state before and after the victim's activity, the attacker is able to infer what the victim was doing during its activity. *Prime+Probe* on a high level consists of the following three steps:

1) Initialize the cache to a known state (i.e., prime);
2) Either wait for or trigger the victim's activity; and
3) Compare the cache state to the known state of Step 1 (i.e., probe).

Priming the cache means that the attacker accesses a set of addresses that cause all the existing cache lines to be evicted. Such a set of addresses is called the *Eviction Set* (ES). The goal of this step is to bring the cache to an attacker-controlled state. After priming, the attacker either waits for or actively triggers accesses by the victim (e.g., by sending requests to the victim). During the probe step, the attacker again accesses all the addresses in the ES, but this time measuring their access times. A slow access time means that the victim has touched a cache line for that specific ES. In contrast, a fast access time means that the cache line is still in the original primed state. This timing information can leak the behavior of the victim process.

C. Intel CAT

Intel CAT [16] is a part of the *L3 Cache Quality of Service feature set* introduced in the *Haswell* server (HSX) CPUs. Initially, the set consisted of two features:

- Cache Management Technology (CMT);
- Cache Allocation Technology (CAT).

The CMT feature enables system software to monitor the LLC usage on a thread, process or virtual machine basis. Typically, the system software assigns a *Resource Monitoring ID* (RMID) to a thread, process or VM being monitored. Moreover, the same RMID can be shared with other threads, processes or VMs to form a monitoring group. The RMID is activated on a logical CPU by the system software using a dedicated Machine-Specific Register (MSR). Each access to the LLC updates the corresponding CMT hardware counters based on the currently active RMID. To reflect the state of the cache lines in the LLC, each cache line's metadata tag (which will be explained later) contains bits to store the corresponding RMID.

The CAT feature also uses the RMID mechanism in a similar way as CMT, but instead of monitoring usage it is used to restrict access to the subset of ways of the LLC on a per thread, process or VM basis. Essentially, CAT can be used to partition the ways of the LLC among the assigned threads, processes or virtual machines. Designed and implemented with performance implications in mind, the CAT feature allows for the enforcement of Class of Service (CoS) on each assigned thread, process, or VM, and isolates those belonging to different CoSs in the LLC.

Liu *et al.* suggested a few schemes where the CAT feature is used as a cache side-channel attack mitigation option [13]. All the proposed schemes are aim at isolating attackers from victims in the LLC by assigning a distinct set of ways to each of them. The authors claim that this prevents an attacker from triggering cache line evictions of a victim or observing cache usage patterns of other cache partitions.

D. Inclusive, Exclusive, and Non-inclusive caches

When a CPU core issues a load or store on a cacheable memory address and it misses in all levels of the cache hierarchy, the memory controller requests access to DRAM and fills the cache with the requested cache line. However, where the cache line ends up depends on whether the cache hierarchy is inclusive, exclusive, or non-inclusive.

In an *inclusive cache* hierarchy, a cache line present in the highest level (i.e., L1) of the cache implies that it is also present in the lower levels (i.e., L2 and L3) of the cache. Most Intel client platforms follow an inclusive cache design.

Inclusive LLCs have been a common implementation model for caches on Intel systems for many years (and still is for client platforms). Inclusive LLCs optimize snoop requests because a cache line that is not present in the LLC is also not present in the L1 or L2 caches and, as such, a snoop request does not have to be propagated to these structures. Therefore, the main burden of servicing snoop traffic is limited to the LLC structure. This property makes inclusive caches particularly prone to cache side-channel attacks on the LLC in a cross-core environment.

In an *exclusive cache*, a cache line present in the L1 cache implies that it is absent in the L2 and L3 caches. While some older microarchitectures such as AMD Athlon used exclusive caches, to the best of our knowledge no modern microarchitectures have fully exclusive caches; therefore, they are beyond the scope of our discussion.

In a *non-inclusive* cache, a cache line present in the L1 cache is not necessarily present in the L2 and L3 caches. AMD CPUs and Intel Skylake as well as some newer server microarchitectures use non-inclusive LLCs. A non-inclusive LLC can be viewed as a victim cache, which represents the last chance for a cache line to remain in the cache hierarchy before being evicted to DRAM. All cache lines evicted from the L2 cache end up in the LLC. But, the LLC does not necessarily have all the cache lines present in L1 or L2 cache. In fact, when a core requests a new cache line from DRAM, it is directly stored in its L1 cache and completely skips the LLC. The only way for a cache line to end up in the LLC is by getting evicted from per-core caches (i.e., L1/L2). As per the ”Attack Directories, not Caches” paper [1], there are certain cache line characteristics related to cache coherency that makes it more likely to remain longer in the LLC, e.g., cache lines in Shared state in contrast to ones in Exclusive state. In other words, the more CPU cores uses a cache line, the higher the chance that it will be needed in the near future. Thus, evicting it from the cache might be suboptimal.

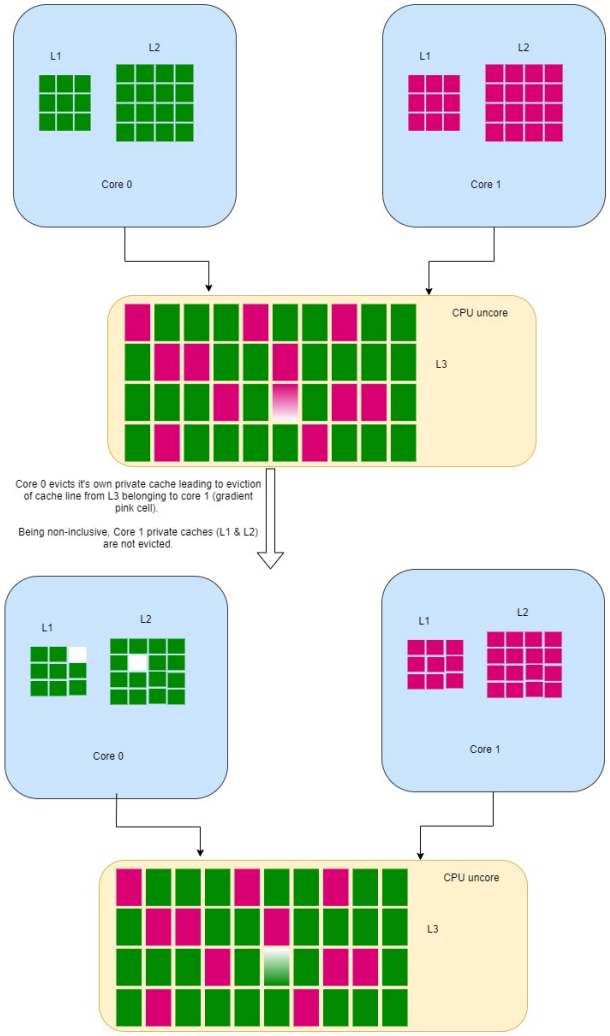

Fig. 1. Non-inclusive caches.

