SecDir: A Secure Directory to Defeat Directory Side-Channel Attacks

Mengjia Yan, Jen-Yang Wen, Christopher W. Fletcher, and Josep Torrellas
University of Illinois at Urbana-Champaign
{myan8,jwen11,cwfletch,torrella}@illinois.edu

ABSTRACT

Directories for cache coherence have been recently shown to be vulnerable to conflict-based side-channel attacks. By forcing directory conflicts, an attacker can evict victim directory entries, which in turn trigger the eviction of victim cache lines from private caches. This evidence strongly suggests that directories need to be redesigned for security. The key to a secure directory is to block interference between processes. Sadly, in an environment with many cores, this is hard or expensive to do.

This paper presents the first design of a scalable secure directory. We call it *SecDir*. SecDir takes part of the storage used by a conventional directory and re-assigns it to per-core private directory areas used in a victim-cache manner called *Victim Directories* (VDs). The partitioned nature of VDs prevents directory interference across cores, defeating directory side-channel attacks. The VD of a core is distributed, and holds as many entries as lines in the private L2 cache of the core. To minimize victim self-conflicts in a VD during an attack, a VD is organized as a *cuckoo* directory. Such a design also obscures the victim's conflict patterns from the attacker. For our evaluation, we model with simulations the directory of an Intel Skylake-X server with and without SecDir. Our results show that SecDir has a negligible performance overhead. Furthermore, SecDir is area-efficient.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; • **Computer systems organization →** *Multicore architectures.*

KEYWORDS

Cache-Coherence Directories, Side-Channel Attacks, Cuckoo Hashing.

ACM Reference Format:
Mengjia Yan, Jen-Yang Wen, Christopher W. Fletcher, and Josep Torrellas. 2019. SecDir: A Secure Directory to Defeat Directory Side-Channel Attacks. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3326635

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3326635

1 INTRODUCTION

The design of directories for cache coherence has been an active area of research for many years (e.g., [5, 6, 11, 17, 18, 37]). Most of the research has focused on making the directories efficient and scalable to large core counts. As a result, commercial machines have incorporated directories (e.g., [26, 39]).

Recent research has shown that directories are vulnerable to conflict-based side-channel attacks [46]. The insight is that every single line in the cache hierarchy has a corresponding directory entry. Since directories are themselves cache structures organized into sets and ways, an attacker can access data whose directory entries conflict in a directory set with entries from the victim. The eviction of victim directory entries automatically triggers the eviction of victim cache lines from the private caches of the victim — irrespective of whether the cache hierarchy is inclusive, non-inclusive, or exclusive. As the victim re-accesses its data, the attacker can indirectly observe the directory state changing, hence succeeding in their purpose.

Directory attacks are wickedly effective, as they only rely on the fact that directories are shared between cores. Also, they become easier to mount with higher core counts in the machine. This is because the attacker can use the private caches of more cores to access more cache lines whose directory entries map into the same directory set as the victim's.

These observations strongly indicate that directories have a fundamental security problem. Hence, they need to be redesigned with security concerns in mind.

The key to a secure directory is to block interference between processes. Sadly, this is hard or expensive to do in an environment with many cores. For example, an apparent solution is to substantially increase the associativity of the directory structures. Unfortunately, it is unrealistic to aim for a directory associativity as high as the associativity of the private L2 cache times the number of cores — which is the total number of different live directory entries that could be mapped to one directory set.

A second approach is to way-partition the directory. Each application is given some of the directory ways, to which it has uncontested use. This solution is similar to cache partitioning schemes [25, 28, 42]. Unfortunately, this approach is inflexible, low performing, and limited, since servers can have many more cores than directory ways.

Ideally, a secure directory has to have several characteristics. First, it should set aside some directory area to support many isolated partitions inexpensively and scalably. Second, each partition should provide high associativity, so that a victim suffers few self-conflicts under the pathological environment of an attack. Finally, the directory should have little area overhead and provide fast look-ups.

In this paper, we use these ideas to design a secure directory for a server multiprocessor. We call it *SecDir*. SecDir takes part of the storage used by a conventional directory and re-assigns it to per-core private directory structures of high effective associativity called *Victim Directories* (VDs). The partitioned design of VDs prevents directory interference across cores, thus defeating directory side-channel attacks. The VD of a core is distributed, and holds as many entries as lines in the private L2 cache of the core. To provide high effective associativity, a VD is organized as a *cuckoo* directory. Such a design also obscures victim self-conflict patterns from the attacker. An insight in SecDir is that conventional directories need substantial storage to keep sharer information — especially in machines with large core counts — while a core-private directory structure like a VD does not need such information. Hence, a VD can boost the number of directory entries for a very modest storage cost.

We model with simulations the directory of an Intel Skylake-X server [21] without and with SecDir. We run SPEC and PARSEC applications. Our results show that SecDir has negligible performance overhead. Furthermore, SecDir is area-efficient: it only needs 28.5KB more directory storage per core than the Skylake-X for an 8-core machine, while it uses less directory storage than the Skylake-X for 44 cores or more. Finally, a cuckoo VD organization eliminates substantial victim self-conflicts in a worst-case attack scenario.

The contributions of this paper are:

- The design of SecDir, the first scalable secure directory.
- An analysis of the characteristics of a secure directory.
- An evaluation of SecDir.

2 BACKGROUND & MOTIVATION

2.1 Directories and Non-Inclusive Caches

Directory-based cache coherence (e.g., [5, 6, 11, 17, 18, 37]) is one of the two ways of implementing hardware cache coherence. It relies on a directory, which is a hardware table with an entry for each of the lines stored in the cache hierarchy. A directory entry keeps the sharer information for a line, namely, which cores cache the line. The simplest encoding of this information is through a Presence bit vector (one bit per core, which is set if the core's caches have a copy of the line); other encodings use a set of pointers to the sharers [18]. For simplicity, this paper uses the design with the presence bit vector by default. In addition, a directory entry contains information on the coherence state of the line. Such information can be as simple as a Dirty bit indicating whether the line is dirty.

Typically, the directory is partitioned and physically distributed into as many *slices* as cores. Each directory slice can store directory entries for a fixed set of physical addresses. A proprietary hash function maps each requested address to its corresponding slice. In an inclusive cache hierarchy, each directory slice is closely associated with a Last-Level Cache (LLC) slice.

In recent years, as the number of cores in a machine has increased, there has been a trend toward using non-inclusive cache hierarchies (NICHs). In NICHs, the LLC (typically, the L3) may not contain all the lines present in the private caches (i.e., the L2s). The reason why NICHs are desirable is that, for large core counts, the latency from a core to remote slices of the LLC is very large; hence, it is attractive to keep the L2 miss rates low by having large L2 caches. Unfortunately, large L2 caches in inclusive cache hierarchies end-up replicating substantial state. It is therefore better to keep large, non-inclusive L2s, and use the LLC as a victim cache. This is the approach used by Intel's Skylake-X/SP [21, 40].

In a NICH, since some cache lines are in L2s and not in the LLC, the directory is organized differently. Zhao et al. [48] propose to have two directory structures per slice. One is like the directory for an inclusive cache hierarchy, with one entry for each line in the LLC slice; the other structure has entries for lines only in L2s. We call these structures the *Traditional Directory* (TD) and the *Extended Directory* (ED), respectively. Both TD and ED are set-associative and queried concurrently.

According to our prior work [46], the Intel Skylake-X has a similar type of directory structure. While the ED and TD in Skylake-X appear to share some state, in this paper, we will assume for Skylake-X a simplified structure where the ED and TD in a slice are separate, as shown in Figure 2(a). As shown in the figure, the ED and TD in a slice can have different associativities. Further, they can have a different number and type of coherence states.

As a core references an address and brings its line from memory to the L2, the slice mapping the address allocates a directory entry in the ED. The entry remains in the ED even if other cores also access the line. An ED entry is moved to the TD in two cases. One is if the ED entry is evicted from the ED due to a conflict in an ED set. The other case is if a cache line is evicted from an L2; as the line is written back to the LLC, its ED entry is moved to the TD.

There is one case when a TD entry is moved to the ED. It is when a core writes to a line that has a directory entry in the TD. In this case, as the hardware invalidates all the other copies of the line in the system, its TD entry moves to the ED. To the best of our knowledge, the eviction of a TD entry due to a conflict in a TD set causes the corresponding cache line to be invalidated from all the caches and the TD entry to be discarded. Some more details are discussed in Appendix A.

2.2 Conflict-Based Cache Attacks

In a cache-based side-channel attack (or cache attack for short), an attacker tries to observe a victim's access patterns on a target address, obtaining secret information in the process. One of the most effective and popular cache attacks is the conflict-based cache attack. It has been demonstrated on a wide range of computing platforms, including Intel [30, 46], ARM [27], and AMD [22] processors, and on many security-sensitive applications, such as encryption algorithms [16, 34] and web browser transactions [14, 32].

