Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects

Miles Dai* *MIT*
Riccardo Paccagnella* *UIUC*
Miguel Gomez-Garcia *MIT*
John McCalpin *TACC*
Mengjia Yan *MIT*

Abstract

This paper studies microarchitectural side-channel attacks and mitigations on the on-chip mesh interconnect used in modern, server-class Intel processors. We find that, though difficult to exploit, the mesh interconnect can be abused by an adversary *even when known attack vectors inside the cores and caches are closed*. We then present novel, non-invasive mitigation mechanisms to interconnect side-channel attacks and offer insights to guide the design of future defenses.

Our analysis starts by thoroughly reverse engineering the mesh interconnect to reveal, for the first time, the precise conditions under which it is susceptible to contention. We show that an attacker can use these conditions to build a cross-core covert channel with a capacity of over 1.5 Mbps. We then demonstrate the feasibility of side-channel attacks that leak keys from vulnerable cryptographic implementations by monitoring mesh interconnect contention. Finally, we present an analytical model to quantify the vulnerability levels of different victim and attacker placements on the chip and use the results to design a software-only mitigation mechanism.

1 Introduction

Microarchitectural attacks have become an increasingly serious threat to system security. In a microarchitectural attack, an attacker infers secrets about a victim program by monitoring the side effects of the victim's execution on shared hardware resources. These types of attacks can be used to leak cryptographic keys [6, 30, 61, 73, 75, 101, 103], browsing activity [72, 83, 84], user keystrokes [37, 38, 56, 57, 77, 78], and other secret information [35, 41, 43, 98]. Moreover, these attacks can employ speculative execution to completely bypass memory isolation and leak arbitrary data [13, 54, 59, 81, 86, 87].

In the past two decades, several hardware resources have been exploited to mount microarchitectural attacks, including caches [61, 73, 75, 101], branch predictors [2, 22, 23], and execution ports [5, 10, 33]. Fortunately, we have also seen more practical mitigation mechanisms. For example, to block attacks targeting SMT contexts, operating systems can disallow programs from different security domains from being scheduled onto the same CPU cores. Multiple effective mitigations to block cache side-channel attacks have also been proposed via spatially partitioning shared caches or flushing cache content upon context switches [12, 28, 32, 53, 60, 100, 107].

In this paper, we study an on-chip microarchitectural attack surface that remains open even if all attack vectors in the cores and caches are closed: the on-chip interconnect. Despite active research on microarchitectural security, very few works [20, 74, 82, 90, 92] have explored attacks and defenses on the on-chip interconnect. Such an interconnect extends across the whole chip and is used to connect all on-chip resources, including shared caches, DRAM controllers, and I/O ports. Specifically, the majority of accesses to the last-level caches and all accesses to DRAM need to travel through certain segments of the on-chip interconnect. Given their high accessibility, on-chip interconnects are likely to become increasingly relevant for microarchitectural security, especially as attack vectors inside the cores and caches are closed.

1.1 Challenges of Exploiting Mesh Interconnects

In particular, we explore microarchitectural attacks and mitigations on the *mesh interconnect* used in Intel server processors since 2016 [96]. On these processors, on-chip resources, including cores, private caches, and LLC slices, are organized into tiles and placed on the die using a 2-dimensional grid layout, shown in Figure 1. The mesh interconnect provides a bidirectional link between each pair of neighboring tiles.

There exist two key challenges to building microarchitectural attacks on the mesh interconnect. First, it is difficult to contend spatially on a mesh topology. For contention to occur, the attacker's and the victim's traffic flows must overlap on the interconnect. Server processors with a mesh interconnect contain many cores, providing numerous placement options for the victim and the attacker. Moreover, the mesh interconnect is designed to distribute traffic to avoid congestion. Consider-

*These authors contributed equally to this work.

ing the numerous placement options and sparsely-distributed traffic flows, blindly trying to detect interconnect contention is ineffective. An effective attacker must carefully consider the scheduling policies used by the mesh interconnect.

Second, it is difficult to create temporal contention. Temporal contention happens when the attacker's and the victim's traffic use a segment of the interconnect simultaneously. A memory access that misses in the L2 cache and hits in the LLC slice of a neighboring tile may spend 50 cycles in the cache but only 2 to 3 cycles on the interconnect. The probability for two such memory accesses to contend on the interconnect is low. Moreover, the extra latency caused by interconnect contention is small and thus is very sensitive to noise.

Taken together, these challenges require the attacker to have a thorough understanding of the mesh interconnect protocols. In fact, we show that uncovering details about the interconnect is not only essential for attacks, but also useful for mitigations.

1.2 This Paper

In this paper, we answer the following two questions. First, is it really *feasible* to construct side-channel attacks by only exploiting contention on a mesh interconnect? Second, are there non-invasive approaches that can mitigate interconnect side channels without requiring hardware modifications?

We start by reverse engineering previously unknown details about Intel's mesh interconnect. First, we reverse engineer the *lane scheduling policy* on the interconnect. We find that each segment in the mesh interconnect consists of multiple lanes, and the lane scheduling policy decides which lane a traffic flow will use based on the flow's source and/or destination tile. Interestingly, the policies for vertical rings and horizontal rings are completely different. Second, we reverse engineer the *priority arbitration policy* used at each tile. We find that the priority of a traffic flow is determined by its source tile. The priority information is important since traffic flows with a high priority cannot be delayed by ones with a low priority and thus cannot be used to observe interconnect contention.

We use the reverse-engineering results to build covert and side-channel attacks that exploit mesh interconnect contention. Our attacks work even if the processor has deployed mitigations against a wide range of microarchitectural attacks, including attacks targeting SMT resources, private and shared caches, and DRAM. Our covert channel can achieve a capacity of 1.53 Mbps. Our side-channel attack can extract keys from vulnerable ECDSA and RSA implementations.

Finally, we offer insights into mitigating interconnect side-channel attacks. Specifically, we find that the victim and attacker placements significantly affect attack efficacy. Importantly, not all the cores are equally vulnerable. We then design an analytical model to quantify vulnerability levels and validate this model using our side channel. We use the findings of our model to guide the design of a non-invasive software-based mitigation to interconnect side-channel attacks.

Disclosure We disclosed our findings to Intel in Q2'21. Intel classified our attack as a "traditional side-channel attack", and referred to their guidance on software-based mitigations [46].

2 Background

2.1 Cache Architecture

The cache is used to store data and instructions for fast access. On modern processors, the cache is typically *set-associative*. A *cache line* can reside in any way of a *cache set*, and the cache set that a line maps to is determined by its address bits.

Modern Intel processors have two levels of *private caches* (L1 and L2), and a *shared L3 cache*, also called last-level cache or LLC. The L1 cache is small (e.g., 32-64 KB) and fast, typically responding within 5 cycles. The L2 cache is slightly bigger (e.g., 256 KB-1 MB) and has a latency of 10-20 cycles. Finally, the shared LLC is 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 (DRAM) access latency, which is on the order of 200-300 cycles.

When a memory access is issued by the core, 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.

LLC Slice Organization The LLC of modern Intel processors is organized into multiple *slices* (partitions). In client-class processors, the number of slices is the same as the number of cores. In server-class processors, the number of slices is sometimes greater than the number of cores (as we see in Section 3). Such an organization is helpful to keep the design modular and scalable. Intel processors map each memory address to a particular *slice ID* using a proprietary mapping function that is designed to keep the distribution of cache lines among slices as uniform as possible [43,44,48,50,64,67,102].

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

2.2 On-chip Interconnect

On multi-core processors, an on-chip interconnect connects physical cores, shared caches, and memory controllers. Since the late 2000s, Intel has used a ring interconnect architecture, known as a *ring bus*. However, rising core counts on Intel's high-end processors revealed scalability issues which led Intel to develop the mesh interconnect. This interconnect first appeared in the Knights Landing microarchitecture in 2016 and has since been used by all Intel processors in the Xeon Scalable server series and the high-end Core X-series [96].

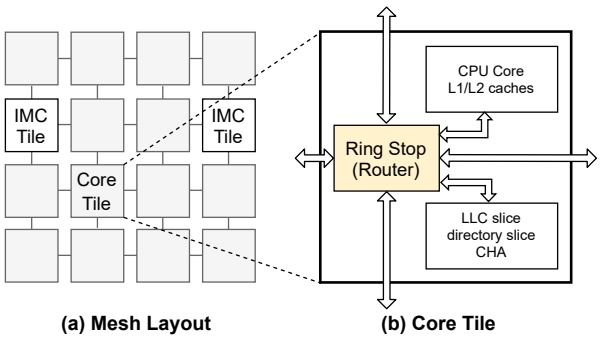


Figure 1: Intel mesh interconnect architecture.

Mesh Topology The interconnect topology determines the physical layout and connections between hardware modules. We call each hardware module a *tile*. Intel's mesh interconnect topology, shown in Figure 1(a), organizes tiles into a 2-dimensional array and forms a grid. Each tile is directly connected to its immediate neighbors [47, 96].

Tile Types Relevant to this paper are two types of tiles: *Core tile*s and *IMC tile*s [96]. As shown in Figure 1(b), a Core tile incorporates a CPU core (including L1 and L2), an LLC slice, and a *ring stop*. The LLC slice includes the data, directory (snoop filter), and a control unit called the Caching/Home Agent (*CHA*), used to maintain cache coherency between tiles. The ring stop is responsible for injecting, forwarding, and receiving traffic to/from the interconnect. It is also referred to as a node router or ring station [74]. An IMC tile includes a ring stop and an integrated memory controller that is connected to off-chip DRAM modules. Cascade Lake processors have two IMC tiles placed symmetrically on the border of the grid.

2.3 Microarchitectural Side Channels

A microarchitectural side channel involves a *transmitter* in the victim's security domain and a *receiver* in the attacker's security domain stealthily communicating with each other. The medium of the communication channel is some microarchitectural structures whose states and occupancy can be modified by the transmitter and the receiver's activities. Similar to prior work [33, 74], we classify microarchitectural side channels into two groups based on the type of resource they exploit: eviction-based attacks and contention-based attacks.