Non-inclusive caches have one particularly relevant implication for resistance against LLC based side-channel attacks. Since the LLC does not have to maintain inclusive property, an eviction from the LLC does not result in an eviction from the private caches. Thus, cross-core evictions from the lower level caches are not trivial.

Figure 1 depicts a non-inclusive cache design, where evictions from a core’s private caches end up in the LLC. In this case, Core 0 evicts a cache line from its private caches (indicated by white cells) and stored in the LLC thereby evicting one of the cache lines (cells with pink gradient fill) belonging to Core 1. However, this does not cause the corresponding cache line from the private caches of Core 1 to be evicted.

Note that cache coherency still has to be maintained among all the cache structures in a non-inclusive cache design, and the cache directory is used to track the state of each cache line. In a non-inclusive design, the directory implementation needs to reflect the state of the private caches. Therefore, the

directory array needs to be extended to also cover the cache lines present in L1/L2 private caches, but not present in the LLC. Yan *et al.* call this the *extended cache directory* [1]. They also describe an ingenious mechanism to trigger cross-core evictions for non-inclusive caches, and thereby create a possibility to implement cross-core cache side-channel attacks on newer CPU microarchitectures.

E. Directory Snooping

Directory Snooping addresses the scaling limitation of *Source Snooping* [17] with a more selective request propagation approach. Therefore, it is preferred on current many-core server systems. Directory Snooping maintains a cache directory structure that tracks whereabouts of cache lines in the system. Instead of broadcasting snoop requests, the directory structure is consulted and only L1/L2-caches of the cores that actually have the relevant cache line get the requests. This avoids a lot of unnecessary traffic on the shared bus.

The directory structure can be a part of the LLC array[3] [1]. The directory shares the same m and n (sets and ways) parameters and is sliced using the same hash function as the LLC.

As the directory maintains the coherence state of the entire cache hierarchy, it must always reflect the current and accurate state of all cache lines in the system. In other words, when a cache-line entry is evicted from the directory, the corresponding cache line must also be evicted from the cache hierarchy including private caches of the cores. This forms the basis of the cache directory attack, which will be discussed in later sections.

III. The Cache Directory Attack

This section takes a closer look at the novel cache side-channel attack proposed by Yan *et al.* [1]. As mentioned in Subsection II-B, one of the requirements for a successful cache-based side-channel attack is to properly prepare the cache state. In this particular case of a cache directory attack, the ability to trigger evictions is necessary to prepare the cache state. Depending on the underlying cache hardware properties (e.g., inclusive vs. non-inclusive) and the exact threat model (e.g., core local vs. cross-core and shared vs. no shared memory), triggering evictions may either be a relatively easy task or a nearly impossible undertaking. Since the discussion on the difference between inclusive vs. non-inclusive was provided in Subsection II-D, the rest of the paper assumes the LLC is non-inclusive.

When an attacker operates within the same core as its victim, it has direct access to the L1 and L2 caches. Thus, triggering evictions is just a matter of issuing enough accesses to prime the cache with its own data [18], e.g., using instructions such as `clflush`. This also works regardless of local vs. remote core setup because `clflush` evicts all cached state for a given memory address within a cache coherency domain.

[3]The exact implementation details are proprietary to the CPU vendors, so it is an assumption based on the commonalities presented next in the paper. This is also suggested by Yan *et al.*

A. Eviction Set Generation

For an attacker, generating evictions on a remote core is very challenging when the following properties exist:

- Non-inclusive LLC;
- No shared memory with a victim; and
- No sharing of CPU cores with a victim.

Prior to the discovery of the cache directory attack, an attacker had no known way to trigger evictions in the private caches (L1/L2) of the remote cores due to the non-inclusive property. An attacker could still trigger evictions within the LLC, but it has to first trigger eviction from the private caches of the remote core.

In order to understand the cache directory attack, the following terms are defined:

- Congruent Addresses (CAs) and Congruent Cache Lines - CAs are addresses whose cache set index bits match. In other words, cache lines with congruent addresses are always cached in the same cache set. Congruent Cache Lines are cache lines with congruent addresses;
- Congruent Set (CS) - CS is a collection of congruent addresses and typically a superset of a resulting Eviction Set (defined below);
- Eviction Set (ES) - ES is a minimal collection of precisely chosen memory addresses, which when accessed guarantees a congruent cache line eviction from the cache hierarchy. Typically, the ES memory addresses are congruent to one another and are congruent to the cache line to be evicted; and
- Eviction Address (EA) - EA is a member of the ES.

Figure 2 is similar to Figure 1 illustrating inclusive and non-inclusive caches except for the introduction of a new structure called *Cache Directory* in between the private caches of the cores and the LLC, which tracks the whereabouts of cache lines in the system for the purpose of cache coherence snooping. In Figure 2, Core 0 is the attacking core while Core 1 is the victim core. Again, all the green cells represent cache lines brought into the cache hierarchy via Core 0 initiated memory accesses, while pink cells represent memory accesses of Core 1. Shared cache line accesses are omitted from this discussion because the focus is on the hardest to evict cache lines (i.e., the non-shared ones). Empty white cells in the private caches of Core 0 represent cache lines that will be filled with data from the newly accessed addresses. To better understand the memory accesses in Core 0 and the corresponding evictions that occur in the Cache Directory, the affected cache lines are shaded with a gradient color representing the corresponding congruent cache line in the Cache Directory. As an example, a white cell with partial green gradient fill represents a pending memory access to a physical address is congruent with physical address of a cache line private to Core 0, and thus is tracked by the Cache Directory. Similarly, a green cell with a partial pink gradient fill represents a physical address of memory access that brought in a cache line to the private cache of Core 0 was congruent with physical address of cache line in the Cache Directory tracked for Core 1. Figure 2

Fig. 2. Non-inclusive caches with cache directory.

shows three different states of the private and shared caches of the cores depending on the memory accesses performed by Core 0:

- *Top left corner*: This represents the initial state. Core 0 performs two memory accesses to fill in two cache lines in its private caches (indicated by two empty white cells);
- *Bottom left corner*: In this figure, Core 0 performs a memory access to a physical address, which is congruent to a physical address with a green cache line in the Cache Directory. This results in the eviction of a cache line belonging to Core 0 in the Cache Directory, and thus those cache lines are also evicted from private caches of Core 0 (represented by two extra white cells). This is not an example of an attack but simply illustrates the fact that anything that gets evicted from the Cache Directory has to be evicted from the private caches of the core as well; and
- *Top right corner*: This state is achieved when Core 0 performs a memory access to a physical address that is congruent with a pink cache line in the Cache Directory, which is private to Core 1. Such a memory access causes a pink cache line from the Cache Directory to be evicted, and thus, it is also evicted from private caches (represented by white cells) of Core 1. This allows the attacking Core 0 to make an observation later using the Prime+Probe side channel primitive.

For systems that use the pseudo Least Recently Used (pLRU) replacement algorithm, a cache line eviction can be triggered when new cache lines are inserted into each available way of the corresponding set. A Cache Set Index can be derived directly from the physical address bits (e.g., bits 6-16 for an LLC with 2,048 sets). However, a modern system software operates on virtual addresses, which shares 12 least significant bits for 4 KB pages with the physical addresses. Therefore, the upper bits of the Cache Set Index are masked and are not available as shown in Figure 3. An attacker could try to use a timing side-channel to find sufficient number of congruent addresses to discover the hidden bits, but there is a more reliable way. Modern system software and architectures (e.g., x86-64) have support for large pages (i.e., instead of 4 KB virtual page to physical frame mappings, 2 MB and 1 GB page mappings can be used). The primary purpose of having a large page size is to improve performance by reducing TLB thrashing and shortening page-table walks. As shown in Figure 3, the lower 21 bits of a 2 MB virtual page address are identical to its physical page address. This is sufficient to give direct access to all the Cache Set Index bits. A 1 GB page has the lower 30 bits identical to its frame, but it may not be available depending on the system capabilities and settings.

To build an ES for a given cache line, an attacker first needs to allocate a set of 2 MB or 1 GB pages and construct an initial CS. The CS is seeded with all the congruent addresses for the victim cache line, i.e., it forms a reservoir of candidate memory addresses from which to build the final ES. To derive the address count for a minimal ES, the attacker needs to determine the associativity of the target cache and multiply

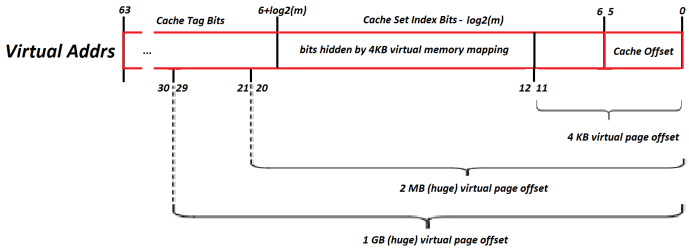

Fig. 3. Virtual memory address for 4 KB and huge pages.

it by the number of slices (i.e., number of cores). Since the process of constructing a minimal ES out of the initial CS is based on noisy and imprecise time measurements, the CS should contain an order of magnitude more addresses than the desired ES.

After the CS is built, the next step is to minimize it into an ES. The LLC is sliced into multiple partitions, each of which is n-way associative. Thus, the total LLC associativity is $n \times S_c$, where S_c is the number of slices/cores. This implies that the ES can be constructed based on the total LLC associativity. This is useful when the attacker cannot determine nor predict precisely which slice the victim's cache line will be stored. As a result, the ES will be large and contain at least $n \times S_c$ addresses, which negatively impacts the granularity of the attack and reduces its applicability. A better alternative is for the attacker to somehow determine the exact slice used by the victim cache line. This can be accomplished by applying probing, brute-force-like trial-and-error mechanism, or some more sophisticated reverse engineering technique. This will limit the address count of the ES to the associativity of a single slice.

The ES covering all the slices is less precise and has coarser granularity for sampling the LLC. The full-set eviction time is significantly longer than the time for a single slice set eviction. This is because the number of CPU cycles needed is a linear function of the number of slices/cores. As a result, using the total associativity of ES has limited practical applicability, especially on systems with many CPU cores. In addition, victim's frequency of operations is higher than the attacker's ability to evict the entire cache set, which make it much harder to sample and analyze. As a consequence, a single-slice based ES is usually desired because it allows for quick evictions and high sampling rates. Therefore, it gives an attacker a good insight into what its victim is doing. Thus, creating a set of ESs for each independent slice would allow for mounting a successful attack.

The attacker uses a special technique described in [1], which is detailed in Subsections III-C and III-D, to turn a single CS with enough addresses into a set of ESs, each of which has a minimal number of addresses needed to trigger evictions in the corresponding slice.

There is an additional complication caused by non-inclusive caches. This is due to the fact that memory accesses do not populate both the LLC and private caches all at once like they do in an inclusive cache design. In order to populate the LLC set with cache lines corresponding to EAs, the attacker core has to first access the cache lines corresponding to the ES to populate its L1 and L2 caches. Then, to evict these cache lines out of the private cache into the LLC, the attacker needs to access a small set of cache lines whose addresses are congruent to the EAs of the cache set in L2, but not congruent to the EAs of the cache set in L3. This requires the following two steps to prime the LLC:

1) Populate the private cache with ES cache lines using simple loads; and
2) Trigger collisions in the private cache to evict the cache lines belonging to the ES to the LLC.

To make the priming process reliable, it is important to trigger the private cache evictions without triggering collateral evictions in the LLC. The easiest way to achieve this is to choose a set of cache lines whose addresses are cache set congruent with the EAs in the L2 cache, but at the same time is not cache set congruent with the EAs in the LLC. Also, the ES for the private cache needs at least L2-associativity cache line members.

There is a simple way of constructing the ES for the private cache (which will be discussed in the next section). Since the number of cache sets in the L2 cache is usually at least $2\times$ smaller than in the LLC, enough EAs with the same L2 and LLC Cache Set Index bits can be found. As Figure 3 shows, L2 Cache Set Index usually requires fewer bits than the LLC set index. By toggling the address bits that are a part of the LLC Set Index but not L2 Cache Set Index, the cache line addresses that are in the L2 cache but are not LLC congruent can be easily obtained.