A conflict-based attack has three steps. In the *Conflict* step, the attacker evicts the target address from the cache by creating conflicts in the cache set where the target address maps to. In a cross-core attack, where attacker and victim run on different cores, this step involves evicting the target line from the victim's private cache, creating a so-called *inclusion victim* [23, 45].

In the *Wait* step, the attacker waits for a predefined interval. Meanwhile, the victim may perform an access to the target address, bringing the line back into the cache. Finally, in the *Analyze* step, the attacker analyzes the cache state to figure out whether the victim has accessed the target line during the Wait interval.

There are many variations of conflict-based attacks, such as prime+probe [34], evict+reload [14], evict+time [33], alias-driven attack [15], evict+prefetch [12], and prime+abort [7]. These attacks are different only in the way they perform the Analyze step. SecDir aims to defend against conflict-based cache attacks by blocking the Conflict step.

There are two other types of cache attacks, which are not considered in this paper. The first one is flush-based attacks [12, 13, 22, 47], which require that attacker and victim share the page with the target address. The attacker uses a special instruction (e.g., `clflush`), to evict the target address from the cache hierarchy. This attack is easy to block. Since attackers and victims can only share read-only and execute-only pages, an effective solution is to disable `clflush` on these pages [45]. The second type of attacks is cache-collision attacks [4], where the attacker does not perform evictions and only passively measures the victim's execution time, which leaks information. This type of attack can be fixed by modifying the victim application or using a random fill cache architecture [29].

2.3 Directory Attacks

Recent work shows various examples of directory attacks on the Intel Skylake-X directory [46]. They exploit the limited associativity of one directory slice compared to the combined associativity of all of the L2 caches in the machine plus one LLC slice. To understand the attacks, consider the parameters of the Skylake-X caches and simplified directories that we use in this paper. In a given slice, the TD and ED have associativities of W_{TD} = 11 and W_{ED} = 12, respectively. Consequently, a given directory slice can hold at most 23 entries mapping to the same set. On the other hand, the associativity of an LLC slice is $W_{LLC} = W_{TD}$ = 11, and that of an L2 cache is W_{L2} = 16. Further, the machine can have many cores — i.e., from N = 8 to 28.

An attacker can use up to N-1 cores to bring enough lines into L2 caches and into one LLC slice to require more than 23 directory entries to be mapped into a single set of a directory slice (Figure 1). The result is that any directory entry in that set belonging to the victim process is evicted, automatically evicting the corresponding line from the victim's L2 cache, which is an inclusion victim. As the victim process later reloads the data, its directory entry is automatically reloaded, which allows the attacker to indirectly observe the victim's action. This attack can be accomplished with a variety of techniques described in Section 2.2, including the popular evict+reload and prime+probe.[1]

For a victim running on a core to be able to keep at least one entry in the directory, a directory slice would have to have an associativity $W_{TD} + W_{ED}$ such that

$$W_{TD} + W_{ED} > W_{L2} \times (N - 1) + W_{LLC}$$

which assumes that the attacker can use all the cores minus one. In a Skylake-X with 8 cores, this requires a directory slice with an associativity higher than 123. With more cores, the required associativity increases rapidly. Since this is an unreasonable associativity, current directories are easy targets of conflict-based side-channel attacks.

[1] In [46], the version of prime+probe attack exploits a limitation of the implementation of the Intel Skylake-X, as discussed in Appendix A.

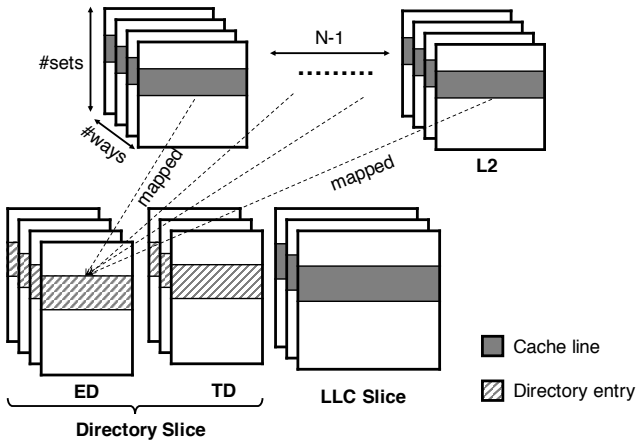


Figure 1: Attackers exploiting the limited associativity of a directory slice.

3 THREAT MODEL AND DESIGN GOALS

Table 1 puts in context the threat model that SecDir targets. We consider two aspects of the threat model, namely, the co-location of victim and attacker, and the attacker's strategy. The threat model targeted by SecDir is marked with an X.

		Co-location	
		Same-core	Cross-core
Attack Strategy	Active		X
	Passive		

Table 1: Classification of threat models. The threat model targeted by SecDir is marked with X.

SecDir targets a scenario where victim and attacker run on different cores (i.e., *cross-core*). We consider server multiprocessors used in cloud setups. The attacker and the victim are processes from different security domains, and run in different virtual machines or containers [8] with different domain tags. The hypervisor or host OS is able to distinguish different domains, and avoids assigning the two processes to the same cores [41]. Thus, since the attacker and the victim are on different cores, they share the LLC, but not private caches.

SecDir targets an *active* attacker, rather than a *passive* one. An active attacker is one that interferes with the victim's cache accesses by using directory conflicts, and exposes some of the victim's security-sensitive cache accesses. It uses prime+probe [46], evict+reload, or other techniques. It is possible that the attacker can control multiple processes, or even the majority of the processes in the chip. Hence, it can issue many requests from many cores to stress the directory and trigger pathological cases.

On the other hand, a passive attacker does not interfere with the victim. It simply monitors the victim's cache accesses or execution time, which may be affected by the reuse or conflict of cache lines or directory entries within the victim itself (i.e., *self-reuse* and *self-conflict*). An example is the cache-collision attack (Section 2.2).

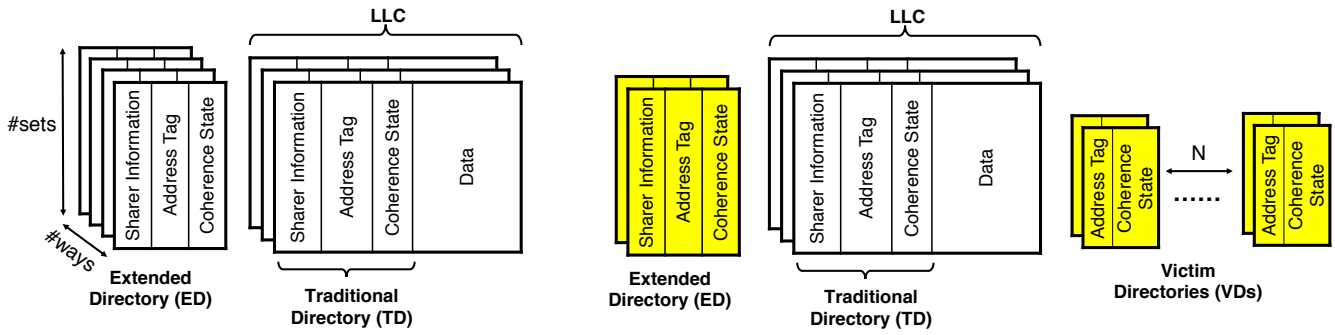


Figure 2: Slice of a conventional directory (a) and SecDir (b).

To summarize, our security goal is for directory conflicts experienced by the victim program to be a function of the victim program only, and not the attacker's active behavior. Note that there is a big difference between active and passive attackers. The effectiveness of a passive attack strategy is highly dependent on how leaky the victim application is, while an active attack strategy can be used to expose more information than the victim itself leaks. Also, since self-reuses and self-conflicts are not triggered by the attacker, their patterns can be obfuscated by introducing randomness into entry replacement algorithms.

We design a solution to defeat cross-core conflict-based active attacks on directories for non-inclusive or exclusive cache hierarchies. Our goal is to prevent an active attacker from creating inclusion victims in the victim's private caches through the eviction of the victim's directory entries. We consider that evictions of victim cache lines or victim directory entries from the victim's private caches or private directories due to *self-conflicts* (i.e., conflicts between victim cache lines or victim directory entries) do not leak information. Prior work [35, 45] has taken a similar approach, as self-conflicts are not triggered by the attacker.

4 DESIGNING A SECURE DIRECTORY

4.1 Main Idea

The root cause of the directory vulnerability is the limited associativity of individual directory slices, given the number of cores in current servers. To defeat directory attacks, we need a new directory organization with three attributes. First, the directory should set aside some storage to support inexpensive and scalable per-core isolated directory partitions. Such support will provide victim isolation, and prevent the attacker from creating inclusion victims. Second, each partition should have high associativity, so that a victim suffers minimal self-conflicts under an attack. Finally, the directory should add little area overhead and provide fast look-ups. In this paper, we propose a new directory organization with these three attributes called *SecDir*.

