2019 IEEE Symposium on Security and Privacy

Attack Directories, Not Caches: Side-Channel Attacks in a Non-Inclusive World

Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, Josep Torrellas
University of Illinois at Urbana Champaign
{myan8, spraber2, gopired2, cwfletch, rhc, torrella}@illinois.edu

***Abstract*—Although clouds have strong virtual memory isolation guarantees, cache attacks stemming from shared caches have proved to be a large security problem. However, despite the past effectiveness of cache attacks, their viability has recently been called into question on modern systems, due to trends in cache hierarchy design moving away from inclusive cache hierarchies.**

In this paper, we reverse engineer the structure of the directory in a *sliced, non-inclusive* cache hierarchy, and prove that the directory can be used to bootstrap conflict-based cache attacks on the last-level cache. We design the first cross-core Prime+Probe attack on non-inclusive caches. This attack works with minimal assumptions: the adversary does not need to share any virtual memory with the victim, nor run on the same processor core. We also show the first high-bandwidth Evict+Reload attack on the same hardware. We demonstrate both attacks by extracting key bits during RSA operations in GnuPG on a state-of-the-art non-inclusive Intel Skylake-X server.

I. Introduction

Cloud computing on shared machines is now ubiquitous. Cloud hypervisors share physical hardware resources between concurrent guest Virtual Machines (VMs), giving each VM the impression that it owns the entire cloud. On the business side, dynamically sharing hardware between tenants is essential to keep cloud computing economically viable. However, in this environment, an obvious challenge is security. Fortunately, researchers and industry have developed a suite of techniques—e.g., the hypervisor-OS privilege hierarchy and hardware security extensions such as Intel SGX [1]—to provide virtual memory isolation between VMs and between processes within a VM.

Unfortunately, virtual memory isolation is insufficient to maintain privacy in the cloud. The very fact that users share the same physical machine leads to *shared resource attacks*, whereby the adversary can infer sensitive information by monitoring how the victim uses available hardware [2]–[6]. Of these, cache attacks [2], [7], [8] are arguably the most popular and powerful, enabling an adversary to learn fine-grain information regarding a victim process' memory access pattern—e.g., attacks can disclose encryption keys [8], user keystrokes [9], user web behavior [10], and more. Worse, these attacks can succeed even when the victim and adversary are run on different processor cores and do not share virtual memory by exploiting hardware characteristics of the last-level cache (LLC), which is shared across cores [8].

A. Challenges for Current Cache Attacks

Despite their past successes, the viability of LLC cache attacks has been called into question on modern systems due to recent trends in processor design. To start with, many prior attacks [7], [9]–[11] can be mitigated out of the gate, as virtualized environments are now advised to disable shared virtual memory between VMs [12].

Without sharing virtual memory with a victim, the adversary must carefully consider the *cache hardware architecture* in mounting a successful attack. This is where problems arise. First, modern cache hierarchies are becoming *non-inclusive or exclusive*. Prior LLC attacks without shared virtual memory (e.g., [8]) rely on LLCs being *inclusive*, as this gives adversaries the ability to evict cache lines that are resident in the victim's private caches. Non-inclusive cache behavior is significantly more complicated than that of inclusive caches (Section III). Second, modern LLCs are physically partitioned into multiple *slices*. Sliced LLCs notoriously complicate attacks, as the mapping between victim cache line address and cache slice is typically proprietary. Taken together, these challenges cause current LLC attacks to fail on modern systems (e.g., the Intel Skylake-X [13]).

Modern systems are moving to non-inclusive cache hierarchies due to the redundant storage that inclusive designs entail. Indeed, AMD servers have always used exclusive LLCs [11], and Intel servers are now moving to this design [13]. We expect the trend of non-inclusive caches to continue, as the cost of inclusive caches grows with core count (Section II).

B. This Paper: Modernizing Cross-Core Cache Attacks

In this paper, we design a novel cross-core cache attack that surmounts all of the above challenges. Specifically, our attack does not require the victim and adversary to share cores or virtual memory, and succeeds on state-of-the-art *sliced non-inclusive caches*, such as those in Skylake-X [13]. Our key insight is that in a machine with non-inclusive cache hierarchies, we can still attack the directory structure. Directories are an essential part of modern cache hierarchies, as they maintain tracking information for each cache line resident in the cache hierarchy.[1] Since the directory must track *all* cache lines, and not just cache lines in the LLC, it offers an attack surface

[1]This should not be confused with the "directory protocol" used in multi-socket attacks that assume shared virtual memory between the adversary and victim [11].

© 2019, Mengjia Yan. Under license to IEEE.
DOI 10.1109/SP.2019.00004

similar to that of an inclusive cache. Indeed, our work suggests that conflict-based LLC attacks (on inclusive, non-inclusive or exclusive cache hierarchies) should target directories, not caches, as directories are a homogeneous resource across these different cache hierarchy designs.

Contributions. To summarize, this paper makes the following contributions:

1) We develop an algorithm to find groups of cache lines that completely fill a given set of a given slice in a non-inclusive LLC (called an *Eviction Set*). This modernizes prior work on Eviction Set creation, which only works for sliced inclusive LLCs.

2) Using our Eviction Sets, we reverse engineer the directory structure in Skylake-X, and identify vulnerabilities in directory design that can be leveraged by cache-based side channel attacks.

3) Based on our insights into the directory, we present two attacks. The first is a Prime+Probe attack on sliced non-inclusive LLCs. Our attack does not require the victim and adversary to share cores or virtual memory. The second attack is a novel, high-bandwidth Evict+Reload attack that uses multi-threaded adversaries to bypass non-inclusive cache replacement policies.

4) We use our two attacks to attack square-and-multiply RSA on the modern Intel Skylake-X server processor. Both of these attacks are firsts: although prior work implemented an Evict+Reload attack on non-inclusive LLCs, it cannot attack RSA due to its low-bandwidth. Finally, we construct efficient covert channels for sliced non-inclusive LLCs.

II. Background

A. Memory Hierarchy and Basic Cache Structures

Modern high-performance processors contain multiple levels of caches that store data and instructions for fast access. The cache structures closer to the core, such as the L1, are the fastest, and are called higher-level caches. The ones farther away from the core and closer to main memory are slower, and are called lower-level caches. High-performance processors typically feature two levels of private caches (L1 and L2), followed by a shared L3 cache—also referred to as LLC for last-level cache.

The L1 cache is designed to be small (e.g., 32-64KB) and to respond very fast, typically within a few cycles. The L2 cache is slightly bigger (e.g., 256KB-1MB) and takes around 10-20 cycles. Finally, the LLC is designed to be large (e.g., several to tens of MBs) and has a latency of 40-60 cycles. The LLC latency is still much lower than the main memory access latency, which is on the order of 200-300 cycles.

A cache consists of the *data array*, which stores the data or code, and the *tag array*, which stores the high-order bits of the addresses of the data or code. The cache is organized in a number of *cache lines*, each one of size *B* bytes. The cache is typically set-associative, with *S* sets and *W* ways. A cache line occupies one way of a cache set. The set in which a cache line belongs is determined by its address bits. A memory address is shown in Figure 1.

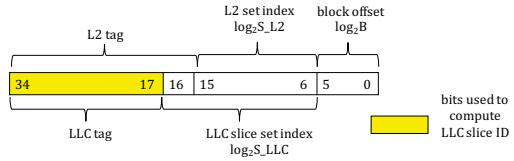


Fig. 1. Example of a memory address broken down into tag, index, and block offset bits. The actual bit field sizes correspond to the L2 and the LLC slice of the Intel Skylake-X system, as we reverse-engineered in Section V. We refer to the LLC slice set index as the LLC set index in this paper.

The lower $\log_2 B$ bits indicate the *block offset* within a cache line. The next $\log_2 S$ bits form the index of the set that the cache line belongs to. The remaining bits of the address form the tag. The tags of all the lines present in the cache are stored in the tag array. When a load or store request is issued by the core, the tag array of the L1 cache is checked to find out if the data is present in the cache. If it is a hit, the data is sent to the core. If it is a miss, the request is sent to the L2 cache. Similarly, if the request misses in L2 it is further sent to the LLC and then to main memory. Note that, generally, lower levels of the cache hierarchy have more sets than higher levels. In that case, cache lines that map to different LLC sets may map to the same L2 set, due to the pigeonhole principle.

B. Multi-Core Cache Organization

The LLC in a modern multi-core is usually organized into as many slices (partitions) as the number of cores. Such an organization, shown in Figure 2, is helpful to keep the design modular and scalable. Each slice has an associativity of W_{slice} and contains S_{slice} sets. S_{slice} is $1/N$ the total number of sets in the LLC, where N is the number of cores.

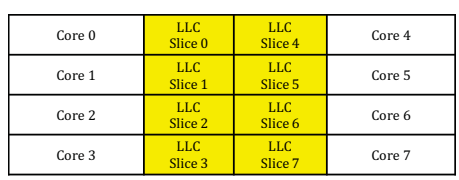


Fig. 2. Example of a sliced LLC design with 8 cores.

Processors often use an undocumented hash function to compute the slice ID to which a particular line address maps to. The hash function is designed to distribute the memory lines uniformly across all the slices. In the absence of knowledge about the hash function used, a given cache line can be present in any of the slices. Therefore, from an attacker's perspective, the effective associativity of the LLC is $N \times W_{\text{slice}}$. The hash function used in Intel's Sandybridge processor has been reconstructed in prior work [14], and found to be an xor of selected address bits. The slice hash function for the Skylake-X is more complex, as we find in Appendix B.

We now discuss two important cache design choices, and the trade offs behind them.

a) Inclusiveness: The LLC can be either *inclusive, exclusive,* or *non-inclusive* of the private caches. In an inclusive LLC, the cache lines in private L2 caches are also present in the LLC, whereas in an exclusive LLC, a cache line is never present in both the private L2 caches and in the LLC. Finally, in a non-inclusive LLC, a cache line in the private L2 caches may or may not be present in the LLC.