In summary, as the authors in [1] suggested, the private cache ES can be derived directly from the LLC ES by simply toggling the bits of EAs that are part of the LLC Set Index, but are not part of L2 Cache Set Index. Thus, depending on the private cache eviction policy, this technique uses a single LLC ES for both private cache and LLC evictions and eliminates the need to maintain multiple eviction sets.

B. Eviction-Set Generation for Inclusive Caches

Yan *et al.* [1] suggested a practical algorithm to construct a set of per-slice ESs based on the previous work by Liu *et al.* [19] for inclusive caches. The main part of the algorithm is a function called *check_conflict*. This function takes a collection of addresses, called candidate address set (CAS), and an address X, and returns true if an address in CAS conflicts with X. Since this function is the essence of the ES generation and the attack, it is essential to make its implementation noise-resistant and false-positive as well as false-negative free.

The function is shown in Listing 1 using a python-like notation:

Listing 1. *check_conflict* function for inclusive cache

```
def check_conflict(X, CAS):
    memory_access(X)
    for addr in CAS:
        memory_access(addr)
```

```
    time = memory_access(X)
    return time > LLC_threshold
```

`LLC_threshold` in Listing 1 denotes the average access time to main memory when cache lines are not present in the LLC. If X is present in the LLC, the access is faster. This value can be determined beforehand by applying some baseline measurements on a few random memory addresses. An average value of the results is usually a good approximation for `LLC_threshold`.

To construct the ES per slice, the CS is sampled for the first slice and minimized until a minimal ES is obtained. For the minimization, the *check_conflict* function is used to determine which addresses from the CS conflict with the addresses of the current ES. For practical reasons, the CS is purposely created to be larger than theoretically needed to guarantee that all per-slice ESs can be derived. It is also important to check during each iteration whether the current CS still conflicts and evict the reference address X (i.e., *check_conflict*(X, CS) returns true). This is because to find all the cache lines that are necessary for the address X eviction, the addresses from CS are being removed one-by-one until the resulting CS no longer evicts X. At this point, the last removed address is essential because it clearly contributed to the earlier eviction, and can be added to the current ES. This process is repeated until |CS| is smaller than associativity of a target cache or |ES| matches the desired associativity. Then, for each additional slice, CS is sampled again so that there are enough candidates to create the next ES, and also checked so that there is no interference with any previous ES. The process completes after ESs are generated for all the slices.

C. Eviction-Set Generation for Non-Inclusive Caches

In order to apply the same technique to non-inclusive caches, Yan *et al.* suggested a modification to the original *check_conflict* function [1]. As explained at the end of Section III-A, in order to evict a cache line from the LLC, the cache lines corresponding to EAs need to be first evicted from the private cache into the LLC. This can be achieved by reusing the same addresses from CAS and toggling their LLC Set Index bits to preserve L2 cache set congruency, but avoid LLC set congruency, as shown in Listing 2.

Listing 2. *check_conflict* function for non-inclusive cache

```
def check_conflict(X, CAS):
    memory_access(X)
    for addr in CAS:
        memory_access(addr)
    for addr in CAS[:L2_associativity]:
        memory_access(llc_cache_set_index_toggle(
            addr))
    time = memory_access(X)
    return time > LLC_threshold
```

However, this code requires that the CAS contains at least the same number of memory addresses as the associativity of the L2 private cache. The associativity of the LLC slice in a non-inclusive design is usually lower than the associativity of L2. But, as shown in Subsection III-D, this does not matter for the effectiveness of the cache directory attack.

By using 2 MB pages to reveal all the Cache Set Index bits of physical addresses and the *check_conflict* function shown in Listing 2, an ES can be reliably generated for non-inclusive caches. Our practical implementation of the cache directory attack is based on the work in [1] with a few approximations and heuristics developed to reduce system noise and make the algorithm easily portable.

The ES generation is essential for Prime+Probe cache side-channel attacks; therefore, the ESs for all the slices are generated. Nevertheless, an attacker still cannot trigger cross-core evictions from a private cache to the LLC and it also cannot observe any activity of the victim. Therefore, the key insight of [1] is to attack the directory structure itself and not the data array of the cache. As already mentioned, the directory based on non-inclusive cache design has to reflect the state of entire cache hierarchy at any point in time. When there is a conflict in the directory structure caused by a new cache line entering the cache hierarchy regardless of the cache level, another cache line has to be evicted from the cache hierarchy to make room for the new one. The eviction has to occur so that the cache directory can continue to reflect the global cache hierarchy state and maintain cache coherency. In other words, *the cache directory in a non-inclusive cache design is inclusive*.

D. The Cache Directory Attack Characteristics

The authors in [1] observed from their reference system with a single per-slice ES that it takes approximately 12 EA accesses to evict a cache line from the private cache to the LLC, which is less than the overall associativity of the L2 cache (i.e., 16 ways). Furthermore, their research showed that approximately 21 EA accesses are needed to evict a cache line from the LLC into DRAM, which is also fewer than the combined associativity of L2 and LLC (i.e., 16 ways + 11 ways = 27 ways).

They reasoned that whenever a cache conflict occurs in the private cache part of the directory, the corresponding directory entry is migrated to the LLC part of the directory. In addition, the corresponding cache line must also be evicted from the private cache to the LLC. Furthermore, whenever a cache conflict occurs in the LLC part of the directory, the directory entry is removed and the corresponding cache line is evicted from the LLC to DRAM.

Our proposed environment is more realistic and provides more constraints to an attacker, e.g., there is no shared memory between the attacker and the victim. The reason to add this additional constraint is that otherwise other side-channels would be possible and the mitigation scenario becomes unrealistic. Due to the added constraint, the attacker has a highly limited influence over the cache coherency states. Our proposed scenario replicates the realities of state-of-the-art Cloud-based deployments, and with the added constraints challenges the practicality of the original directory attack.

In summary, the key to cache directory attacks is to create conflicts in the directory to evict cache lines from the private caches to the LLC or even to DRAM. These evictions can

also be triggered cross-core. Furthermore, it is not necessary to evict cache lines all the way to DRAM. Instead, the attacker can prime the selected LLC set with its own cache lines corresponding to CAs and then leverage cache directory conflicts to trigger cross-core private-to-LLC evictions of the victim's cache line. If the victim's cache line has been cached, its eviction to the LLC will in turn evict one of the attacker's cache lines corresponding to CAs used for priming. This opens up an observable side-channel since the victim's cache line and all the attacker's cache lines except the one that has been evicted still remain in the cache hierarchy (i.e., not evicted to DRAM).