Figure 2(a) shows a slice of a conventional directory for non-inclusive cache hierarchies, such as that of Intel's Skylake-X, and how we change it into SecDir in Figure 2(b). The key idea of SecDir is to take a portion of the ED (e.g., some of its ways) and re-assign the storage to per-core private directory structures called the *Victim Directories* (VDs) (Figure 2(b)). We do not modify the TD because that would also require modifying the LLC.

A given core's VD in the slice shown is only *one bank* of the core's total VD. In other words, a core has a VD bank in every slice, each mapping a different set of addresses. This distributed VD of a core is sized such that it can hold as many directory entries as cache lines can be in a private L2 cache. This size will minimize self-conflicts in the VD in benign applications. Further, to provide high effective associativity, each VD bank uses *Cuckoo hashing* [10, 11]. Such a design also obscures any victim conflict patterns from the attacker. Finally, since, to a large extent, VD access hits should occur mostly during attacks, we simplify the hardware and access the VD only after ED and TD.

A slice has as many VDs as cores. Like the ED and TD, the VDs in a slice only contain directory entries for lines mapped to the local slice. A VD is set associative and, because it keeps information for a single core, it does not need sharer information bits. Such information is effectively encoded in the VD ID.

The VD of a core loads an entry when the local ED plus TD have a conflict and need to evict the directory entry for a line that lives in that core's L2 cache. The VD evicts an entry in two cases. The first one is when the entry suffers a conflict; in this case, if the entry belonged to a dirty line, the line is written back from the core's L2 to memory. The second case is when the corresponding data line is evicted from the core's L2 into the LLC; in this case, the directory entry is moved from the VD to the TD. Full details of the VD operation are presented in Section 4.2.

The VD is accessed after the ED/TD declare a miss. Hence, the VD complements the ED/TD: the ED/TD provide fast directory lookup, while the VD blocks interference of directory entries used by different cores in an attack. Overall, SecDir has the attributes discussed above:

Provides Isolation Inexpensively and Scalably. In a slice, there are as many VDs as cores, each owned by a core. Hence, directory entries used by different cores are isolated, and cannot interfere with each other. A given core has one VD bank in each slice. We size the banks so that, together, all the banks for a core across slices can accommodate as many entries as lines fit in an L2 cache. This helps minimize self-conflicts in the distributed VD of a core running a victim program, as we can assume that the lines referenced by a benign victim program are largely uniformly distributed across the different slices.

Note that the VD design is *scalable* with the number of cores in the machine: irrespective of the number of cores in the machine, the size of the distributed VD for a core is practically constant. As more cores are added, the size of a VD bank in each slice decreases, but the number of slices and, therefore, the number of VD banks, increases.

Provides High Associativity. A slice of SecDir has a high effective associativity. It has the associativity of the ED plus TD (available to all processes) plus the associativity of the private VD bank augmented with cuckoo hashing (Section 5.2.1). Under benign conditions, the VD is unlikely to be highly utilized. Under attack conditions, the victim can utilize its core's VD banks across all the slices to isolate the directory entries corresponding to its L2 lines.

Uses Low Area. SecDir reassigns some storage from ED to VD. An important insight in SecDir is that the ED needs substantial storage to keep sharer information, while a core-private directory structure like VD does not need such information (Figure 2(b)). Hence, SecDir takes ED tags, which include sharer information, and converts them to VD tags, which do not. The VD is area efficient.

Importantly, the overhead of the sharer information in an ED entry tends to increase with the number of cores in the machine (e.g., more presence bits). Hence, as the number of cores increases, we can add more VD entries per core, as the VDs reuse more sharer information bits.

This fact produces a surprising effect. Suppose that we redesign Skylake-X's directory into SecDir's, using the following guidelines: the number of entries in an ED slice, and the number of entries in a core's VD machine-wide is *each* equal to the number of lines in a private L2 cache. In this case, SecDir only needs 28.5KB more directory storage per slice than the Skylake-X directory for an 8-core machine, and it uses *less* directory storage than the Skylake-X for 44 cores or more. We present more details in Section 7.

Delivers Efficient Directory Lookup. Under ordinary, attack-free conditions, most of the directory hits are satisfied by the TD or ED. When TD and ED miss and a VD bank is accessed, the VD typically misses. At that point, a main memory access is initiated. Compared to a main memory access latency, a VD access latency is very small. However, it is still important for the VD accesses to be efficient. Consequently, as we will see, VDs have an *Empty Bit* (EB), which helps to avoid unnecessary VD accesses. The EB saves substantial energy and some latency (Section 5.2.2).

Note that a given directory entry can be, at the same time, in multiple VD banks in the slice (i.e., banks of different cores). This is a security requirement, as we will see. Hence, on a VD access, we sometimes need to search all the banks. Searching multiple banks does not increase the VD access latency, as the different banks are independent (Section 5.1).

4.2 SecDir Directory Operation

Any line in the cache hierarchy has to have a corresponding directory entry. In this section, we describe how the SecDir directory operates. First, however, we recall how a traditional directory operates. For simplicity, the following discussion assumes a MESI cache-coherence protocol. SecDir can work with any protocol — e.g., our evaluation in Section 8 uses a MOESI protocol.

4.2.1 Traditional Directory Operation. Figure 3(a) shows the operation of a traditional directory in a non-inclusive cache hierarchy. When a core accesses a line that does not exist in the cache hierarchy, the line is directly brought from main memory into the core's private cache. No data is inserted into the LLC. At the same time, a directory entry for the line is inserted in the ED (①). As discussed in Section 2.1, a directory entry can migrate from ED to TD due to a conflict in an ED set, or due to a cache line eviction from an L2; a directory entry can migrate from TD to ED due to a core writing to a line that has a directory entry in the TD.

When a TD conflict occurs, the conflicting TD entry cannot be moved to the ED (where it could have possibly come from), as it could cause a conflict deadlock going back and forth. Instead, the conflicting TD entry is discarded, which automatically causes all copies of the corresponding cache line to be evicted from the cache hierarchy (②). This is the transition that an attacker uses to create an inclusion victim in the victim's private cache. Specifically, the attacker first forces the eviction of a victim directory entry from ED, and then from TD.

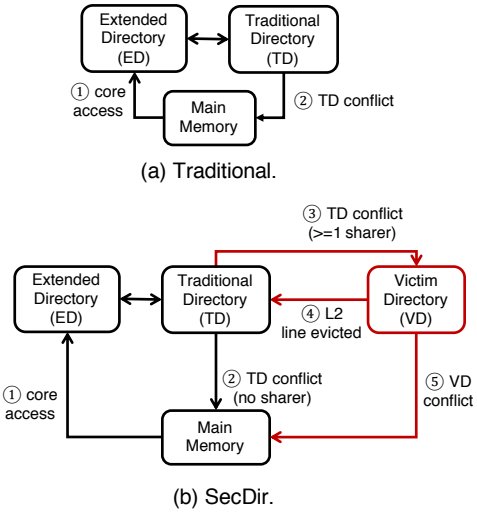


Figure 3: Operation of a traditional (a) and SecDir (b) directory.

4.2.2 SecDir Operation. Figure 3(b) shows the operation of SecDir. As in the traditional directory, when a line is fetched from main memory to a core's private cache, its directory information is stored in the ED (①). Further, the directory state can migrate between ED and TD exactly like in the traditional directory.

The difference occurs on a TD conflict. Depending on the sharer information in the conflicting directory entry, two different transitions may occur. First, if the directory entry shows that there are no sharers of the cache line (i.e., the cache line is only in the LLC), then the conflicting TD entry is discarded and, if the cache line is dirty, the cache line is written back to main memory (②). Note that it is secure to allow attackers to evict victim lines in such a way. The reason is that, as we will see, for a victim line to be only in LLC, it means that the victim process has evicted the line from its private L2 due to a self-conflict. As per Section 3, we do not

Transition	VD Access Type	Coherence Transaction	Security
②: TD → Memory	—	If cache line in Dirty state in LLC, write it back to memory; invalidate line from LLC	No leakage
③: TD → VD	Insert the directory entry into the VDs of all the sharers	—	No leakage
④: VD → TD	Search all VD banks to remove any matching directory entry	Write back the cache line to the LLC	Leak only L2 self-conflicts (safe)
⑤: VD → Memory	Remove the conflicting directory entry from the VD bank	Write back the corresponding cache line from the core's L2 to memory if in Dirty state; invalidate the line from that L2	Leak only VD self-conflicts (safe)

Table 2: Summary of SecDir transitions.

consider a victim's self-conflicts in its private caches as part of the attack model, since they are not caused by an active attacker.

The second case is when the conflicting TD entry shows that there are one or more caches with a copy of the cache line. Hence, evicting the cache line from the L2s would create a vulnerability like in conventional directories (Section 4.2.1). Consequently, in this case, the state in the conflicting TD entry is migrated to VD (③). Specifically, for each of the sharers of the line, as specified in the TD entry, SecDir creates an entry in the corresponding local VD bank. To be secure, every single sharer needs to have a VD entry, because every sharer needs to retain the line in its L2. Fortunately, this operation is local to the directory, does not generate cache coherence transactions, and has no impact on L2 cache states. In particular, the victim process is unaffected: it continues to access the cache line out of its L1/L2 caches, completely unaware that the directory entry has moved from the TD to the VD. Further, after the entry is inserted in the VD, it cannot be tampered with by the actions of other cores, thanks to the partitioned nature of the VD.