The inclusive design wastes chip area and power due to the replication of data. Typically, as the number of cores increases, the LLC size must increase, and hence the average LLC access latency increases [15], [16]. This suggests the use of large L2s, which minimize the number of LLC accesses and, therefore, improve performance. However, increasing the L2 size results in a higher waste of chip area in inclusive designs, due to the replication of data. The replication of data can be as high as the L2 capacity times the number of cores. Therefore, non-inclusive cache hierarchies have recently become more common. For example, the most recent server processors by Intel use non-inclusive caches [13], [17]. AMD has always used non-inclusive L3s in their processors [11].

b) Cache Coherence and Directories: When multiple cores read from or write to the same cache line, the caches should be kept coherent to prevent the use of stale data. Therefore, each cache line is assigned a state to indicate whether it is shared, modified, invalid, etc. A few state bits are required to keep track of this per-line state in hardware in the cache tag array or directory.

Two types of hardware protocols are used to maintain cache coherence—*snoop-based* and *directory-based*. The snoop-based protocols rely on a centralized bus to order and broadcast the different messages and requests. As the number of cores is increased, the centralized bus quickly proves to be a bottleneck. Therefore, most modern processors use a directory-based protocol, which uses point-to-point communication. In a directory-based protocol, a *directory* structure is used to keep track of which cores contain a copy of a given line in their caches, and whether the line is dirty or clean in those caches.

In an inclusive LLC design, the directory information can be conveniently co-located with the tag array of the LLC slice. Since the LLC is inclusive of all the private caches, the directory state of all the cache lines in any private cache is present in such a directory. The hardware can obtain the list of sharer cores of a particular line by simply checking the line's directory entry in the LLC. There is no need to query all the cores. However, the directory in a non-inclusive cache hierarchy design is more complicated, as we reverse engineer in Section V.

C. Cache-based Side Channel Attacks

Cache-based side channel attacks are a serious threat to secure computing, and have been demonstrated on a variety of platforms, from mobile devices [18] and desktop computers [2], [19] to server deployments [8], [10], [20]. Side channel attacks bypass software isolation mechanisms and are difficult to detect. They can detect coarse-grained information such as when a user is typing [18] down to much more fine-grained information such as a user's behavior on the web [21], and even RSA [8], [22] and AES [19], [20], [23]–[25] encryption keys.

Cache-based side channel timing attacks leverage timing differences in memory accesses to deduce information about a victim workload. There are many such attacks (e.g., [2], [7]–[11], [18]–[21], [25]–[33]). In this paper, we refer to *target address* as the address which, when accessed, reveals information about victim behavior.

As an example of cache-based attack, consider the square-and-multiply exponentiation algorithm, which is widely used in many encryption algorithms such as RSA and ElGamal. Algorithm 1 shows an implementation. In the process of computing its output, the algorithm iterates over exponent bits from high to low. For each bit, it performs a *sqr* and a *mod* operation. Then, if the exponent bit is "1", the algorithm performs a *mul* and a *mod* operation that are otherwise skipped. The target addresses can be the addresses of Line 3 and Line 6 in Algorithm 1. The instruction in Line 3 is executed as many times as the number of bits in the exponent. The instruction in Line 6 is only executed if the corresponding bit is set. If, at every iteration, an attacker can evict both instructions from the cache, and probe to see if the victim has brought them back into the cache, then the attacker can track the execution of loop iterations and reveal the full exponent.

Algorithm 1: Square-and-multiply exponentiation.

Input : base b, modulo m, exponent $e = (e_{n-1}...e_0)_2$
Output : $b^e \ mod \ m$

```
1  r = 1
2  for i = n − 1 downto 0 do
3      r = sqr(r)
4      r = mod(r, m)
5      if e_i == 1 then
6          r = mul(r, b)
7          r = mod(r, m)
8      end
9  end
10 return r
```

In the first phase of a cache-based attack, the attacker first identifies the target address. This can be done using source code analysis or through a cache template attack [9]. In the second phase, the attacker gathers timing information to carry out the attack. In this phase, the attacker follows three steps:

1) Evict the target address from the resource in which it is resident.
2) Wait a time period during which the victim may access the target address.
3) Measure the timing of certain accesses to determine the location of the target address.

Three existing attacks highlight the three-step process outlined above. These are listed in order of difficulty, with the last being the most difficult for an attacker.

Flush+Reload [7], [33]: This attack is easiest thanks to the use of shared memory between the attacker and the victim. Shared memory is possible on many platforms due to page-deduplication and shared libraries [7]. The attacker can simply flush the target address using the `clflush` instruction (step

1). After waiting for a period of time (step 2), the attacker re-accesses the target address and measures the latency (step 3). The attacker will expect a cache hit when accessing the memory flushed in step 1 if the victim has accessed the memory in the interval, and a miss otherwise. This attack is also referred to as a *flush-based* attack. This type of attack has serious limitations: it relies on `clflush`, and cloud platforms are now advised to turn sharing off, which disables this attack [34], [35].

Evict+Reload [18]: This attack also relies on shared memory for the reload operation, but does not flush or evict data using `clflush`, as not all architectures have a `clflush` instruction, and some defenses have suggested making `clflush` a privileged instruction [34] or disabling it all together [35]. In this case, the attacker uses cache conflicts to evict the target address (step 1). Specifically, the attacker accesses enough addresses mapped to the same cache set as the target address to evict the target address. The other steps are the same as Flush+Reload. Besides the additional complexity of creating such conflicts, this attack is otherwise the same as Flush+Reload, with an alternative flushing mechanism.

Prime+Probe [19]: This attack does not need shared memory. The attack steps are called prime, wait, and probe. In the prime step, the attacker evicts the target address from the cache by accessing a group of addresses mapped to the same cache set as the target address. During the wait step, the attacker waits. Finally, in the probe step, the attacker re-accesses the group of addresses used in the prime step, to measure victim activity. A cache miss in the probe step indicates that the victim has accessed the target address during the interval, and caused the eviction of one of the addresses accessed during the prime step from the cache.

Evict+Reload and Prime+Probe are referred to as *conflict-based* attacks due to fact that they exploit conflicts in cache structures.

D. Eviction Set

An *Eviction Set (EV)* is a collection of addresses that are all mapped to a specific cache set of a specific cache slice, and that are able to evict the current contents of the whole set in that slice. In a slice with W_{slice} ways, an eviction set must contain at least W_{slice} addresses to occupy all the ways and evict the complete contents of the set. We refer to *Eviction Addresses* as the addresses in an Eviction Set. In an inclusive LLC, both Evict+Reload and Prime+Probe use Eviction Sets to evict the target address from the private caches. Further, the probe operation in Prime+Probe measures the latency of accessing an Eviction Set to deduce the victim's activity.

III. The Challenge of Non-Inclusive Caches

Previous cross-core cache side-channel attacks only work for inclusive cache hierarchies (e.g., [8], [34]). In a non-inclusive cache hierarchy, attackers must overcome the two main challenges that we describe next. In this discussion, we assume that the `clflush` instruction is disabled (Section II) [36] and that shared memory between attacker and victim has been disabled [12].

A. Lack of Visibility into the Victim's Private Cache

In a non-inclusive cache hierarchy, an attacker running on a core seemingly cannot evict an address from another core's private cache — i.e., it cannot create an *Inclusion Victim* in the second core's cache. To see why, consider Figure 3, which shows a shared LLC and two private caches. The attacker runs on Cache 1 and the victim on Cache 0. The target line is shown in a light shade in Cache 0.

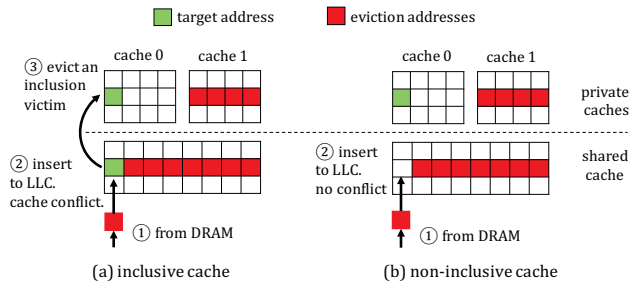


Fig. 3. Attempting to evict a target line from the victim's private cache in inclusive (a) and non-inclusive (b) cache hierarchies.

Figure 3(a) shows an inclusive hierarchy. An LLC set contains lines from the attacker (in a dark shade) plus the target line from the victim (in a light shade). The attacker references an additional line that maps to the same LLC set. That line will evict the target line from the LLC, and because of inclusivity, also from private Cache 0, creating an inclusion victim. The ability to create these inclusion victims on another cache is what enables cross-core attacks.

Figure 3(b) shows a non-inclusive hierarchy. In this case, the target line is in the victim's cache, and not in the LLC. Consequently, when the attacker references an additional line that maps to the same LLC set, there is no invalidation sent to Cache 0. The attacker has no way to create inclusion victims in the victim's cache.

B. Eviction Set Construction is Hard

In a later section, we will show that we perform Prime+Probe and Evict+Reload attacks in non-inclusive cache hierarchies using an Eviction Set (EV). However, the algorithm used to create an EV in inclusive cache hierarchies [8] does not work for non-inclusive hierarchies. Creating an EV in non-inclusive hierarchies is harder. The reason is that it is less obvious what memory accesses are required to reliably evict the target line, which is currently in the private cache, from the entire cache hierarchy.

To see why, consider Figure 4, which shows a private cache and two slices of the shared LLC. Victim and attacker run on the same core. Figure 4(a) shows an inclusive hierarchy. The target line is in the private cache and in one slice of the LLC. To evict the target from the cache hierarchy, the attacker only needs to reference enough lines to fill the relevant set in the corresponding slice of the LLC. This is because, as these lines fill the set, they will also fill the set in the private cache, and evict the target line from it. This is the EV, shown in a dark shade. The order and number of accesses to each of the lines

in the EV required to evict the target address is determined by the replacement algorithm used in the LLC slice.

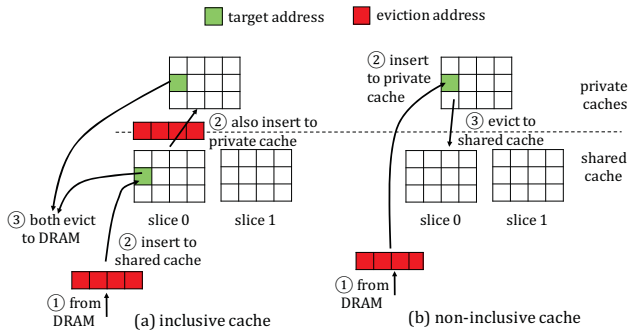


Fig. 4. Attempting to evict a target line in inclusive (a) and non-inclusive (b) cache hierarchies. Victim and attacker run on the same core.

