Calibration Done Right: Noiseless Flush+Flush Attacks

Guillaume Didier[1,3,4] and Clémentine Maurice[2]

[1] Direction Générale de l'Armement
[2] Univ Lille, CNRS, Inria
[3] DIENS, École normale supérieure, CNRS, PSL University, Paris, France
[4] Univ Rennes, CNRS, IRISA
guillaume.didier@inria.fr, clementine.maurice@inria.fr

Abstract. Caches leak information through timing measurements and side-channel attacks. Several attack primitives exist with different requirements and trade-offs. Flush+Flush is a stealthy and fast one that uses the timing of the `clflush` instruction depending on whether a line is cached. We show that the CPU interconnect plays a bigger role than previously thought in these timings and in Flush+Flush error rate.
In this paper, we show that a naive implementation that does not account for the topology of the interconnect yields very high error rates, especially on modern CPUs as the number of cores increases. We therefore reverse-engineer this topology and revisit the calibration phase of Flush+Flush for different attacker models to determine the correct threshold for `clflush` hits and misses. We show that our method yields close-to-noiseless side-channel attacks by attacking the AES T-tables implementation of OpenSSL, and by building a covert channel. We obtain a maximal capacity of 5.8 Mbit/s with our method, compared to 1.9 Mbit/s with a naive Flush+Flush implementation on an Intel Core i9-9900 CPU.

1 Introduction

The cache hierarchy is a key component of modern CPUs, and relies on the principle of making the common case fast [13,3]. Caches have been extensively studied with respect to side-channel attacks, resulting in several primitives such as Prime+Probe, Evict+Time [24], Flush+Reload [35], and Flush+Flush [10]. These can be used to build covert channels and side-channel attacks, e.g., on cryptographic libraries. Flush+Reload is a popular choice due to ease of implementation, reliability, and reasonable requirements on x86 platforms: for example, a variety of transient execution attacks [15,17,9] used it as a covert channel.

These primitives aim to observe memory accesses from other processes, through cache timings. Flush+Reload resets the state using the x86_64 `clflush` instruction, which ensures that the latest value of a cache line is flushed back to memory, with no copy remaining in the cache hierarchy. It then makes a costly reload to check if the line is cached. Flush+Flush is a variant that uses the execution time of the `clflush` instruction itself to do the check. Flush+Flush is thus faster and stealthier, as it causes no memory accesses by the attacking process.

Calibration is a critical step of the attack where an attacker chooses a threshold between `clflush` hits (timing of the `clflush` instruction when the line is present in the cache) and `clflush` misses (timing when the line is absent). A sub-optimal threshold leads to errors in a covert channel or a side-channel attack. The main source of Flush+Flush noise comes from the fact that the median execution time of `clflush` hits is close to the median value for misses, whereas the distributions of load execution time for hits and misses are more separated.

Our experiments show that the timing of the `clflush` instruction actually suffers from multiple sources of variability, which impairs the threshold and the subsequent attacks. A careful analysis of these execution timings unmasks a major culprit: the CPU interconnect. We uncover the various contributions of the CPU interconnect between the attacker, the cache slice, the victim core, and the system agent accessing the main memory, and propose a method to find the topology of recent Intel CPUs. Accounting for this topology, we significantly reduce this noise, making Flush+Flush a low-noise attack primitive that remains both fast and stealthy, and, thus, a realistic alternative to Flush+Reload. Our evaluation shows that a higher number of cores and larger caches distributed in more slices increases Flush+Flush noise on modern single-socket machines.

We show that our calibration improvements to Flush+Flush improve covert channel capacity. A naive Flush+Flush implementation has a 20% error rate while our improved Flush+Flush has a negligible error rate and a bandwidth 3× higher. The latter's bandwidth is also 3 to 4 % higher than Flush+Reload.

In summary, we thus make the following key contributions[5]:

1. We present a method to uncover the interconnect topology of Intel CPUs, and apply it on Coffee Lake CPUs. We explain the variation of the execution time of `clflush` caused by topology on single-socket systems (Section 5).
2. We measure the resulting error rate depending on the location of the attacker, victim, and cache slice accessed on single-socket machines, and analyze the differences with dual-socket machines (Section 6).
3. We benchmark the improved covert channel ideal capacity that results, compared with Flush+Reload and a naive implementation of Flush+Flush. We show how these improvements make Flush+Flush a reliable side-channel primitive, on par with Flush+Reload (Section 7).

2 Background

In this section, we describe some necessary background on CPU caches and the CPU uncore and interconnect, multi-socket systems, and cache side-channel attacks. We focus on Intel CPUs in the remainder of this paper.

2.1 CPU caches

DRAM-based main memory is slow compared to the CPU frequency. *Caches* are smaller but faster SRAM-based memories placed in front of the main memory

[5] Code: `https://github.com/GuillaumeDIDIER/calibration-done-right`

to speed up accesses, applying the "make the common case fast" principle [13]. Caches exploit access locality to keep blocks of memory that are likely to be accessed soon. An access to a block currently in the cache is a cache hit, a fast access. Otherwise, the request gets served at the next level, until main memory.

Modern Intel CPUs typically have a three-level hierarchy. At the first level, on each core, the instruction and data memory access paths each hit their own small caches in 4-5 cycles (L1-D and L1-I). At the second level, each core has an L2 cache, that serves the L1 misses in 15-20 cycles. At the last level, the chip has a shared L3, which acts as the last-level cache and answers in 50-100 cycles, while memory takes over 200 cycles.

Cache associativity and eviction policy. Caches store fixed-size chunks of contiguous memory called cache *lines*, typically 64 bytes in size. The 6 least significant bits of the address determine the offset within the line, while the remainder gives the location of the line. Caches are generally organized, from an abstract interface point of view, as an array of cache sets. Addresses are assigned to a deterministic set by a hash function that usually corresponds to a few bits of the address, next to the offset bits. Each set is composed of a fixed number of *ways*, each containing a cache line, along with metadata identifying the line cached in the way. This metadata usually comprises the coherence state (see below) and a tag, corresponding to the address bits that are not used for index and offset. The tag, index and offset can be used together to check whether the requested address is cached in this way. The number of ways is called *associativity*. A direct-mapped cache has an associativity of 1, while a fully-associative cache has a single set with as many ways as the number of lines in the cache. Most large caches associativity is of the order of 10, and a large number of sets [13].

In each set, the *eviction policy* determines what to do when a new line needs to be inserted in a set full of valid lines. Modern CPUs usually use a variant of the *least recently used* policy, that evicts the line whose last use is the furthest in the past, but the exact policy is undocumented [29].

