Flush-based Cache Attacks on Modern / Multi-Socket x86 Systems

Guillaume Didier, Thomas Rokicki, Augustin Lucas

Flush-based Cache Attacks on Modern / Multi-Socket x86 Systems

Guillaume Didier
Universität des Saarlandes, Germany;
Formerly: DGA; Univ. Rennes, Inria, IRISA

Thomas Rokicki
CentraleSupélec, Inria, CNRS,
IRISA, Université de Rennes; France

Augustin Lucas
Département d'Informatique, ENS de Lyon;
Univ. Rennes, Inria, IRISA; France

Abstract

Flush-based cache attacks have been extensively studied and leveraged, yet their behavior on today's complex x86 platforms is not fully understood. Notably, as cache and memory latency depend on the physical layout of cores, caches, and Numa nodes, system topology increasingly influences the latencies underlying such attacks. We thus investigate the impact of this growing complexity on the effectiveness of flush-based attacks. We present a large-scale, topology-aware study of Flush+Reload and Flush+Flush across 36 Intel and AMD, single- and multi-socket systems, with server and client CPUs. We show topology-induced contributions dominate the latency variations. In particular, Numa's contribution on multi-socket systems makes topology-unaware attacks unreliable. Our topology-aware calibration accounts for topological parameters like attacker/victim cores, memory Numa node, and target address, improving error rates and attack viability. We demonstrate Flush+Flush works on AMD targets, and is often more accurate than Flush+Reload on modern x86 platforms. Finally, we introduce Load/Flush+Reload, a covert-channel comparing invalid to shared loads, with double the true capacity than Flush+Reload/Flush+Flush. Our results show that topology awareness is required for dependable cache attacks on recent platforms, and provide practical guidance for attackers and defenders.

1 Introduction

Cache-based side and covert channels are among the oldest and most widely exploited microarchitectural leakage primitives [16, 17]. They remain essential to both offensive and defensive research: most transient execution attacks rely on such channels to transfer data from the transient to the architectural domains [20]. They are also used to reverse engineer undocumented aspects of modern microarchitectures [6].

On x86, the `clflush` instruction provides an unprivileged way to evict cache lines from all levels of the cache hierarchy, enabling fine-grained observation of memory access behavior within (read-only) memory shared by the attacker and victim. It forms the basis of the Flush+Reload attack [22], which remains a fundamental primitive in cache-based side-channel attacks. Gruss et al. [8] also observed that `clflush` latency depends on the cache line's coherence state on Intel CPUs, leading to the Flush+Flush attack. To our knowledge, Flush+Flush has never been demonstrated to work on AMD CPUs [6].

Modern CPUs have grown increasingly complex: As illustrated in Figure 1, modern systems feature several levels of caches, with per-core slices in the last-level cache, complex on-chip and off-chip interconnect, and multiple Numa nodes in multi-socket systems. Consequently, the latency of memory accesses depends strongly on the relative placement of cores, cache slices, and Numa nodes within the system topology, as illustrated in Figure 2. For example, reading data from memory (cache miss) located in a remote Numa node incurs a higher latency (600 cycles on 2× Sapphire Rapids system) than reading from the local node's memory (only 400). While this behavior is intuitive, its impact on the accuracy, reliability, and timing characteristics of cache-based side-channel attacks has yet to be quantified. This gap motivates our study and leads to our main research question:

To what extent does the growing complexity of x86 systems affect the effectiveness of flush-based cache attacks?

Our study seeks to determine which topological factors most strongly influence the variability of Flush+Flush and Flush+Reload attacks, and how attacker awareness of these factors improves accuracy and bandwidth. We perform a detailed analysis of memory latency across all relevant topological dimensions, including attacker and victim core placement, cache slice, and Numa node, to quantify their respective impact on timing variability and error rates. To further quantify the gains of topology awareness, we evaluate the bandwidth and error rates of different topological configurations in a covert-channel setting, assessing the speed and robustness of these primitives in practice. To ensure that our observations generalize across platforms, we conduct a large-scale empirical study spanning 36 Intel and AMD systems, encompassing client, server, and multi-socket architectures.