Figure 4(b) shows a non-inclusive hierarchy. In this case, the target line is only in the private cache. As the core accesses the same dark cache lines as in Figure 4(a), the lines go first to the private cache, bypassing the LLC. The replacement algorithm used in the private cache will determine when the target line is evicted from the private cache into the LLC. When the target is evicted, it will go to one of the LLC slices, depending on the mapping of addresses to slices. Then, the core needs to evict enough lines into that LLC slice to create enough conflicts to evict the target line from the slice.

Overall, the order and number of accesses to each of the lines in the EV required to evict the target address is determined by multiple factors, including the replacement algorithm used in the private cache, the mapping of the line addresses to LLC slices, and the replacement algorithm used in the LLC slice. In the inclusive case, only the replacement algorithm in the LLC affects evictions.

We note that, in non-inclusive cache hierarchies, the replacement algorithms in the private caches and LLC slices can be quite sophisticated. What specific line is chosen as a replacement victim depends not only on the number and order of accesses to the lines, but also on the coherence state of the lines in the cache as well. Specifically, we have empirically observed that the replacement algorithm in the LLC slices tries to minimize the eviction of lines that are present in multiple private caches. This particular heuristic affects the ability to create an effective EV for Evict+Reload attacks, where lines are shared between attacker and victim.

C. Attack Overview

We address these two challenges in two novel ways. To handle the difficulty of creating EVs, we propose a novel way to create EVs for non-inclusive caches in Section IV. Using EVs and other techniques, we reverse engineer the Intel Skylake-X directory structure (Section V). This process reveals key insights into directory entry replacement policies and inclusivity properties. In particular, we derive conditions for when an attacker is able to use the directory to create inclusion victims in the private caches of a non-inclusive cache hierarchy. Based on our reverse engineering results, and the new EV construction methodology, we design effective "Prime+Probe" and "Evict+Reload" attacks in non-inclusive caches hierarchies in Section VI.

IV. Constructing Eviction Sets

In this section, we present an EV construction algorithm for non-inclusive caches. Recall that an EV is a collection of memory addresses that fully occupy a specific cache set of a specific LLC slice. This is a core primitive that we use to reverse engineer the directory (Section V) and later complete our attacks (Section VI).

Liu et al. [8] proposed an EV construction algorithm for sliced inclusive caches. However, it does not work for non-inclusive caches. The reason was discussed in Section III: line eviction from the cache hierarchy is less predictable because it depends on the L2 replacement algorithm, the mapping of line addresses to LLC slices, and the replacement algorithm used in the LLC slices. We fix these issues by developing a new implementation for `check_conflict`, an important subroutine used in Liu et al., that works on non-inclusive caches. We present evaluation results to demonstrate the effectiveness of our algorithm in Section VII-A.

A. The Role of check_conflict in EV Construction Algorithm

The EV construction algorithm by Liu et al. [8] uses a function that we call `check_conflict(Address` x`, Collection` U`)` (called `probe` in [8]), shown in Algorithm 2. This function checks if the addresses in Collection U conflict with x in the LLC. The function should return $true$ if U contains W_{slice} or more addresses, which are mapped to the same slice and set as x. The function should return $false$ otherwise. The EV construction algorithm works only if this function has very low false positive and false negative rates.

Algorithm 2: Baseline check_conflict for inclusive caches.

```
Function check_conflict (x, U):
    access x
    for each addr in U do
        access addr
    end
    t = measure time of accessing x
    return t ≥ LLC_miss_threshold
end
```

To see why these requirements are important, consider how `check_conflict` is used in the EV construction algorithm. The high-level idea is to start with a collection U known to conflict with x in an LLC slice. Then, one removes an address y from the collection U and obtains a new collection $U' = U - y$. If the conflict with x disappears when checking against U', then we know that y must contribute to the conflict. In such a case, y is considered to be in the EV for x. Clearly, the operation needs low false positive and false negative rates to precisely observe the disappearance of conflicts. Appendix A provides more details on how the algorithm is used, and why a high-accuracy implementation is important.

B. New check_conflict Function

We first discuss why the `check_conflict` function designed by Liu et al. [8] has a high false negative rate when applied naïvely to non-inclusive caches. We then show how the function can be modified to work in non-inclusive caches. In the following discussion, we assume that all the addresses in U have the same LLC set index bits as x.

Baseline `check_conflict` [8]. In Algorithm 2, the base function first accesses the target address x, ensuring that the line is cached. It then accesses all the addresses in U. If a later access to line x takes a short time, it means that the line is still cached. Otherwise, it means that the line has been evicted out of the cache due to cache conflicts caused by U. Thus, the access latency can be used to determine whether U contains enough addresses to evict x.

When applied to non-inclusive caches, this function has a high false negative rate. Specifically, when U contains enough addresses that, if they all were in the LLC, they would evict x, the function is supposed to return $true$. However, it may return $false$. To see how this false negative happens, consider a minimal U, which has exactly W_{slice} addresses mapped to the same LLC slice as x. On non-inclusive caches, when accessing U, some of these W_{slice} lines may remain in L2 and never be evicted from the L2 into the LLC. Hence, these addresses do not have a chance to conflict with x in the LLC, and x is not evicted, resulting in a false negative. Moreover, since the replacement algorithm of L2 is neither LRU nor pseudo-LRU, simply accessing U multiple times does not guarantee a small false negative rate, as we validate in Section VII-A.

Naïve New `check_conflict`. To reduce the false negative rate, we need to flush all the lines in U from the L2 to the LLC. It would be convenient if we had a special instruction to do so, but such an instruction does not exist in x86. Hence, we leverage L2 conflicts to achieve the flush effect.

We create an extra collection of addresses, called $L2_occupy_set$, which contains W_{L2} addresses mapped to the same L2 set as U. When accessed, $L2_occupy_set$ forces all lines in U to be evicted to the LLC. Our modified `check_conflict` function is shown in Algorithm 3. After accessing x and all the addresses in U as in the base function (line 2-5), the addresses in $L2_occupy_set$ are accessed (line 6-8). In this way, every line in U gets evicted to the LLC slice where x is, and we can significantly reduce the false negative rate.

Algorithm 3: New check_conflict for non-inclusive caches.

```
1  Function check_conflict (x, U):
2      access x
3      for each addr in U do
4          access addr
5      end
6      for each addr in L2_occupy_set do
           // this evicts U from L2 to LLC
7          access addr
8      end
9      t = measure time of accessing x
10     return t ≥ LLC_miss_threshold
11 end
```

However, this naïve approach has a high false positive rate. A false positive can occur when U does not contain enough addresses to evict x from the LLC slice, but with the help of some addresses in $L2_occupy_set$ that end up getting evicted to the LLC, they evict x from the LLC. In this case, the function is supposed to return $false$, but it returns $true$.

Reliable New `check_conflict`. To reduce the false positive rate in the naïve new `check_conflict` function, we need to make sure accesses to $L2_occupy_set$ do not interfere with the conflicts between U and x in the LLC. We can achieve this by leveraging the one-to-many set mapping relationship between L2s and LLCs.

For a reliable design, we select $L2_occupy_set$ such that its addresses are mapped to the same L2 set as addresses in U, but to a *different* LLC set than used by addresses in U (and x). As mentioned before, upper level caches like the L2 contain fewer cache sets than lower level caches like the LLC (Section II-A). For example, in Skylake-X, the L2 has 1024 sets, while an LLC slice has 2048 sets. Correspondingly, the L2 uses 10 bits (bits 6-15) from the physical address as the set index, while the LLC slice uses 11 bits (bits 6-16). Therefore, $L2_occupy_set$ can be constructed by simply flipping bit 16 of W_{L2} addresses in U. Such addresses can be used to evict U from the L2 but do not conflict with U in the LLC.

In summary, we design a reliable `check_conflict` function with both low false positive rate and low false negative rate. This function can be used in the EV construction algorithm of Liu et al. [8] to construct an EV for non-inclusive caches. We evaluate the effectiveness of the function in Section VII-A. For independent interest, we use our EV creation routine to partially reverse engineer the Skylake-X slice hash function in Appendix B.

V. Reverse Engineering the Directory Structure in Intel Skylake-X Processors

We leverage our EV creation function to verify the existence of the directory structure in an 8-core Intel Core i7-7820X processor, which uses the Intel Skylake-X series microarchitecture. We also provide detailed information about the directory's associativity, inclusivity, replacement policies (for both private and shared data), and interactions with the non-inclusive caches. These insights will be used for the attack in Section VI. Skylake-X is a server processor for cloud computing and datacenters. A comparison of the cache parameters in this processor with previous Skylake series processors is listed in Table I.

	Skylake-S	Skylake-X/Skylake-SP
L1-I	32KB, 8-way	32KB, 8-way
L1-D	32KB, 8-way	32KB, 8-way
L2	256KB/core 16-way, inclusive	1MB/core 16-way, inclusive
LLC	2MB/core 16-way, inclusive	1.375MB/core 11-way, non-inclusive

TABLE I
Cache Structures in Skylake Processors.

Relative to the older Skylake-S processor, the Skylake-X/SP LLC is non-inclusive and, correspondingly, Skylake-X/SP can

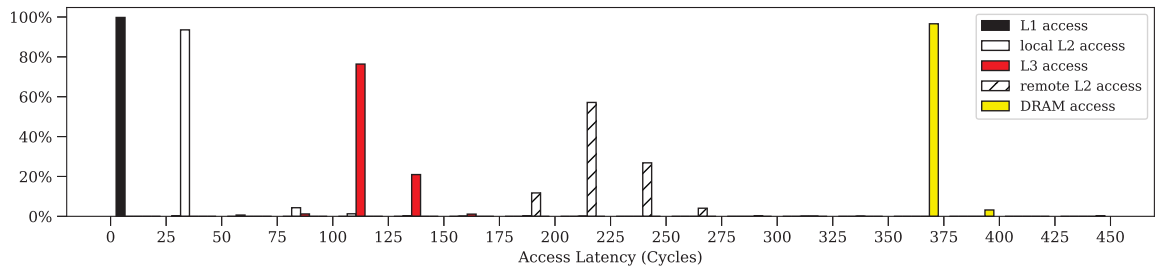


Fig. 5. Latency of a cache line access when the line is in different locations in the Intel Skylake-X cache hierarchy.

support larger L2 caches relative to the LLC. The L2 in Skylake-X/SP grows to 1 MB per core, which is 4 times larger than before, while the LLC size shrinks from 2 MB per core to 1.375 MB per core. The associativity in the LLC slice is also reduced from 16-way to 11-way.