Cache coherence. Due to the cache hierarchy, the same memory location may be simultaneously stored in several different places. It is thus important to ensure all these locations store the same value. This is achieved using a cache coherence protocol, which enforces a Single Writer or Multiple Reader invariant. Intel uses a variation of the MESI cache coherence protocol [14], in which a line can be:

- *Invalid* (I): The cache does not store a valid value, accesses are misses and require making a request to the next level.
- *Shared* (S): The cache holds a *clean* copy of the correct value, matching the one in memory, but other caches may also own one. The line can be read with no further request, but a write requires communicating with the other caches.
- *Exclusive* (E): The cache holds the correct value, as in the shared case, but it is additionally the only cache to do so. The line can be modified (and can transition to the Modified state) without any further request to the hierarchy.
- *Modified* (M): The cache holds exclusively a modified value. The stale value in memory must be updated before this *dirty* line can be evicted from the cache.

This protocol guarantees that, for a given cache line, all cores will see the same sequence of values, but it offers no guarantee about the order in which each core sees changes to different locations. This ordering is governed by the memory consistency model. In non-server Intel CPUs, an inclusive L3 cache maintains cache coherence. It includes a copy of all lines cached in lower-level caches, and keeps track of the coherency state of each line.

Cache slicing. The bigger SRAM is, the slower it is to access. Moreover, more cores mean higher request traffic to the cache hierarchy. To make the last-level cache scale properly with multiple cores, it is split into several slices, each associated with a core. Chips with more cores have proportionally more slices, which can proportionally serve a greater number of requests.

Physical addresses are mapped to a single slice, using a hash function. The first sliced caches simply used specific bits of addresses, similar to set indexes. However, as uncovered by Maurice et al. [20], modern Intel CPUs use a complex function, which uses the XOR of several bits of the physical address to generate each bit of the output. This was introduced in the Sandy Bridge microarchitecture, and is still present in client Skylake derivatives such as Coffee Lake. Examples of such functions are given in Appendix A. On CPUs whose number of cores is a power of two, the resulting function is linear, otherwise, a non-linear component is required [36].

`clflush` instruction. The x86 ISA includes an instruction to flush a cache line, `clflush`. Executing this instruction causes the cache hierarchy to make sure the memory contains the latest value and evicts the cache line from all levels of cache. Such an instruction is privileged in many ISAs as its main use is in device drivers using DMA. However, x86 also *allows it in user mode*, where it can be used to manually evict lines from the cache in the unlikely case manual cache management improves performance. This instruction can thus be used in cache side-channel attacks. A significant property of `clflush` is that calling `clflush` on one core evicts it from the all of the coherency domain, usually all the cores.

2.2 CPU uncore and interconnect

Modern CPUs tend to have several distinct clock domains. Each core can vary its frequency independently, but there is also a significant part of the system that is not part of a core. A common clock domain is needed for the interconnection network in between the cores, the GPU, the memory, and I/O systems. This part of the core, *i.e.*, everything that is not a specific core, is called the *uncore*.

Prominent in the uncore is the core interconnect, which is not well documented by Intel apart from stating it is a bidirectional ring ([14], Section 2.4.5.3). This leaves room for several interpretations and topologies. The last-level cache is distributed among the nodes of the interconnect network, with each slice being associated with a core. While it was usually assumed that each core had exactly one slice, it no longer the case on some recent Intel systems [30].

Figure 1a is a die shot, annotated by WikiChip [1] of the 8 core Coffee Lake CPUs, this layout is used to produce among others the Intel Core i9-9900 CPU.

2.3 Multi-socket systems

A multi-socket system is a system where several multi-core CPUs, each with its cache system, share a single physical memory space with an interconnect between the two packages. In multi-socket systems, there is no single last-level cache ensuring the coherence between the caches of the two cores. It appears that some of the ECC bits inside the DRAM are used to maintain some coherency metadata, and requests may need to flow in between the two sockets [14,22].

2.4 Cache side-channel attacks

The cache hierarchy contains a global state that is shared among processes. The cache impacts timing but not the correctness of code since its memory permissions are enforced and the value it stores is preserved whether a line is cached or not. However, the time it takes to access a line leaks information to the party that performs the measurement. There are two common scenarios. First, in a *covert channel*, two processes can cooperate to communicate when they are monitored on other channels or simply when not allowed to. Second, in a *side-channel attack*, an attacker process measures which cache line an unwitting victim accesses, leaking access pattern information.

There exist two main techniques of cache attacks: Prime+Probe [24,18] and Flush+Reload [12,35]. In Prime+Probe, the attacker fills a set with her cache lines, and the victim accesses a line within that set. This causes an eviction of one of these lines, which the attacker then measures. Prime+Probe has the least requirements, as the attacker does not need to share memory with her victim, and does not require any specific instruction to evict cache lines.

However, in many settings, the attacker and victim can share read-only memory, in which case the attacker can probe a specific shared cache line. The Flush+Reload attack uses the `clflush` instruction to flush a line that may or may not be accessed by the victim, and then times how long it takes to reload it. The attacker therefore detects whether the victim accessed this line.

Flush+Flush [10] is a variant of Flush+Reload, in which the attacker times a cache line flush instead of a reload. Indeed, `clflush` takes a different time depending on whether the line is cached or not. This is however a very small variation of time, around 10 cycles, and the measurements have a large standard deviation, leading to a significant amount of noise. This attack is noisier than Flush+Reload, but faster and stealthier, as it avoids costly misses, and cannot be detected using performance counter to monitor suspicious cache misses.

One of the sources of timing variation is the scaling of the CPU frequency, for which Saxena and Panda [28] have proposed a solution.

3 Motivation

Using `clflush` as the measurement, such as in Flush+Flush, has the potential of monitoring several addresses with less interference than Flush+Reload, as

flushing does not trigger the prefetcher. It is also fast, as a reload operation is slower than a flush on a miss, which Flush+Reload attacks cause frequently.

However, one point that has been overlooked for this attack is the choice of the threshold to distinguish between a flush hit and a flush miss. This threshold is crucial to avoid noise. When looking at the timings for a single address and on a single run, it appears that there is a good separation between the hits (slower) and the misses (faster), for a single-socket system. However, from one run to another, the exact threshold may change, even with a fixed frequency. The threshold also differs for different addresses.

We hypothesize that the variability is due to the complex topology of sliced caches, and that accounting for these sources of variability improves significantly the quality of the channel, especially as the number of cores grows. Our experiments show that ignoring CPU topology can result in very poor error rates, e.g., in some cases, a 45% error rate for a covert channel using a naive method for choosing the threshold. In the remainder of the paper, we show that taking into account the topology and slices to compute tailored thresholds allows us to build a side channel with an error rate well under 0.01%. Flush+Flush is therefore, contrary to what was thought before, not a noisy attack when crafted carefully.