A directory entry in the VD can be moved out of the VD in two cases. The first one is when, due to a conflict in an L2 cache, a cache line is evicted from an L2. In this case, SecDir consolidates all the VD entries for the line (which result from multiple cores sharing the line) into a single TD entry (④). It also inserts the line into the LLC, so that future accesses to this cache line get it from the LLC.

Note that SecDir has to consolidate entries from as many VD banks as cores share the line. Hence, this operation requires searching all the VD banks in the slice and, for each match found, removing the entry from the VD bank and setting a bit in the presence bit vector in the new directory entry in the TD. Fortunately, an L2 line eviction is not on the critical path of serving cache accesses, and this search overhead can be hidden. Further, Section 5 shows how to optimize this operation. Finally, note that this transition does not create a vulnerability, since it is created by a victim self-conflict in the victim's L2. The resulting self-eviction is not part of our attack model because it is not caused by an active attacker.

The second case when SecDir moves an entry out of the VD is a conflict in the VD. The conflicting entry cannot be moved back to TD (where it came from) because it could cause a conflict deadlock. Instead, SecDir discards the entry, and invalidates the corresponding cache line from the corresponding L2 — writing it back to main memory if dirty (⑤). Any other copies of the line in other L2s, together with their VD entries, are undisturbed.

This operation is secure according to our model because, as the VD is partitioned, such VD conflicts can only be self-conflicts among directory entries owned by the same core. An active attacker attempting a cross-core cache attack has no way to directly enforce such VD self-conflicts. Also, recall that we size the distributed VD for a core across all the slices to be similar to the size of the core's L2. In this case, even in the worst case when an attacker forces all of the victim's directory entries into the victim's VD, the victim will likely still be able to retain most of its L2 lines.

The SecDir transitions are summarized in Table 2.

5 VICTIM DIRECTORY DESIGN

This section discusses how to access the VD, and VD's features for security and efficiency.

5.1 Accessing the Victim Directory

The VD is accessed differently than the ED and TD. Let us assume that the ED and TD have associativities equal to W_{ED} and W_{TD}, respectively, and that they have the same number of sets. As shown in Figure 4(a), the ED/TD are accessed as a conventional cache structure of $W_{ED} + W_{TD}$ associativity. For simplicity, the figure neglects the Coherence State bits of each directory line. When an address tag match is found, the sharer information is read.

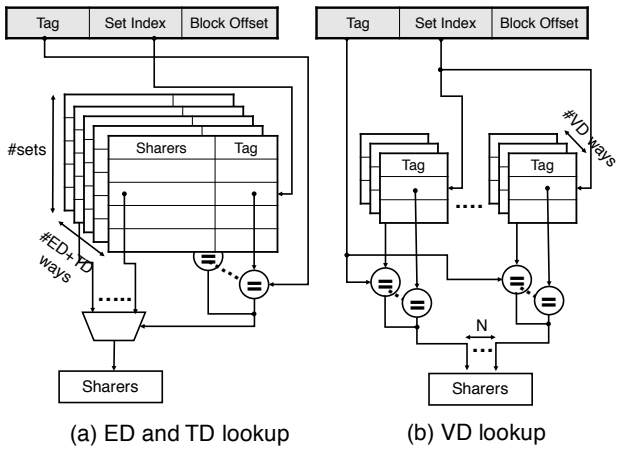


Figure 4: Accessing directories. For simplicity, we neglect the Coherence State bits.

In a VD bank, there is no sharer information field. An access only provides a single bit (hit or miss). A directory entry can be in multiple VD banks. Hence, as shown in Figure 4(b), SecDir needs to potentially search all the local VD banks. Each VD bank may match, in which case it contributes with a set bit to a presence bit vector. The vector gives the sharing information for the cache line.

Given the size of the ED and TD, most accesses under attack-free conditions hit in the ED/TD. However, we still want the VD access to be fast and, therefore, SecDir keeps the associativity of each VD bank (W_{VD}) modest. Note that searching all the local banks in parallel does not slow down the VD access, as the searches are independent. Unlike searches in an associative cache, this design does not need an additional multiplexer to select one of the banks, as it produces a bit vector.

There are three types of searches performed on the VDs of a slice. On a read request, SecDir only needs to find *one* VD bank with the matching address tag. The coherence protocol will access the L2 of the core that owns the bank, and retrieve the line. On a write, SecDir searches all the local VD banks to obtain the complete sharer bit vector. A VD entry for the writing core is allocated (if it does not exist already), and all the other matching entries are invalidated. Finally, if SecDir performs transition ④ in Table 2 (i.e., a cache line is evicted from a private cache), the VD operation involves finding all the matching entries in VD, generating a complete sharer bit vector, creating a TD entry, and invalidating all the matching entries in the VD.

In machines with many cores, SecDir can save hardware by performing the VD search operation in batches — e.g., by accessing and searching 8 VD banks at a time. This implementation saves hardware, but results in slower searches. In this case, on a read operation, SecDir calls off the search as soon as one matching entry is found.

5.2 Victim Directory Features

The VD has two features that are helpful in two different scenarios: one helps in pathological directory conflicts caused by attackers, and the other in executions without attacks.

5.2.1 VD Bank Organization as a Cuckoo Directory. In a directory attack, the attacker tries to cache in the private caches of multiple cores many lines that map to a single set of a single directory slice. In the worst case, the attacker completely fills the set of both ED and TD in the slice, and SecDir has to move all the victim directory entries in that set to the victim's VD bank.

While the attacker concentrates its accesses on lines that map to specific directory sets and slices, a benign victim application generally distributes its directory entries across directory sets and slices evenly. Consequently, in our design, we size the distributed VD for a core across all the slices so that it holds as many entries as the number of lines that fit in an L2 cache. This design should allow the VD to retain many of the directory entries needed by the victim. However, the victim may still suffer self-conflicts in the VD. To minimize the number of self-conflicts in the VD, SecDir organizes each VD bank as a *Cuckoo Directory* [10, 11].

A cuckoo directory is an organization that increases the occupancy of a directory by using multiple hash functions to insert an entry in the directory. The result is a higher effective associativity and, hence, fewer evictions. In SecDir, a cuckoo directory has a second advantage: it obscures victim self-conflict patterns from the attacker.

A cuckoo directory admits multiple organizations, some more sophisticated than others. In SecDir, we use a very simple design, to show the potential of the scheme. Specifically, to insert or look-up an entry, a VD bank is accessed with two hash functions (i.e., $h_1(x)$ and $h_2(x)$) in a pipelined manner. Consider the insertion of an entry first. If any of the hash functions picks a set with an empty slot, the entry is stored in an empty slot. If both hash functions pick full sets, an entry in one of the sets is evicted, leaving an empty slot for the incoming entry to use. If the evicted entry had been hashed with $h_1(x)$, it is now hashed with $h_2(x)$, or vice-versa, landing in another set. If there is an empty slot there, it takes it; otherwise it evicts an entry there, which is rehashed again. This process is repeated for up to *NumRelocations* times, before an entry is evicted to memory for good. Appendix B shows an example of this mechanism.

On a VD bank look-up, the two hash functions can return at most one hit. To confirm a VD bank miss, both functions have to miss. Note that the cuckoo organization requires one extra bit per VD entry, to indicate which hash function was used (*Cuckoo* bit).

The cuckoo operation is useful during an attack, as it increases the occupancy of victim VD banks and reduces victim self-conflicts in VD. Moreover, even if the victim suffers self-conflicts in its VD banks, the randomization of the conflicts obscures the conflict patterns from the attacker. This increases the victim's resistance against even a passive attacker.

5.2.2 Early Detection of VD Misses. In an execution without attacks, the VD is likely to be highly underutilized. In this case, it is desirable to quickly detect when a VD bank access is guaranteed to miss, and save energy by skipping the access. SecDir supports this operation by adding an *Empty Bit* (EB) to each set of each VD bank. The EB bit is wired to the NOR of the Valid bits of all the entries in the set of the VD bank. Hence, if an EB bit is set, it means that the corresponding set in the VD bank is empty.

With this support, VD bank queries that search for a certain directory entry proceed as follows. First, the hardware accesses the EB arrays of the VD banks with the correct set index bits. If an EB array returns a logic one, the hardware skips the ordinary access to the VD bank array; otherwise, it proceeds with the access. This design saves energy. Moreover, since accessing the EB arrays is faster than accessing the VD bank arrays, this design also saves some access latency.

The EB hardware can be organized in different ways. One implementation has a separate EB array per VD bank. Another combines the EB bits of all the VD banks into a single EB array of width N.