A. Timing Characteristics of Cache Access Latencies

We first conduct a detailed analysis of the timing characteristics of the cache access latencies on Skylake-X. This information can be used to infer the location of a specified cache line, and is useful in reverse engineering the directory structure.

For each cache location, we measure the access latency by using the `rdtsc` instruction to count the cycles for one access. We use the `lfence` instruction to make sure we get the timestamp counter after the memory access is complete as suggested in [37]. Thus, all the latencies presented include delays introduced by the execution of `lfence`.

Figure 5 shows the distribution of latencies to access lines in different cache layers. For each cache layer, we perform 1,000 accesses. The latency is for accessing a single cache line. From the figure, we see that L1 and local L2 access latencies are below 50 cycles. An LLC access takes around 100 cycles, and a DRAM access around 350 cycles.

A remote L2 access occurs when a thread accesses a line that is currently in another core's L2. From the figure, a remote L2 access takes around 200 cycles, which is shorter than the DRAM latency. We leverage the difference between the remote L2 latency and the DRAM latency in the "Evict+Reload" attack to infer the victim's accesses.

B. Existence of the Sliced Directory

Our first experiment is designed to verify the existence of a directory and its structure. In each round of the experiment, a single thread conducts the following three steps in order:

1) Access target cache line x.
2) Access a set of N eviction addresses. In a Reference setup, these are cache line addresses that have the same LLC set index bits as x, and can be mapped to different LLC slices. In a SameEV setup, these are cache line addresses that have the same LLC set index bits as x, and are mapped to the same LLC slice as x.
3) Access the target cache line x again while measuring the access latency.

Generally, step 2 is repeated multiple times (100 times) to avoid the noise due to cache replacement policy in both Reference and SameEV setups. The medium access latency over 1,000 measurements for step 3 is shown in Figure 6, as a function of the number of eviction addresses accessed. We validated that the experiment results are consistent for x mapped to different slices and sets.

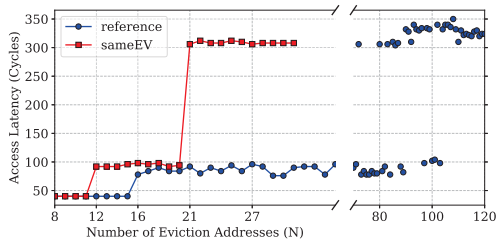


Fig. 6. Target line access latency as a function of the number of eviction addresses, in experiments to verify the existence of the directory.

According to the timing characteristics in Figure 5, we know that latencies around 40, 100 and 300 cycles indicate that the target line is in local L2, LLC and DRAM, respectively. In the Reference configuration, if 16 or more lines are accessed in step 2, the target line is evicted from L2 to LLC. These evictions are caused by L2 conflicts because the L2 associativity (W_{L2}) is 16. Later, we start to observe that the target line is evicted to DRAM when more than 75 addresses are accessed in step 2. This number is less than 104 ($W_{L2} + N_{slice} \times W_{slice}$) because the hash function makes the addresses used in the experiment distribute unevenly across the different slices.

In the SameEV setup, we observe L2 misses when 12 cache lines are accessed in step 2, before reaching the L2 associativity. Moreover, the target line is evicted out of the LLC when 21 lines are accessed, even though the L2 cache and one LLC slice should be able to hold up to 27 lines ($W_{L2} + W_{LLC}$). The difference between 12 and 16 (L2 case), and between 21 and 27 (LLC case) indicates that there exists some bottleneck, other than the L2 and LLC slice associativity. This indicates the presence of some set-associative on-chip structure, where conflicts can cause L2 and LLC evictions. The structure's associativity seen for L2 lines is 12, and the associativity seen for L2 and LLC lines is 21.

In addition, we know that the structure is sliced and looked-up using the LLC slice hash function. Notice that addresses conflict in this structure only if they are from the same LLC slice, as in the SameEV configuration. Addresses from different LLC slices do not cause conflicts in this structure, as seen in the Reference configuration.

Finally, we can also reverse engineer the number of sets in each slice of the structure by testing which bits are used to determine the set index. We analyzed the EVs that we derived for this structure, and found that the addresses in a given EV always have the same value in bits 6-16. The addresses that belong to the same EV are mapped to the same set and slice, and should share the same set index bits and slice id. Since none of the bits 6-16 are used for slice hash function (see Appendix B), we know that these bits are used as set index bits. Hence, the structure has 2048 sets, the same number of sets as an LLC slice.

> 1. There exists a set-associative structure that operates alongside the non-inclusive caches. Contention on this structure can interfere with cache line states in both the L2 and LLC.
> 2. The structure is sliced and is looked up using the LLC slice hash function. Each slice has the same number of sets as an LLC slice.
> 3. The associativity of the structure for L2 lines is 12; the associativity of the structure for L2 and LLC lines is 21.

C. Inclusivity and Associativity for Private Cache Lines

We use the term *Private* cache line to refer to a line that has been accessed by a single core; we use the term *Shared* cache line to refer to a line that has been accessed by multiple cores. We observed that the non-inclusive LLC cache in Skylake-X behaves differently towards private and shared cache lines.

We conduct a two-thread experiment to reverse engineer the inclusivity of the set-associative structure that we found and the cache for private lines. The two threads are pinned to different cores.

1) Thread A accesses target line x.
2) Thread B accesses N eviction addresses. The addresses are selected for the Reference and SameEV setups as in the previous experiment.
3) Thread A accesses target line x again and measures the access latency.

The access latencies in step 3 are shown in Figure 7, as a function of the number of eviction addresses.

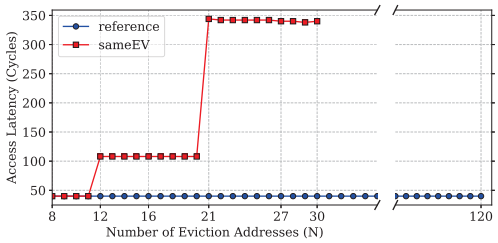


Fig. 7. Target line access latency as a function of the number of eviction addresses in experiments to analyze the inclusive property for private lines.

In the Reference setup, the target line in thread A's private L2 is never evicted. Due to the non-inclusive property of the LLC, thread B is unable to interfere in thread A's private cache state, and hence loses any visibility into thread A's private cache accesses. However, in the SameEV setup, the target line is evicted from the L2 to LLC when 12 lines are accessed by thread B, and it is further evicted to DRAM when thread B accesses 21 lines.

This experiment shows that the structure is shared by all the cores, and that conflicts on this structure can interfere with L2 and LLC cache states. In particular, the SameEV configuration shows that we *can create inclusion victims across cores*, since lines in the L2 can be evicted due to the contention on the structure. We can safely conclude that the structure is inclusive to all the lines in the cache hierarchy, including L2 and LLC. This characteristic can be leveraged by an attacker to gain visibility into a victim's private cache and build Prime+Probe attacks. Moreover, the experiment also confirms the same associativity across cores as the last experiment.

> 4. The structure is shared by all cores.
> 5. The structure is inclusive, and contention on the structure can cause inclusion victims across cores.

D. Inclusivity and Associativity for Shared Cache Lines

To reverse engineer the inclusivity and associativity of the structure for shared cache lines, we use 2 or 3 threads in different modes.

1) Thread A and B both access the target cache line x to ensure the line has been marked as shared by the processor.
2) In *1evictor* mode, thread B accesses N cache line eviction addresses; in the *2evictors* mode, thread B and C access N cache line eviction addresses to put those lines into the Shared state. In both modes, different eviction cache lines are selected for Reference and SameEV setups as in our previous experiments.
3) Thread A accesses the target line x again and measures the access latency.

From this discussion, in the 1evictor mode, only x is in the shared state; in the 2evictors mode, both x and the N eviction lines are in the shared state. Figure 8 shows the access latencies for step 3.

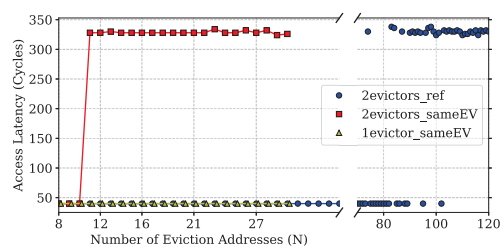


Fig. 8. Target line access latency as a function of the number of eviction addresses in experiments to analyze the inclusive property for shared lines.

In the 1evictor_sameEV setup, the shared target line is never evicted out of thread A's private L2. However, we showed in Figure 7 that this pattern does cause remote L2 evictions of *private* lines. Comparing the two cases, we can infer that the cache coherence state–namely whether a line is shared or not–plays a role in the cache line replacement policy. The replacement policy prefers not to evict shared cache lines.

In the 2evictors_sameEV setup, threads B and C are able to evict the target line out of the LLC by accessing 11 shared lines, while in the 2evictors_ref setup, we begin to observe stable LLC misses when around 85 lines are accessed. The characteristics in 2evictors_sameEV indicates the associativity of the inclusive structure for shared cache lines is 11. Moreover, this experiment indicates how an attacker can use shared eviction lines to evict a shared target line out of the cache hierarchy, which we will leverage to build stable and efficient Evict+Reload attacks.

6. The cache replacement policy takes into account the coherence state and prefers not to evict cache lines which have been accessed by multiple cores.
7. The associativity of the inclusive structure for shared cache lines is 11.

E. Putting It All Together: the Directory Structure

We infer that the inclusive structure is a directory. Indeed, Intel has used a directory-based coherence protocol since Nehalem [38].[2] Supporting a directory-based protocol requires structures that store presence information for all the lines in the cache hierarchy. Thus the directory, if it exists, must be inclusive, like the structure we found. In the rest of the paper, we will use the term directory to refer to the inclusive structure we found.

Overall structure. Figure 9 shows a possible structure of the directory in one LLC slice. From Section V-B, we found that the directory is sliced, is looked-up using the LLC slice hash function, and has the same number of sets as the LLC. An LLC slice can co-locate with its directory, enabling concurrent LLC slice and directory look-up.