4 Experimental setup

We run experiments on two single socket systems:

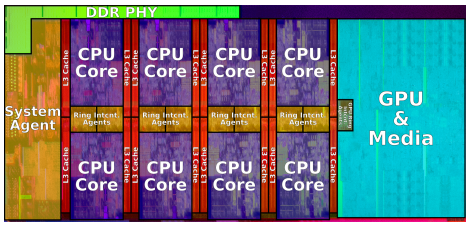
- *4-core machine*: a Dell Latitude 7400 machine with an Intel Core i5-8365U CPU (Whiskey Lake, 4 cores, 8 threads). We have validated that it uses the cache slicing functions that were previously reverse-engineered from Sandy Bridge to Skylake [20] (see Appendix A). It runs Fedora 30.
- *8-core machine*: a Dell Precision 3630 machine with an Intel Core i9-9900 CPU (Coffee Lake, 8 cores, 16 threads). We have reverse-engineered its last-level cache hash functions (see Appendix A). It runs Ubuntu 18.04.5 LTS.

We enable hyper-threading, but disable turbo boost on those machines. The intel_pstate driver is set to performance mode on all cores, to stabilize the core frequencies. Additionally, we write a non-null value in each page before use, this prevents any optimization and involuntary page sharing involving the zero-page.

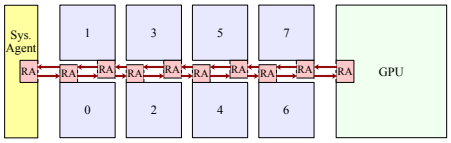
5 Topology Modeling

In this section, we investigate the factors that influence the execution time of `clflush` to improve the Flush+Flush attack, and propose a mathematical model with an associated ring topology. The only information we have from the Intel documentation is that the interconnect is a “bidirectional ring”.

A `clflush` miss occurs when a cache line is not validly cached, which corresponds to a line in the I state. A line that has just been flushed is in the I state — the cache may have an entry in the I state or no entry at all, but it is equivalent



a: Intel Coffee Lake 8-core die shot.
Image by Intel, annotated by WikiChip [1].



b: Proposed i9-9900/8-core topology.
Each core (0 to 7, blue) has a ring agent (RA, red), handling accesses to the local L3 slice and communication over the ring. The GPU and the system agent, handling outside world communication, such as memory, also have their own RA.

Fig. 1: Core i9-9900 die shot and topology.

at the cache coherency protocol level. A `clflush` hit occurs when the line is in any valid state. However, in practice in a Flush+Flush attack, the cache line of interest transitions from an I to an E state when the single victim core loads the line that has just been flushed. Therefore, the two relevant timings are `clflush` of a line in the E state for a hit, and in the I state for a miss. We study these timings depending on three parameters:

1. A: the attacker core that executes `clflush` on an address,
2. V: the victim core that accesses the address and caches it in its L1 or L2,
3. S: the core that contains the last-level cache slice that this address maps to.

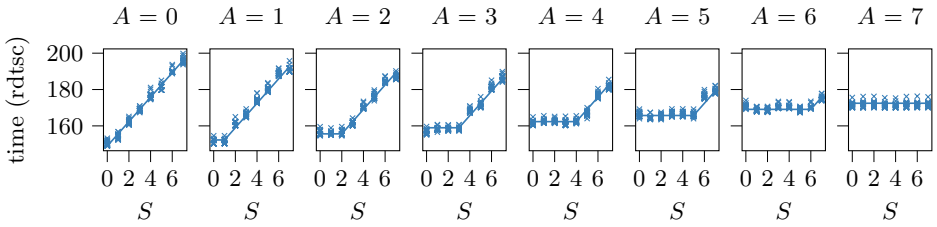
V doesn't contribute to miss timing as invalid lines are not cached in any L1/L2.

Measurements and topology. For each attacker core, Figure 2a shows the time it takes to execute a `clflush` instruction on a cache line in the I state, depending on the slice. The first finding is that all 8 cores have a distinct timing pattern, which implies that the ring has no symmetry.[6] For each attacker, we notice that slices with a lower core number than the attacker all have the same timing, while for slices with a higher number the time increases with the distance between the attacker and the slice. Such a pattern only makes sense if the nodes are aligned in a linear fashion, and if the attacker sends a message to the slice, which then sends a message to the system agent, and then back to the slice and finally to the attacker. Consequently the miss time is more variable for attackers closer to the system agent than for one further away.

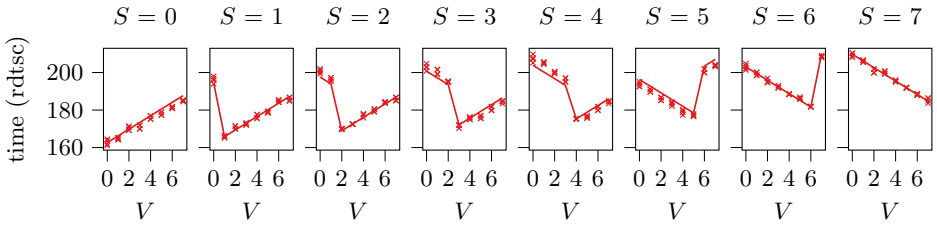
Figure 2b shows the time it takes to execute a `clflush` instruction on a cache line in the E state. Here, we notice an asymmetry in the core, which can be explained if the recall request is always sent by the slice in the same direction without knowing in which core the line is cached. We omit the graphs for other A as they only show a simple linear offset depending on $|A - S|$.

Given that the topology is described as a ring, given the die shot in Figure 1a and our results, we thus propose the topology in Figure 1b, with 8 cores aligned in a linear graph with forward and backward links. For a 4-core machine, similar measurements lead to a similar topology with only cores 0-3.

[6] Unlike the figure in Intel documentation [14] and the figure by WikiChip [1].



a: For a cache line in the I state, depending on its slice S for each attacker core A. There are 32 points per slice S, as we made one measurement for each attacker hyper-thread (2) and for each victim logic core (16). Victim logic core has no impact on a miss.



b: For a cache line in the E state, depending on the victim core V for each slice S, using a fixed attacker core ($A = 0$). There are 4 points per victim core V, one for each attacker and victim hyper-thread.

Fig. 2: Median timings of `clflush` on the 8-core machine depending on the victim core V, the slice S, and the attacker A, along with the fitted model according to our proposed topology, which corresponds to our measurements.

Mathematical model. The above timing measurements can be interpreted within the proposed topology as follows, leading to a mathematical model that can be fitted and compared with the measurements. Misses result in a request to be sent on the ring from the core requesting the flush to the slice, which then sends a message to the memory, and then answers the same path in reverse, using each time the shortest path. The eviction time in state I, $t_I(A, S)$ is thus:

$$t_I(A, S) = C + h \times |A - S| + h \times |S - M|,$$

in which:

– C is a constant base timing,
– h is a constant corresponding to the time a round-trip hop on the ring takes,
– M corresponds to the system agent location, which is -1.