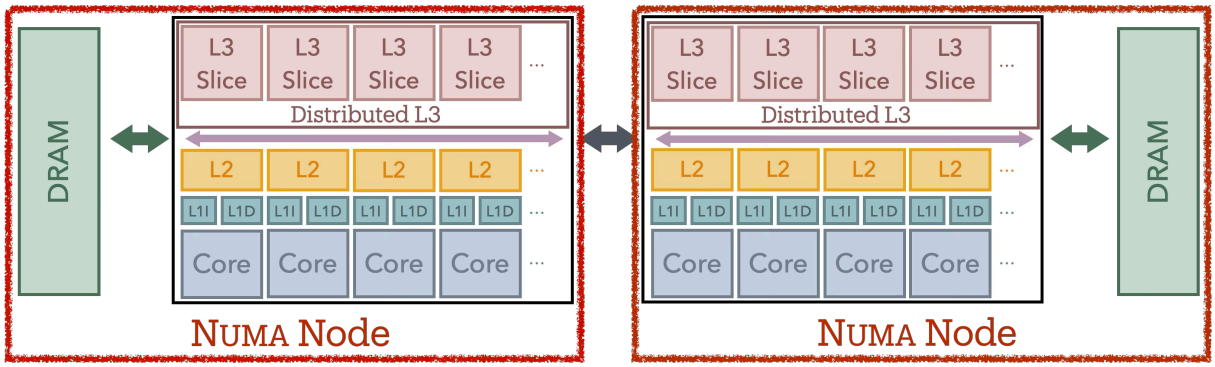


Figure 1: Example of the topology of a multi-socket system, with non-uniform memory access (NUMA).

We find that topology awareness, in particular NUMA awareness, can reduce error rates of attacks by up to an order of magnitude compared to topology-unaware methods. For instance, on `esterel41`, a two-socket Intel Sapphire Rapids machine, topology-unaware Flush+Flush has an average error rate of 27.55%, while the topology-aware average is 1.53%. If the attacker is not merely aware, but is allowed to select the best configuration, then the error rate drops below 0.01%.

We also present Load/Flush+Reload, a more accurate variant of Flush+Reload, which provides a higher bandwidth, however limited to a covert-channel threat. On a single socket Zen 5 machine, it achieves a true capacity of 4.98 Mbit/s, a 2× improvement over Flush+Flush and Flush+Reload, which only achieve, respectively, 1.95 Mbit/s and 2.27 Mbit/s.

To summarize, we make the following contributions:

— We demonstrate that topology is a major contributor to load and `clflush` timing, with NUMA being the most significant factor in multi-socket systems.
— We show that, on modern systems, accurate Flush+Reload and Flush+Flush attacks require topology awareness.
— We present an evaluation on 36 x86 systems, of the impact of various parameters on cache attack performance.
— We show that Flush+Flush is widely applicable on x86 machines, including AMD and multi-socket systems, and may be more accurate than Flush+Reload in many cases.
— We present a new covert-channel primitive, Load/Flush+Reload, whose bandwidth can reach 2× that of Flush+Reload and Flush+Flush, and far lower error rates.

This paper is organized as follows: Section 2 recalls relevant background and prior work. Section 3 introduces the parameters, threat model, and an overview of our experiments. Section 4 evaluates the contribution of system topology on `clflush`-based attacks. Then, Section 5 evaluates the benefits of topology awareness on covert channels. Finally, we discuss limitations, related, and future work in Section 6.

2 Background and Related Work

2.1 Modern System Memory Hierarchy

2.1.1 Cache Hierarchy and Structure

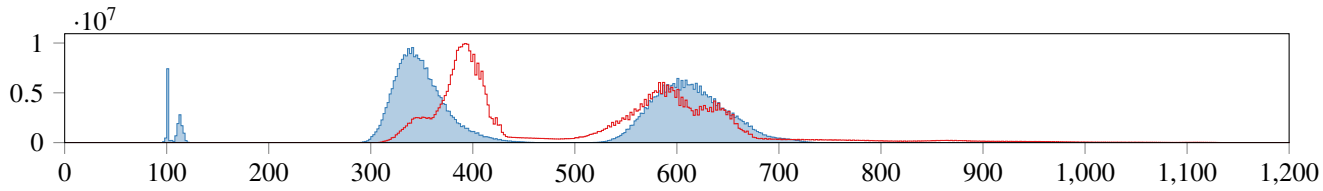
In order to hide main memory latency, modern processors include smaller and faster memories called *caches*. Caches keep copies of recently used *cache blocks* (or memory blocks), made of a fixed number of bytes. Usually, x86 systems use 64-byte cache blocks. We refer to the combination of a cache block and its associated cache metadata as a *cache line*. Cache lines usually include a tag, to identify the memory address of the block within, and extra bits, such as the line's validity. Most caches are accessed using the physical addresses rather than the virtual addresses, to avoid synonym problems [10].