6 DICUSSION: VD TIMING CONSIDERATIONS

The fact that the VD is accessed after the ED/TD are accessed raises the question of whether an attacker could exploit a timing side channel. Specifically, an attacker could push a victim's directory entries from the ED/TD to the VD, and then time the execution of the victim's program to find whether it takes longer.

If the victim is a single-threaded program, there is no such timing side channel. A victim's execution is oblivious to whether the directory entry is in ED/TD or VD. In either case, the corresponding cache line is in the victim's private cache. On accessing the line, the victim hits in its private cache, and does not access the directory.

The situation is different in a multi-threaded victim where two or more victim threads share writable data. Specifically, every time that one core writes a line and sends an invalidation to another core, the directory is accessed. Similarly, when a core accesses

the line and obtains it from another core's cache, the directory is accessed. In both cases, accessing the directory in the VD rather than in the ED/TD makes the coherence transaction take a bit longer. For example, using the parameters of the system we evaluate in Section 8, accessing the VD extends by about 7 cycles a transaction that would otherwise typically take about 100 cycles.

While this timing side channel may be hard to exploit due to the non-determinism of cross-thread communication (i.e., each thread's accesses occur asynchronously to other threads'), we need to disable this side channel. One way to do so is by artificially slowing down a response from the ED/TD by the time it would take to additionally access the VD. A naive solution would apply such slowdown to every ED/TD-satisfied transaction. A more advanced solution would apply such slowdown only to ED/TD-satisfied transactions that involve invalidating or querying another core's cache. We leave the implementation and evaluation of this solution to future work.

Parameter	Value	Parameter	Value
Physical Address		**Extended Directory (ED) in SecDir**	
Line address	40 bits	# ways	W_{ED}
Line offset	6 bits	# sets	2048
L2 Cache		**Victim Directory (VD) in SecDir**	
# ways	16	# VD banks/slice	N
# sets	1024	# ways per VD bank	W_{VD}
Traditional Directory (TD)		# sets per VD bank	S_{VD}
# ways	11	Cuckoo bit (per entry)	1 bit
# sets	2048	Empty bit (per set)	1 bit
Address tag	29 bits		
Extended Directory (ED)			
# ways	12		
# sets	2048		
Address tag	29 bits		

Table 3: Parameters of the Intel Skylake-X directory and SecDir. The number of cores is represented by N.

7 A POSSIBLE DESIGN OF SECDIR

As an example of a possible SecDir design, we take the parameters of the Intel Skylake-X directory [46] and modify them to support SecDir. We are interested in comparing the Skylake-X and SecDir directories for the same total directory storage. For simplicity, in our analysis, we make a few assumptions on the cache coherence protocol and the encoding of the cache coherence states. Specifically, we use the MESI coherence protocol, and encode the sharer information in each directory entry as a "full-mapped" bit vector of N presence bits (where N is the number of cores in the machine) [5]. Using a full-mapped bit vector is reasonable for modest core counts. Also, we neglect any extra bits needed to encode transient cache coherence states. With these assumptions, the structures of Figure 2 only need the following Coherence State bits: TD entries need a Dirty and a Valid bit, while ED and VD entries only need a Valid bit.

The storage of the Skylake-X directory includes TD and ED; the storage of SecDir includes TD, a new ED, and VD. To size VD, we partition the original (i.e., Skylake-X's) ED into a new (i.e., SecDir's) ED and VD. Specifically, we take some ways off the original ED and give the storage to the VD. Hence, the original ED and new ED have the same number of sets but different number of ways. This is the simplest reorganization strategy. We use random replacement in ED and VD, and conservatively neglect the storage taken by any replacement algorithm bits in TD.

Table 3 lists the relevant parameters of Intel's Skylake-X to the best of our knowledge. They include physical address, L2 cache, TD, and ED parameters. The table also shows the SecDir parameters for ED and VD. Since the values of SecDir's parameters will change in our experiments, we refer to them as variables W_{ED}, W_{VD}, and S_{VD}. Recall that each entry in a VD bank has a Cuckoo bit, and each set in a VD bank has an Empty bit (EB).

The ED in Skylake-X has an associativity of 12, and the ED in SecDir has an associativity of only W_{ED}. We use the difference in ways (i.e., $12 - W_{ED}$ ways) to build the VD banks in a slice. Specifically, we consider SecDir designs where W_{ED} is 6, 7, 8, 9, or 10. For a given W_{ED} and core count, we design the VD as follows. We consider VD bank associativities (W_{VD}) ranging from 3 to 8. We choose the VD design with the highest directory entry count and a power-of-two number of sets (S_{VD}) that fits in the storage available.

Once we pick a VD design, we count the number of directory entries that such a design provides to a single core across all the slices. In case of an attack, these are the directory entries that the victim can use in an isolated manner. We compare this number to the number of lines in an L2 cache. The ratio between these two numbers is shown in Figure 5. Values above 1 mean that the per-core VD contains more directory entries than lines in L2.

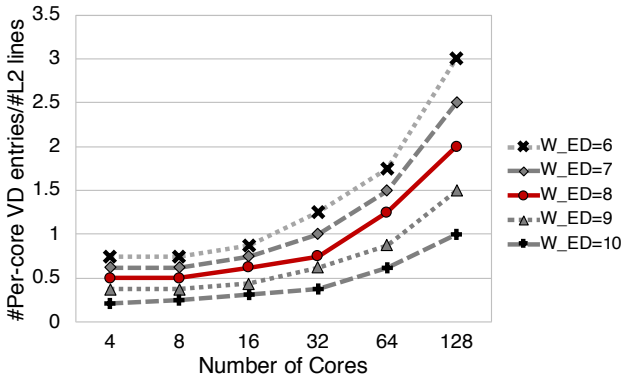


Figure 5: Comparing the number of per-core VD entries machine-wide to the number of lines in an L2 cache. Values above 1 mean that the per-core VD has more entries than lines in an L2. We use a SecDir design with the same directory storage as a Skylake-X.

In SecDir, we want to have at least as many directory entries in the per-core VD as lines in L2. The figure shows that, even allowing the ED to retain a large number of ways (i.e., W_{ED} out of the original 12), we quickly attain a per-core VD that has as many entries as lines in L2. This is because the VD, unlike the ED, does not store sharer information. Specifically, the Skylake-X ED has 12 ways and holds 1.5× as many entries as L2 lines. SecDir can keep 8 ways for the ED (ensuring that the ED holds as many entries as L2 lines), and reassign 4 ways to the per-core VD. At 44 cores or

more, such per-core VD can also hold as many entries as L2 lines or more. If we only have 8 cores, which is the design we evaluate in Section 8, and we still want to keep 8 ways for the ED, the per-core VD needs extra storage to have as many entries as the L2. It can be easily computed that it only needs 28.5 Kbytes per slice to have as many entries as the L2. This is a very small overhead compared to the sizes of an L2 and an LLC slice.

8 EXPERIMENTAL SETUP

We evaluate SecDir and compare it to Skylake-X's directory using simulations with Gem5 [3]. The parameters of the architecture with SecDir are shown in Table 4, which augments the parameters in Table 3. We implement a directory-based MOESI cache coherence protocol. Recall from Section 7 that both SecDir and Skylake-X use the same cache and TD configurations. As shown in the tables, the EDs of Skylake-X and SecDir in a slice have the same number of sets (2048) but different set-associativity, namely 12 and 8, respectively. This means that the ED of SecDir in a slice has as many entries as lines in L2. The VD is designed so that, per core across all the slices, it also has as many entries as lines in L2. As indicated in Section 7, with these parameters, SecDir needs 28.5KB more directory storage per slice than the Skylake-X directory for an 8-core machine. This is a very small overhead considering the sizes of the L2 and the LLC slice. We call the Skylake-X architecture *Baseline*.

Parameter	Value
Architecture	8 cores at 2.0GHz using MOESI dir coherence
Core	8 issue, out-of-order, no SMT, 32 load queue entries, 32 store queue entries, 192 ROB entries
Private L1-I	32KB, 64B line, 4-way, 4 cycle round-trip (RT) latency
Private L1-D	32KB, 64B line, 8-way, 4 cycle RT latency
Private L2	1MB, 64B line, 16-way, 10 cycles RT latency
Shared L3 (per slice)	1.375MB, 64B line, 11-way, 30 cycles RT local latency, 50 cycles RT remote latency
Directory (per slice)	TD: 11-way, 2048 sets; ED: 8-way, 2048 sets; num VD banks: 8; VD bank: 4-way, 512 sets; NumRelocations: 8
Directory RT latency	To TD/ED: same as L3. To VD: over L3, add 2 cycles (EB access) and, if miss in EB, add 5 cycles (VD access)
Network	4×2 mesh, 128b link width
DRAM	RT latency: 50 ns after L3

Table 4: Parameters of the SecDir architecture.

In our cuckoo directory implementation, we use the skewing hash functions proposed by Seznec and Bodin [38] as our $h_1(x)$ and $h_2(x)$ functions. These functions distribute cache lines equally among sets, possess local and inter-bank dispersion properties, and can be easily implemented in hardware. We set the cuckoo NumRelocations to 8.