Upon receipt of a request to flush a line in the E state, the slice sends a single message along the ring, in one privileged direction. For core numbered from 0

to $\frac{n_{core}}{2}$ included, this is towards the higher numbered cores (and the GPU), otherwise, it is towards the lower numbered cores. This message is passed around the ring until the victim core V that has the line cached in its lower level cache (L1/L2) receives it. If the core is not in the initial direction, the message will follow the ring back in the other direction until it reaches the victim core. The victim core then discards the line, which is clean, and sends a reply to the slice, along the shortest path. The eviction time in state E, $t_E(A, S, V)$ is thus:

$$t_E(A,V,S) = \begin{cases} C' + h \times |A-S| + h \times |R-(V-M)| & \text{if } S \leqslant \frac{N}{2} \text{ and } V < S \\ C' + h \times |A-S| + h \times |S-V| & \text{if } S \leqslant \frac{N}{2} \text{ and } V \geqslant S \\ C' + h \times |A-S| + h \times |S-V| & \text{if } S > \frac{N}{2} \text{ and } V \leqslant S \\ C' + h \times |A-S| + h \times |M-V| & \text{if } S > \frac{N}{2} \text{ and } V > S, \end{cases}$$

where:

- C' is a different base time constant,
- h is a constant, roughly how long a round-trip hop on the ring takes,
- N is the number of cores,
- R is the ring diameter in hops, corresponding to how many hops there are between the system agent and the GPU, and thus, in our case, $R = N + 1$.

In addition to our measurements, Figure 2a and Figure 2b present the fitted model for the 8-core machine, which appears to explain the behavior consistently.

Summary. We have uncovered that while CPUs appeared to be arranged symmetrically in Intel's bidirectional ring, they are in fact aligned one after the other in a linear graph, with the system agent at an end and the GPU at the other end. First, the `clflush` instruction timing is always influenced by the distance between the core requesting the flush and the slice where the address lives in the last-level cache. Second, in the I state the timing will depend on the distance between the slice and the system agent, whereas in the E state, it will depend on how long a message sent along the ring will need to reach the core that currently has the line, and then go back to the slice. These finding are consistent with those by Paccagnella et al. [25].

6 Improving error rate accounting for topology

6.1 Attacker models

We define different attacker models depending on attacker capabilities. We measure the error rate that can be achieved for each triple consisting of an attacker core, a victim core, and a slice. We also compute the average over all triples.

The attacker core can be set using the `sched_set_affinity` Linux system call. We therefore assume that the attacker always chooses the core with the lowest error rate. In some cases, the attacker may also control the victim core,

e.g., if she launches the process. The victim core can always be found using the `/proc/pid` file system that gives the core affinity and the last core used.

The slice can be found using the physical address but this information is usually unavailable to an unprivileged attacker. However, when the hash function is linear, it is possible to define an equivalence class of addresses within a page that belong to the same slice. It is not possible to know which equivalence class corresponds to which physical slice *a priori*, but the pair of page and result of the hash function defines an equivalence class of virtual address with the same timing impact. We name this equivalence class $\tilde{S}$. Using timing measurements, each equivalence class can be, *a posteriori*, attributed to a precise physical slice, on a per page basis, but we do not use this attribution for our attacks.

If the attacker launches a covert channel, she can pick the addresses used to communicate, and therefore the optimal equivalence class. In a side-channel attack, the attacker cannot pick the addresses to monitor, but usually knows the equivalence class, as she knows both addresses and hash functions. We still present models where the attacker has no knowledge of the slices to compare the previous naive models with the ones that yield the best attacks.

- *Global Threshold (GT)*: The simplest model, using a single threshold that minimizes the average error rate over all triples of attacker, victim, and slice. This is a topology oblivious attacker, as in the initial Flush+Flush attack [10].
- *Best A, Known V*: The attacker knows on which core the victim is running and chooses the attacker core it runs on. The attacker computes a single threshold for all addresses, therefore ignoring the impact of cache slices.
- *Best AV*: The attacker can pick the cores both the victim and the attacker are running on, e.g., in the case of a covert channel or a side-channel attack in which the attacker launches the victim process. It ignores the impact of slices.
- *Known* $\tilde{S}$: The attacker does not know on which core she or her victim runs, but takes into account the slices, using per-slice thresholds. We use this model for comparison with the GT model.
- *Best A, Known* $\tilde{S}V$: The attacker pins her process to the best core, knows the victim core and takes into account the slices. This is a realistic attacker model. To be compared with *Best A, Known V* model.
- *Best AV, Known* $\tilde{S}$: This is the most powerful side-channel attacker, that can pin both the attacker and victim.
- *Best AV*$\tilde{S}$: This is the best covert channel attack model, where the attacker chooses the cores and an address in a slice that yields the best results.

6.2 Experimental results on error rate

For each (A,V,$\tilde{S}$) we make 2^{20} measurements, 2^{19} hits (in E state), and 2^{19} misses (in I state). We time how long `clflush` takes to execute in each case using the `rdtsc` instruction and build a histogram of the execution time distribution. From these histograms, we can evaluate the number of hits and misses that would be correctly or incorrectly classified using a threshold, and determine thresholds that minimize the average error rate for each model, along with the corresponding average error rate. We present three such histograms above:

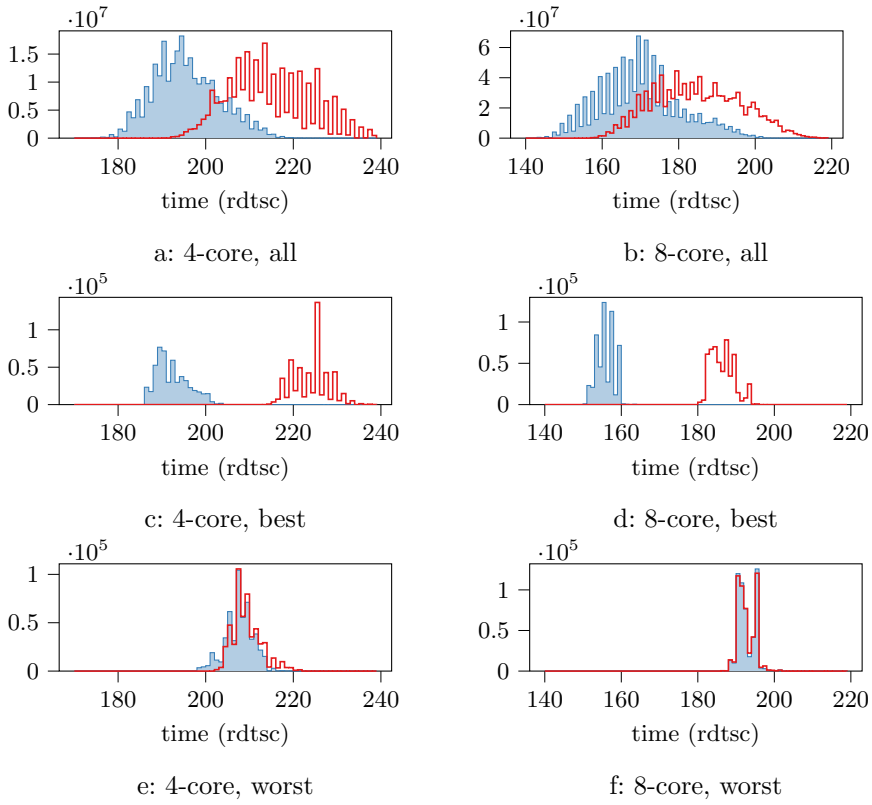


Fig. 3: Histograms for both machines of hit (outlined, red) and miss (filled, blue) `clflush` timing distributions for: – a, b: the superposition of all possible $(A, V, \tilde{S})$ triples (Average in the GT model). – c, d: the best possible $(A, V, \tilde{S})$ choice (*Best* $AV\tilde{S}$ model) – e, f: the worst possible $(A, V, \tilde{S})$ choice.