Evicting a cache line to DRAM and fetching it back is a costly operation (∼300-400 CPU cycles depending on the hardware). Private cache to LLC evictions are significantly faster (100-200 CPU cycles). As a consequence, using private cache to LLC evictions improves the cache sampling granularity. An attacker can use this approach because the cache directory associativity for the private cache is 12, while LLC associativity is 11 on the reference system. The way the LLC is primed and probed incurs a small penalty cost. The cache lines corresponding to CAs are first put into the private cache and then evicted to the LLC. In the worst case, the LLC priming requires LLC-associativity plus L2-associativity number of accesses (reference system: 11 + 16 = 27).

The conclusion by the authors in [1] is "the root cause of the vulnerability is that the directory associativity is smaller than the sum of the associativities of the caches that the directory is supposed to support", which is supported by the observations presented in this research. The authors also believe that the reasons behind this are scalability and performance (look-up latency and energy consumption), but also the cost of more expensive cache hardware.

Appendix V provides a detailed implementation of the attack leveraging the theory and techniques described until now.

E. Evaluation of Intel CAT Against the Cache Directory Attack

Yan *et al.* discussed a few potential countermeasures to the cache directory attack [1]. One solution is to completely eliminate cache directory conflicts by extending its associativity. This is a simple solution, but it is also impractical due to its hardware real estate cost and unknown performance implications. Thus, it is unlikely that the future designs would implement this approach. As an alternative, they also suggested the centralization of the cache directory at the cost of reduced performance and scalability. The last option is to revert back from the directory snooping scheme to the source snooping scheme as briefly discussed in Subsection II-E. However, to the best of our knowledge, such an option is not available in any of the currently implemented designs. The authors further stated that another potential solution would be to partition the cache directory entries among the cores in a manner similar to the way Intel CAT partitions the cache [1]. The same method was also proposed by academics working with Intel researchers in the CATalyst paper [13] against LLC side-channels. In other words, Intel CAT can be used to partition the cache and achieve isolation for the conflicting workloads [16].

F. Intel CAT is Not a Security Feature

This subsection demonstrates that Intel CAT is *not* an effective mitigation against the cache directory attack.

The cache directory attack was reproduced on a system with the CAT feature enabled and configured. In order to accomplish this, a few extra steps are added to the example mentioned earlier to enable CAT and partition the LLC into two distinct sets of ways – one assigned to the receiver (RX) and the other to the transmitter (TX).

Intel CAT partitioning was applied using the Xen's hypervisor feature Platform Shared Resource (PSR) Monitoring/Control using the boot command line parameter: *psr=cmt:1,rmid_max:255,cat:1,cos_max:255,cdp:1*. Listing 3 shows the hardware information for the test platform with the enabled CAT configuration.

Listing 3. hwinfo output

```
# xl psr-hwinfo
Cache Monitoring Technology (CMT):
Enabled         : 1
Total RMID      : 191
Supported monitor types:
cache-occupancy
total-mem-bandwidth
local-mem-bandwidth
Cache Allocation Technology (CAT):
Socket ID       : 0
L3 Cache        : 33792KB
CDP Status      : Enabled
Maximum COS     : 7
CBM length      : 11
Default CBM     : 0x7ff
Socket ID       : 1
L3 Cache        : 33792KB
CDP Status      : Enabled
Maximum COS     : 7
CBM length      : 11
Default CBM     : 0x7ff
Memory Bandwidth Allocation (MBA):
Socket ID               : 0
Linear Mode             : Enabled
Maximum COS             : 7
Maximum Throttling Value: 90
Default Throttling Value: 0
Socket ID               : 1
Linear Mode             : Enabled
Maximum COS             : 7
Maximum Throttling Value: 90
Default Throttling Value: 0
```

The RX virtual machine was configured to boot with the restricted CAT partition applied (CBM: 0x7f0 - 7 ways) as shown in Listing 4.

Listing 4. CBM configuration output for the Receiver VM

```
Socket ID       : 1
L3 Cache        : 33792KB
Default CBM     : 0x7ff
ID      NAME            CBM (code)      CBM (data)
0       Domain-0        0x7ff           0x7ff
1       Xenstore        0x7ff           0x7ff
499     receiver        0x7f0           0x7f0
```

Also, the TX boots with a restricted CAT partition (CBM: 0x00f - 4 ways) and is shown in Listing 5.

Listing 5. CBM configuration output for the Transmitter VM

```
Socket ID        : 1
L3 Cache         : 33792KB
Default CBM      : 0x7ff
ID       NAME             CBM (code)       CBM (data)
0        Domain-0         0x7ff            0x7ff
1        Xenstore         0x7ff            0x7ff
499      receiver         0x7f0            0x7f0
522      transmitter      0xf              0xf
```

As such, both RX and TX operate in the LLC on a fully distinct sets of ways.

When the proof-of-concept was executed without any CAT-related modifications, i.e., just with the enforced CAT partitioning, the ES generation algorithm did not work. This was not surprising since the CAT partitioning changed the effective associativity of the LLC for both RX and TX. The LLC associativity changed from 11 to 7 for RX, and from 11 to 4 for TX. Therefore, the algorithm needed to be adjusted in order to support the new values. With the correct associativity parameters, the ES generation worked correctly. Also, another interesting observation was made: the ES generation was 10-20% slower with the CAT partitioning, but both RX and TX start generating their ESs simultaneously because the partitioning protected them from mutually generated noise.

After the ES generation problem was resolved, the rest of the experiment was straight-forward. In the end, the CAT enforced LLC partitioning did not affect the effectiveness of the cache directory attack in any significant way. The covert channel was successfully established between the LLC-isolated RX and TX, and presented the same quality and bandwidth characteristics as the one without CAT partitioning.

This is not really surprising after all since CAT was designed as a QoS feature that enforces partitioning based on the ways of the cache data array. On the other hand, the cache directory attack does not depend on having access to all the ways of the cache data array, as is the case for other inclusive-cache-based side-channel attacks. In fact, the cache directory attack focuses primarily on triggering cross-core evictions from the private cache (i.e., L2) to the LLC in a way that affects attacker's portion of the LLC. In other words, an attacker controls the partition of the LLC and by triggering an eviction of a victim's cache line from victim's private cache into victim's LLC partition, forces victim to re-fetch evicted cache line back into the private cache. This may create contention in the cache directory, which in turn triggers another cache-directory-induced private cache to LLC eviction of one of the attacker's cache lines. Such event is observable by the attacker, and this is exactly what creates cross-core cache side-channel. This cache side-channel attack cannot be blocked by CAT partitioning.