We evaluate SecDir and Skylake-X with 12 mixes of single-threaded SPEC applications [19] and 10 multi-threaded PARSEC applications [2]. We use the same approach as Jaleel et al. [23] to pick the mixes of SPEC applications. Specifically, we run 23 individual SPECInt2006 and SPECFP2006 applications on a single core and one slice of the non-inclusive LLC structure. These applications are classified into three categories, namely core cache fitting (*CCF*), last-level cache fitting (*LLCF*), and last-level cache thrashing (*LLCT*), according to their L2 and L3 miss rates. We consider the 6 possible combinations of two of these categories, and select 2 application mixes in each combination, as listed in Table 5.

Category	Name & Applications	Name & Applications
CCF, CCF	mix0: 4 gobmk + 4 sjeng	mix1: 4 hmmer + 4 gamess
LLCF, LLCF	mix2: 4 bzip2 + 4 omnetpp	mix3: 4 gromacs + 4 zeusmp
LLCT, LLCT	mix4: 4 libquantum + 4 lbm	mix5: 4 bwaves + 4 sphinx3
CCF, LLCF	mix6: 4 sjeng + 4 omnetpp	mix7: 4 h264ref + 4 zeusmp
CCF, LLCT	mix8: 4 gobmk + 4 libquantum	mix9: 4 namd + 4 bwaves
LLCF, LLCT	mix10: 4 omnetpp + 4 bwaves	mix11: 4 zeusmp + 4 lbm

Table 5: SPEC workload mixes.

We use the reference input size for the SPEC applications. When running these mixes on 8 cores, we run 4 copies of each application, and assign them to different cores. We skip the first 10 billion instructions, and report simulation results for 500 million cycles. For the PARSEC applications, we use the simmedium input size and report simulation results for the region-of-interest (ROI).

9 SECURITY EVALUATION

We evaluate the security properties of SecDir on the AES encryption algorithm [44]. Software implementations of AES usually use 4 look-up tables, called T-tables, to improve the performance of the cipher computation. The encryption process involves multiple rounds, and each round has a round key. To generate the result for a certain round, the algorithm uses the last round's result to look-up the T-tables, and then perform an XOR operation on the obtained value and the round key. This implementation is known to be vulnerable to conflict-based cache attacks [34], as the access patterns on the T-tables leak intermediate encryption results and round keys.

In a conflict-based cache attack, the attacker aims to observe the victim's access patterns on the T-tables. First, it evicts all the T-table entries from the cache hierarchy, and then tests which entries have been accessed by the victim in each round of encryption by analyzing the resulting cache states.

SecDir can effectively block such attacks. In SecDir, the ED and TD are shared by all the cores, but the VDs are per-core private. The most powerful adversary can take full control of the ED and TD, but is unable to interfere with the entries in the victim's VD. In order to emulate such a powerful attacker, we simulate SecDir without ED or TD, assuming that the attacker fully controls these two structures. This is the most pathological scenario that an attacker can create.

In this scenario, we run the AES encryption implementation from OpenSSL 0.9.8, and record the access patterns on the T_0 T-table. The table uses 16 memory lines. Figure 6 shows the addresses of the T_0 memory lines accessed as a function of time. Accesses are classified as: (i) main memory accesses or (ii) L1 or L2 hits. Note that, in this experiment, if a line is evicted from L2, its VD entry is evicted too because there is no TD. Consequently, since this is a single-threaded application, no L2 cache miss will hit in the VD; all L2 cache misses will access main memory.

From the figure, we see that the first access to each memory line of the table misses in the cache hierarchy and causes a main memory access. As the line is fetched from memory, its directory entry is inserted into VD, since there is no TD or ED. As seen in the figure, all of the victim's subsequent accesses to the T_0 table lines hit in the private L1/L2 caches. The attacker cannot observe these

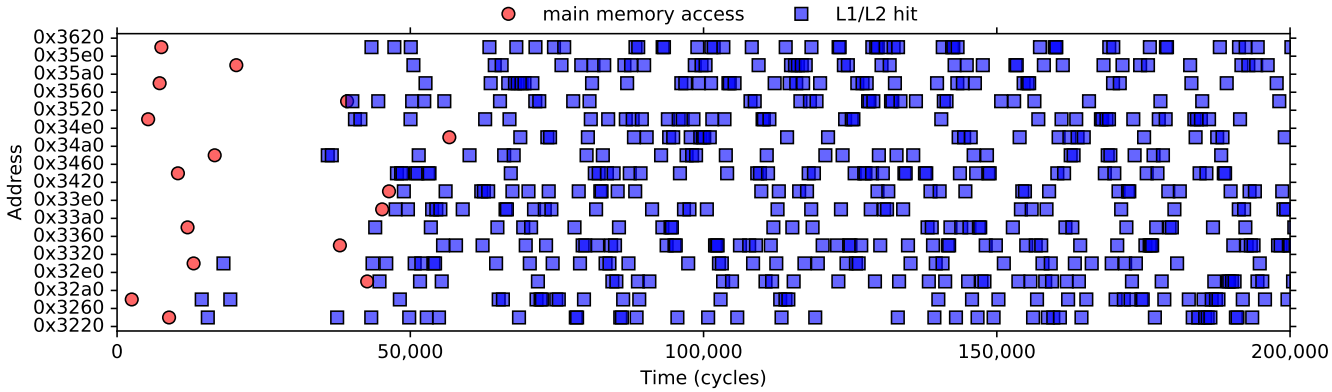


Figure 6: Trace of accesses to the T_0 table in AES encryption. The application runs on SecDir with VD but no ED or TD.

private cache hits, and cannot directly interfere with the entries in the victim's VD.

SecDir is effective at protecting other applications, such as the square-and-multiply operations in the RSA encryption algorithm. The data in the leaky region of RSA is much smaller than the T-table. Hence, it fits in the L2 cache and its directory entries fit in the VD. An attacker cannot evict the target lines from L2.

If the victim's security-sensitive data is large, the victim may suffer self-conflicts in its L2 and/or in its VD. In this case, the victim may leak information. However, recall that protecting against such leakage is not within the scope of SecDir, as such leakage is not created by an active attacker.

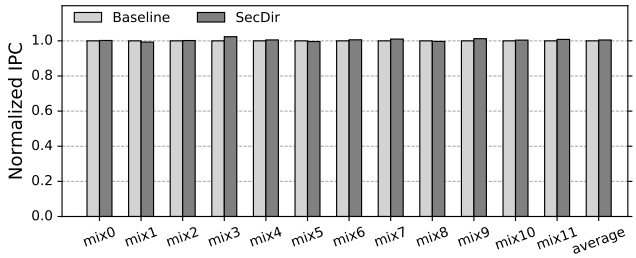
10 PERFORMANCE & AREA EVALUATION

10.1 Evaluation of SPEC Application Mixes

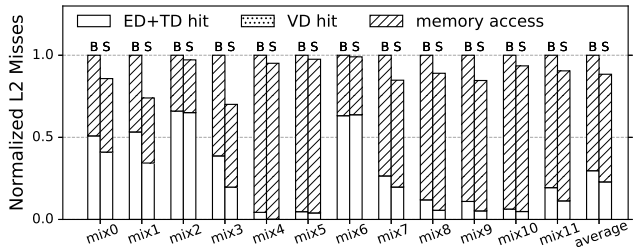
Figure 7 evaluates SecDir executing SPEC mixes. Figure 7(a) shows the average instructions per cycle (IPC) of the SPEC mixes running on SecDir and Baseline. For each mix, the bars are normalized to Baseline. From the figure, we see that the IPC of the mixes changes little across architectures. The reason is that SecDir has a positive and a negative effect on the execution time, and both effects tend to cancel out.

SecDir's negative effect is that it slightly increases the latency of a main memory access. This is because a request that misses in ED and TD checks the VD on its way to main memory. As indicated in Table 4, checking the VD takes 2 cycles (if the EB array satisfies the request) or 2 plus 5 cycles (if it does not). Under attack-free conditions, the VD is not highly utilized, and most VD accesses miss. However, this does not mean that the VD is ineffective, since whatever entries the VD contains are very useful. Indeed, such entries enable the L2 to retain cache lines and, therefore, avoid an L2 miss and directory access in the first place.

SecDir's positive effect is that it reduces the number of directory entry conflicts and, therefore, the number of cache line inclusion victims. In Baseline, a TD entry conflict typically results in an L2 cache line eviction to DRAM. In SecDir, instead, the directory entry is moved to VD and the corresponding cache line remains in L2, avoiding an inclusion victim. Avoiding L2 inclusion victims results in fewer memory accesses and fewer ED/TD accesses. This effect improves performance.



(a) Normalized IPC.



(b) L2 miss characterization. In the figure, *B* is Baseline, and *S* is SecDir.

Figure 7: Evaluation of the SPEC mixes.