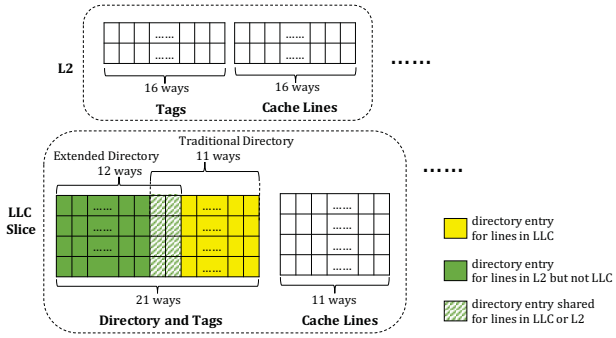


Fig. 9. Reverse engineered directory structure.

From Section V-B, each directory slice has 21 ways in total (denoted $W_{\text{dir}} = 21$) for all the lines in the cache, including the L2 and LLC. From Section V-B, there are maximally 12 ways can be used for lines present in the L2 but not in the LLC. We call the directory for these lines the *Extended Directory* (ED). We denote the ED associativity as $W_{\text{ED}} = 12$. From the public documentation in Table I, we know that the LLC slice and its directory (which we call the *Traditional Directory* is 11-way set associative. We denote such associativity as $W_{\text{TD}} = W_{\text{slice}} = 11$. One might expect $W_{\text{dir}} = W_{\text{ED}} + W_{\text{TD}}$, but $21 < 12 + 11$. From this mismatch, we infer that 2 ways in each directory slice are dynamically shared between the traditional directory and the extended directory. How these ways are shared is determined by the replacement policy, which prefers to hold lines that are in state shared.

Migration between directories. The migration between the ED and the traditional directory operates as follows. An ED conflict causes a directory entry to be migrated from the ED to the traditional directory, and the corresponding cache line to be evicted from an L2 to the LLC. A conflict in the traditional directory causes a directory entry to be removed from the whole directory slice, which causes the corresponding cache line to be evicted out of the entire cache hierarchy. For private lines, 21 addresses are needed to cause a traditional directory conflict. From Section V-B, the private lines will take up the ED and traditional directory, after which we see conflicts. For shared lines, only 11 addresses are needed to cause conflicts. We found that shared lines, after being accessed a sufficient number of times, allocate a data entry in the LLC and migrate their directory entry from the ED to the traditional directory. In this case, the line lives in multiple L2s and in the LLC at the same time.[3] This is likely a performance optimization as LLC hits are faster than remote L2 hits (Figure 5). Thus, heavily shared lines should be accessible from the LLC.

This directory structure matches our reverse engineered results. Even though the actual implementation may use a slightly different design, our interpretation of the structure is helpful in understanding the directory and cache interactions, and in designing the attacks.

F. The Root Cause of the Vulnerability

The directory in non-inclusive caches is *inclusive*, since it needs to keep information for *all* the cache lines that are present in the cache hierarchy. This inclusivity can be leveraged to build cache-based attacks. An attacker can create conflicts in the directory to force cache line evictions from a victim's private cache, and create inclusion victims.

Considering the usage of directories, the directory capacity ($W_{\text{dir}} \times S_{\text{dir}}$) should be large enough to hold information for all the cache lines. In this case, how can it be possible to cause a directory conflict before causing cache conflicts? This section answers this question by analyzing the root cause of the vulnerability that we exploit.

The root cause is that *the directory associativity is smaller than the sum of the associativities of the caches that the directory is supposed to support*. More specifically, a directory conflict can occur before a cache conflict if any of the following conditions is true, where ED indicates the directory entries used by L2 cache lines.

$$W_{\text{ED}} < W_{\text{L2}} \times N_{L2}$$
$$\textit{or} \quad W_{\text{dir}} < W_{\text{L2}} \times N_{L2} + W_{\text{slice}}$$

where N_{L2} is the number of L2s in the system (usually equal to the number of cores).

[2]Sometimes, the directory structure is called "core valid bits".

[3]This is consistent with the cache being non-inclusive. Non-inclusive means that the cache may be inclusive for certain lines in certain circumstances.

We believe that, for performance reasons, these conditions should be common. First, as the number of cores on chip increases, architects want to avoid centralized directory designs and, therefore, create directory slices. At the same time, architects try to limit the associativity of a directory slice to minimize look-up latency, energy consumption, and area. As a result, it is unlikely that each directory slice will be as associative as the sum of the associativities of all the L2 caches. For example, an 8-core Intel Skylake-X processor with 16-way L2s would require each directory slice to have a 128-way ED to avoid ED conflicts before L2 conflicts. This is expensive.

We found that the above conditions do not hold in some AMD processors. Consequently, our attack does not work on these AMD processors. We also found that memory and coherence operations on some AMD machines are slower than on Intel machines. This may suggest that these AMD machines do not use sliced directories. Appendix C describes the experiments we performed.

VI. Attack Strategies

Leveraging the EV construction algorithm in Section IV and our reverse engineering of the directory structure in Section V, we can now build effective side channel attacks in non-inclusive cache hierarchies. In this section, we first present our Prime+Probe attack targeting the ED. We then show a multi-threaded Evict+Reload attack to achieve fine-grained monitoring granularity. Finally, for completeness, we provide a brief discussion on Flush+Reload attacks.

A. Prime+Probe

To the best of our knowledge, this is the first Prime+Probe attack on non-inclusive caches. We first present our customized cross-core attack on Skylake-X, and then discuss how to generalize the attack to other vulnerable platforms.

On Intel Skylake-X, an attacker can leverage the inclusivity of the ED to gain visibility into a victim's private cache state. Before the attack, the attacker uses the EV construction algorithm of Section IV to construct an EV that is mapped to the same LLC slice and set as the target address. Since the ED is both sliced and looked-up using the LLC slice hash function, it follows that the EV is mapped to the same ED set and slice as the target address. Thus, the EV can be used in the prime and probe steps.

Our cross-core Prime+Probe attack follows the same steps as a general Prime+Probe attack. In the prime step, the attacker accesses W_{ED} EV lines to occupy all the ways within the ED set, and evict the target line from the victim's L2 to the LLC. During the wait interval, if the victim accesses the target address, it causes an ED conflict and one of the EV addresses will be evicted from the attacker's private L2 cache to the LLC. In the probe step, when the attacker accesses the EV addresses again, it will observe the LLC access. Alternatively, if the victim does not access the target line in the wait interval, the attacker will observe only L2 hits in the probe step. After the probe step, the ED set is fully occupied by EV addresses, and can be used as the prime step for the next attack iteration.

Attack granularity. The attack granularity is determined by the time per attack iteration, which is composed of the wait time and the probe time. The more efficient the probe operation is, the finer granularity an attack can achieve.

In our ED-based Prime+Probe attack, the probe time is very short. The attacker only needs to distinguish between local L2 latency and LLC latency, which is shorter than the probe time in inclusive cache attacks, where the attacker needs to distinguish between LLC latency and DRAM latency.

Generalizing the attack. The attack above is customized for Intel Skylake-X. We now discuss how to generalize the attack to other vulnerable platforms which satisfy the conditions discussed in Section V-F.

First, a characteristic of the Skylake-X is that the ED associativity is not higher than the L2 associativity ($W_{\text{ED}} \leq W_{\text{L2}}$), which allows us to trigger ED conflicts using a single attacker thread. If this condition is not satisfied, we can still mount a Prime+Probe attack with *multiple* attacker threads, running on different cores, as long as $W_{\text{ED}} \leq W_{\text{L2}} \times (N_{L2} - 1)$. For example, consider the case where $W_{\text{ED}} = W_{\text{L2}} \times (N_{L2} - 1)$. The attacker can use $(N_{L2} - 1)$ threads running on all the cores except for the victim's core, where each thread accesses W_{L2} addresses to occupy the ED set.

Second, the directory in Skylake-X uses the same hash function as the LLC. Therefore, we can directly use the EVs constructed for LLC slices to create directory conflicts. If the sliced ED uses a different hash function, the attack should still work but will need a new EV construction algorithm for the directory.

B. Evict+Reload

On non-inclusive caches, an attacker could leverage the directory's inclusivity to build Evict+Reload attacks using a similar approach as in Prime+Probe. However, the evict operation in Evict+Reload is more challenging than the prime operation, since the target line is shared by the attacker and the victim. As we showed in Section V-D, the cache replacement policy takes into account the coherency state—namely that the target line is *shared*—and prefers not to evict the directory entries for shared lines.

We propose a novel multi-threaded Evict+Reload attack that can achieve fine-grained monitoring granularity by taking advantage of the characteristics of the replacement policy. The attack involves two tricks, namely, to upgrade the eviction addresses to a higher replacement priority, and to downgrade the target address to a lower replacement priority.

The attacker consists of three threads: a main thread which executes the evict and reload operations, and two helper threads to assist evicting the shared target line. The two helper threads share all the eviction addresses, and thus are able to switch the eviction addresses to the *shared* coherence state, similar to the 2evictors_sameEV setup in Section V-D. This brings eviction addresses to the same replacement priority as the target address in the directory. In addition, the main attacker thread evicts the target address from its private cache to the LLC by creating L2 conflicts, which makes the target address non-shared.

Throughout the entire attack, the helper threads run in the background, continuously accessing the addresses in the EV in order to keep these addresses in their caches. In the eviction step, the main attacker thread introduces conflicts in its L2 cache to evict the target line from its L2 to the LLC. The helper threads then evict the target line (which they do not share) from the LLC to DRAM, by accessing the shared EV lines. If the victim accesses the target line during the wait interval, it will bring the line into its L2. Then, in the reload step, the main attacker will see a remote L2 access latency. Otherwise, the attacker will observe a DRAM access latency.

C. Flush+Reload

A Flush+Reload attack on non-inclusive caches follows the same procedure as the one on inclusive caches. This process has been referred to as Invalidate+Transfer [11] on AMD's non-inclusive caches. We evaluate this attack on Intel's platform for completeness, though it is not necessary to demonstrate our new attack channel on directories.

The attacker uses the `clflush` instruction to evict the target address from all levels of the cache hierarchy to DRAM. If, during the wait interval, the victim accesses the target address, the line will be brought into the victim's local L2. In the measurement phase, the attacker reloads the target address and measures the access latency. If the victim had accessed the line, the attacker will see a remote L2 access latency; otherwise, it will observe a DRAM access latency.