Eviction-based attacks (also called "stateful") focus on microarchitectural resources that hold shared states, such as caches [1, 3, 6, 17, 19, 30, 37–40, 49, 52, 58, 61, 65, 69, 72, 73, 75, 80, 83, 84, 101, 105, 106], TLBs [34], DRAM row buffers [77], and Branch Target Buffers (BTB) [2, 22, 23]. An eviction-based attack generally involves three steps. First, the receiver executes and brings a shared microarchitectural structure into a known state. Second, the transmitter is triggered to modify the shared state based on some secret value. Third, the receiver probes the structure to learn the modified state and infer the secret. The classical cache attacks such as Flush+Reload [101] and Prime+Probe [73, 75] follow the three steps above.

Contention-based attacks (also called "stateless") exploit the finite bandwidth capacity of a resource that is shared by multiple programs. Such resources include functional units [4, 93], cache banks [103], execution ports [5, 10, 33], the memory bus [97], random number generators [21], the on-chip interconnect [74], and the off-chip interconnect [51]. When multiple parties concurrently use such a resource, delays occur which might result in information leaks. In a contention-based attack, information leakage happens only *during the time* when the victim is utilizing the shared resource, in contrast to an eviction-based attack, where the attacker and the victim do not need to simultaneously use the shared resource.

3 Target Architecture and Tile Layout

In this section, we describe some basic architectural parameters of the Intel Xeon Scalable Family Processors on the Purley platform launched in 2017, which implement the mesh interconnect. These parameters include the cache configurations, the tile layout, and the tile mapping which are necessary to reverse engineer the mesh interconnect in Section 6 and carry out the attacks in Sections 7 and 8. Throughout the paper, we run our experiments on a 24-core Intel Xeon Gold 5220R (Cascade Lake) processor running Ubuntu 18.04.

Cache Configurations Our processor features three levels of caches. The L1 is 32 KB with 64 sets and 8 ways. The L2 is 1 MB with 1024 sets and 16 ways. Each LLC slice is 1.375 MB with 2048 sets and 11 ways. Importantly, the shared LLC is non-inclusive. To generate interconnect traffic between two given tiles, we need to generate cache accesses that miss in the private caches and hit in a specific LLC slice. We describe how to do this with a non-inclusive LLC in Section 5.1.

Tile Layout Intel Cascade Lake processors come with three different die configurations, namely, LCC (low core count), HCC (high core count), and XCC (extreme core count). We focus our analysis on the XCC configuration, which consists of 30 tiles, organized into a 5×6 grid, shown in Figure 2.

Prior work [42, 55, 68] provides multiple approaches to reverse engineer the tile layout. We used an approach similar to the one from McCalpin [68], that we describe in Appendix B.

Figure 2 shows the results of our reverse engineering. We label each tile using a 2D coordinate (x, y), where x indicates the row number and y indicates the column number. For example, tile $(0,0)$ is at the top left corner of the chip.

In this processor, there are two IMC tiles, located symmetrically at $(1,0)$ and $(1,5)$. The remaining 28 tiles are all Core tiles. Two Core tiles, $(3,3)$ and $(4,2)$, are completely disabled (black), and another two tiles, $(4,0)$ and $(4,5)$, are partially disabled (yellow). In a partially disabled Core tile, the LLC slice, directory slice, and CHA are enabled while the core and

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)
cpu 0 slice 0	cpu 1 slice 4	cpu 15 slice 9	cpu 16 slice 13	cpu 17 slice 17	cpu 12 slice 22
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)
IMC 0	cpu 14 slice 5	cpu 9 slice 10	cpu 10 slice 14	cpu 11 slice 18	IMC 1
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)
cpu 13 slice 1	cpu 8 slice 6	cpu 20 slice 11	cpu 21 slice 15	cpu 22 slice 19	cpu 23 slice 23
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)
cpu 7 slice 2	cpu 19 slice 7	cpu 3 slice 12	X	cpu 5 slice 20	cpu 6 slice 24
(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)
slice 3	cpu 2 slice 8	X	cpu 4 slice 16	cpu 18 slice 21	slice 25

Figure 2: An example tile layout of an Intel Cascade Lake processor with 24 active cores and 26 active LLC slices.

private caches are disabled. As a result, our processor has 24 active cores and 26 active LLC slices.

The position of the disabled tiles may vary between chips of the same model [68]. For example, we found that on another chip with the same processor model, the completely disabled tiles are located at $(3,1)$ and $(4,5)$ and the partially disabled ones at $(4,0)$ and $(3,5)$. We describe how this variability can be accounted for by an attacker in Appendix B.

Tile Mapping To reverse engineer the mesh interconnect protocols, we also need to know 1) which tile a given CPU ID maps to, and 2) which tile a given LLC slice ID maps to. The CPU ID is used by the operating system (OS). Since our 24-core processor has hyper-threading (SMT) enabled, the OS sees 48 logical CPUs. CPU c and CPU $c+24$ always map to the same tile. The LLC slice ID is kept internally by the hardware and is referred to as the *CHA ID* by Intel.

Our approach to inferring the tile mapping information is similar to the one from McCalpin [68] and is described in Appendix B. Figure 2 shows the two mapping relationships we found using such an approach on our processor. For clarity, we use the 2D coordinates to refer to tiles, CPU IDs, and LLC slices in the remainder of this paper. The reader can use Figure 2 to figure out CPU and slice IDs if needed.

To generate and monitor traffic between two tiles (x_0, y_0) and (x_1, y_1), we pin a process to Core (x_0, y_0) and make it perform cache accesses to LLC slice (x_1, y_1). For conciseness, we use the following format to describe the above configuration:

$$\text{Core}(x_0, y_0) \leftrightarrow \text{Slice}(x_1, y_1)$$

The bidirectional arrow represents the multiple traffic flows in both directions, which we explain in Section 6.

4 Threat Model

Like prior work, we assume that the victim and the attacker are co-located on the same machine and run on the same processor. They belong to different security domains, do not share memory [89, 108], and can run on different processes or virtual machines. We assume a restrictive scenario where the system has adopted effective defense mechanisms against known on-chip side-channel attacks. For example, the system may disallow software from different security domains from concurrently running on the same core [7, 16, 63] and adopt LLC partitioning to prevent cross-core cache attacks [12, 60, 85].[1] Assuming the presence of such mechanisms allows us to study the microarchitectural attack surface beyond known on-chip side-channel attacks. Indeed, the goal is to highlight that even if known on-chip side channels are mitigated, we are still vulnerable to interconnect side-channel attacks.

In the cross-process setup, we consider an attacker who is able to choose its placement (which core to execute on) using the `set-affinity` command. When targeting cryptographic implementations, we also make the standard assumption that the attacker knows the code of the victim as most cryptographic libraries are open source. Finally, we assume that the attacker can observe multiple victim executions to leak multiple bits of the key and increase the efficacy of the attack.

5 Designing Receivers and Transmitters

In this section, we describe the design of the receiver and the transmitter that we use to reverse engineer the mesh interconnect (Section 6) and mount our attacks (Sections 7 and 8).

5.1 Designing the Receiver

The goal of the receiver is to detect contention on the mesh interconnect. Since interconnect contention can be small (a few cycles per load), it is important for the receiver to make accurate and reliable measurements.

Baseline Receiver The receiver monitors the interconnect by pinning itself to a given core and accessing addresses that map to a given LLC slice. These accesses will travel through the mesh interconnect and may be delayed by interconnect contention from other applications. The receiver can then time these accesses to determine whether contention happened.

To generate *reliable* accesses to a given LLC slice, the receiver's accesses need to miss in the L2 and hit in the LLC while avoiding L2 hits and DRAM accesses. To this end, we use two sets of addresses called a *monitoring set* and an *eviction set* (*EV*). The monitoring set monitors traffic between a remote LLC slice and the receiver's core, and the EV evicts the monitoring set from the L2 cache to the LLC. The addresses in the monitoring set are mapped to the target LLC slice and to one or more L2 sets. The addresses in the EV are mapped to the receiver's local LLC slice (to avoid

[1] This is partially possible on today's processors using Intel CAT, which allows the creation of way-based partitions [70]. Additionally, mechanisms that partition cache directories (to block [99]) have been proposed in the literature [11, 15, 100] and may be deployed in future hardware.

generating unnecessary interconnect traffic) and to the same L2 set(s) as the monitoring-set addresses. We can obtain addresses that map to the desired L2 set and LLC slice using prior approaches [74, 102], discussed in Appendix A.

The receiver works in three steps:

1. **Preparation:** The receiver accesses the addresses in the monitoring set, bringing them into the L2 cache of the receiver's core. Since the LLC is non-inclusive, these addresses may not be present in the LLC.
2. **Eviction:** The receiver accesses the addresses in the EV multiple times to evict the addresses in the monitoring set from the L2 cache of the receiver's core. Any addresses in the monitoring set that were not in the LLC will be written back to the corresponding LLC slice [99].
3. **Measurement:** The receiver accesses the addresses in the monitoring set and times the latency of *each access* using `rdtsc`. This latency includes the time for the accesses to travel through the interconnect. Note that this step collects multiple latency samples. The number of samples is determined by the size of the monitoring set.

The last two steps (Eviction and Measurement) can be repeated to collect more latency samples.

Tuning the Receiver We tune the following knobs to find the receiver configuration that gives the most reliable measurements: 1) monitoring set size, 2) EV size, and 3) number of times we access the EV during the Eviction step.

The monitoring set contains addresses that map to multiple L2 sets and multiple addresses mapped to each L2 set. We denote the number of L2 sets as N_{L2} and the number of addresses mapped to each L2 set as W_{L2}. The size of the monitoring set is $N_{L2} \times W_{L2}$. Using a larger monitoring set allows for more consecutive latency samples in the Measurement step (step 3 above), leading to a larger consecutive monitoring window. However, constructing a larger monitoring set takes longer and also requires constructing N_{L2} eviction sets.

We adjust N_{L2} and W_{L2} to tailor the transmitter and receiver to different tasks. In the reverse engineering of Section 6, we set $N_{L2} = 1$ to reduce the time spent creating the monitoring set. We set $W_{L2} = 16$. Recall that each LLC slice has 11 ways but has twice as many sets as the L2 cache. Thus, 22 addresses per L2 set could ideally fit in an LLC slice. In practice, we found that 16 addresses per L2 set works more reliably. When measuring the receiver's temporal resolution (Section 7) and executing side-channel attacks (Section 8), we set $N_{L2} = 32$ to have a large consecutive monitoring window. For the covert-channel (Section 7), we also design a version of the receiver that has an infinite monitoring window and does not require eviction-set accesses, at the cost of a lower sampling density.