To understand the positive effect, Figure 7(b) shows the number of L2 misses and breaks them into: (i) hits in the ED or TD, (ii) hits in the VD, and (iii) misses in the directory, which cause main memory accesses. For each SPEC mix, the figure shows bars for Baseline (B) and SecDir (S), which are normalized to the former.

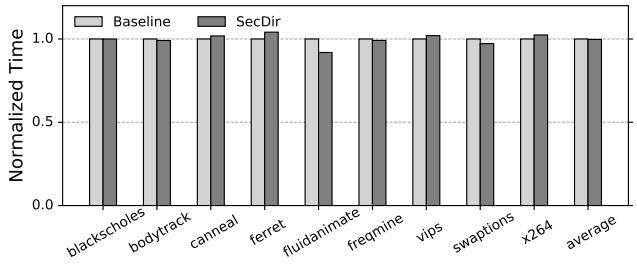
From the figure, we see that SecDir decreases the number of L2 misses in practically all the mixes. On average, the reduction is 11.4%. The reduction comes from reducing L2 inclusion victims. It results in both fewer memory accesses and fewer ED/TD hits.

The bars show that there are no VD hits. This is the normal behavior in single-threaded applications and, as indicated before, it does not mean that the VD is useless. On the contrary, for VD entries in use, there is no reason to access the VD: the L1/L2 caches contain the line and intercept any access to the line from the core. When the line is evicted from L2 and written back to the LLC, SecDir migrates the VD entry to TD (④ in Figure 3(b)). Hence, subsequent L2 misses will either hit in the TD or, if the entry is evicted from TD

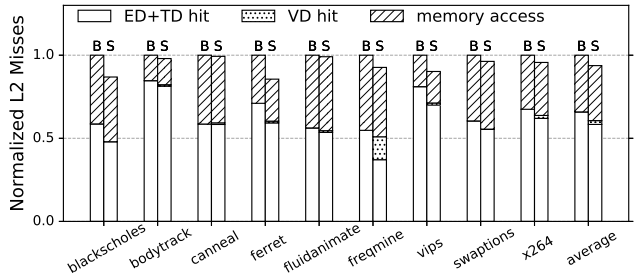
before the line is re-referenced, obtain the data from main memory. In none of the cases will the VD hit.

10.2 Evaluation of PARSEC Applications

Figure 8 shows the same data as Figure 7 for the PARSEC applications. Figure 8(a) shows the execution time on Baseline and on SecDir, normalized to the Baseline. Similar to Figure 7(a), SecDir has a very small impact on the performance of PARSEC applications. As before, this is a combination of both positive and negative effects.



(a) Normalized execution time.



(b) L2 miss characterization. In the figure, *B* is Baseline, and *S* is SecDir.

Figure 8: Evaluation of the PARSEC Applications.

Figure 8(b) shows the L2 miss count of PARSEC applications in Baseline and in SecDir, normalized to Baseline. As in Figure 7(b), the bars are broken down into: (i) hits in the ED or TD, (ii) hits in the VD, and (iii) main memory accesses. Like in SPEC mixes, SecDir reduces the L2 misses in most applications, causing a reduction in the number of ED/TD hits and, to a lesser extent, in the number of memory accesses. The average L2 miss reduction is 7%.

The figure shows that, on average, the number of VD hits is very small. However, in the *freqmine* application, nearly 14% of the L2 misses are intercepted by VD. This scenario is possible in multi-threaded applications. It occurs when the L2 of a core misses on a cache line present in another core's L2 and the line has its directory entry in the VD of the second core. We expect this case to occur in applications with high data sharing between cores. In this case, a fast VD would improve performance.

Overall, in both single-threaded and multi-threaded applications, SecDir provides an effective defense against directory based side-channel attacks, while introducing a negligible performance cost in the absence of attacks.

10.3 Evaluating VD Features

Table 6 examines two features of the VD, namely, the Empty bit (EB) and the cuckoo organization. To assess the impact of the EB, we compare the number of VD bank look-ups performed when using the EB (EBVD) to the number of VD bank look-ups performed when not using the EB (NoEBVD). Recall that, without EB, we need to perform a look-up of the N VD banks in a slice every time that we miss in the ED/TD directory.

SPEC Mix	EBVD / NoEBVD	CKVD / NoCKVD	PARSEC Appl.	EBVD / NoEBVD	CKVD / NoCKVD
mix0	0.45	0.75	blackschol.	0.01	0.63
mix1	0.18	0.95	bodytrack	0.18	0.55
mix2	0.36	0.53	canneal	0.09	0.51
mix3	0.53	0.93	ferret	0.23	0.59
mix4	0.38	1.00	fluidanim.	0.08	0.46
mix5	0.44	0.84	freqmine	0.38	0.62
mix6	0.43	0.41	vips	0.24	0.83
mix7	0.49	0.81	swaptions	0.01	0.56
mix8	0.45	0.96	x264	0.34	0.59
mix9	0.52	0.90	Avg.	0.17	0.59
mix10	0.48	0.73			
mix11	0.47	1.03			
Avg.	0.43	0.82			

Table 6: Evaluating the Empty bit and cuckoo organization.

The columns in the table labeled EBVD/NoEBVD show the ratio between the two measures. Since the VD is under-utilized in executions without attacks, the EB effectively decreases the number of VD bank look-ups across all applications. On average, only 43% and 17% of the VD bank accesses are needed with EB in SPEC and PARSEC applications, respectively. For two PARSEC applications, namely blackscholes and swaptions, EB eliminates practically all of the VD look-ups, as the VD remains highly unused during execution.

We also evaluate the effectiveness of using the cuckoo organization in the VD under the worst possible attack conditions. Specifically, we assume the case where the adversary fully controls the TD and ED directories, and the victim can only use the VD. Consequently, we disable TD and ED, and the application can only use its VD. We compare the number of VD self-conflicts when the VDs are used as cuckoo directories (CKVD) and when they are used as plain directories (NoCKVD). CKVD uses two of the skewing hash functions proposed by Seznec and Bodin [38] (Section 8); NoCKVD simply uses one of them.

The columns in the table labeled CKVD/NoCKVD show the ratio between the two self-conflict counts. We see that the cuckoo organization often eliminates a substantial fraction of the self-conflicts. The impact on a given application depends on a variety of factors, including the application's access patterns and its working set. On average, with SecDir's simple cuckoo organization, 82% and 59% of the VD self-conflicts remain in SPEC and PARSEC applications, respectively. Note that there are two SPEC mixes (i.e., mix4 and mix11) for which our cuckoo design does not reduce the number of conflicts. As shown in Table 5, these mixes contain last-level cache thrashing (LLCT) applications. In these cases, the VD bank in one or more slices is full, and the cuckoo approach is unable to reduce self-conflicts. To reduce the self-conflicts in these mixes, we need to either increase the size or associativity of VD, or make the cuckoo implementation more sophisticated — e.g., by improving the hash

functions used, or by increasing NumRelocations. We leave these avenues to future work. In any case, although our threat model does not target victim self-conflicts, we note that the cuckoo operation effectively reduces self-conflicts and obscures victim self-conflict patterns from the attacker.

10.4 Storage and Area Overhead

We compute the storage and area required by the directory of the baseline Skylake-X architecture and of SecDir. We use the parameters of Section 8. We compute storage in Kbytes and area in mm^2 as given by CACTI 7 [1] using 22 *nm* technology. Table 7 shows the per-slice results for TD, ED, and VD.

Baseline Structure	Storage (KB)	Area (mm^2)	SecDir Structure	Storage (KB)	Area (mm^2)
TD	107.25	0.080	TD	107.25	0.080
ED	114.00	0.087	ED	76.00	0.057
—	—	—	VD	66.50	0.057
Total	221.25	0.167	Total	249.75	0.194

Table 7: Storage and area used by the directory structures in a slice in the baseline Skylake-X and in SecDir.

We see that SecDir requires 28.5 KB additional storage per LLC slice, which is 12.9% more than the baseline architecture. Also, the SecDir directory structures take 0.027 mm^2 more area, which is 16.2% more than the baseline. Overall, these are modest numbers. Also, this analysis is for a machine with 8 cores. As indicated in Section 7, SecDir uses less directory area than the baseline for 44 cores or more.

11 RELATED WORK

Many works have been proposed to defeat cache-based side-channel attacks. We classify these defense mechanisms into two categories, isolation-based and randomization-based. These approaches all have limitations when applied to directory structures, especially when the number of cores is high.

Isolation-based defenses. Isolation-based defense mechanisms rely on cache partitioning techniques to block unintended cache interference. There are two types of cache-partitioning techniques depending on the total number of partitions required.

The first type partitions the cache into as many regions as the number of security domains. Static way-partition [36] provides isolation by statically assigning certain cache ways to each security domain. Unfortunately, this approach can introduce serious performance overhead, since the cache cannot be dynamically shared. Moreover, when the number of cores is higher than the cache associativity, some cores cannot get a cache partition, resulting in serious under-utilization of core resources. Note that this scenario is very common in modern server processors, which calls for scalable defense solutions.