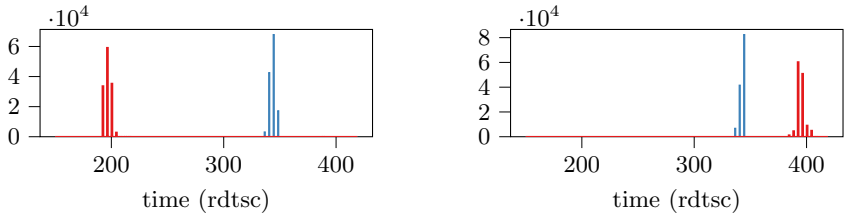
– In Figure 3a and 3b the histograms for all attackers, victim, and slices.
– In Figure 3c and 3d the histograms on the best choice of attacker, victim, and slice equivalence class in the *Best* $AV\tilde{S}$ attacker model.
– In Figure 3e and 3f the histograms on the most unfavorable choice of attacker, victim, and slice, with severe overlap between the two distributions.

Table 1 shows the results for the 4-core and 8-core machines, indicating for A, V and $\tilde{S}$ whether they are unknown, known or chosen in each case. For the 8-core machine, we observe a staggering difference between the 25% error rate of the GT attacker model, to the close to 0% error rate of the *Best* $AV\tilde{S}$ model (less than 1 error per 2^{20} measures).

Summary. Choosing the attacker and victim locations significantly improves the accuracy over the very unreliable global threshold. On top of that, using a

Table 1: Results for each attacker model on the 4-core and 8-core machines. U. means Unknown, and K. Known.

	4-core machine				8-core machine			
	Error rate	A	V	$\tilde{S}$	Error rate	A	V	$\tilde{S}$
GT	14.0%	U.	U.	U.	25.1%	U.	U.	U.
Best A, Known V	6.07%	3	K.	U.	10.5%	7	K.	U.
Best AV	0.176%	7	0	U.	0.115%	7	8	U.
Known $\tilde{S}$	11.6%	U.	U.	K.	22.8%	U.	U.	K.
Best A, Known $\tilde{S}V$	3.16%	5	K.	K.	7.18%	7	K.	K.
Best AV, Known $\tilde{S}$	0.103%	7	0	K.	0.0174%	1	0	K.
Best $AV\tilde{S}$	4.96×10^{-3}%	3	3	3	0 ($< 2^{-20}$)	2	7	14



a: A and V in the same socket. b: A and V on different sockets.

Fig. 4: Histograms of hit (red) and miss (blue, around 340) `clflush` timing distributions, for two different (A, V) pairs on a 2x Intel Xeon E5-2630 v3 machine.

per-slice threshold provides a further boost. However, when the victim cannot be chosen, accounting for slices gives a much greater boost. Lastly, choosing the best combination of attacker, victim, and slice gives close-to-perfect error rates.

6.3 The case of dual-socket machines

In dual-socket machines, there is no cache shared between all of the coherency domain. Coherence is maintained using bus snooping and using ECC bits in the DRAM to store some coherency information [14,22]. Thus, `clflush` behavior differs significantly from single-socket systems, depending on the attacker and victim location. The slice is not attached to a specific socket as each socket has its own last-level cache, and thus its contribution here was not studied in detail.

Figure 4a shows that when the victim is in the same socket, we observe that a hit is faster than a miss. This makes sense if the socket last-level cache has the coherency info of the accessed line in the E state, whereas it needs to reach out to the DRAM directory on a miss. However, when the victim is located in the other socket, *a hit is slower than a miss* as shown by Figure 4b. This can probably be explained because more communication is required in the former case, to cause the remote core to evict and then update the DRAM directory.

Overall, if the sockets on which the attacker and victim reside are not controlled, a simple threshold model will give poor results. A dual threshold model may give good quality results, separating same-socket hits, misses and remote-socket hits, or a detailed model accounting for attacker and victim location.

7 Evaluation

In this section, we evaluate our improved Flush+Flush primitive on a covert channel and on a side-channel attack on the AES T-tables implementation.

7.1 Building a better channel

Protocol. We implement a framework to benchmark covert channel ideal bandwidth with different primitives. We use the same protocol for each primitive. The benchmark uses two threads in the same process, and an optimized synchronization primitive. Such an ideal synchronization is unlikely to exist in real-world implementation but it allows us to measure theoretical limits of the channel itself. Real-world channels are likely to observe a lower bit-rate, and a corresponding decrease in true capacity, but with similar error rates.

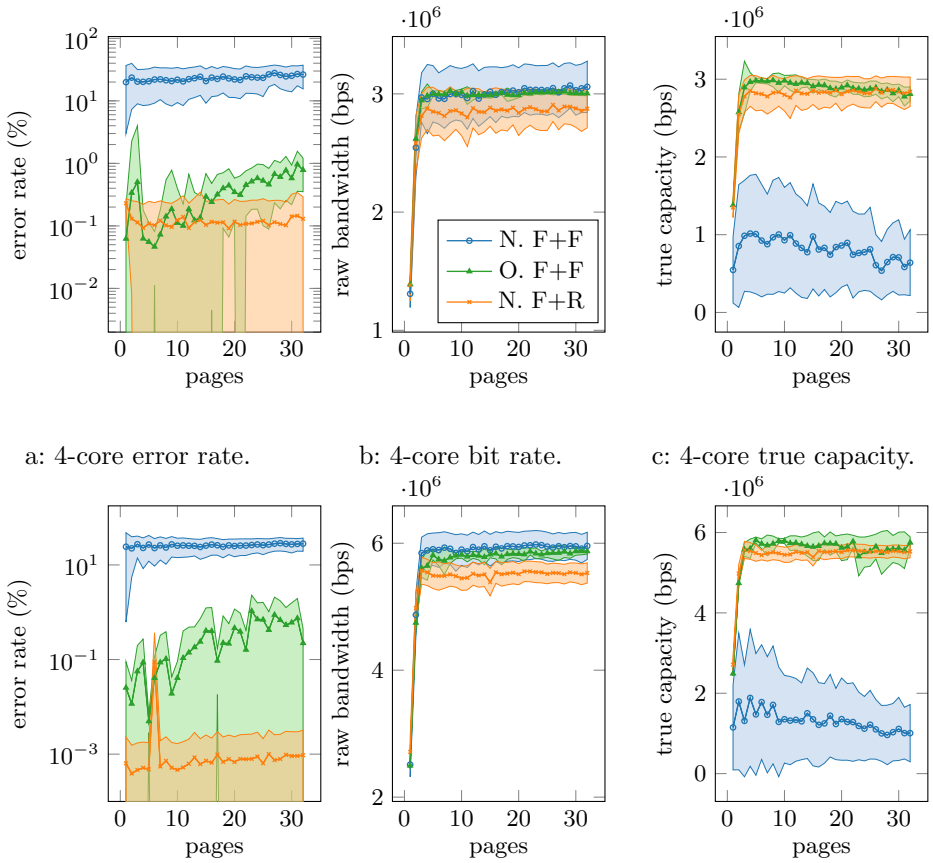
In practice, we use several shared pages, and within each, we pick an address in the optimal $\tilde{S}$. We also synchronize on a per page basis indicating which thread can currently access the page (to transmit or receive), using mutable shared memory, as the ideal synchronization primitive. Once done with a page, threads hand the page over to the other threads by flipping the per-page bit.