Finally, we consider the EV size and the number of times to access the EV. The goal of the EV is to evict all the monitoring set addresses from the L2 cache to the LLC. Through experimentation, we found that accessing an EV with 16 addresses per L2 set 4 times can achieve 100% eviction rate.

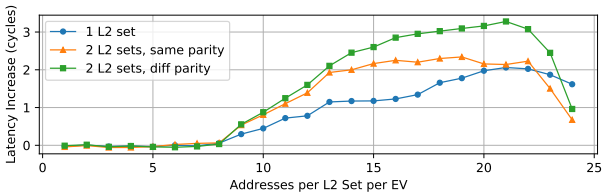


Figure 3: Increased access latency of the receiver for different transmitter configurations.

5.2 Designing the Transmitter

The goal of the transmitter is to reliably generate contention on the interconnect. To this end, it must satisfy the following requirements. First, the transmitter's traffic should incur high contention to help distinguish different levels of contention. Second, the traffic should be reliable, introducing low variance in the receiver's signal. Third, the transmitter should generate as few coherence transactions as possible to facilitate analyzing the traffic patterns.

Baseline Transmitter The baseline transmitter configuration resembles that of the receiver in Section 5.1. The transmitter uses 2 eviction sets: a *target EV* that maps to a given target LLC slice and a *local EV* that maps to the local LLC slice. The two EVs map to the same L2 sets and can evict each other from the L2 to their LLC slices. The transmitter alternately accesses the two EVs to generate traffic to the target LLC slice. The local EV only generates local traffic and does not use the interconnect, simplifying our reverse-engineering process. When addresses in each EV are mapped to multiple L2 sets, the accesses to the different L2 sets are interleaved.

Tuning Transmitter Configurations We adjust the following to find the best transmitter configuration: 1) the number of L2 sets that are mapped to by addresses within each EV, and 2) the number of addresses that map to each of those L2 sets. To compare configurations, we use the receiver from Section 5.1 to monitor the transmitter's traffic and measure the increase in average latency when the transmitter is on. The receiver monitors on $\text{Core}(0,3) \leftrightarrow \text{Slice}(0,2)$, and the transmitter generates traffic on $\text{Core}(0,5) \leftrightarrow \text{Slice}(0,1)$. This configuration was found experimentally to exhibit interconnect contention.

Figure 3 shows the receiver's observation when using different transmitter configurations. A larger latency increase suggests that the transmitter introduced more contention.

Regarding the number of addresses per L2 set, we see that when the number of addresses per L2 set is 8 or lower, the latency difference is zero. Since the L2 associativity is 16, the transmitter is unable to generate LLC hits when both the local and target EVs have 8 or fewer addresses. Thus, no interconnect traffic is generated. When there are between 9 and 22 addresses per L2 set, the latency difference increases, indicating that the transmitter generates increasing levels of

contention. Above 22 addresses per L2 set, the difference decreases as the transmitter experiences LLC misses. The loads are served from DRAM, slowing down the transmitter.

Regarding the number of L2 sets used by each EV, we observe that when the transmitter uses two L2 sets instead of one, the receiver observes higher contention. For example, when the transmitter uses two L2 sets and 15 addresses per L2 set, the receiver observes a 2.1-cycle average difference, which is nearly twice the 1.1-cycle average difference when using a single L2 set. Further increasing the number of L2 sets to 3 or above had negligible impact.

We also found that the parity of the set indices of the two L2 sets has an impact on the receiver's observation. Specifically, when the two sets have both even or both odd set numbers (same parity), the latency difference saturates when the number of addresses per L2 set reaches 15. However, when the parities of the two sets are different, the average latency difference keeps increasing and can reach as high as 3.28 cycles.

Optimal Transmitter Configuration Considering all the above factors, we pick the following transmitter configuration since it generates high contention and shows high reliability.

- The addresses in each EV map to two L2 sets.
- The two L2 set indices have different parities.
- The number of addresses per L2 set is 20.

6 Reverse Engineering the Mesh Interconnect

In this section, we use the transmitter and the receiver from Section 5 to reverse engineer the characteristics of Intel's mesh interconnect. In particular, we determine the precise conditions necessary for contention to occur. For both the covert channel (Section 7) and the side-channel attack (Section 8), these findings inform the optimal receiver placement to leak data with a low error rate and high bandwidth.

Overview Intel's mesh interconnect is implemented as a 2-dimensional array of ring interconnects [62, 96]. Traffic on this array follows a Y-X routing policy, meaning that it always travels vertically first and then horizontally [68, 96]. When changing direction, the traffic must jump from a vertical ring to a horizontal ring. Each ring is made of 4 functionally-separate rings: 1) a *request ring*, also known as address ring, 2) a *data ring*, also known as block ring, 3) an *acknowledge ring*, and 4) an *invalidate ring*, also known as snoop ring [47].

Intuitively, contention on the mesh interconnect happens when two memory accesses use the same physical ring, in the same direction, and on overlapping segments. However, this is *not sufficient* to guarantee observable contention. Determining the precise necessary conditions requires answering the following three questions. First, what traffic flows and rings are used by different memory transactions? Second, what is the scheduling policy that allocates these traffic flows to physical lanes? Prior work [74] found that Intel's ring interconnects use a multi-lane organization, but the policy on our processor is different from that of a 1-dimensional ring interconnect. Finally, what is the arbitration policy that determines the priority between multiple in-flight traffic flows?

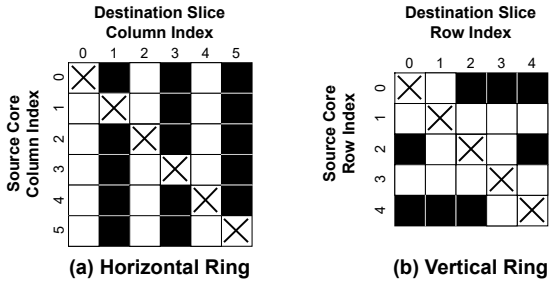


Figure 4: Lane scheduling policy for Core→Slice traffic on horizontal rings and vertical rings. Black and white squares indicate different lanes. For each pair of source and destination tiles, the Slice→Core traffic uses the opposite lane.

In this section, we answer all three questions. Note that all the reverse engineering experiments in this section are carried out with hardware prefetchers disabled [45]. The prefetchers are enabled to model a realistic setup when we perform the covert channel and side-channel attacks.

6.1 Traffic Flows

We first figure out which rings are used by different traffic flows using Intel performance counters (or PMON counters).

We run the transmitter (from Section 5.2) and configure the PMON counters to measure the number of cycles that each ring is used for at each ring stop.[2] These numbers reveal the precise rings and segments of the mesh interconnect that are used by the transmitter.

On running the transmitter, we observed traffic flows from the core to the LLC slice on the request ring and the data ring. Traffic was also observed going from the LLC slice to the core on the data ring and the acknowledge ring. According to our analysis, the data traffic flow from core to slice is due to writeback data. On a non-inclusive cache, when an L2 line is replaced, the data needs to be written back to the LLC [99].

6.2 Lane Scheduling Policy

Prior work discovered that each of the request, data, and acknowledge rings features two physical "lanes" [74]. Intel's Uncore PMON guide for our processor [47] confirms the existence of such lanes and refers to them as "odd/even rings". The guide also describes PMON counter unit masks that can monitor traffic on the two lanes of each ring separately which we use to reverse engineer the lane scheduling policy. Specifically, we run our transmitter and use the PMON counters to identify the lanes used by various transmitter placements.

[2] This is done using the "In Use RING Events". For example, the horizontal acknowledge ring can be monitored with `HORZ_RING_AK_IN_USE` [47].

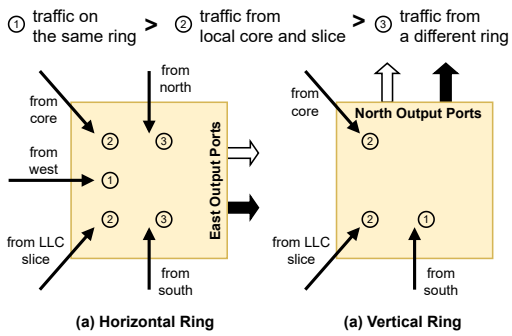


Figure 5: Priority arbitration policy for different traffic flows.

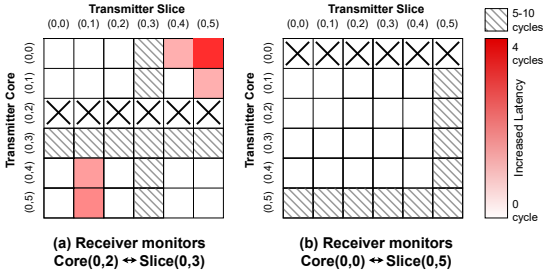


Figure 6: Observed increased access latency by the receiver when (a) the receiver monitors Core$(0,2) \leftrightarrow$ Slice$(0,3)$ and (b) the receiver monitors Core$(0,0) \leftrightarrow$ Slice$(0,5)$.

We find that the lane scheduling policy is a *static* policy that is determined by the source and/or the destination of a given traffic flow. The scheduling policy on the horizontal rings and the vertical rings are *independent* from each other.

Traffic From a Core to an LLC Slice Figure 4 summarizes the scheduling policy, where black and white indicate different lanes. Figure 4(a) shows the lane scheduling policy for horizontal rings. Each row is a column index for the source core and each column is a column index for the destination slice. The horizontal lane policy is destination-based. If the destination slice of a traffic flow is on column 0, 2, or 4, the white lane is used. Otherwise, the black lane is used.

Figure 4(b) shows the lane scheduling policy for vertical rings. The vertical lane policy is source-based except for 4 special cases. If the source core is on row 0, 2, or 4, the black lane is used; otherwise, the white lane is used. The 4 special cases are Core$(0,*)\rightarrow$Slice$(1,*)$, Core$(2,*)\rightarrow$Slice$(1,*)$, Core$(2,*)\rightarrow$Slice$(3,*)$, Core$(4,*)\rightarrow$Slice$(3,*)$. Since the mesh interconnect of our processor has an odd number of rows (5 rows), we believe these 4 special cases were introduced by hardware designers to distribute the traffic served by each link more evenly between the two lanes.