DAWG [25] is a dynamic way-partitioning technique, which is designed to address side channels via cache occupancy states, coherence states, and replacement information. However, when the number of security domains is higher than the cache associativity, DAWG is forced to frequently re-assign cache ways from one domain to another. These re-assignment operations can leak cache occupancy information from the victim to the attacker. Other dynamic partitioning techniques, such as SecDCP [42], NoMo cache [9], and SHARP [45] have similar security issues. SecDir, instead, can scale to high numbers of cores without losing the security guarantees, since the per-core VD structure can flexibly provide as many partitions as the number of cores.

The second type of partitioning technique, used by CATalyst [28] and STEALTHMEM [24], partitions the cache into two regions, i.e., a security-sensitive region and a non-secure shared region. The first region is reserved for security-sensitive data accesses. Cache interference within the security region is blocked via page coloring. The non-secure region can be dynamically shared by other applications. However, both approaches require programmers to provide information about security-sensitive data or instruction accesses. Such information is not easy to obtain for many applications.

Randomization-based defenses. There are several mitigation techniques that rely on the randomization of the address mapping logic or system timing components. CEASER [35] and RPcache [43] randomize the mapping of addresses to cache sets to prevent the attacker from evicting target lines from the cache. For example, CEASER dynamically remaps cache lines, so that the attacker cannot find an effective group of addresses that are mapped to the same set as the target address. However, both techniques can only reduce the bandwidth of the attack, instead of eliminating it. The attacker can still perform the evict operation when it accesses enough lines across a large number of cache sets.

TimeWarp [31] and FuzzyTime [20] disrupt timing measurements by adding noise to the system clock. They can protect against attacks which measure cache access latency and execution time, but they are unable to prevent alias-driven attacks [15]. Furthermore, they hurt benign programs that require a high-precision clock. Liu et al. [29] proposed the random fill cache architecture for the L1 cache to defeat the cache-collision attack (i.e., the reuse-based attack). However, this approach may suffer substantial performance degradation if applied to the much larger last-level cache.

12 CONCLUSIONS

This paper presented *SecDir*, a secure directory to defeat directory side-channel attacks. SecDir takes part of the storage used by a conventional directory and re-assigns it to per-core private directory areas used in a victim-cache manner called Victim Directories (VDs). To minimize victim self-conflicts in a VD during an attack, a VD is organized as a cuckoo directory. Such a design also obscures any conflict patterns from the attacker. We modeled with simulations the directory of an Intel Skylake-X server and a modified design that supports SecDir. Our results showed that SecDir has a negligible performance impact. Furthermore, SecDir is area-efficient: it only needs 28.5KB more directory storage than the Skylake-X per slice for an 8-core machine, while it uses less storage than the Skylake-X for 44 cores or more. Finally, a cuckoo VD organization eliminated substantial victim self-conflicts in a worst-case attack scenario.

ACKNOWLEDGMENTS

This work was funded in part by NSF under grants CCF-1725734 and CNS-1816226, and by an Intel Strategic Research Alliance (ISRA) grant.

APPENDIX A: A LIMITATION IN SKYLAKE-X

Among the attacks that Yan et al. [46] have demonstrated on the non-inclusive caches of Intel's Skylake-X processor, one of them does not exploit the limited associativity of the overall directory structure — which, as discussed in Section 2.3, is the root cause of directory attacks. Instead, Yan et al.'s prime+probe attack exploits a limitation in the implementation of the cache coherence states in Skylake-X's cache hierarchy. In this section, we discuss this limitation and suggest a simple method to fix it. Such a fix has been incorporated in our SecDir implementation.

Consider a victim with the target cache line in its private cache in the Exclusive coherence state, and the line's directory entry in the ED. The attacker creates conflicts in the ED, evicting the target line's directory entry from ED to TD. In Skylake-X, each TD entry must be associated with data — i.e., it must have a corresponding cache line in the LLC. Consequently, as the directory entry moves from ED to TD, the target line is copied to LLC. Unfortunately, the line cannot remain, at the same time, in Exclusive state in the victim's private cache. Hence, it is invalidated from the victim's private cache. This invalidation causes an inclusion victim, which is leveraged by the attacker to complete the prime step. This is a limitation of Skylake-X's implementation, since this invalidation is unnecessary.

To fix this limitation, we suggest to allow TD entries to be associated with empty LLC lines. In the example presented, as the directory entry is moved from ED to TD, we propose to keep the LLC entry empty, and retain the Exclusive cache line in the private cache. In this way, ED conflicts do not cause L2 evictions. After adopting this mechanism, the only way to create an L2 eviction is to exploit the limited associativity of the combined TD plus ED directory structure, as discussed in Section 2.3.

APPENDIX B: CUCKOO DIRECTORY EXAMPLE

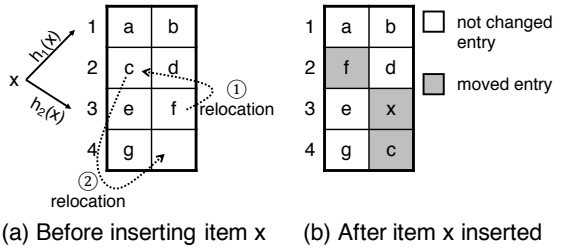


Figure 9: Example of cuckoo directory insertion operation.

While a cuckoo directory admits sophisticated organizations, SecDir uses a very simple design. Figure 9 shows an example of how to insert an item in a two-way set-associative directory that uses SecDir's design. When inserting a new item x, assume that the two hash functions $h_1(x)$ and $h_2(x)$ select sets 1 and 3 (Figure 9(a)). Since both sets are full, one of them is selected and one entry in that set is evicted to provide space for x. Assume that item f in set 3 is selected for eviction. After its eviction, f is re-inserted in a line of its alternative set — Set 2 in the figure (①). Since there is no space in Set 2, one item needs to be evicted and relocated. Assume that it is item c, which is moved from Set 2 to 4 (②). Since there is space in Set 4, c is inserted and there is no additional eviction. Figure 9(b) shows the resulting directory state. Shaded entries are those that have been moved.

In the general case, the relocation procedure is repeated until it either finds an empty slot in the directory, or until a maximum number of relocations (*NumRelocations*) is reached. In the later case, an item is kicked out of the directory structure, which is not likely to be from the same cache set as the item that was first inserted. This fact confuses the attacker.

REFERENCES

[1] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* (2017).

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* (2011).

[4] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer.

[5] Lucien M. Censier and Paul Feautrier. 1978. A new solution to coherence problems in multicache systems. *IEEE transactions on computers* (1978).

[6] David Chaiken, John Kubiatowicz, and Anant Agarwal. 1991. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[7] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security)*.

[8] Docker. 2019. Docker: What is a Container? A standardized unit of software. https://www.docker.com/resources/what-container

[9] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* (2012).

[10] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. ACM.

[11] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. 2011. Cuckoo directory: A scalable directory for many-core systems. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE.

[12] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[13] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[14] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security)*.

[15] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *IEEE Symposium on Security and Privacy*. IEEE.

[16] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy*. IEEE.

[17] Anoop Gupta and Wolf-Dietrich Weber. 1992. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Trans. Comput.* (1992).

[18] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. 1990. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *International Conference on Parallel Processing*.

[19] John L Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* (2006).

[20] Wei-Ming Hu. 1992. Reducing timing channels with fuzzy time. *Journal of computer security* (1992).

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

[22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *Proceedings of the 11th Asia Conference on Computer and Communications Security*. ACM.

[23] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C Steely Jr, and Joel Emer. 2010. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*.

[24] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*.

[25] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *International Symposium on Microarchitecture (MICRO)*.

[26] James Laudon and Daniel Lenoski. 1997. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*.

[27] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security)*.

[28] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[29] Fangfei Liu and Ruby B Lee. 2014. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE.

[30] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*.

[31] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*.

[32] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. 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.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA*. Springer.

[34] Colin Percival. 2005. Cache Missing for Fun and Profit. http://www.daemonology.net/papers/htt.pdf

[35] Moinuddin K Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[36] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. 2009. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM.

[37] Daniel Sanchez and Christos Kozyrakis. 2012. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA)*.

[38] André Seznec and Francois Bodin. 1993. Skewed-Associative Caches. In *International Conference on Parallel Architectures and Languages Europe (PARLE)*. Springer.

[39] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE micro* (2016).

[40] Don Soltis, Irma Esmer, Adi Yoaz, and Sailesh Kottapalli. 2017. The New Intel Xeon Processor Scalable Family (Formerly Skylake-SP). In *Hot Chips: A Symposium on High Performance Chips*.

[41] VMWare. 2014. Transparent Page Sharing: New default setting. http://blogs.vmware.com/security/2014/10

[42] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[43] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*.

[44] Wikipedia. 2019. Advanced Encryption Standard. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[45] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. 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*.

[46] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE Symposium on Security and Privacy*.

[47] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security)*.

[48] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. 2010. 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.