VII. Evaluation

A. Effectiveness of the check_conflict Function

We evaluate the effectiveness of the `check_conflict` function by measuring the false positive rates and the false negative rates. We consider three designs, the baseline function proposed by Liu et al. (no_flushL2), and the two modified functions discussed in Section IV, i.e. flushL2_naive and flushL2_reliable.

We obtained 8 EVs and confirmed their correctness by checking their conflicting behaviors as in Section V. All the addresses in the 8 EVs have the same LLC set index bits, and each EV is mapped to a different LLC slice. To measure the false positive rate, we select an address x and set the argument U of `check_conflict` to be a collection of addresses with 10 addresses ($< W_{\text{slice}}$) from the same EV as x, and 5 addresses from each of the other EVs. The function should return $false$. We then count the number of times when the function mistakenly returns $true$. To measure the false negative rate, an extra address from the same EV as x is added to the collection U, so that U contains 11 eviction addresses ($= W_{\text{slice}}$). The function should return $true$. Then, we count the number of times when the function mistakenly returns $false$. In each of the three `check_conflict` implementations, the eviction operation (line 3-5 of Algorithm 2, line 3-8 in Algorithm 3) is repeated multiple times. Figure 10 shows how the false positive rate and the false negative rate change with the number of eviction operations performed.

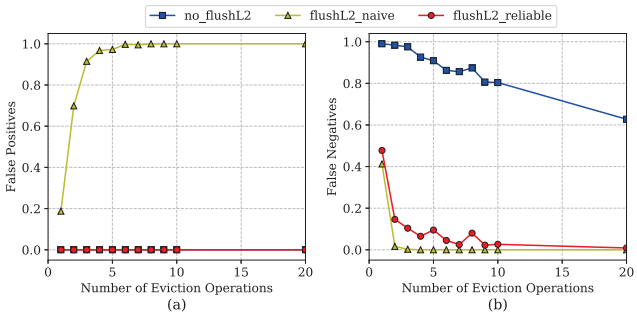


Fig. 10. Comparing effectiveness of different check_conflict functions.

In Figure 10(a), both no_flushL2 and flushL2_reliable have no false positives. flushL2_naive has a much higher false positive rate due to the extra conflicts introduced by the $L2_occupy_set$. In Figure 10(b), both flushL2_naive and flushL2_reliable can achieve very low false negative rate when eviction operations are repeated around 10 times. The false negative rate of the no_flushL2 approach stays high even though the evictions are performed 20 times. In conclusion, our reliable flushL2 approach in `check_conflict` function is effective and can achieve both low false negative rate and false positive rate.

B. Extended Directory Timing Characteristics

As we leverage ED conflicts to construct our Prime+Probe attack, it is very important to understand their timing impact on cache access latencies, as shown in Figure 11. The figure shows the access latency of a number of addresses from the same EV. In the "no_EDconf" case, we simply measure the latency of EV accesses. In the "1_EDconf" case, between two measurements, we use a different thread on another core to issue one access to the same ED set to cause one ED conflict. Thus, the latency in "no_EDconf" is the expected probe latency with no victim accesses during wait intervals, while the "1_EDconf" latency corresponds to the expected probe latency when victim accesses the target line.

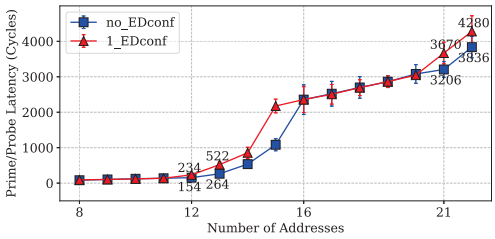


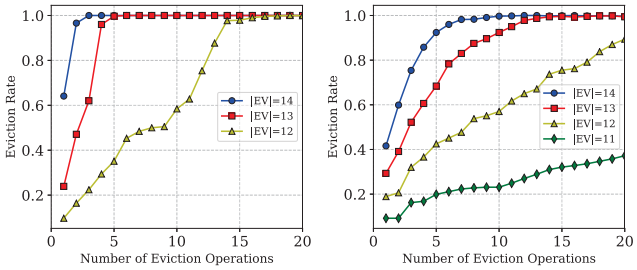
Fig. 11. Prime/Probe Latency

A high-resolution low-noise Prime+Probe attack requires the probe operation to be efficient and clearly distinguishable. From Figure 11, W_{ED} (12) is the optimal number of probe addresses we should use in Prime+Probe. First, the impact of ED conflicts is large and clearly observable. The timing difference between no ED conflicts and a single ED conflict is around 80 cycles. Second, accessing 12 addresses takes very short time, around 230 cycles with a ED conflict. With

such efficient prime/probe operation, we can do fine-grained monitoring. It is also feasible to use 13-15 addresses, but it is not optimal due to the longer access latency and larger variance. Note that the variance in Figure 11 is measured in a clean environment, there is more noise when running the attacker code with the victim code.

C. Directory Replacement Policy Analysis

As discussed before, the directory uses a complex replacement policy. We analyze how the replacement policy affects the effectiveness of eviction operations on a private and a shared cache line in Figure 12. This is an important factor an attacker needs to consider in designing efficient cache attacks.



(a) Evict a private line (a) Evict a shared line

Fig. 12. Analysis of Directory Replacement Policy

Figure 12(a) shows the eviction rate of evicting a private cache line from a remote L2 to the LLC by creating ED conflicts. To repeat the eviction operation, we simply re-access each address in the EV in the same order. When using 12 EV addresses, the eviction rate reaches 100% after accessing the EV for 14 times, while the eviction rate increases much faster when we increase the size of the EV. For example, accessing 13 EV addresses for 5 times can ensure eviction. Figure 12(b) shows the eviction rate of evicting a shared cache line from a remote L2 to DRAM by creating directory conflicts with 2 eviction threads. It turns out when using 14 EV addresses, it requires repeating the eviction operation 9 times to ensure complete eviction. This indicates the necessity to downgrade the target line replacement priority to achieve fine-grained attack granularity, as we discussed in Section VI-B.

In summary, due to the complexity of the directory replacement policy, we find it difficult to come up with an efficient eviction strategy. A possible approach would be to try all the combinations of EV sizes and access orders as in [18]. Nevertheless, in this paper, we show that our attacks can tolerate this imperfect eviction rate.

D. Covert Channel on Extended Directories

We demonstrate a covert channel between two different processors that utilizes the extended directory between two different processes. One process serves as the sender and the other as the receiver. The sender and receiver run on separate cores, and each utilizes 7 addresses that are mapped to the same LLC slice. Together there are 14 addresses, which are enough to cause measurable ED conflicts.

Since we have not reverse engineered the slice hash function, the sender and the receiver cannot directly negotiate which slice

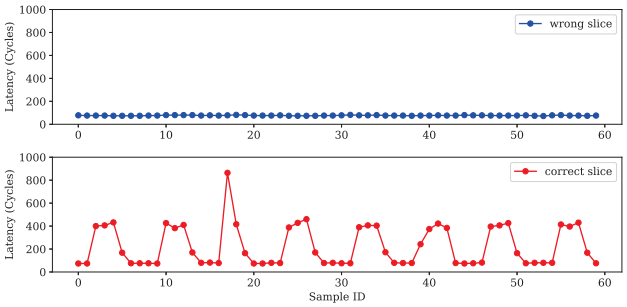


Fig. 13. The upper plot shows receiver's access latencies on a slice not being used for the covert channel, while the lower one shows the one used in the covert channel. Sender transmits sequence "101010...".

to use. Before communication, the receiver scans all slices to find the one the sender is using. The sender transmits a bit "0" by idling for 5000 cycles, and keeps accessing the 7 addresses for 5000 cycles to transmit a bit "1". The receiver decodes the two states by taking latency samples every 1000 cycles. On our 3.6GHz machine, it takes 5000 cycles to transmit one bit, thus the bandwidth is $0.2Mbit/s$. With a better protocol than we are using, the bandwidth can be further improved.

Figure 13 shows the results of our reliable covert communication channel. The upper plot shows the latencies that the receiver observes when accessing the wrong slice. All the latencies are low, as they correspond to L1 cache hits. In the lower plot, it is clear to see the sender's message of "101010". The receiver observes a ~ 400 cycle latency due to ED conflicts when decoding a "1" bit, which is easily differentiated from the ~ 80 cycle L1 hits for a "0" bit.

E. Side Channel Attacks on the Square-and-Multiply Exponentiation Algorithm

We evaluate the effectiveness of our side channel attacks on the square-and-multiple exponentiation vulnerability in GnuPG 1.4.13. The implementation is similar to the one presented in Algorithm 1 in Section II. As discussed before, a victim's accesses on function `sqr` and `mul` can leak the value of exponent. In GnuPG, these two functions are implemented recursively, thus the target address identifying each function will be accessed multiple times throughout the execution of either operation. We show how this algorithm remains vulnerable on non-inclusive caches by attacking it with Prime+Probe, Evict+Reload and Flush+Reload attacks.

1) Flush+Reload: We evaluate a cross-core Flush+Reload attack on this new platform for completeness. The victim and the attacker run on separate cores. The flush and reload operations are used on the addresses located at the entry of the `sqr` and `mul` functions. We use a wait time of 2000 cycles between the flush and reload.

Figure 14 shows the time measurement of the reload operation for 100 samples. A low latency reload operation, less than 250 cycles, indicates the victim has accessed the target address during the wait interval. A high latency, around 350 cycles, means the victim has not accessed the target address. According to the algorithm, an access on `sqr` followed by

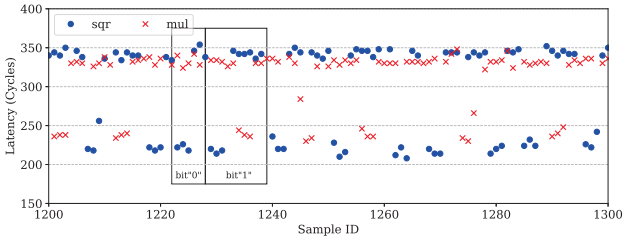


Fig. 14. Access latencies measured in the reload operation in Flush+Reload. A sequence of "1001110101" can be deduced as part of the exponent.

an access on `mul` indicates a bit "1", and two consecutive accesses on `sqr` without `mul` accesses in the between indicate a bit "0". From Figure 14, we can see that each `sqr` operation completes after 3 samples, or about 6000 cycles. Leveraging this information, the attacker is able to deduce part of the exponent as "1001110101".