Implementation. We implement three covert channels with different primitives: 1. a single threshold *naive* Flush+Flush, with no core pinning (GT model), 2. a single threshold Flush+Reload that doesn't need to account for topology, and 3. a topology-aware Flush+Flush using the *Best* $AV\tilde{S}$ attacker model.

Evaluation. For each channel on each machine, we evaluate the raw bit rate C, the error rate p, and the (true) capacity $T = C \times (1 + p \log_2 p + (1-p) \log_2 (1-p))$ [23].

Results. We run our experiments on both machines mentioned in Section 4. Figure 5 shows statistics on the performance of the covert channels depending on the number of pages used, for each machine: the average error rate, the raw bit rate, and the true capacity of the resulting channel.

As shown by table Table 2, our carefully calibrated Flush+Flush yields a threefold increase in bandwidth on both machines compared to the naive Flush+Flush, and provides a bandwidth higher than Flush+Reload by 3 to 4 %. We conclude that Flush+Flush is now a compelling alternative to Flush+Reload.



a: 4-core error rate. b: 4-core bit rate. c: 4-core true capacity.

d: 8-core error rate. e: 8-core bit rate. f: 8-core true capacity.

Fig. 5: Covert channel performance depending on the number of pages used.

7.2 AES T-tables attack using Flush+Flush

AES T-tables implementation. The AES T-tables implementation is well-known to be vulnerable to side-channel attacks, we, therefore, use it as a benchmark to compare our Flush+Flush implementation [2,7,12,4,5,31,11,24]. We compare our improved Flush+Flush implementation with per-slice thresholds to the naive version of Flush+Flush and to the Flush+Reload attack. We attack the OpenSSL 1.1.1g library, compiled with `no-asm` and `no-hw` to enable T-tables. For this experiment, prefetchers are enabled on both machines.

T-tables are an implementation of an AES round using lookups in tables. The lookup in the first round is $T_j[p_i \oplus k_i]$, where $0 \leq i \leq 16$ and j is the remainder of i divided by 4 (`j = i & 0x3`). With 4-byte elements and 64-byte cache lines,

Table 2: Result of covert channel benchmarking

	4-core machine			8-core machine		
Channel	Capacity	Bit rate	Err. rate	Capacity	Bit rate	Err. rate
Naive F+F	1.01 Mbit/s	2.96 Mbit/s	20%	1.88 Mbit/s	5.89 Mbit/s	23%
Opt. F+F	2.99 Mbit/s	3.03 Mbit/s	0.1%	5.81 Mbit/s	5.81 Mbit/s	0.005%
F+R	2.88 Mbit/s	2.91 Mbit/s	0.1%	5.57 Mbit/s	5.57 Mbit/s	0.0005%

there are 16 entries per cache line, and a cache attack can only monitor the upper 4 bits of $p_i \oplus k_i$. See Osvik et al. [24] for the detailed explanation.

Attacking the T-tables. We run the attack using all three side channels with the attacker and the victim in the same thread. For the naive Flush+Flush and Flush+Reload, the core is chosen randomly. For our improved Flush+Flush, we chose the best core according to Section 6, with the model *Best AV, Known* $\tilde{S}$.

We run the experiment with two different keys. One of them is the null key, and the other is a key with $k_0 =$ `0x51`. In this chosen-plaintext attack, a byte of the plaintext is set to fixed values ($0x00$, $0x10$, $0x20$, by increment of $+0x10$), while the remainder is chosen randomly. In this case, one of the cache lines (depending on the fixed byte value) of the T-table is deterministically accessed, while the other ones are not always accessed, and have a higher number of misses. Plotting the misses, such cache lines show distinctive pattern that identifies a byte of the key. Notably, the null-key pattern is diagonal.

Results. We observe that a naive Flush+Flush attack will show some lines with all hits or all misses, due to the threshold depending on the slice (see Figures 6a, 6b, 6c and 6d). Using a per-slice threshold (see Figures 6e, 6f, 6g and 6h) allows us to achieve an accuracy similar to Flush+Reload (see Figures 6i, 6j, 6k and 6l). Again, accounting for the contribution of slices and CPU interconnect to `clflush` timing variations makes an optimized Flush+Flush channel competitive with Flush+Reload, and improves the reliability over naive Flush+Flush.