To better bridge the gap between memory and core speed, modern systems generally have three levels of caches. First, each physical core has a private level 1 instruction cache (L1I) and a private level 1 data cache (L1D). Both of these caches then fall back to a level 2 private cache (L2). Last, a shared level 3 cache can fulfill requests for all cores in the chip, as the *last-level cache* (LLC). However, in systems with multiple sockets, each socket has a separate last-level cache [10]. This can also be the case in multi-chip modules, a technology that is used by AMD in its larger processors.

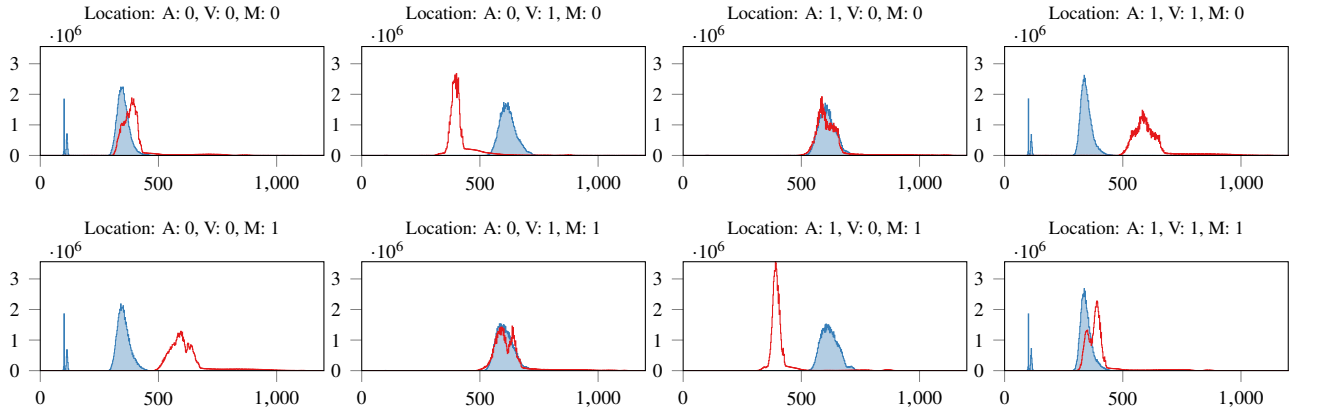
A cache is *inclusive* when it must contain a copy of every line in the lower level caches [10]. This is true for LLCs of older Intel CPUs. However, Intel introduced non-inclusive LLCs in the Skylake-SP server micro-architecture [11].

2.1.2 Cache Coherence

As modern systems include multiple cores, and both private and shared caches, cores may observe different values for the same memory location, due to stale copies in private caches. To avoid such unintuitive behavior, modern CPUs provide *cache coherence* to ensure all cores agree on the sequence of values taken by a memory location (usually at the cache block granularity). This is done by enforcing a *single-writer or multiple-reader* (SWMR) invariant [10, 14].



(a) Topology-unaware histogram, weighing equally all possible configurations



(b) NUMA-aware histograms, under the NUMA-AVM model.

Figure 2: Flush+Reload timing histograms on `esterel41`, *cf.* Table 11 (2× Sapphire Rapids). In red, outlined, loads of an invalid line (*I*) *i.e.*, cache misses. In blue, filled, loads of a victim-exclusive line (*E*), *i.e.*, cache hits.

Cache coherence is implemented using protocols, in which each cache line has a state. Variants of the classical MESI protocol are used by Intel (MESIF) and AMD CPUs (MOESI). In MESI, a cache line can be in the following state:

Modified: It holds the modified copy of the cache block, and must write it back before other cores may access it[1].

Exclusive: It holds the only clean copy of the block, and may be upgraded to the Modified (M) state at no cost.

Shared: It holds one of many clean copies of the block, other copies must be invalidated before writing.

Invalid: It contains no valid copy of any memory block.

Single-socket systems usually include a global directory to track the state of lines in the socket, while multi-socket systems need to implement a distributed protocol to maintain cache coherence across all sockets. Inclusive last-level caches often act as the directory, but modern systems increasingly have non-inclusive last-level caches, and separate directories [21].