In Flush+Reload, errors stem from times when the attacker's flush operation overlaps with victim accesses. Such occurrences cause lost bits.

2) Prime+Probe: In our Prime+Probe attacks, we use 12 probe addresses from an eviction set for the target address, and use 500 cycles as the attacker wait interval. [4] We are able to monitor with such small granularity due to the efficient probe operation on the ED. We only monitor one target address, i.e. the address located at the entry of `mul` function, which is good enough.

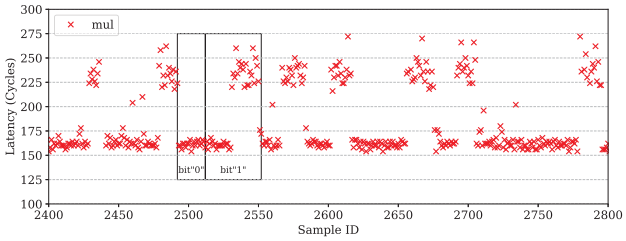


Fig. 15. Access latencies measured in the probe operation in Prime+Probe. A sequence of "01010111011001" can be deduced as part of the exponent.

Figure 15 shows the access latencies measured in the probe operation as results of our Prime+Probe attack for 400 samples. If there is no victim access of the target address, the probe operation will see L2 hits for all the probe addresses without ED conflicts, taking around 160 cycles. Otherwise, if the victim accesses the target address, ED conflicts will be observed, resulting in long access latency, around 230 cycles. We do not track victim accesses on the `sqr` function; this the same approach taken in [8]. Instead, the number of `sqr` operations can be deduced from the length of the interval between two consecutive multiply operations. The attacker can deduce a sequence of "01010111011001" as part of the exponent from Figure 15.

[4]We use 12 EV addresses instead of 13 addresses, because we can get more precise and clean measurements of accessing 12 addresses, even though we suffer some noise due to relatively low eviction rate.

In Prime+Probe attacks, most errors stem from the imperfect eviction rate, which leads to observing a multiply operation for more samples than it actually executed.

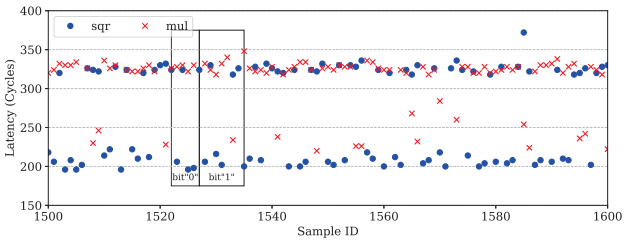


Fig. 16. Access latencies measured in the reload operation in Evict+Reload. A sequence of "0101011110110101" can be deduced as part of the exponent.

3) Evict+Reload: Our novel Evict+Reload attack utilizes 1 attacker thread and 2 helper threads. The 2 helper threads access the same EV with 11 (W_{TD}) addresses mapped to the same LLC slice and set as the target address. The attacker thread accesses 16 (W_{L2}) addresses mapped to the same L2 set as the target line 6 times. We tested multiple eviction approaches, and found this method is highly reliable, and also very efficient, only taking around 1200 cycles. We monitor both the square and multiply operations and use 4000 cycles as the wait interval.

Figure 16 shows the access latencies measured in the reload step as the results for the Evict+Reload attack for 100 samples. The figure can be interpreted in the same way as the one for Flush+Reload, and the attacker can decode the part of the exponent as sequence "0101011110110101".

Compared to Flush+Reload, the Evict+Reload attack on non-inclusive caches tends to suffer more errors. Since the evict operation takes longer than the flush operation, the probability that the evict step overlaps with the victim's access is higher.

VIII. Related Work

There have been a variety of cache-based side channel attacks in the literature. We start by reviewing the attacks most closely related to our attack, namely those on non-inclusive caches. We then briefly discuss side-channel attacks on inclusive caches.

A. Attacks on Non-Inclusive Caches

There are two known attacks on non-inclusive caches that require page sharing [11], [18]. ARMageddon [18] leverages Evict+Reload to attack a non-inclusive ARM LLC. Irazoqui et al. [11] leverage Flush+Reload to attack a non-inclusive AMD dual-socket machine. Both works rely on shared virtual memory. Moreover, neither of these works addresses the complexities stemming from sliced caches. Thus, our work is more general.

ARMageddon's [18] usage of Evict+Reload on non-inclusive caches is slower than our Evict+Reload attack, as it must access many more addresses in the evict phase. Its method will be even slower for larger caches. For example, ARMageddon attacks L1 caches that are at most 32KB, while their shared L2 cache is at most 2048KB. On the Skylake-X system that we attack, the L2 is 1MB and the LLC is 11MB. Despite having larger caches, our attack succeeds and with finer granularity

than ARMageddon. Additionally, we overcome issues related to sliced caches, which are not present on ARM architectures.

B. Attacks on Inclusive Caches

Same-core side channel attacks [27], [39], [40] leverage hyper-threading to co-locate victims and attackers on the same core, and exploit cache timing differences between L1 and L2 cache accesses. Other attacks exploit the operating system scheduler to achieve core-co-residency, overcoming the need for hyper-threading [19]. Cross-core attacks are more difficult, as timing information comes from a much larger LLC, which increases noise as it is shared across many cores. Yarum et al. [7] proposed a cross-core, cross-VM Flush+Reload attack on an LLC by leveraging shared memory stemming from memory deduplication. Liu et al. [8] proposed a practical Prime+Probe attack on an inclusive LLC, which does not rely on shared memory, as we discussed earlier.

IX. Countermeasures

Hardware-based. Our attack causes conflicts on the limited number of directory entries (including ED) to create inclusion victims. One approach to prevent the attack is to eliminate contention for directory entries. This can be realized in a few different ways, some of which introduce severe performance degradation. First, we can increase the associativity of the ED in each LLC slice, so that it is equal to the maximum possible number of entries in a set, i.e. $N_{L2} \times W_{\mathrm{L2}}$. In a sliced directory design, the total number of ED entries will then be $N_{ED} \times S_{\mathrm{ED}} \times N_{L2} \times W_{\mathrm{L2}}$. This results in a large amount of wasted area on the chip. Second, we can build a centralized directory structure. However, such centralized structure is not scalable and will be a serious performance bottleneck. Third, we can eliminate directories and use a snoopy-based coherence protocol. However, snoopy protocols do not scale with the core count.

Beyond general architectural changes, we can prevent the attack by applying several side channel prevention techniques that have been used for inclusive caches [34], [41], [42]. For example, the directory entry replacement policy can be modified to mimic SHARP [34], which prevents the creation of inclusion victims in the LLC. By preventing the replacement of directory entries occupied by a different core than the requesting one, the proposed attack can be prevented. Alternatively, one can partition the directory entries among the cores in a manner similar to the way Intel CAT partitions the cache.

Software-based. Software-only cache side channel defenses suffer from a variety of drawbacks. Some of these defenses use cache-coloring techniques [43]–[47] or constant time program transformation [48], which incur potentially large overheads, or costly application level changes. Compiler level defenses are transparent to developers, but incur large runtime overhead [49]. Other transparent techniques focus on kernel level changes, but remain probabilistic [50], [51]. Nomad [52] is a probabilistic defense that operates at the cloud scheduler level to keep two tenants from being co-scheduled on the same host for long periods. It is challenging to mount a probabilistic defense against fine-grained attacks such as the one presented in Section VI.

X. Conclusion

In this paper, we identified the directory as a unifying structure across different cache hierarchies on which to mount a conflict-based side channel attack. Based on this insight, we presented two attacks on non-inclusive cache hierarchies. The first one is a Prime+Probe attack. Our attack does not require the victim and adversary to share cores or virtual memory, and succeeds in state-of-the-art non-inclusive sliced caches such as those of Skylake-X [13]. The second attack is a novel, high-bandwidth Evict+Reload attack that uses a multi-threaded adversary to bypass non-inclusive cache replacement policies. We attacked square-and-multiply RSA on the modern Intel Skylake-X processor, using both of our attacks. Moreover, we also conducted an extensive study to reverse engineer the directory structure of the Intel Skylake-X processor. Finally, we developed a new eviction set construction methodology to find groups of cache lines that completely fill a given set of a given slice in a non-inclusive LLC.

Acknowledgments

This work was supported in part by NSF under grant CCF 1725734.

REFERENCES

[1] Intel, "Intel Software Guard Extensions Programming Reference," 2013.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology–CT-RSA*. Springer, 2006.

[3] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers Track at the RSA Conference*. Springer, 2007.

[4] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *HPCA'14*.

[5] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2015.

[6] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[7] Y. Yarom and K. Falkner, "Flush+Reload: a high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security)*, 2014.

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

[9] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security)*, 2015.

[10] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[11] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016.

[12] VMWare, "Transparent page sharing: new default setting." 2014. [Online]. Available: http://blogs.vmware.com/security/2014/10

[13] Intel, "6th Gen Intel Core X-Series Processor Family Datasheet - 7800X, 7820X, 7900X," 2017. [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/core/6th-gen-x-series-datasheet-vol-1.html

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

[15] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer, "High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015.

[16] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng, "NCID: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies," in *Proceedings of the 7th ACM international conference on Computing frontiers*. ACM, 2010.

[17] D. Mulnix, "Intel Xeon Processor Scalable Family Technical Overview," 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview

[18] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association.

[19] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *Selected Areas in Cryptography*. Springer, 2006.

[20] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing and its application to AES," in *IEEE Symposium on Security and Privacy*. IEEE, 2015.

[21] 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*. ACM, 2015.

[22] D. M. Gordon, "A survey of fast exponentiation methods," *Journal of algorithms*, vol. 27, no. 1, 1998.

[23] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games–Bringing Access-Based Cache Attacks on AES to Practice," in *IEEE Symposium on Security and Privacy*. IEEE, 2011.

[24] B. Gülmezoğlu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+reload attack on AES," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2015.

[25] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2006.

[26] C. Percival, "Cache Missing for Fun and Profit," Oct. 2005. [Online]. Available: http://www.daemonology.net/papers/htt.pdf

[27] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009.

[28] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011.