IV. Conclusions

The cache directory attack was considered novel, but had some hard constraints on it. In addition, it was not tried against platforms that used Intel's CAT (cache partitioning) as a mitigation. In this work, the original cache directory attack is extended to cover a more practical scenario, considers realistic constraints to the setup, and CAT is used by the hypervisor in order to demonstrate that even a state-of-the-art cloud-based platform would still be vulnerable if it depended only on CAT as a cache-based side-channel mitigation between instances.

In light of the demonstrated attack, the CAT feature can no longer be considered as a viable lonely cache-based side-channel mitigation option, even on systems with non-inclusive cache designs.

A general protection mechanism against cache directory attack is still a subject of research. Protection options to defend secrets at known locations do exist though (such as, marking areas that contain secrets un-cacheable [20]).

As new attack classes are discovered, past mitigations have to be re-visited with the englightment of the new attack possibilities/capabilities. Sharing attack code, environments and methods are the ways for the 'offense to drive defense' [21].

Acknowledgment

We would like to thank Michael Kurth, Bjorn Doebel, Martin Pohlack for the insightful discussions on the topic. We would also like to thank Intel Corporation for confirming that CAT (Cache Allocation Technology) is not intended as a mitigation against LLC side-channels.

V. APPENDIX

VI. An implementation of a cache directory attack: covert channel

Our covert channel implementation follows the approach presented in [1] and consists of the following:

- CPU: Intel(R) Xeon(R) Platinum 8175M CPU @2.50GHz (Skylake)
- Hypervisor: Xen 4.12.1_04-3.6 as provided by SUSE Linux Server 15
- Guest OS: 'Kernel Testing Framework' (KTF) [22].

There are two parties communicating over the side-channel, receiver (RX) and transmitter (TX), and each is a single-slice based channel with approximately 7-16 cache line accesses. The following describes their functionalities. The full implementation will be made available as part of the KTF Project [22], which was used for the scenario.

General Receiver activities:

- Detect a slice used by TX for transmission.
- Create a minimal eviction set for each slice of the socket (24 slices in the test setup).
- Periodically probe each slice looking for sufficient number of 1's in the channel indicating the presence of TX transmission.
- After both RX and TX establish a common slice used for transmitting bits, start receiving data by priming the cache directory set with a predefined number of cache lines.

In order to successfully trigger a collision in the cache directory set and cause a victim cache line eviction, RX needs to issue 7 to 16 cache accesses priming the set. This is because the cache directory set associativity is 12. Depending on which way of the set the victim's cache line resides in, it was empirically observed that at least 7 cache accesses are needed to observe a signal. It was also observed that 16 cache accesses guarantee a reliable signal.

General Transmitter activities:

1) Establish communication channel.
2) Creates a minimal eviction set for a single slice.
3) Starts transmitting data over the covert channel.

In order to transmit a bit of information, the TX touches a predefined number of cache lines from the eviction set. Similarly to the RX counterpart, it has to issue between 7 and 16 cache accesses in order to saturate the cache directory associativity and trigger a collision.

In our basic implementation, the RX probes the channel with a resolution of 1000 cycles. The transmitter emits bit of data with a resolution of 6500 cycles. With the CPU frequency used in our test setup, the covert channel reaches an approximate bandwidth of 50 Kb/s as shown in Table I.

Listing 6 shows an example output for the traffic.

Listing 6. Example Traffic

TX Resolution (cycles)	Error Rate (%)	Bandwidth (Kb/s)
20000	3	15 Kb/s
10000	4	30 Kb/s
5000	40	60 Kb/s

TABLE I
COVERT CHANNEL BANDWIDTH.

```
00: 00000000 11111111 00000000 11111111 00000000
    11111111 00000000 11111111
01: 00000000 11111111 00000000 11111111 00000000
    11111111 00000000 11111111
02: 00000000 11111111 00000000 01111111 00000000
    01111111 00000000 01111111
03: 00000000 01111111 00000000 01111111 00000000
    01111111 00000000 01111111
04: 00000000 01111111 00000000 01111111 00000000
    01111111 00000000 01111111
05: 00000000 01111111 00000000 01111111 00000000
    01111111 10000000 01111111
06: 00000000 01111111 00000000 01111111 00000000
    01111111 00000000 01111111
07: 00000000 01111111 00000000 01111111 00000000
    11111111 00000000 11111111
08: 00000000 11111111 00000000 11111111 00000000
    11111111 00000000 11111111
09: 00000000 11111111 00000000 11111111 00000000
    11111111 00000000 11111111
10: 00000000 11111111 00000000 11111111 00000000
    11111111 00000000 11111111
11: 00000000 11111111 00000000 11111111 00000000
    11111111 00000000 11111111
12: 00000000 11111111 00000000 11111111 00000000
    11111111 10000000 11111111
13: 00000000 11111111 00000000 11111111 00000000
    11111111 00000000 11111111
14: 00000000 11111111 00000000 11111111 00000000
    11111111 00000000 11111111

Transmission bit stats #0:499 #1:461 #total:960 #
    errors:38
```

A. RX guest OS activities

In this subsection, the RX activities are enumerated with examples of the observed output of the proof-of-concept implementation.

1-) The RX boots on a CPU (e.g., a physical CPU20 on socket 1) and detects the details of the cache hierarchy by using the 'CPUID' instruction as shown in Listing 7.

Listing 7. Receiver Cache Hierarchy Output

```
KTF - Kernel Test Framework!

Cache info:
L1d: (non-inclusive) Line size: 64, Ways: 8, Sets:
    64, Slices: 1, Partitions: 1, Size: 32 KB
L1i: (non-inclusive) Line size: 64, Ways: 8, Sets:
    64, Slices: 1, Partitions: 1, Size: 32 KB
L2u: (non-inclusive) Line size: 64, Ways: 16, Sets:
    1024, Slices: 1, Partitions: 1, Size: 1024 KB
L3u: (non-inclusive) Line size: 64, Ways: 11, Sets:
    2048, Slices: 24, Partitions: 1, Size: 33792 KB
```

2-) The receiver obtains the number of slices for the LLC by obtaining the number of available CPU cores on a socket (e.g., socket 1)

3-) The RX allocates a reference cache line and selects a desired cache set to build the congruent and eviction sets for it. For this purpose, a large page must be used to provide access to all the cache set bits. A reference cache line belonging to a desired cache set is derived by adding the set number multiplied by the cache line size (64 bytes) to its memory address.