2.1.3 Multi-socket Systems and NUMA

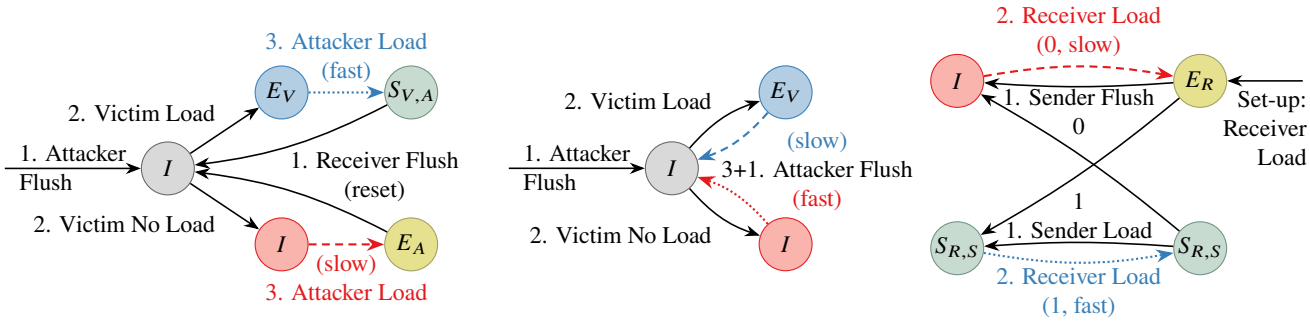
In modern multi-socket systems, each socket has its own memory controllers, and associated DRAM attached. Consequently, the access time to a given main memory location can vary significantly depending on whether the request is fulfilled by a memory controller on the same socket, or must traverse the socket interconnect to reach the correct memory controller. Systems where this is a significant issue are deemed to have *Non-Uniform Memory Access*, or NUMA. To optimize the performance of memory-heavy processes, the system is divided into NUMA-nodes. Each NUMA-node contains a group of physical cores with roughly the same memory access latencies, and the closest memory controller(s), giving them the best memory latency. For instance, pinning the memory of a process to the memory controller of a node and running the process' threads on physical cores in that node may reduce the memory latency. In other cases, maximizing the processing bandwidth involves spreading the threads over all NUMA-node, each thread processing data from the memory of its node.

Hence, operating systems usually allow processes to make placement requests for their threads and memory. Modern Linux kernels can also rebalance NUMA-node usage dynamically, migrating physical pages, from one node to another, to optimize the performance.

[1] Our attacks, as they target read-only memory, do not involve the M state.

2.2 Micro-architectural Attacks

Micro-architectural attacks violate confidentiality guarantees, meant to be provided by process isolation, by exploiting the ISA's micro-architectural implementation through timing or performance counters. We typically distinguish *side channels* in which an unsuspecting process leaks sensitive data through micro-architectural state, which the attacker can measure; from *covert channels* in which two attackers cooperate to stealthily exchange information across isolation boundaries.



(a) Flush+Reload (b) Flush+Flush (c) Load/Flush+Reload (New, *cf.* Section 3.1)

Figure 3: Cache coherence transition diagrams of the three primitives. Fast transitions in dotted, slow transitions dashed. The arrow colors match the histogram. I: Invalid state. E_C: Exclusively state on core C. $S_{C,D}$: Shared state between cores C and D.

2.3 `clflush`-based Attacks

`clflush` is an x86 *unprivileged* instruction that evicts the cache line given as an operand from the entire cache coherence domain, and ensures it is written back to main memory [11]. From a cache coherence perspective, `clflush` reliably transitions lines into the *Invalid* state.

This instruction enables the Flush+Reload and Flush+Flush attacks. These two attacks require shared memory between the attacker and victim, usually read-only. The attacker can then observe the addresses of memory accesses by the victim, in the shared memory, at cache-block granularity.

In **Flush+Reload** [22], the attacker first flushes the target cache line, and waits for the victim to, possibly, read that line. If the victim accesses the target line, it transitions into a Victim-Exclusive state, otherwise the line keeps its Invalid state. The attacker then measures the execution time to load the target line, distinguishing the two possible coherence states. The line transitions[2] respectively into a Shared or an Attacker-Exclusive state. The attacker then repeats the process. Figure 3a presents the coherence state transition for this attack, with the attacker and victim on distinct physical cores.