[29] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-speed covert channel attacks in the cloud," in *USENIX Security symposium*, 2012.

[30] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: cross-cores cache covert channel," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015.

[31] T. Hornby, "Side-channel attacks on everyday applications: Distinguishing inputs with flush+reload," in *BackHat 2016*.

[32] G. Irazoqui, M. S. IncI, T. Eisenbarth, and B. Sunar, "Know thy neighbor: Crypto library detection in cloud," *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 1, 2015.

[33] D. Gruss, C. Maurice, and K. Wagner, "Flush+Flush: A stealthier last-level cache attack," *arXiv preprint arXiv:1511.04594*, 2015.

[34] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[35] Chrome Developers Native Client, "Security: Disallow the x86 "clflush" instruction due to DRAM "rowhammer" problem," 2014. [Online]. Available: https://bugs.chromium.org/p/nativeclient/issues/detail?id=3944

[36] Intel, "The Intel 64 and IA-32 architectures software developer's manual," vol. 2A: Instruction Set Reference A-Z, no. 325383, 2016.

[37] G. Paoloni, "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures," 2010.

[38] R. Singhal, "Inside Intel next generation Nehalem microarchitecture," in *Hot Chips*, vol. 20, 2008.

[39] D. J. Bernstein, "Cache-timing attacks on AES," Technical Report, University of Illinois at Chicago, 2005.

[40] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, 2010.

[41] 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 *22nd IEEE Symposium on High Performance Computer Architecture*, 2016.

[42] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016.

[43] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016.

[44] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud," in *USENIX Security symposium*, 2012.

[45] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2011.

[46] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: a dynamic cache partitioning system using page coloring," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014.

[47] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM, 2009.

[48] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution." in *USENIX Security Symposium*, 2015.
[49] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *30th IEEE Symposium on Security and Privacy*. IEEE, 2009.
[50] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.
[51] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-VM side-channels," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.
[52] S.-J. Moon, V. Sekar, and M. K. Reiter, "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration," in *Proceedings of the 22nd acm sigsac conference on computer and communications security*. ACM, 2015.
[53] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *Research in Attacks, Intrusions, and Defenses*. Springer, 2015.
[54] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in Intel processors," in *Euromicro Conference on Digital System Design (DSD)*. IEEE, 2015.
[55] W. V. Quine, "The problem of simplifying truth functions," *The American Mathematical Monthly*, vol. 59, no. 8, 1952.
[56] D. Molka, D. Hackenberg, and R. Schöne, "Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014.
[57] M. Clark, "A new X86 core architecture for the next generation of computing," in *Hot Chips 28 Symposium (HCS), 2016 IEEE*. IEEE, 2016.
[58] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer *et al.*, "Zen: A next-generation high-performance x86 core," in *IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2017.

APPENDIX

A. The Eviction Set Construction Algorithm

The complete EV construction algorithm that we used for non-inclusive caches is a slightly modified version of the algorithm proposed by Liu et al. [8], as shown in Algorithm 4.

The `find_EV(Collection CS)` function above takes a collection of addresses, which we call a Candidate Set (CS) as input, and outputs an eviction set (EV) for one slice as output. For the algorithm to work, it is required that all the addresses in CS have the same LLC set index bits, and CS contains more than W_{slice} addresses for each slice. Such CS can be easily obtained by using a large number of addresses. To find EVs for all the slices within CS, we need to run the function the same number of times as the number of slices.

The function initializes EV as an empty set and selects a random address $test_addr$ in CS (line 2-3). It then tries to construct an EV containing all the addresses which are mapped to the same slice and set as $test_addr$ from CS. First, it creates a new set CS' by removing $test_addr$ from CS (line 4), and then performs a sanity check to make sure CS' contains enough addresses to evict $test_addr$ out of LLC using `check_conflict` (line 5-7).

The loop (line 8-14) performs the bulk of the work, checking whether an address is mapped to the same slice as $test_addr$. Since CS' conflicts with $test_addr$, if removing an address $addr$ causes the conflict disappear, we know that $addr$ contributes to the conflict, and $addr$ should be added to EV

Algorithm 4: Constructing an eviction set.

```
   Input  : candidate set CS
   Output : EV
1  Function find_EV(CS):
2      EV = {}
3      test_addr = get a random addr from CS
4      CS' = CS − test_addr
       // make sure there are enough addresses to conflict with test_addr
5      if check_conflict(test_addr, CS')==false then
6          return fail
7      end
8      for each addr in CS' do
9          if check_conflict(test_addr, CS' − addr)==false then
               // if conflict disappears, we know addr contributes to
               // the conflict, and it should be in the EV
10             insert addr to EV
11         else
12             CS' = CS' − addr
13         end
14     end
15     for each addr in CS do
16         if check_conflict(test_addr, EV)==true then
17             insert addr to EV
18         end
19     end
20     return EV
21 end
```

(line 9-10). Such addresses are kept in CS'. Addresses which are not strictly necessary to cause conflicts with $test_addr$ are removed from CS' (line 12), and CS' should still conflict with $test_addr$ after the remove operations. After the loop, we obtain a minimal EV with exactly W_{slice} number of addresses.

It is possible that there are more than W_{slice} addresses from the same slice as $test_addr$ which have been conservatively removed in the loop. We use an extra loop (line 15-19) to find these addresses, by iteratively checking each address in the original CS to determine whether it conflicts with the obtained EV.

The `check_conflict` function is extensively used in this algorithm. On line 9, the function is used to test whether removing an address from a set can cause LLC conflicts to disappear. This operation requires the function to have both a low false positive rate and a low false negative rate, as discussed in Section IV.

B. Slice Hash Function

Based on our EV construction results, we are able to reverse engineer part of the slice hash function in the Intel Skylake-X processor. Our goal here is to show that the slice hash function is not a simple XOR operation of selected physical address bits. This design is significantly different from the one in previous Intel processors such as SandyBridge and IvyBridge. Considering that all of the previous works on reverse-engineering slice hash functions [53], [54] rely on the use of a simple XOR hash function, our results identify the need for more advanced reverse-engineering approaches.

We briefly discuss how to get the partial hash function. We select 128 addresses with the same LLC set index bits to form a Candidate Set (CS). Bits 6-16 of these addresses are set to the same value, while bits 17-23 are varied. The goal is to

reverse engineer how bits 17-23 affect the output of the slice hash function.

First, we run `find_EV` on the 128-address CS and obtain 8 EVs. Each EV is mapped to one cache slice. Second, we try to figure out the slice id for each EV. Since the Skylake-X processor uses a mesh network-on-chip to connect L2s and LLC slices [13], a local LLC slice access takes shorter time than a remote slice access. We check the access latency of each EV from each core. We then get the id of the core from which the access latency is the lowest, and assign the core id to the EV. Finally, we use the Quine-McCluskey solver [55] to get the simplified boolean functions from the input bits to the slice id as below. In the following, o_i is the ith bit in the slice id, and b_i is the ith bit in the physical address.

$$
\begin{aligned}
o_2 =& b'_{23}b'_{19} + b'_{22}b'_{19} + b_{23}b_{22}b_{19} \\
o_1 =& (b_{23} + b_{22})(b_{20} \oplus b_{19} \oplus b_{18} \oplus b_{17}) + \\
& b'_{23}b'_{22}(b_{20} \oplus b_{19} \oplus b_{18} \oplus b_{17})' \\
o_0 =& b'_{22}(b_{19} \oplus b_{18}) + b_{22}(b_{23} \oplus b_{21} \oplus b_{19} \oplus b_{18})' \\
& where \quad \{b_{63}...b_{24}\} = 0x810
\end{aligned}
$$

These functions can not be further reduced to a simple XOR function. According to our observations, some of the higher bits (bits 24-63) also affect the hash function, which we have not fully reverse engineered.

C. Attacking AMD Non-Inclusive Caches

We tried to reverse engineer the non-inclusive cache hierarchy in an 8-core AMD FX-8320 processor, which uses the AMD Piledriver microarchitecture. The cache parameters in this processor are listed in Table II.

	AMD Piledriver	AMD Zen (4-core CCX)
L1-I	64KB/2cores, 2-way	64KB, 4-way
L1-D	16KB, 4-way	32KB, 8-way
L2	2MB/2cores, 16-way, inclusive	512KB, 8-way, inclusive
LLC	8MB/8cores 64-way, non-inclusive	2MB/core 16-way, non-inclusive

TABLE II
CACHE STRUCTURES IN AMD PROCESSORS.

We found that the L2 caches in this processor are inclusive and shared by two cores. We verified that previous inclusive cache attacks work well, if the attacker and the victim are located on neighboring cores and share the same L2.

To see whether the non-inclusive LLC is vulnerable to cache attacks, we tried the reverse engineering experiments in Section V to detect the existence of directories. We did not observe extra conflicts besides cache conflicts. It is possible that the processor uses a snoopy-based cache coherence protocol [56], in which case there is no directory. It is also possible that the processor uses a centralized and high-associativity directory design, such that the directory associativity is at least as high as the total cache associativity. In this case, the directory for L2 lines needs to have ≥ 64 ways. Overall, the conditions in Section V-F do not hold.

We also evaluated our attack on an 8-core Ryzen 1700 processor, which uses the latest AMD Zen microarchitecture. The cache parameters are also listed in Table II. The processor consists of 2 Core Complexes (CCX). A CCX is a module containing 4 cores, which can connect to other CCX modules via Infinity Fabric [57], [58]. We did not observe extra conflicts other than the cache conflicts on this processor either. Given the small number of cores on each die and low L2 associativity (8 on AMD CCX compared to 16 on Intel Skylake-X), we hypothesize that this processor either uses a snoopy-based protocol or a 32-way centralized directory for L2 lines.

Performance implications for AMD designs. We measured the remote L2 access latency for the Piledriver processor, and found that it was about as long as a DRAM access. This time is significantly longer than the corresponding operation in the Intel Skylake-X (Figure 5). This observation backs up our claim that sliced directories are important structures in high performance processors. For the Ryzen processor, we have a similar result. Specifically, a cross-CCX access takes a similar amount of time as a DRAM access. The Skylake-X/Skylake-SP processors can support up to 28 cores. Since each CCX is only 4 cores, constructing a similarly provisioned Ryzen system can mean that most cross-core accesses turn into cross-CCX accesses.