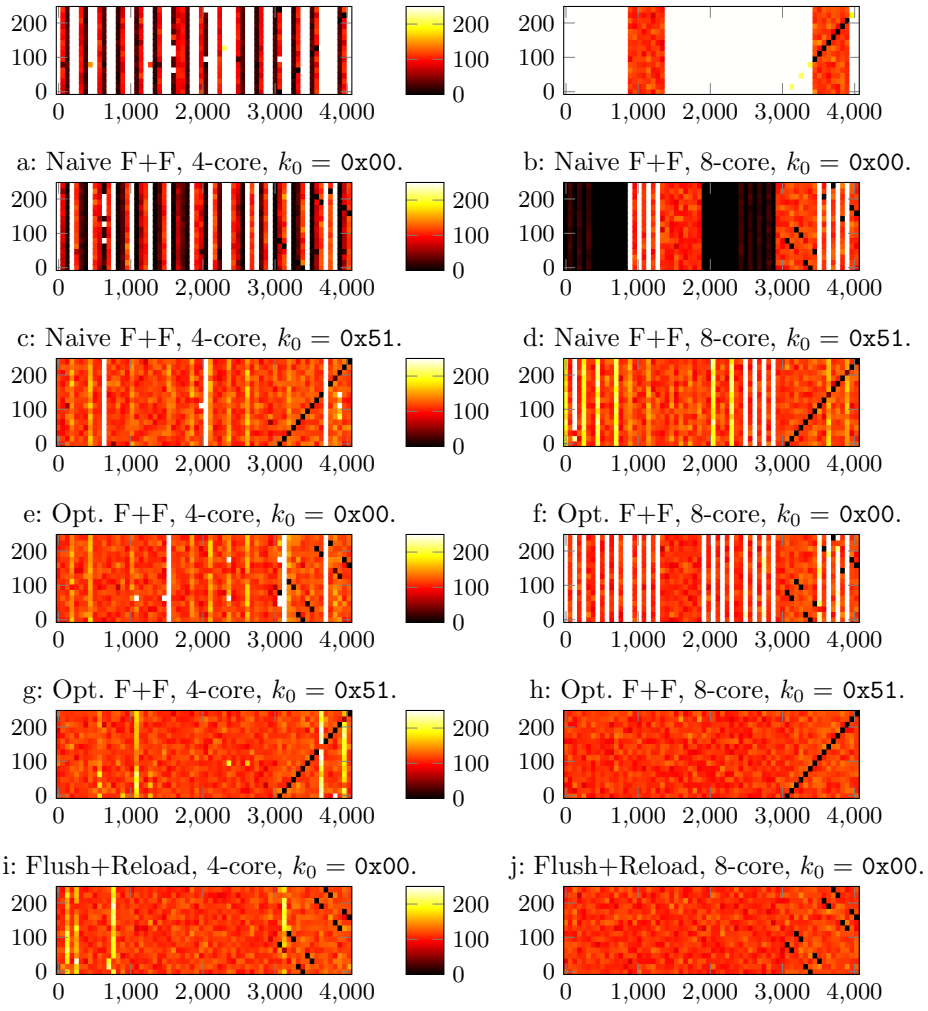
8 Related work

Cache attacks are a rich field, with several primitives extensively studied and new emerging ones. The AES T-tables implementation is well-known to be vulnerable to side channels, with various ways of exploiting it. Moreover, reliable covert channels are one of the key elements for transient microarchitectural attacks.

8.1 Cache attacks primitives

The first cache-based attacks were published around 2005. Percival [26] attacked RSA while Osvik et al. [24] attacked AES and were the first to define the Prime+

Probe primitive. The "high resolution [and] low noise" Flush+Reload primitive was defined by Yarom et al. [35], which was then automated by Gruss et al. [11] with Cache Template Attacks. Gruss et al. [10] then introduced the stealthy Flush+Flush primitive, a variant of Flush+Reload.



a: Naive F+F, 4-core, $k_0 =$ `0x00`.

b: Naive F+F, 8-core, $k_0 =$ `0x00`.

c: Naive F+F, 4-core, $k_0 =$ `0x51`.

d: Naive F+F, 8-core, $k_0 =$ `0x51`.

e: Opt. F+F, 4-core, $k_0 =$ `0x00`.

f: Opt. F+F, 8-core, $k_0 =$ `0x00`.

g: Opt. F+F, 4-core, $k_0 =$ `0x51`.

h: Opt. F+F, 8-core, $k_0 =$ `0x51`.

i: Flush+Reload, 4-core, $k_0 =$ `0x00`.

j: Flush+Reload, 8-core, $k_0 =$ `0x00`.

k: Flush+Reload, 4-core, $k_0 =$ `0x51`.

l: Flush+Reload, 8-core, $k_0 =$ `0x51`.

Fig. 6: Results of the T-table attack using a Naive Flush+Flush, Optimized Flush+Flush and Flush+Reload side channels. Each column represents an address and each row corresponds to a different value of the first byte of the chosen plaintext, with the remaining bytes filled randomly. The color scale cuts off lines with too many misses, T-tables that are deterministically accessed have very few misses and reveal key bits.

Cache attacks on cloud computing and virtualized environments [27,33,32], were shown to be a practical threat [19,18]. Maurice et al. [21] also studied protocols that could obtain a reliable channel on top of various primitives.

In recent years, various other primitives have been developed to adapt to evolutions in modern CPUs. Yan et al. [34] reverse-engineered non-inclusive caches directories to mount an attack on CPUs with non-inclusive caches, while Saxena et al. [28] tackled dynamic frequency scaling, and pointed out a first difference between same core and different core attackers. Briongos et al. [8] uncovered the replacement policy of some Intel CPUs and built an attack that avoids causing misses to the victim, whereas Flush+Flush avoids causing for the attacker.

Since 2018, transient microarchitectural attacks, such as Meltdown [17], Spectre [15] and Fallout [9] make extensive use of reliable cache-based covert channels.

In concurrent work, Paccagnella et al. [25] have built a contention-based channel on the ring interconnect, reversing in detail the protocol for memory loads and the finer structure of the interconnect.

8.2 Attacking AES T-tables

Koeune and Quisquater [16] uncovered an implementation issue in AES that caused a timing attack. Bernstein [6] also developed a timing-based attack and uncovered various sources of variability including caches. Osvik et al. [24] then published the first attack based on monitoring the T-tables accesses. Many related publications [2,7,12,4,5,31,11] now use the AES T-tables as a benchmark.

9 Future work

We have explored the timing of `clflush` for two coherence states, but using our framework, it should be possible to set-up lines in other coherence states, such as shared (S) and modified (M), that do not impact side-channel research, but can help to better understand CPU memory hierarchy and performance.

The impact of frequency on timing channels, especially those relying on small differences is significant. Most attacks are described at a steady frequency, but in a real setting, frequency scaling can severely hamper them. A model instruction execution time depending on the frequency could mitigate this variability.

Intel large server CPUs starting with Skylake Scalable Processors (SP) no longer use inclusive caches. However the ISA still requires that `clflush` flushes a cache line from all the coherency domain. It should thus be possible to use the `clflush` instruction to attack such systems, an approach that [34] has not covered. These systems also use a different topology that warrants further inquiry.

`clflush` also behaves differently on multi-socket systems, as shown in Section 6.3, in a way that is not always tractable with a simple global threshold model. Further work could evaluate the benefits of dual-threshold versus per $A, V, \tilde{S}$ threshold models, and the performance of channels built in this way.

10 Conclusion

The interconnect topology of Intel CPU plays a larger role than was previously known in cache attacks, and in particular Flush+Flush. A naive Flush+Flush implementation that does not account for the topology yields poor error rates, especially as the number of cores increases. We reverse-engineer this topology and study its timing impact on the `clflush` instruction. Using these insights, we significantly enhance the Flush+Flush primitive by accounting for the topology during the calibration step. Consequently, we recommend taking into account these findings into the calibration step, measuring timings for all possible combinations of attacker, victim, and home slice location, and then determining the best thresholds depending on the attacker model. Our results therefore demonstrate that the Flush+Flush primitive is as reliable as Flush+Reload, with the further advantages in stealth and being less affected by prefetcher noise.