Flush+Flush [8] exploits the fact that on Intel CPUs, `clflush` execution time itself depends on the state of the target cache line. It is thus possible to avoid the Reload step of Flush+Reload and instead time a `clflush` instruction, which both measures and resets the state. As such, the target line alternates between the Invalid state and the Victim-Exclusive state, the latter following a memory access by the victim. Figure 3b presents the corresponding coherence states. Whether it works on AMD CPUs has yet to be ascertained.

2.4 Calibration of Timing Attacks

Cache-based side and covert channels rely on timing measurements to infer whether data was cached. In either case, the execution time must be mapped into one of the possible outcomes. The simplest way is often to classify by comparing the measured execution time to a threshold: intuitively, a cache hit is fast and a miss is slow. However, `clflush` is often faster when its operand is not in the cache.

In more complex systems, the timing distribution of cache operations may not be cleanly separable, and classification may benefit from using a range defined by two thresholds rather than a single one. Selecting the optimal thresholds requires collecting representative timing data for each outcome and analyzing the corresponding histograms of execution times. When multiple factors influence timing, such as which cores the attacker and victim execute on, or the NUMA node to which the target memory belongs, classification must account for each configuration independently. This approach, which we refer to as *topology-aware calibration*, assigns separate classifiers or thresholds to each relevant attack configuration to minimize misclassification rates.

3 Methodology

Our goal is to characterize the impact of the topology of the memory hierarchy on cache attacks in modern computer systems. To do so, we aim to explore all possible combinations of parameters that influence the latencies observed in these attacks. One such parameter is the memory controller that serves a cache miss in NUMA systems. In a two-socket system, a request to the other socket's memory controller incurs the extra cost of crossing the inter-chip interconnect[3], and of traversing two on-chip networks instead of one.

We call *topology* the combination of all the sources of variability induced by the difference in paths within the memory hierarchy. In practice, this covers the respective physical cores on which the **attacker** (A) and **victim** (V) processes run, the **memory** controller (M) to which the target memory belongs, and the latency caused by the internal organization of caches. For instance, on Intel CPUs, this is the cache slice to which

[2] Unless attacker and victim share the same physical core

[3] UPI on modern Intel systems

the target physical memory maps. We use the virtual **address** ($Addr$) of the target cache block as a proxy to this factor, as the hashing functions used by the last-level caches of our machines are not generally known, and attackers usually cannot access the physical addresses that would be needed.

Three research questions thus follow from our goal:

RQ1: Which attack is applicable on each micro-architecture?

RQ2: What is the impact of topology on attack accuracy, and what gains can be obtained with topology awareness?

RQ3: What covert channel true capacity can be achieved using the best trade-off between accuracy and attack speed?

We define a *configuration* as a tuple, $(A, V, M, Addr)$, of the attacker and victim physical core (and socket), target memory, and virtual address. Our experiments thus iterate over the entire set of possible configurations, and make a series of measurements for each. To answer these questions, we run two sets of experiments: The first one aims at measuring latencies for hits and misses in each primitive, building histograms such as Figure 2, to answer **RQ1** and **RQ2**. The second one aims at evaluating the performance of covert channels, in an end-to-end setting, answering **RQ3**.

3.1 Attack Primitives

We consider three attack primitives: First, the classical Flush+Reload (FR) and Flush+Flush (FF), already described previously (Section 2.3). In addition, due to a bug while implementing Flush+Reload, we discovered Load/Flush+Reload (LFR), a primitive which rearranges the elements of Flush+Reload, resulting in a different set of cache coherence transitions. While Flush+Reload compares the execution time of a load from an Invalid state (I) to a Load from an Exclusive state on the victim/sender core (E_V), this primitive compares the execution time of an Invalid load (I), with that of a load from a Shared state between the sender and the receiver ($S_{R,S}$). The receiver only issues loads, while the sender issues either a load (to encode a 1), or a flush, (to encode a 0). Since the receiver has previously loaded, the sender's action brings the cache line into either the Invalid or the Shared state. The attacker's loads leave Shared states unchanged, and turn Invalid states into a Receiver-Exclusive states. This results in a much faster attack, makes transmitting series of ones inexpensive, and increases the timing difference between the two transitions.