We verified that all horizontal rings use the same lane scheduling policy, including the ones with disabled tiles and the one with IMC tiles. The same applies to the vertical rings.

Traffic From an LLC Slice to a Core For each pair of source and destination tiles in Figure 4, the traffic from a slice to a core uses the opposite lane of the traffic from a core to a slice. For example, Figure 4(a) indicates that traffic from Core$(*,0)\rightarrow$Slice$(*,1)$ uses the black lane. This means that traffic from Slice$(*,0)\rightarrow$Core$(*,1)$ uses the white lane.

6.3 Priority Arbitration Policy

The last missing piece required to fully understand the conditions for *observable* contention to occur on the mesh interconnect is the priority arbitration policy. The priority of different traffic flows is important because a flow with high priority cannot be delayed by a flow with low priority, leading to unobservable contention. To reverse engineer the priority arbitration policy, we use the transmitter and receiver from Section 5 and place them on different cores and slices.

To compare the priority of two traffic flows, we pick a transmitter placement ("*A*"), and a receiver placement ("*B*") such that the receiver can observe contention from the transmitter. To simplify our analysis, *A* and *B* are selected such that the transmitter and the receiver contend on exactly one ring. We then swap the placements. If the contention is still observable, then the traffic flows for *A* and *B* have the same priority. If the receiver can no longer observe contention, then the traffic flow for *A* has a higher priority than the flow for *B*.

Figure 5 shows the results, where we rank the priority of different traffic flows (with ① being the highest priority). When using the horizontal rings (Figure 5(a)), traffic already on the ring has the highest priority; traffic from the local core and slice takes second priority; traffic from the vertical rings has the lowest priority. Figure 5(b) shows the priority on the vertical rings. Once again, traffic already on the ring has a higher priority than the traffic injected from the local core or slice. Due to the Y-X routing policy, there is no traffic flow that switches from a horizontal ring to a vertical ring.

6.4 Case Studies of Timing Measurement

We performed a comprehensive timing analysis of the lane scheduling policy and the priority arbitration policy by running the transmitter and the receiver (Section 5) on all possible combinations of cores and slices. In this section, we show two case studies of the horizontal ring on Row 0 and demonstrate that the reverse-engineering results play an important role in designing effective covert channel and side-channel attacks.

A Case Study for the Lane Scheduling Policy In the case study shown in Figure 6(a), the receiver monitors Core$(0,2)$ $\leftrightarrow$ Slice$(0,3)$. The transmitter placement is varied by trying all pairs of cores and slices. Each row indicates the transmitter's core, and each column indicates the transmitter's LLC slice. The row for Core$(0,2)$ is empty since we do not pin the transmitter and receiver to the same core, which would cause contention on pipeline and private cache structures.

First, we observe high contention (5-10 extra cycles per load) on the Slice$(0,3)$ column and the Core$(0,3)$ row. In the column for Slice$(0,3)$, the transmitter's target EV shares a slice with the receiver's monitoring set, and in the row for Core$(0,3)$, the transmitter's local EV shares a slice with the receiver's monitoring set. In both cases, the high contention is caused by LLC *slice port contention*.[3]

Second, there are 5 transmitter placements that cause interconnect contention with the receiver's traffic, resulting in delays of 1-5 cycles per load. The transmitter's and receiver's traffic contend *only if they use the same lane and ring on overlapping segments and travel in the same direction*. For example, when the transmitter uses Core$(0,1) \leftrightarrow$ Slice$(0,4)$, its traffic overlaps with the receiver's traffic on the east-to-west direction of the request ring and on the west-to-east direction of the data and acknowledge rings. However, Figure 4 shows that the receiver uses the black lane of the request, data, and acknowledge rings, while the transmitter uses the white lane of the same rings, so no contention is observed.

The case study also shows that our receiver is able to observe different levels of contention. Contention on multiple rings can lead to a larger delay. For example, when the transmitter uses Core$(0,0) \leftrightarrow$ Slice$(0,5)$ (the top right cell in Figure 6(a)), the transmitter and the receiver contend on three rings—the request, data, and acknowledge rings. We see a delay of 1-2 extra cycles compared to the other transmitter placement where contention only happens on 1 or 2 rings.

A Case Study for the Priority Arbitration Policy We show another case study where the receiver monitors Core$(0,0) \leftrightarrow$ Slice$(0,5)$. We vary the transmitter placement in the same way as in the previous case study. Similarly, we observe high contention when the transmitter uses Core $(0,5)$ and when the transmitter uses Slice $(0,5)$ from slice port contention. However, we do not observe any interconnect contention despite the receiver monitoring all segments on Row 0 since the traffic injected by the receiver from Tile $(0,0)$ and Tile $(0,5)$ has priority over the traffic injected by the transmitter from the other tiles (Figure 5). Therefore, the receiver's traffic is never delayed due to interconnect contention.

7 Interconnect Covert Channel Attacks

In this section, we use the results from Section 6 to build a cross-core covert channel on the mesh interconnect.

Our attack builds on the receiver and transmitter from Section 5. The transmitter generates traffic to send a bit "1" and remains idle to send a bit "0". We use the version of our receiver optimized for continuous monitoring at the cost of a lower sampling density. In particular, we keep $N_{L2} = 1$ but set $W_{L2} = 24$ for the monitoring set. Because 24 exceeds the L2 associativity, accessing new monitoring set addresses evicts previously-accessed addresses from the L2 into the LLC. This removes the need to access a separate eviction set, allowing for more evenly-spaced measurements. In practice, this self-eviction method results in some private cache hits, so we measure four concurrent loads. The measured latency is the maximum latency of any of the four loads which significantly increases the probability of measuring an LLC hit.

[3] We make the transmitter and the receiver access different LLC sets to ensure that we do not introduce any cache line conflicts and LLC misses.

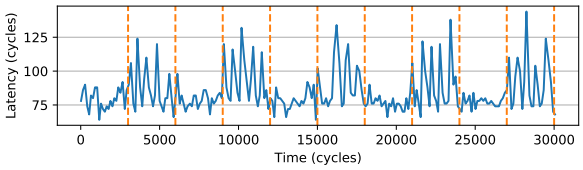


Figure 7: Trace collected by the receiver when the transmitter sends alternating 0s and 1s, with the receiver placement Core$(3,1) \leftrightarrow$ Slice$(2,1)$ and sender placement Core$(4,1) \leftrightarrow$ Slice$(1,1)$, using a transmission interval of 3000 cycles.

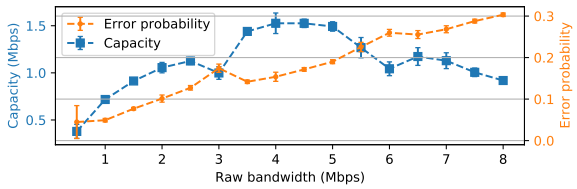


Figure 8: Covert channel capacity and error probability with decreasing interval size (mean across 5 runs).

Figure 7 shows an example trace when placing the transmitter on Core$(4,1) \leftrightarrow$ Slice$(1,1)$ and the receiver on Core$(3,1)$ $\leftrightarrow$ Slice$(2,1)$, and the transmitter sends an alternating sequence of 1s and 0s with an interval of 3000 cycles. Every other interval contains high latency measurements caused by the transmitter's traffic delaying the receiver's measurements and should be decoded as bit "1". The intervals with low latency measurements should be decoded as bit "0".

In this setup, the transmitter and the receiver run on different cores and load from different slices. Hence, they do not share any cache structures (sets, directories, or slice ports). This implies that our covert channel works due to contention on the mesh interconnect only.

Covert Channel Capacity To evaluate the performance of our covert channel, we compute the channel capacity metric, which accounts for both the raw bandwidth and the error probability [71, 74, 77]. Specifically, we vary the interval size and track the error probability over a transmission of 100,000 *random bits*. We perform each experiment 5 times and show the results in Figure 8. We achieve a maximum average channel capacity of 1.53 Mbps ($\sigma = 0.04$) with an interval size of 488 cycles. This capacity is in the same order of magnitude as prior interconnect covert channels [74].

Cross-VM Setup We also build a proof-of-concept where the transmitter and the receiver run on separate virtual ma-

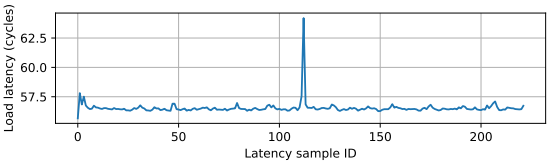


Figure 9: Temporal resolution trace with the transmitter using 16 accesses. For clarity, this plot is averaged across 300 traces.

chines (VMs). We spawn two VMs using QEMU/KVM and pin the transmitter VM to Core $(4,1)$ and the receiver VM to Core $(3,1)$. We use the algorithm described by Yan et al. [99] to construct monitoring and eviction sets and rely on timing measurements to determine the mapping of sets to LLC slices. The receiver targets slices near $(2,1)$ while the transmitter targets slices near $(1,1)$. We mounted the attack successfully with an interval of 1000 cycles and observed a 12.5% average error rate across 4 runs. This corresponds to an average channel capacity of 1.00 Mbps ($\sigma = 0.04$), confirming the feasibility of interconnect cross-VM covert channels.

Running the attack in a real cloud environment also requires two additional steps: 1) the transmitter and receiver need to infer the mesh topology and the ID of the physical core where they are running; and 2) the transmitter and the receiver need to perform an agreement phase where they synchronize with each other. We discuss how to perform the first step in Appendix B. For the second step, our VM configuration allowed for synchronization via the wall clock, but on machines where this is not possible, existing synchronization protocols based on preambles may be used (e.g., [66]).