Acknowledgements This work has been partly funded by the French Direction Générale de l'Armement, and by the ANR-19-CE39-0007 MIAOUS. Some experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

1. Coffee Lake - Microarchitectures - Intel - WikiChip (2020), `https://en.wikichip.org/w/index.php?title=intel/microarchitectures/coffee_lake&oldid=97412#Octa-Core`, last edited 2020-07-03
2. Aciiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (short paper). In: Information and Communications Security, ICICS (2006)
3. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. p. 483–485. AFIPS '67 (Spring), ACM (1967)
4. Apecechea, G.I., Inci, M.S., Eisenbarth, T., Sunar, B.: Fine grain cross-VM attacks on Xen and VMware are possible! IACR Cryptol. ePrint Arch. **2014**, 248 (2014), `http://eprint.iacr.org/2014/248`
5. Apecechea, G.I., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! A fast, cross-vm attack on AES. In: RAID (2014)
6. Bernstein, D.J.: Cache-timing attacks on aes (2005)
7. Bogdanov, A., Eisenbarth, T., Paar, C., Wienecke, M.: Differential cache-collision timing attacks on AES with applications to embedded cpus. In: CT-RSA (2010)
8. Briongos, S., Malagón, P., Moya, J.M., Eisenbarth, T.: RELOAD+REFRESH: abusing cache replacement policies to perform stealthy cache attacks. In: USENIX Security Symposium (2020)
9. Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Bulck, J.V., Yarom, Y.: Fallout: Leaking data on meltdown-resistant cpus. In: CCS (2019)

10. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+flush: A fast and stealthy cache attack. In: DIMVA (2016)
11. Gruss, D., Spreitzer, R., Mangard, S.: Cache template attacks: Automating attacks on inclusive last-level caches. In: USENIX Security Symposium (2015)
12. Gullasch, D., Bangerter, E., Krenn, S.: Cache games - bringing access-based cache attacks on AES to practice. In: S&P (2011)
13. Hennessy, J.L., Patterson, D.A.: Computer Architecture - A Quantitative Approach, 6th Edition. Morgan Kaufmann (2019)
14. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual (2018), https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf
15. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: S&P (2019)
16. Koeune, F., Koeune, F., Quisquater, J.J., jacques Quisquater, J.: A timing attack against rijndael. Tech. rep. (1999)
17. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: USENIX Security (2018)
18. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: S&P (2015)
19. Maurice, C., Neumann, C., Heen, O., Francillon, A.: C5: cross-cores cache covert channel. In: DIMVA (2015)
20. Maurice, C., Scouarnec, N.L., Neumann, C., Heen, O., Francillon, A.: Reverse engineering intel last-level cache complex addressing using performance counters. In: RAID (2015)
21. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C.A., Mangard, S., Römer, K.: Hello from the other side: SSH over robust cache covert channels in the cloud. In: NDSS (2017)
22. Molka, D., Hackenberg, D., Schöne, R., Nagel, W.E.: Cache coherence protocol and memory performance of the intel haswell-ep architecture. In: 44th International Conference on Parallel Processing, ICPP (2015)
23. Okhravi, H., Bak, S., King, S.T.: Design, implementation and evaluation of covert channel attacks. In: 2010 IEEE International Conference on Technologies for Homeland Security (HST). pp. 481–487 (2010). https://doi.org/10.1109/THS.2010.5654967
24. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: CT-RSA (2006)
25. Paccagnella, R., Luo, L., Fletcher, C.W.: Lord of the ring(s): Side channel attacks on the cpu on-chip ring interconnect are practical. In: S&P (2021)
26. Percival, C.: Cache missing for fun and profit. In: Proc. of BSDCan 2005 (2005)
27. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: CCS (2009)
28. Saxena, A., Panda, B.: DABANGG: time for fearless flush based cache attacks. IACR Cryptol. ePrint Arch. (2020)
29. Vila, P., Ganty, P., Guarnieri, M., Köpf, B.: Cachequery: learning replacement policies from hardware caches. In: PLDI (2020)
30. Vila, P., Köpf, B., Morales, J.F.: Theory and practice of finding eviction sets. In: S&P (2019)

31. Weiß, M., Heinz, B., Stumpf, F.: A cache timing attack on AES in virtualization environments. In: Keromytis, A.D. (ed.) Financial Cryptography and Data Security, FC (2012)
32. Wu, Z., Xu, Z., Wang, H.: Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. IEEE/ACM Trans. Netw. **23**(2), 603–615 (2015)
33. Xu, Y., Bailey, M., Jahanian, F., Joshi, K.R., Hiltunen, M.A., Schlichting, R.D.: An exploration of L2 cache covert channels in virtualized environments. In: Cloud Computing Security Workshop, CCSW. pp. 29–40. ACM (2011)
34. Yan, M., Sprabery, R., Gopireddy, B., Fletcher, C.W., Campbell, R.H., Torrellas, J.: Attack directories, not caches: Side channel attacks in a non-inclusive world. In: S&P (2019)
35. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security Symposium (2014)
36. Yarom, Y., Ge, Q., Liu, F., Lee, R.B., Heiser, G.: Mapping the intel last-level cache. IACR Cryptol. ePrint Arch. (2015)

A Cache slicing functions uncovered

Our research relies on having prior knowledge of the cache slicing functions. We have updated the code base used by Maurice et al. [20] to support newer architectures and used it to uncover the slicing functions of the i9-9900 (Coffee Lake R, 8 cores) and the older i7-4980HQ (Crystal Well, 4 core Haswell with an eDRAM L4 cache), which differ from the previously known functions (see Table 3) that applied to most CPUs from Sandy Bridge to Broadwell. The CPU in our 4-core machine also uses those well known functions. The most significant bits of the functions uncovered are limited by the available memory.

This method uses performance counters located in a per physical core structure called CBox. The uncovered functions map addresses onto each CBox. However, it is suspected that starting with Skylake there are two slices within the same CBox [30], which we cannot detect with this method.

Table 3: Functions from [20] for the 2-, 4- and 8-core Xeon and Core CPU and new functions for the Intel Core i7-4980HQ and i9-9900.

		Address Bit																														
		37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
Sandy Bridge & later [20]	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	⊕		⊕	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕			⊕	
	o_2	⊕	⊕	⊕	⊕			⊕	⊕			⊕	⊕			⊕	⊕			⊕			⊕			⊕	⊕			⊕			
(New) i7-4980HQ	o_0					⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				
	o_1					⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕					
(New) i9-9900	o_0					⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1						⊕	⊕		⊕	⊕	⊕	⊕	⊕		⊕	⊕	⊕		⊕		⊕	⊕				⊕			⊕			
	o_2					⊕	⊕	⊕	⊕	⊕	⊕	⊕				⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕			⊕		⊕	⊕				