3.2 Attacker Models

We identify above several parameters that influence the latencies of hits and misses in cache attack primitives. The attacker aims to distinguish between two *outcomes*: hits and misses. Hence, she wants the distribution of timings for hits and misses to be well separated. However, if she accounts for none of these topology parameters, the distributions get wider and separation can get worse. The attacker thus wants to account for as many parameters as possible to narrow the timing distributions, and hopefully make *classification* easier. We consider two types of classifiers: a *threshold*, and a *range*, respectively denoted **ST** and **DT**[4]. A *threshold* maps values below it to one outcome, and above it to the other, while a *range* maps values inside the range, *i.e.*, between the two thresholds, to one outcome, and those outside to the other.

In this work, we consider different attacker models, representing the extent of knowledge and control the attacker has over these parameters. Each parameter can be either *Unknown* U ❓ (the same classifier is used for all possible values of the parameter), *Known* K ✏ (a distinct classifier is used for each possible value), or *Chosen* C ✅, (the attacker selects the parameter's value to get the best average result).

Generally, the NUMA node of the target memory (M) can be pinned to nodes using `libnuma`, which provides an ease-of-use interface on top of the Linux kernel system calls. These system calls are usually available to the unprivileged user, unless purposefully restricted using capabilities.

The address ($Addr$) being known means using one classifier per cache line. A covert channel attacker may choose the $Addr$ used, while a side-channel attacker may usually not[5].

The attacker and victim [19] core can be controlled precisely through the kernel `sched_setaffinity` system call, a more powerful interface than the attacker and victim NUMA-node selection from `libnuma`. Additionally, the victim core can also be found using `/proc/pid`, which provides the core affinity and the last core used.

Consequently, all these parameters can be realistically controlled by an attacker. We thus define the attacker models in Table 1, respectively Table 1a for multi-socket systems and Table 1b for single-socket ones.

3.3 Experiment 1: Calibration

This first experiment collects latency data for the hits and misses of each primitive. This experiment iterates over the space of $(A, V, M, Addr)$ configurations. We successively pin our process's memory to each NUMA-node M, and iterate over all (A, V) pairs of cores, and over all cache lines within a 4 KiB page. This is achieved by using `libnuma` and `sched_setaffinity`. For each of those configurations, we then take measurements for a set of *operations*, consisting of a victim execution to set up the cache state followed by the attacker execution of the primitive. We have three primitives, with two outcomes each, so we would expect six different operations. However, the miss in both Flush+Reload and Load/Flush+Reload is a load to an invalid cache line. Consequently, we thus only need five operations, described in Table 2, with the line *Load I* being used by two different primitives. For each operation, we take 1024 measurements

[4] For *Single Threshold* and *Dual Threshold*.

[5] It has not been explored if repeated NUMA migrations can allow the attacker to select a more suitable address.

Table 1: Attacker Models in this paper.

(a) Multi-socket

Model	Memory		Attacker (A)		Victim (V)	
	Numa node (M)	Address ($Addr$)	Numa node	Core	Numa node	Core
Topology-Unaware (TU)	U ?	U ?	U ?	U ?	U ?	U ?
Numa-AVM	K ✎	U ?	K ✎	U ?	K ✎	U ?
Numa-M-Core-AV	K ✎	U ?	K ✎	K ✎	K ✎	K ✎
Numa-M-Core-AV-Best	C ✓	U ?	C ✓	C ✓	C ✓	C ✓
Numa-M-Core-AV-Addr (TA)	K ✎	K ✎	K ✎	K ✎	K ✎	K ✎
Numa-M-Core-AV-Addr-Best (Best-TA)	C ✓	C ✓	C ✓	C ✓	C ✓	C ✓

(b) Single-socket

Model	$Addr$	A	V
Topology-Unaware (TU)	U ?	U ?	U ?
Addr	K ✎	U ?	U ?
Core-AV	U ?	K ✎	K ✎
Core-AV-Best	U ?	C ✓	C ✓
Core-AV-Addr (TA)	K ✎	K ✎	K ✎
Core-AV-Addr-Best (Best-TA)	C ✓	C ✓	C ✓

More attacker models are considered in the online appendix (cf. Appendix C).

In single-socket systems (Table 1b), the Numa columns vanish, making some models from Table 1a equivalent.