Temporal Resolution We now measure the temporal resolution of the interconnect channel by configuring the transmitter to spin for 10,000 cycles, issue k LLC loads, and then spin again for 10,000 cycles. We then plot the latency samples collected by the receiver during this time. We use $N_{L2} = 32$ and $W_{L2} = 16$ as our receiver configuration to give us a long enough monitoring window without EV accesses. If we see a spike in the plot for a given k (e.g., Figure 9), then we say that the temporal resolution of the mesh interconnect side channel is at least k loads. The lower k is, the more fine-grained the resolution is. We find that using the same placements of the covert channels above, we reliably observe a contention peak when $k \geq 7$. This resolution is coarser grained than that of traditional cache attacks, which can detect single memory accesses. However, as we will show in Section 8, it is still fine-grained enough to detect secret-dependent LLC loads performed by vulnerable ECDSA/RSA implementations.

8 Interconnect Side-Channel Attacks

We now demonstrate that interconnect side-channel attacks can be used to leak keys from vulnerable cryptographic implementations. We start by describing the basic idea and setup of the attack, followed by examples of attacking fast implementations of ECDSA and RSA. We find that the placements of the victim and the attacker play a critical role in the efficacy of the attack. We then design an analytical model to thoroughly analyze all the placements in Section 8.5.

Algorithm 1: Secret-dependent victim behavior.

```
1 for bit b in secret key do
2     Func1();
3     if b == 1 then
4         Func2();
```

Table 1: Execution time (in cycles) of `Func1` and `Func2` in the fast implementations of ECDSA and RSA.

	`Func1` (miss in L2)	`Func1` (hit in L2)	`Func2` (miss in L2)
ECDSA	17,000 – 18,000	9,050	15,000 – 16,000
RSA	4,000 – 4,100	3,500	3,900 – 4,000

8.1 Victim Setup

Our attack targets fast (insecure) implementations of two cryptographic victims, ECDSA and RSA. These implementations use the code pattern shown in Algorithm 1, which has been targeted in several prior works on microarchitectural side-channel attacks [9, 33, 34, 61, 74–76, 99, 101, 105].

The code iterates over every bit in the secret key and calls two functions based on the secret bit. If the secret bit is 0, only `Func1` executes; otherwise, both `Func1` and `Func2` execute. Therefore, an attacker can infer the secret bit used in each iteration by monitoring whether or not `Func2` is executed.

We use the victim implementations from the libgcrypt library [31]. Specifically, we target the fast implementations[4] of 1) scalar point multiplication (`_gcry_mpi_ec_mul_point`) used in ECDSA during signature generation[5] and 2) modular exponentiation (`_gcry_mpi_powm`) used in RSA during decryption. We measure the execution time for the 2 functions in ECDSA and RSA for the cases when the function hits and misses in the private caches, shown in Table 1. The attacker can obtain this information by profiling the victim application offline. This information is useful when we try to align the latency traces with the victim's execution during the attack.

8.2 Attacker Setup

The attack works as follows. Given a victim placement, the attacker first analyzes the victim's traffic flows, considering

[4] These fast implementations were used in versions 1.6.3 and 1.5.2, respectively. Newer versions, by default, use secure (slower) implementations.

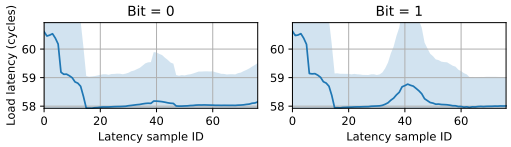
[5] The target function is used to multiply the secret nonce with the group generator. An attacker who learns the nonce can use it to recover the secret key. Our victim runs code from the `pubkey.c` test of libgcrypt, which uses the curve Ed25519 and generates the 256-bit long nonce deterministically.

that the victim's memory accesses are generally uniformly distributed across all LLC slices due to the slice hash function's design. Second, the attacker picks an optimal placement to maximize the observable contention. This placement can be found using the analytical model described in Section 8.5. Next, the attacker triggers the victim to execute and starts collecting latency sample traces. The attacker uses the receiver configuration that can achieve a large consecutive monitoring window (Section 5.1). Specifically, the attacker uses a monitoring set with 32 L2 sets ($N_{L2} = 32$) and 16 addresses mapped to each L2 set ($W_{L2} = 16$). Thus, the attacker can collect 512 consecutive latency samples. Since it takes around 105 cycles to collect and save each sample, the 512 consecutive samples cover more than $50,000$ cycles, enough for one iteration of Algorithm 1 in ECDSA and RSA (see Table 1).

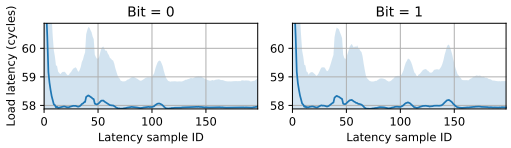
Generating Secret-Dependent Interconnect Traffic A key remaining challenge is how to force the victim to generate secret-dependent interconnect traffic. The victim's memory accesses only use the interconnect if they miss in the private caches and need to access a remote LLC slice or DRAM. The code and data accessed in each iteration of Algorithm 1 fit in our processor's 1 MB L2 cache. Therefore, without extra interference with the victim, the interconnect side channel can only observe the execution of the first iteration of the victim and leak the first bit of the secret key.

To force the victim to generate interconnect traffic for every iteration, we use an approach similar to prior work [26, 74]. The approach requires the system to use a specific defense mechanism against side-channel attacks on private caches that flushes the private caches upon context switches (as suggested by prior work [12,25,27–29,32,39,40,73,75,85,88,107]). The attacker can use this defense mechanism to their advantage. If the attacker can trigger a context switch on the victim, the victim's private cache will be automatically flushed. When the victim resumes execution, the memory accesses will then generate interconnect traffic. Several approaches have been proposed to preempt a victim program from an unprivileged process by exploiting the Linux scheduler [8, 40, 69, 79].

We remark that the assumption of flushing the private caches upon context switches limits the applicability of our attack. However, such an assumption is fairly reasonable. Given that interconnect side-channel attacks are much more difficult to carry out than cache side-channel attacks, it is not necessary to exploit the interconnect side channel on an insecure processor without any protection of the caches. In the following experiments, like prior work [1,3,23,24,36,39,73,74,91], we simulate the preemption and the cache flushing operations by manually stopping the victim at the beginning of an iteration and evicting the victim's core private caches. The eviction is done by accessing an eviction set with W_{L2} addresses for each L2 set (as in [26]). A practical implementation of the attack on the Linux scheduler is beyond the scope of this work.



(a) Results for the RSA victim.



(b) Results for the ECDSA victim.

Figure 10: Example of latency traces with the victim on Core$(0,0)$ and the attacker monitoring Core$(0,2) \leftrightarrow$ Slice$(0,3)$. The light-blue shade is the standard deviation across traces.

8.3 Demonstrating Example of Leaking a Single Bit

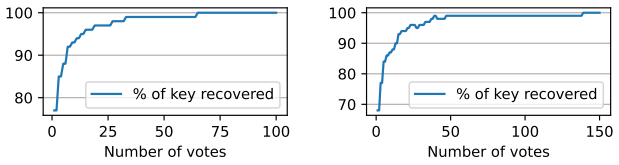
We now present a proof-of-concept demonstration of the attack against ECDSA and RSA. We start with a demonstration of single-bit leakage in this section, followed by the full-key recovery demonstration in Section 8.4. We evaluate how different placements affect attack accuracy in Section 8.5.

For the single-bit leakage demonstration, we target the first bit of the key and thus focus on monitoring the first iteration of the victim loop (Algorithm 1). We run the victim on Core$(0,0)$ and choose a valid placement for the attacker which monitors interconnect contention on Core$(0,2) \leftrightarrow$ Slice$(0,3)$. According to our reverse engineering results, such a placement allows the attacker to observe a moderate amount of the victim's network traffic. A detailed traffic flow analysis of this placement can be found in Appendix C.

Example Traces For both the ECDSA and RSA victims, we generate 5000 random keys. In expectation, half of the traces have their first secret bit=0, and the rest have their first secret bit=1. For each key, we collect a trace during the first iteration of the victim loop. That is, each trace corresponds to the first iteration of a different random key. We then group the traces by bit and plot the average for both groups in Figure 10. We additionally plot the standard deviation of each sample with light-blue shades to show variations across samples.

Figure 10a shows the results when attacking RSA for the case when the secret bit is 0 and when the bit is 1. The plot for bit=1 has an *extra* spike around cycle 40, corresponding to the contention caused by `Func2`. Similarly, Figure 10b shows the results for ECDSA, where we can observe an extra spike around cycle 140 only when the secret bit is 1. These results demonstrate that the interconnect side channel can be used to effectively leak secret key bits from both implementations.

The specific characteristics of the traces of Figure 10 vary depending on the physical memory pages used by the vic-



(a) RSA victim. (b) ECDSA victim.

Figure 11: Percentage of the full key recovered using multiple traces (i.e., votes) per iteration. For RSA and ECDSA, we recovered the full key using 65 or 140 traces, respectively.

tim, which influence the mapping of the victim's code and data to different LLC slices. However, since the slice hash function is designed to uniformly distribute memory across all slices, even when the pages changed, we could always observe distinguishable signals for different secret bits.

We also tested the susceptibility of the attack to noise by repeating it while running the `stress-ng` tool in the background. When running CPU-intensive tasks (`--cpu` stress methods), the secret-dependent spikes in interconnect contention were still distinguishable even when 12 of the other 22 cores were active. However, when running tasks designed to stress the CPU cache (`--cache` stress methods), the secret-dependent spikes in interconnect contention became hard to see when more than 3 cores were active.

Single-Bit Classification Accuracy We use a random forest classifier to simplify the inference of the secret bit used in each latency trace. Our methodology is similar to that of prior work [26, 74]. We use 75% of the 5000 traces as the training set and the remaining 25% as the validation set, which we compute the prediction accuracy for. To account for the different page mappings across runs, we repeat the experiment 10 times and report a range of accuracies.[6]

For the placement of Figure 10, we obtain an accuracy of 69%–71% for ECDSA and 71%–73% for RSA. Next, we show that the attacker can make predictions using multiple traces to further boost the accuracy and simplify the process of full-key recovery on both ECDSA and RSA.