4-) The RX generates the congruent set using the algorithm in Listing 8.

Listing 8. *congruent_set* code

```
reference_cacheline += SET * CACHE_LINE_SIZE

memory_pool = allocate_hugepage(PAGE_SIZE_1G)
CS = set()
for cacheline in memory_pool:
        if llc_cache_set_index(cacheline) ==
            llc_cache_set_index(reference_cacheline)
            :
        CS.add(cacheline)
```

The congruent set consists of all the cache lines from a 1 GB memory pool, whose LLC cache set index matches with the LLC cache set index of the reference cache line.

5-) The RX derives the minimal eviction sets for all slices out of the congruent set prepared during Step 4.

Listing 9 shows the algorithm used in Step 5 with markings A, B and C to explain the specific logic.

Listing 9. *derive_sets* algorithm

```
ES = [set() for i in [1, 2, ..., SLICES]]
for slice_a in [1, 2, ..., SLICES]:
start_over:
    for slice_b in [1, ..., slice_a]:

        # A
        if triggers_conflict(reference_cacheline, ES
            [slice_b]):
            # Find another reference cache line
                belonging to a new LLC slice
            reference_cacheline += SET *
                CACHE_LINE_SIZE
            goto start_over

    ES[slice_a] = set()
    while not verify_eviction_set(ES[slice_a],
        reference_cacheline):              # B
        ES[slice_a] = build_eviction_set(CS,
            reference_cacheline)                # C
```

The code builds an individual eviction set for each slice using the *build_eviction_set* function (C). This is repeated for each slice until the resulting eviction set is verified to evict the reference cache line with the *verify_eviction_set()* function (B). For each new slice, a new reference cache line is selected by ensuring that it does not cause conflicts with any of the previously generated eviction sets for the previous slices (A).

The *build_eviction_set()* and *verify_eviction_set()* functions are shown in Listing 10 and Listing 11, respectively.

Listing 10. *build_eviction_set* function

```
def build_eviction_set(CS, reference_cacheline):
    ES = set()
    CS2 = set()
    if not triggers_conflict(reference_cacheline, CS
        ):                          # A
        return ERROR

    CS2 = copy(CS)
    for cacheline in CS2:

        # B
        if not triggers_conflict(reference_cacheline
            , CS2 - cacheline):
            ES += cacheline
        else:
            CS2 -= cacheline

    for cacheline in CS:

        # C
        if cacheline in ES:
            continue

        if triggers_conflict(cacheline, ES):
            ES += cl

    return ES
```

The *build_eviction_set* function first checks if the provided reference cache line triggers a cache set conflict with the congruent set (A). Next, all the cache lines from the congruent set that do not contribute to the conflict triggering between the congruent set and the reference cache line (B) are eliminated. As a final step, each cache line from the congruent set, which is not already in the eviction set, is checked against the triggering conflict with the current eviction set (C). The cache lines that do trigger a conflict were likely missed in the previous step (B), and are added to the eviction set.

Listing 11. *verify_eviction_set* function

```
def verify_eviction_set(ES, reference_cacheline):
    evict_cacheline(ES)
    time = memory_access(reference_cacheline)
    return time > LLC_threshold:
```

The *verify_eviction_set* function first attempts to evict the reference cache line using the eviction set and then checks if the memory access to the reference cache line is above the LLC access time threshold calculated earlier.

The *evict_cacheline* implementation is shown in Listing 12.

Listing 12. *evict_cacheline* function

```
def evict_cacheline(ES):
    for i in min(LLC_associativity, ES.count()):
                                                # A
        memory_access(ES[i])
        memory_barrier()

    for i in min(L2_associativity, ES.count()):
                                                 # B
        # Toggle the cache line address bit which is
             part of LLC cache set index,
        # but not L2 cache set index in order to
            make the cache line L2 set
        # congruent, but not LLC set congruent.
        toggle_bit(ES[i],  log2(LLC_SETS_COUNT) +
            log2(CACHE_LINE_SIZE) - 1)
        memory_access(ES[i])
        memory_barrier()
```

The *evict_cacheline* function first primes the entire LLC cache set using either the entire eviction set (if it contains fewer cache lines than the LLC associativity) or a sub-set of the eviction set (if it is sufficient to use the LLC associativity number of cache lines) (A). Therefore, a primed cache line will be evicted out of the LLC.

Next, the *evict_cacheline* function repeats with the earlier set (A), but this time modifying the cache line addresses in a way to obtain a cache line which is L2 set congruent, but not LLC set congruent. Therefore, by priming the L2 cache set with congruent addresses, the cache lines is evicted out of the L2 cache set.

6-) When all the eviction sets for all the slices are generated, the RX starts searching for the slice used by the TX.

This is achieved by periodically scanning each slice for a signal (i.e., high bits transmitted by the TX) and staggering the signal reads with a predefined (i.e., shared with the TX) time interval. As soon as the RX probing phase synchronizes with the TX sending phase, the RX will be able to distinguish slices with a very variadic occurrence of high bits (i.e., noise), from a slice with a stable signal emitted by the TX.

7-) After identifying the slice used by the transmitter, receiver starts receiving data.

The implementation of the *receive_data()* and *receive_bit()* functions are shown in Listing 13 and Listing 14, respectively.

Listing 13. *receive_data* function

```
def receive_data(ES):
    slice = find_active_slice(ES)

    while True:
        if receive_bit(ES[slice], INTERVAL):
            print('1')
        else:
            print('0')
```

Listing 14. receive_bit function

```
def receive_bit(ES, INTERVAL):
    bit = 0

    wait_cycles(INTERVAL)

    for cl in ES[:min(
        L2_to_LLC_directory_associativity, ES.count
        ())]:
       if memory_access(cl) > LLC_threshold:
           bit = 1
       # Keep priming for the next round
    return bit
```

The *receive_bit* function first sleeps for a predefined time interval (again shared with the TX) and then primes the cache directory set with its own cache lines measuring access time of each cache line. Upon detection of a cache miss (i.e., memory access time above the LLC threshold), a high bit transmission is assumed as the cause of the cache line eviction.