Table 2: Operations used in the *Calibration* experiment

Operation	Used by Primitive	Victim action	Attacker times	Attacker reset
Load I	F+R & LF+R (miss)	—	Load	`clflush`
Load E_V	F+R (hit)	Load	Load	`clflush`
Load $S_{A,V}$	LF+R (hit)	Load	Load	—
`clflush` I	F+F (miss)	—	`clflush`	—
`clflush` E_V	F+F (hit)	Load	`clflush`	—

after 128 warm-up iterations, whose timing measurement is discarded, before moving on to the next operation.

This results in a large set of histograms, associating 1024-sample histograms of the latencies of each operation with each configuration. Combining the respective hit and miss operations thus gives us histograms for each primitive which can be used to estimate the error rate of classifiers, and identify the best one. Following the attacker model, we select and stack together histograms that will use the same classifier, and obtain the average (**Avg.**) error on these. We also compute several statistics on the underlying series of histograms: minimum (**Min.**), maximum (**Max.**), median (**Med.**), and quartiles (**Q1** and **Q3**) of the observed error rates.

These predictions give an optimistic estimation of the error observed in a real attack: Our measurements, executed with no other activity on the system can benefit with the DRAM row remaining open, which speeds up the cache misses. In a real attack, this might not be the case, and the branch predictor state could also vary and induce more variability than here. This limitation motivates the second experiment, which evaluates the resulting classifiers in an end-to-end attack.

3.4 Experiment 2: Covert Channel Benchmark

To answer **RQ3**, this second experiment aims to evaluate end-to-end attacks, and measure the bandwidth of covert channels built with each attack primitive. This gives a more realistic representation of the error rates that can be achieved. We transmit 1024 bits between two threads, over a channel using n cache lines in separate pages, in a round-robin fashion. This transmission is repeated 16 times for each configuration, to obtain an average and a standard deviation. We explored values of n between 1 and 10 and found that 3 cache lines offered the best tradeoff between bandwidth and accuracy.

We use a generic implementation, parametrized with the primitives. It uses the algorithm from the first experiment to self-calibrate and select the classifier used. Because of that in-situ calibration, it requires a longer runtime than the first calibration experiment. Due to the longer runtime, we cover a smaller set of attacker models, with the following shorthands: **TU** (Topology-Unaware), **TA** (Topology-Aware) **Best-TA** (Best Topology-Aware), see Table 1. One notable caveat is that the Best-TA model selects its configuration to minimize the error rate. This can then result in choosing a configuration causing a decrease in attack speed.

To speed up the experiment, we use the symmetries of the system to reduce the number of configuration to be explored, pinning memory to a single Numa-node and deduplicating equivalent hyper-threads. We verify, in Section 4.4 the underlying assumptions.

From the raw bandwidth (C) and error rate (p), we then derive the true capacity T, using the following formula [15]:

$$T = C \times (1 + (1 - p)\lg(1 - p) + p\lg(p))$$

We thus present the error rate, raw bandwidth, and true capacity in Section 5, answering **RQ3**.

3.5 Experimental Setup

We run our experiments on 36 machines, described in Appendix B. The multi-socket machines use Debian 11, while the single-socket machines use a mix of Ubuntu releases.

In order to get reproducible results, we make a series of measurements with dynamic voltage and frequency scaling

(DVFS) and prefetchers disabled. We also took a second series of measurements with those features enabled in their default configuration. We include the corresponding data in our online supplement, *cf.* Appendix C.

In both experiments, fences and mutual exclusion prevent race conditions and enforce the correct ordering of operations.

In this section, we have thus described, our three research questions, **RQ1**, **RQ2**, and **RQ3**, and described the two experiments we use to answer them. Section 4 thus answers **RQ1** and **RQ2**, while Section 5 answers **RQ3**.

4 Results for Experiment 1: Calibration

In this section, we aim to answer **RQ1**: "Which attack is applicable on each micro-architectures?" and **RQ2**: "What is the impact of topology in attack accuracy, and what gains can be obtained with topology awareness?" To do so, we measured the latency distributions of hits and misses for each primitive (Section 3.1) under each $(A, V, M, Addr)$ configuration, and used them to build the histograms under each attacker model (Table 1). Table 12, in Appendix D includes a summary of the average predicted error rates for all 36 machines under the topology unaware (TU), the Numa-M-Core-AV-Addr (TA), and the Numa-M-Core-AV-Addr-Best (Best-TA) models. We successively discuss the results of single-socket client and server Intel CPUs, single-socket AMD CPUs, before moving to multi-socket systems. After verifying the symmetry assumptions needed for Experiment 2, we discuss results on two-socket Intel servers, two-socket AMD servers, and finish with the four-socket Intel server in our data set.