8.4 Full Key Recovery

We now show how our attack can be used to recover full cryptographic keys (a 1024-bit long key for the RSA victim and 256-bit long nonce for the ECDSA victim). Instead of focusing on the first bit, we show that our attack can distinguish traces collected at different iterations (i.e., different bits) and can account for variations across both iterations and different keys. For ECDSA and RSA, we generate 800 and 200 random keys respectively for training. For each training key, we collect one trace for each iteration of the loop when using that key. We end up with over 200,000 traces across ECDSA and RSA with approximately half of the traces for a secret bit=1 and half for a secret bit=0. We then group these traces by bit, regardless of which iteration and which key the traces correspond to, and use them to train a random forest classifier that is able to make a prediction on secret bits for any iteration.

To evaluate the classification accuracy, we generate a *new*, random test key for both RSA and ECDSA that is not in the training set and collect traces for each bit of the test key. We apply the classifier to the traces and count the correctly classified bits. Given a single trace per iteration, the classifier can correctly identify 77% of the bits for RSA and 68% of the bits for ECDSA. To boost the prediction accuracy, we collect additional traces per iteration of the testing key and use majority voting to merge prediction results. Figure 11 shows how the number of correctly predicted bits changes when we use multiple traces per iteration with majority voting. As expected, the number of correctly predicted bits increases rapidly as the number of traces increases. For RSA, 65 traces per bit were enough to recover 100% of the key. For ECDSA, 140 traces per bit were enough to recover 100% of the key.

8.5 Impact of Attacker and Victim Placements

Due to the lane scheduling policy and the priority arbitration policy used by the interconnect, different placements of the attacker and the victim incur different amounts of *observable contention*. To study the impact of different placements, we design an analytical model to rank tiles by the level of vulnerability and empirically validate the resulting vulnerability scores using the single-bit attack from Section 8.3.

The Analytical Model The analytical model computes the amount of observable interconnect contention for a given mesh layout, victim placement, and attacker placement. The victim's placement contains a core location, and the attacker's placement contains a core location and an LLC slice location. The model assumes an even distribution of the victim's memory addresses across all LLC slices, meaning an equal amount of traffic flows between the victim's core and each LLC slice. Next, the model compares each of the victim's traffic flows with the attacker's traffic flow on the corresponding ring. If both flows share the same direction, lane, and segment, and the victim's flow has higher priority, we say that the victim's flow causes observable contention. We assign a score of 1 to contention on the request and acknowledge rings and a score of 2 to contention on the data ring. This follows from our reverse engineering which showed that contention on the data ring generates larger delays than contention on the request and acknowledge rings (due to each message on the data ring requiring two packets [47]). We then sum the scores for all the victim's traffic flows to quantify the observable contention.

The search tool ranks the vulnerability level for each tile as follows. For each tile used as the victim's core, we search

[6] For simplicity, our proof-of-concept attack trains and tests the classifier on a single machine. However, since we observed similar secret-dependent spikes in interconnect contention traces also on a different machine, it should be feasible for an attacker to train a cross-machine classifier too.

32	32	32	32	32	31
x	20	20	20	20	x
29	13	27	27	13	29
32	20	30	x	20	20
x	32	x	20	32	x

Figure 12: Heatmap of the vulnerability level for different tiles. The number inside each tile represents the vulnerability score output by our analytical model.

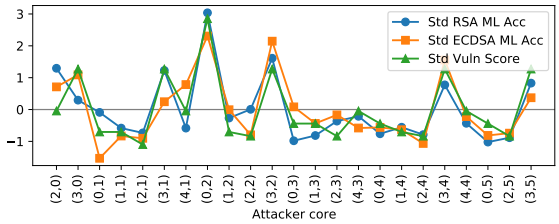


Figure 13: Correlation between standardized analytical model vulnerability scores and standardized observed attack accuracies across all attacker core placements.

for an attacker placement that maximizes the observable contention which is computed using the analytical model. We iterate over all the tiles on the chip and rank these tiles based on their corresponding scores of optimal observable contention. The tile with the smallest score of optimal observable contention is the least vulnerable. Figure 12 shows our results.

Our results highlight a key insight: the victim, when placed onto different cores, experiences different levels of vulnerability. For example, when placing the victim on the cores in the first row of the mesh, the vulnerability scores are consistently high, either 32 or 31, indicating that the attacker is able to find a placement with high levels of observable contention. However, if the victim is on Core $(2,1)$ or Core $(2,4)$, the vulnerability score is 13, meaning that the attacker can only observe much less contention even when placed optimally.

Model Validation We validate our analytical model by computing the single-bit classification accuracy for the attack on different placements. We pin the victim process to Core $(0,0)$, and for every possible attacker core, we use our model to find the optimal attacker slice and run the attack using this placement. To compare the vulnerability scores to the observed classification accuracies, we standardize both values by subtracting the mean and dividing by the standard deviation. Figure 13 shows the strong correlation between the scores and the attack accuracies, demonstrating that our analytical model effectively predicts leakage. For example, placing the attacker on Core $(0,2)$, the predicted best attacker core for a victim at $(0,0)$, results in an accuracy of 87.4% on the RSA victim and 75.8% on ECDSA. However, placing the attacker on Core $(2,1)$, the predicted worst attacker core for a victim at $(0,0)$, results in an accuracy of 65.4% on the RSA victim and 61.5% on the ECDSA victim, much closer to random guessing. We include the raw data in Appendix D.

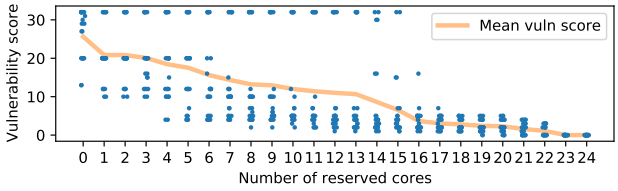


Figure 14: Effect of reserving cores on vulnerability scores.

9 Software-Based Mitigations

The reverse engineering results and the analytical model provide useful insights into designing software-based mitigations. To start, as discussed in Section 8.5, the operating system or hypervisor can schedule the victim to the least vulnerable cores. For example, security-sensitive applications should run on cores with lower scores such as 13 rather than 32. This already makes the attacker suffer from a higher error rate.

In addition to the victim's placement, our analytical model suggests that the attacker's placement also affects the attack accuracy. Thus, the model can guide the design of an intelligent scheduler that reduces leakage through interconnect contention. Given a victim's core location, the scheduler will reserve the cores that contribute to the top k vulnerability scores for applications that belong to the same security domain as the victim. This lightweight and practical mitigation mechanism can significantly reduce the accuracy of the attack.

We evaluate this mitigation mechanism as follows. For each victim placement, we reserve the k cores with the highest vulnerability scores by making them unavailable to the attacker and re-calculate the vulnerability score. Figure 14 and Appendix E show how the scores change across all victim placements with increasing k. The mean score decreases as we reserve more cores. However, this mechanism affects different victim placements differently. For example, the mitigation is highly effective for victim placements such as $(2,0)$ and $(2,5)$, where the vulnerability score is reduced from 29 to 4 after reserving only 4 cores. However, the mitigation is less effective for victim placements such as $(4,1)$ and $(4,4)$, where the score drops below 32 only after reserving 16 cores. These results highlight the importance of scheduling security-sensitive software on cores where the mitigation is most effective.

Discussion The software-based mitigation above reduces the efficacy of interconnect side-channel attacks but does not fully close the channel. We discuss more extensive hardware mechanisms to fully block interconnect leakage in Section 10. Our mitigation aims to be non-invasive and flexible. Since it only requires scheduler modifications, it can be more conveniently adopted on today's systems. In addition, applications

can adjust their degree of isolation. Less sensitive applications can reserve fewer cores to conserve resources.

Since our scheme reduces the flexibility of the scheduler, it may negatively impact system performance. As we reserve cores for a domain, other security domains need to compete for unreserved cores, resulting in potential performance degradation. Even intelligently scheduling a multi-threaded victim (e.g., a VM) that runs on n cores may require reserving more than n cores. In particular, selecting the minimal set of $m \geq n$ cores such that these m cores sufficiently reduce the victim's vulnerability score is a complex (and orthogonal) partitioning problem. However, considering that modern cloud resources are generally over-provisioned [14, 104], we believe that, in practice, the performance impact of our scheme would be low.

10 Related Work

Attacks We provided a classification of microarchitectural attacks in Section 2.3. We now discuss prior works that explored the security of on-chip CPU interconnects specifically.

Wang et al. [92] were the first to consider side-channel attacks on the on-chip interconnect. However, their attack was only demonstrated on a simplified architectural simulator.

Paccagnella et al. [74] described attacks exploiting contention on the ring interconnect used by client-class Intel processors. Our work builds on their techniques but is different in two main ways. First, we study a more complex, 2-dimensional interconnect with different traffic flows and lane scheduling policies, as well as more complex priority arbitration policies. Second, our work handles a much larger number of placement options for the attacker. For example, if we fix the victim's core, there are 598 attacker placements on our processor, as opposed to only 56 placements on an 8-core desktop processor. Hence, our work includes a novel analytical model to find the optimal attacker placement.

Dutta et al. demonstrated a cross-component covert channel that exploits contention on the ring interconnect [20]. There are also works that used information about the on-chip interconnect to improve cache side channels [18, 82]. These works demonstrate additional benefits of reverse engineering the interconnect to attackers. However, these attacks are fundamentally dependent on shared cache structures whereas our work focuses on side channels outside the cache.

Most recently, concurrent work from Wan et al. [90] described a side-channel attack that also targets Intel's mesh interconnect. However, their work does not include lane scheduling and priority arbitration policy details. Further, given the large delays (up to 1000 cycles) they report and the significant memory footprint of their victim, it is unclear whether their attack works due to contention on the interconnect or on other shared structures, e.g., shared cache directories or slice ports, both of which are not partitioned by Intel CAT [70]. In contrast, our work establishes the precise conditions for creating contention on the mesh interconnect, and our experiments are carefully designed to rule out other contention sources.