B. TX guest OS activities

1-) The TX boots on a CPU (e.g., CPU22 on socket 1) and detects the details of the cache hierarchy by using the 'CPUID' instruction as shown in Listing 15.

Listing 15. Transmitter Cache Hierarchy Output

```
KTF - Kernel Test Framework!

Cache info:
L1d: (non-inclusive) Line size: 64, Ways: 8, Sets:
    64, Slices: 1, Partitions: 1, Size: 32 KB
L1i: (non-inclusive) Line size: 64, Ways: 8, Sets:
    64, Slices: 1, Partitions: 1, Size: 32 KB
L2u: (non-inclusive) Line size: 64, Ways: 16, Sets:
    1024, Slices: 1, Partitions: 1, Size: 1024 KB
L3u: (non-inclusive) Line size: 64, Ways: 11, Sets:
    2048, Slices: 24, Partitions: 1, Size: 33792 KB
```

2-) The TX obtains the number of slices available on the LLC by obtaining the number of available CPU cores on a socket (e.g., Socket 1).

3-) The TX allocates a reference cache line and selects a desired cache set to build the congruent and eviction sets for it. For this purpose, a large page must be used to provide access to all cache set bits. A reference cache line belonging to the desired cache set is derived by adding the set number multiplied by the cache line size (i.e., 64 bytes) to its memory address.

4-) The TX generates a congruent set using the same algorithm used by the RX.

5-) The TX derives a minimal eviction set for a single slices (i.e., the slice that will be used for transmitting data) out of the congruent set prepared earlier.

6-) The TX starts transmitting data.

The *transmit_data* function is shown in Listing 16.

Listing 16. transmit_data function

```
def transmit_data(ES, slice_nr, data):
    es = ES[slice_nr]

    for bit in data:
        if bit == '1':
            transmit_one(es, TXD_DELAY)
        else if bit == '0':
            transmit_zero(TXD_DELAY)
```

After selecting a slice and obtaining a corresponding eviction set, the TX uses the *transmit_one* function for high bits and the *transmit_zero* function for low bits of information to be sent out. The *transmit_zero* function is an idle function with no cache activity and shown on Listing 17.

Listing 17. *transmit_zero()* function

```
def transmit_zero(TXD_DELAY):
    wait_cycles(TXD_DELAY)
```

The *transmit_one* implementation is presented in Listing 18.

Listing 18. *transmit_one()* function

```
def transmit_one(ES, TXD_DELAY):
```

```
start_time = get_cpu_ticks()

while get_cpu_ticks() - start_time < TXD_DELAY:
    for cl in ES[:min(
        L2_to_LLC_directory_associativity, ES.
        count())]:
        memory_access(cl)
```

In order to transmit a high bit, the *transmit_one* function triggers cache directory set collisions leading to the receiver cache lines being evicted from the L2 cache. This is done for a specified period of time, *TXD_DELAY*, which is measured in CPU ticks. Therefore, the RX is able to receive a high bit if it probes the cache for the specified amount of time and each time it detects a miss (i.e., eviction by the TX is detected). The low bit is received if the RX does not detect a cache miss for most of its sampling period. Note that occasionally a cache miss might be caused by unrelated factors, i.e., noise.

Our proposed covert channel implementation uses an approach where the TX and the RX are loosely synchronized. The TX sends the current round of information multiple times over a certain amount of time. If a '1' needs to be transmitted, the TX starts priming the selected cache set of a given slice. For a transmission of '0', the TX does not generate any activity. The RX measures the cache activity during the same time window.

In our example transmission in Listing 6, the TX repeatedly transmits a sequence of '0000111111110000' to observe two types of errors: the cache activity of the sender was not captured ('0' instead of '1') and missing transmissions (marked as '+'). In both cases, missing '1' is observed either as not measured or misclassified. The main source of error comes from the fact that the TX and the RX may occasionally desynchronize with each other, in which a few bits of information might become missing. In other words, the RX may mistakenly merge two transmission windows. For example, when the TX emits two instances of '1' and the RX may only measure one instance of '1'. In the other case where '10' is sent, the RX may measure both cache and idle activity and the confidence leans towards to receiving a '0'. Such errors could be detected with improvements in the algorithm, but are beyond the scope of this research.

REFERENCES

[1] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 888–904.

[2] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, pp. 1–27, 04 2018.

[3] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 191–205.

[4] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard., "Practical keystroke timing attacks in sandboxed javascript," European Symposium on Research in Computer Security, 2017, https://gruss.cc/files/keystroke_js.pdf.

[5] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 769–784. [Online]. Available: https://doi.org/10.1145/3319535.3363219

[6] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[7] B. Gras, K. Razavi, and H. e. a. Bos, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 955–972.

[8] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on AES," in *RAID*, 2014.

[9] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1406–1418. [Online]. Available: https://doi.org/10.1145/2810103.2813708

[10] Intel Corporation, "Guidelines for mitigating timing side channels against cryptographic implementations." [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html

[11] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '96. Berlin, Heidelberg: Springer-Verlag, 1996, p. 104–113.

[12] O. Aciicmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," Cryptology ePrint Archive, Report 2007/039, 2007, https://ia.cr/2007/039.

[13] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 406–418.

[14] R. Branco and B. Lee, "Cache-related hardware capabilities and their impact on information security," *ACM Comput. Surv.*, vol. 55, no. 6, dec 2022. [Online]. Available: https://doi.org/10.1145/3534962

[15] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: A secure directory to defeat directory side-channel attacks," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 332–345.

[16] Intel Corporation, "Improving real-time performance by utilizing cache allocation technology," 2015. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf

[17] J. Goodman, "Source snooping cache coherence protocols," 2009. [Online]. Available: https://parlab.eecs.berkeley.edu/sites/all/parlab/files/20091029-goodman-ssccp.pdf

[18] C. Percival, "Cache missing for fun and profit," 08 2009.

[19] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[20] K. Sun, K. Hu, and R. Branco, "A new memory type against speculative side channel attacks." [Online]. Available: https://github.com/rrbranco/Papers/blob/master/2019-A_New_Memory_Type_Against_Speculative_Side_Channel_Attacks.pdf

[21] B. Hawkes, "Project zero: Make 0days hard," 2015.

[22] K. T. F. Team, "Kernel test framework." [Online]. Available: https://github.com/KernelTestFramework/ktf