Mitigations Existing mitigations to our attack can be classified into software and hardware mitigations. Among software mitigations, the recommended strategy is to use constant-time cryptographic implementations [46]. Mitigations at the hardware level that separate the traffic flows of different security domains have also been proposed. Wang and Suh investigated domain-aware priority arbitration policies which give low-security traffic precedence over high-security traffic at the router [92]. Wassel et al. propose a time-multiplexed scheduling policy in which network links may only carry traffic from a predefined security domain at each instant in time [94]. While effective, these approaches require hardware modifications to the interconnect and cannot be adjusted to accommodate more security domains once deployed. In contrast, our proposed mitigation is non-intrusive and does not require hardware changes. Alternatively, a limited form of spatial partitioning may be accomplished using Intel's Sub-NUMA Clustering (SNC), which splits the LLC slices into two disjoint clusters, each bound to a single memory controller [48]. However, SNC only focuses on memory mappings, so while it may reduce interconnect contention in particular cases, it makes no guarantees about isolating interconnect traffic in general. Further, it only supports two domains.

11 Conclusion

In this paper, we reverse engineered the lane scheduling and priority arbitration policies used by Intel's mesh interconnect. We demonstrated covert channel and side-channel attacks that exploit contention on the mesh interconnect. We then used an analytical model to quantify the vulnerability of different cores and proposed a non-invasive software mitigation.

Our results underscore that, though difficult to exploit, on-chip interconnects remain an overlooked microarchitectural attack surface and that additional work is necessary to enforce security-by-design against these attacks in future server processors. We made a first step towards this goal by introducing a non-invasive mitigation to interconnect side channels. Going forward, we expect that our work will facilitate future research into the security of on-chip interconnects. More broadly, we hope that our findings motivate the development of principled, holistic mitigations against microarchitectural attacks, as opposed to the current per-resource, "spot" mitigations.

Acknowledgments

We thank our shepherd Michael Schwarz and the anonymous reviewers for their valuable feedback. This work was funded in part through NSF grants 2046359 and 1954521, and AFOSR grant FA9550-20-1-0402.

References

[1] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *CSAW*, 2007.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[3] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *CT-RSA*, 2008.

[4] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*, 2007.

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[6] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In *CCS*, 2020.

[7] Lucian Armasu. OpenBSD will disable Intel Hyper-Threading to avoid Spectre-like exploits (updated). https://www.tomshardware.com/news/openbsd-disables-intel-hyper-threading-spectre,37332.html. Accessed on Jun 12, 2022.

[8] C Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. Highly efficient algorithms for AES key retrieval in cache access attacks. In *EuroS&P*, 2016.

[9] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh aah... just a little bit": A small amount of side channel can go a long way. In *CHES*, 2014.

[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[11] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In *MICRO*, 2019.

[12] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. Preprint, arXiv:1506.00189 [cs.CR], 2015.

[13] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, 2019.

[14] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE transactions on services Computing*, 5(2):164–177, 2011.

[15] Mainak Chaudhuri. Zero directory eviction victim: Unbounded coherence directory and core cache isolation. In *HPCA*, 2021.

[16] Thomas Claburn. RIP Hyper-Threading? ChromeOS axes key Intel CPU feature over data-leak flaws – Microsoft, Apple suggest snub. https://www.theregister.co.uk/2019/05/14/intel_hyper_threading_mitigations/. Accessed on Jun 12, 2022.

[17] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR DRBG. In *S&P*, 2020.

[18] Guillaume Didier and Clémentine Maurice. Calibration done right: Noiseless Flush+Flush attacks. In *DIMVA*, 2021.

[19] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 cache attack using Intel TSX. In *USENIX Security*, 2017.

[20] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated CPU-GPU systems. In *ISCA*, 2021.

[21] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[22] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *TACO*, 13(1), 2016.

[24] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

[25] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *CCS*, 2018.

[26] FPSG-UIUC. lotr. https://github.com/FPSG-UIUC/lotr, 2021. Accessed on Jun 12, 2022.

[27] Janosch Frank. The common challenges of secure VMs. https://static.sched.com/hosted_files/kvmforum2020/a3/Janosch%20Frank.pdf, 2020. Accessed on Jun 12, 2022.

[28] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *JCEN*, 8(1), 2018.

[29] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *APSys*, 2018.

[30] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In *CCS*, 2017.

[31] GnuPG. Libgcrypt. https://gnupg.org/software/libgcrypt/index.html, 2021. Accessed on Jun 12, 2022.

[32] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *CLOUD*, 2013.

[33] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[34] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[35] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[36] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *CHES*, 2016.

[37] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, 2016.

[38] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[39] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *S&P*, 2016.

[40] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *S&P*, 2011.

[41] Berk Gulmezoglu, Andreas Zankl, M Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using AI. In *CCS*, 2019.

[42] Marcos Horro, Mahmut T Kandemir, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. Effect of distributed directories in mesh interconnects. In *DAC*, 2019.

[43] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[44] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015.

[45] Intel. Disclosure of hardware prefetcher control on some Intel processors. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html. Accessed on Jun 8, 2021.

[46] Intel. Guidelines for mitigating timing side channels against cryptographic implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html. Accessed on May 27, 2022.

[47] Intel. *Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring*, July 2017.

[48] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, May 2020.

[49] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *S&P*, 2015.

[50] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *DSD*, 2015.

[51] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *ASIACCS*, 2016.

[52] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on the last level cache. In *DAC*, 2016.

[53] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

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

[55] Steve Kommrusch, Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. Optimizing coherence traffic in manycore processors using closed-form caching/home agent mappings. *IEEE Access*, 9:28930–28945, 2021.

[56] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *S&P*, 2020.

[57] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *ESORICS*, 2017.

[58] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In *ASIACCS*, 2020.

[59] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[60] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[61] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, 2015.

[62] Yen-Cheng Liu, Jason W Horihan, Krishnakumar Ganapathy, Umit Y Ogras, Allen W Chu, and Ganapati N Srinivasa. On-chip mesh interconnect, Patent US20150006776A1, 2013.

[63] Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. *Security Best Practices For Developing Windows Azure Applications*. Microsoft, June 2010.

[64] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, 2015.

[65] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *DIMVA*, 2015.

[66] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*, 2017.

[67] John D. McCalpin. Address hashing in Intel processors. In *IXPUG*, 2018.

[68] John D. McCalpin. Mapping core and L3 slice numbering to die location in Intel Xeon Scalable processors. Technical report, Texas Advanced Computing Center (TACC), 2020.

[69] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *SAC*, 2006.

[70] Khang T Nguyen. Introduction to cache allocation technology in the Intel Xeon processor E5 v4 family. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html. Accessed on Jun 12, 2022.

[71] Hamed Okhravi, Stanley Bak, and Samuel T King. Design, implementation and evaluation of covert channel attacks. In *HST*, 2010.

[72] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, 2015.

[73] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, 2006.

[74] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security*, 2021.

[75] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[76] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure DSA signing exponentiations really are constant-time. In *CCS*, 2016.

[77] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 2016.

[78] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[79] Bholanath Roy, Ravi Prakash Giri, C Ashokkumar, and Bernard Menezes. Design and implementation of an espionage network for cache-based side channel attacks on AES. In *ICETE*, 2015.

[80] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS*, 2018.

[81] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[82] Johanna Sepúlveda, Mathieu Gross, Andreas Zankl, and Georg Sigl. Beyond cache attacks: Exploiting the bus-based communication structure for powerful on-chip microarchitectural attacks. *TECS*, 20(2), 2021.

[83] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, 2021.

[84] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, 2019.

[85] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *IC2E*, 2018.

[86] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[87] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[88] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-VM side-channels. In *USENIX Security*, 2014.

[89] VMware Knowledge Base. Security considerations and disallowing inter-virtual machine transparent page sharing (2080735). https://kb.vmware.com/s/article/2080735. Accessed on Jun 12, 2022.

[90] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. MeshUp: Stateless cache side-channel attack on CPU mesh. In *S&P*, 2022.

[91] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. PAPP: Prefetcher-aware prime and probe side-channel attack. In *DAC*, 2019.

[92] Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *NOCS*, 2012.

[93] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *ACSAC*, 2006.

[94] Hassan MG Wassel, Ying Gao, Jason K Oberg, Ted Huffmire, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. *ACM SIGARCH Computer Architecture News*, 41(3), 2013.

[95] WikiChip. Cascade Lake - microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake. Accessed on Jun 12, 2022.

[96] WikiChip. Mesh interconnect architecture - Intel. https://en.wikichip.org/wiki/intel/mesh_interconnect_architecture. Accessed on Jun 12, 2022.

[97] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security*, 2012.

[98] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, 2020.

[99] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*, 2019.

[100] Mengjia Yan, Jen-Yang Wen, Christopher W Fletcher, and Josep Torrellas. SecDir: A secure directory to defeat directory side-channel attacks. In *ISCA*, 2019.

[101] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[102] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015.

[103] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. *JCEN*, 7(2), 2017.

[104] Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys (CSUR)*, 47(4):1–33, 2015.

[105] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

[106] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, 2014.

[107] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

[108] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *CCS*, 2016.

Appendix

A Determining Address Mapping

When designing the transmitter and receiver, we need addresses that map to a given L2 set and LLC slice. To determine the L2 set of a virtual address, we use 2MB hugepages, as in [49,61]. To determine the LLC slice ID of a virtual address, there exist two approaches. If the attacker has root privileges, such as when reverse engineering a local machine, the attacker can use Intel PMON counters, as in prior work [68]. Without root privileges, the attacker can leverage timing information to map addresses to LLC slices in a coarse-grained fashion. For example, since accesses to the local slice are faster than accesses to remote slices, the attacker can pin a program to a core and find addresses that map to the local slice (as in prior work [102]). On our 24-core and 26-slice processor, the attacker will find 26 groups of addresses of which 24 map to slices on active cores and 2 map to partially-disabled tiles. The attacker then figures out the slice ID of each group based on the tile-mapping information (as explained in Appendix B).

B Inferring Tile Layout and Mapping

We describe how we reverse-engineered the tile layout and tile mapping of our Intel Xeon Gold 5220R processor shown in Figure 2. Specifically, we want to obtain the following: 1) the positions of disabled tiles; 2) the mapping from an LLC slice ID to a tile; 3) the mapping from a core ID to a tile.

We start by obtaining basic processor information using the `lscpu` command, which indicates that the chip has 24 cores. Next, we find the number of LLC slices. As documented by Intel [47], bits 27:0 in the `CAPID6` register indicate the number

of LLC slices, which we find to be 26 on our setup. Our processor uses an XCC configuration with 30 tiles, organized into a 5×6 grid. According to public information [95], an XCC chip has 2 IMC tiles and 28 Core tiles. Therefore, we infer that there are 4 disabled cores and 2 disabled LLC slices.

We first describe an approach to locate disabled tiles that require privileged access to a machine and then describe alternative approaches that do not require privileged access.

Locating Disabled Tiles Using the `CAPID6` Register Prior work found that each bit of the `CAPID6` register corresponds to a Core tile on the die [68]. The Core tiles are numbered in column-major order from the top-left corner. Hence, by querying the `CAPID6` register, we learn which tiles are disabled. For example, on our processor, the `CAPID6` register contains `0x0ffddfff`, where bits 13 and 17 are unset, indicating that tiles 13 and 17 are disabled. According to the numbering mechanism above, tile 13 is located at $(4,2)$ and tile 17 is located in $(3,3)$ in Figure 2.

Reverse Engineering Tile Mappings The LLC slice IDs are kept internally by the hardware and are referred to as CHA IDs by Intel. They are used by the PMON counters and are very useful for our reverse engineering process. McCalpin [68] has verified that CHA IDs map to active Core tiles in column-major order.

The last step is to figure out which tile each CPU ID maps to. We use the approach in [68] as follows. Recall that CPU IDs are used by the OS to schedule processes onto different cores. We pin a program on a given CPU and repeatedly load from DRAM (assisted by the `clflush` instruction) to generate traffic to the two IMC tiles. We then use PMON counters to monitor the data ring utilization on each tile. According to the traffic flow analysis in Section 6.1, the only traffic on the data ring is from the IMC to the tile that the logical CPU maps to. With two IMC tiles and a Y-X routing policy, only one tile observes 2 incoming traffic flows, and that is the tile the logical CPU maps to. We repeat this approach for every logical CPU to completely recover the mapping.

Locating Disabled Tiles With Unprivileged Access Prior work has observed that the position of disabled tiles may vary across different units of the same processor model [68]. An unprivileged attacker may need to locate the disabled tiles on the victim's machine. We discuss two methods that do not rely on privileged access, i.e., access to the `CAPID6` register.

One method uses timing information by having the attacker first profile the cache access latency for all core-slice pairs on a local machine with privileged access. The attacker then obtains the same profile on the victim's machine. By analyzing how the two profiles differ, the attacker can infer the locations of the disabled tiles. An alternative method uses interconnect contention. The attacker again profiles the interconnect contention pattern (cf. Section 6) on both a local machine and the victim's machine. By analyzing how the contention pattern on the victim's machine deviates from the reference pattern, the attacker can infer the locations of the disabled tiles.

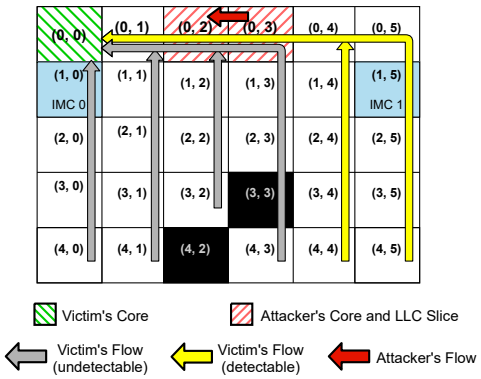


Figure 15: Traffic flows on the data and acknowledge rings when the transmitter executes on Core $(0,0)$ and the attacker monitors traffic between Core$(0,2) \leftrightarrow$ Slice$(0,3)$.

C Side-Channel Attack Placement

Figure 15 explains why the attacker's placement in Section 8.3 allows it to observe a moderate amount of network traffic by showing the traffic flows on the data and acknowledge rings. The victim has traffic flows from all 25 remote slices. The attacker's traffic uses the same lane as the victim's horizontal traffic (both use the white lane per Figure 4). Crucially, the attacker's traffic also has lower priority than the traffic from the tiles in the two rightmost columns. Therefore, this attacker can monitor the victim's traffic flows from 9 out of 24 slices.

Note that while this attacker placement is effective, it is in fact a suboptimal placement for a victim at $(0,0)$. Placing the receiver at Core$(0,2) \leftrightarrow$ Slice$(0,3)$ results in a vulnerability score of 29 whereas Core$(0,2) \leftrightarrow$ Slice$(2,3)$, the optimal placement, has a vulnerability score of 32.

D Model Verification Data

Table 2 shows the results used to verify our analytical model from Section 8.5. In general, higher vulnerability scores correspond to better bit-classification accuracies. For example, positioning the attacker on Core$(0,2) \leftrightarrow$ Slice$(2,3)$ (vulnerability score of 32) gives an 87.4% accuracy when leaking an RSA key bit while positioning the attacker on Core$(2,1) \leftrightarrow$ Slice$(4,5)$ (vulnerability score of 2) yields a 65.4% accuracy.

E Mitigation Evaluation Data

Table 3 shows the results used in the attack mitigation analysis from Section 9. For every possible victim core location, we compute the vulnerability score after reserving the cores that contribute to the top k vulnerability scores.

Attacker	Attacker	Vuln	RSA			ECDSA		
Core ID	Slice ID	Score	Acc	Prec	Rec	Acc	Prec	Rec
(2,0)	(4,5)	10	77.2	76.0	60.2	68.7	71.0	62.9
(3,0)	(2,0)	20	71.4	71.7	49.9	70.4	68.0	81.4
(0,1)	(2,5)	5	69.1	65.2	51.6	58.7	57.5	60.4
(1,1)	(2,5)	5	66.2	59.4	50.7	61.8	60.5	64.9
(2,1)	(4,5)	2	65.4	56.8	52.3	61.5	60.8	62.8
(3,1)	(2,0)	20	76.8	76.9	59.6	66.6	68.0	60.6
(4,1)	(3,0)	10	66.2	62.0	47.1	69.0	71.7	65.8
(0,2)	(2,3)	32	87.4	83.6	83.6	75.8	77.4	70.8
(1,2)	(2,5)	5	68.1	59.0	48.8	65.5	65.0	69.2
(2,2)	(4,1)	4	69.7	65.4	51.7	62.0	60.0	61.5
(3,2)	(2,0)	20	79.0	75.4	65.7	75.1	81.6	66.8
(0,3)	(2,1)	7	63.9	56.4	45.0	65.9	64.4	70.0
(1,3)	(2,1)	7	64.9	61.0	42.3	63.6	64.8	63.9
(2,3)	(4,1)	4	67.5	60.6	47.1	64.8	63.0	66.6
(4,3)	(3,0)	10	68.4	77.9	32.3	63.0	61.0	68.2
(0,4)	(1,2)	7	65.2	58.1	44.2	63.0	64.5	60.4
(1,4)	(2,2)	5	66.4	59.0	50.7	62.7	61.1	70.2
(2,4)	(3,2)	4	65.0	57.8	48.2	60.8	61.2	59.4
(3,4)	(2,0)	20	74.2	74.0	57.1	72.5	74.2	67.5
(4,4)	(3,0)	10	67.1	67.2	50.4	64.6	62.9	64.5
(0,5)	(2,1)	7	63.7	53.0	44.6	61.9	62.0	62.4
(2,5)	(4,1)	4	64.5	55.2	47.5	62.2	60.3	63.8
(3,5)	(2,0)	20	74.5	71.6	58.4	67.2	73.2	58.4

Table 2: Classification percent accuracy (Acc), precision (Prec), and recall (Rec) for various optimal attacker placements. The victim was pinned on $(0,0)$.

Victim Core	Number of reserved cores (k)																							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(0,0)	32	20	20	20	20	20	10	10	10	10	7	7	7	7	5	5	5	5	4	4	4	4	2	0	0
(2,0)	29	10	10	10	4	4	4	4	4	4	4	4	4	2	2	2	2	2	2	2	2	0	0	0	0
(3,0)	32	20	20	20	20	20	20	10	7	7	5	5	5	5	4	4	4	4	4	2	2	2	2	0	0
(0,1)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	7	5	5	5	5	5	3	3	0	0
(1,1)	20	20	20	16	10	5	5	5	5	5	5	4	4	3	3	2	2	2	2	2	2	0	0	0	0
(2,1)	13	12	12	12	12	12	12	10	10	10	4	2	2	2	2	2	2	2	2	0	0	0	0	0	0
(3,1)	20	20	20	20	20	20	10	7	6	5	5	5	5	4	3	3	2	2	2	2	2	2	0	0	0
(4,1)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	7	5	5	5	3	3	0	0
(0,2)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	30	15	5	4	4	4	4	4	3	0	0
(1,2)	20	20	20	15	10	5	5	5	4	4	4	4	3	3	2	2	2	2	1	1	1	0	0	0	0
(2,2)	27	12	12	12	12	12	12	12	10	10	10	2	2	2	1	1	1	1	1	1	0	0	0	0	0
(3,2)	30	20	20	20	20	20	20	15	10	6	5	5	4	4	3	2	2	1	1	1	1	1	0	0	0
(0,3)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	30	15	5	4	4	4	4	4	3	0	0
(1,3)	20	20	20	15	10	4	4	4	4	4	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0
(2,3)	27	12	12	12	12	12	12	12	2	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0
(4,3)	20	20	20	20	15	12	12	12	12	12	12	10	5	5	4	4	3	3	2	2	1	1	0	0	0
(0,4)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	16	7	5	5	5	4	4	3	3	0	0
(1,4)	20	20	20	16	10	7	5	5	5	5	5	4	4	4	3	3	2	2	2	2	2	0	0	0	0
(2,4)	13	12	12	12	12	12	12	10	10	10	4	4	2	2	2	2	2	2	2	0	0	0	0	0	0
(3,4)	20	20	20	20	20	20	16	10	7	6	5	5	5	4	3	3	2	2	2	2	2	2	0	0	0
(4,4)	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	7	5	5	5	5	3	3	0	0
(0,5)	31	20	20	20	20	20	10	10	10	10	7	7	7	7	5	5	5	5	4	4	4	4	2	0	0
(2,5)	29	10	10	10	4	4	4	4	4	4	4	4	4	2	2	2	2	2	2	2	2	0	0	0	0
(3,5)	20	20	20	20	20	20	10	7	5	5	5	5	5	5	4	4	4	4	4	2	2	2	2	0	0

Table 3: Vulnerability scores for each victim core for all numbers of reserved cores.