As a general note, most of the Load/Flush+Reload histograms tend to have a very narrow and very high peak for hits. To allow reading the wider, much less tall miss distribution, we usually truncate the peak for hits.

Concerning single-socket systems, Numa is not a concern. In addition, Didier and Maurice [5] demonstrated the benefits of topology awareness for Flush+Flush on Coffee Lake-R[6] client CPUs. We thus focused on checking the most recent Intel micro-architectures, Arrow Lake and Emerald Rapids, while looking at more AMD micro-architectures, which had no prior coverage for Flush+Flush. Table 12, in Appendix D presents the error rate predictions for all 36 tested machines.

To illustrate the behavior, we discuss the results for the most recent systems in each category: Intel single-socket client, Intel single-socket server, Intel two-socket (server only), AMD single-socket and AMD two-socket. We also include the only four socket machine we have.

4.1 Calibration of Intel Client CPUs

Arrow Lake (late 2024) is Intel's latest client micro-architecture. Under a *topology-unaware* model, Flush+Flush and Flush+Reload histograms exhibit pretty good separation, while Load/Flush+Reload shows even greater one, as shown in Figure 4. Hence, on ARL, topology awareness helps marginally, only reducing the worst-case error rate. Table 3 displays the predicted error rates for a selection of attacker models, where the additional parameters make a difference.

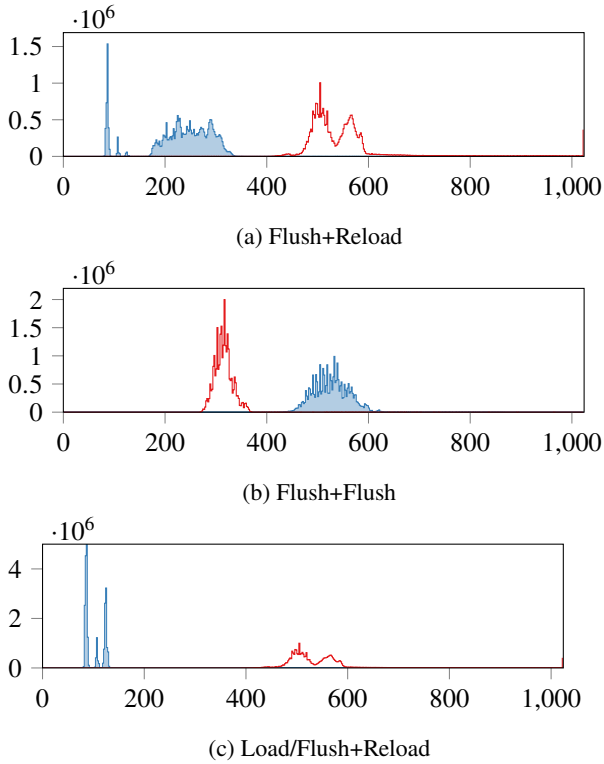


Figure 4: Flush+Reload, Flush+Flush, and Load/Flush+Reload topology-unaware histograms, on ARL, (1× Arrow Lake). Cache misses in red, outlined; hits in blue, filled.

We conclude that all three attacks work on recent Intel client CPUs (**RQ1**). On some systems, topology awareness benefits are marginal, however, our result on Coffee Lake (*cf.* Appendix C), and those of Didier and Maurice [5] indicate that in some cases, Flush+Flush accuracy may degrade significantly without topology awareness (**RQ2**).

4.2 Calibration of Intel Server CPUs

Emerald Rapids (2024) is Intel's latest server micro-architecture, and a minor evolution of Sapphire Rapids.

Figure 5 shows the topology-unaware histograms for all three primitives. For loads, cache misses overlap with exclusive hits in another core (250–400 cycles), such that topology-unaware Flush+Reload is ineffective. Flush+Flush exhibits good separation, and Load/Flush+Reload an even better one.

Table 4 presents a subset of the results under various attacker models, where once again, the additional parameters make a difference. Flush+Flush and Flush+Reload improve with more detailed attacker models, but that Flush+Reload

[6] Skylake 4th refresh, with minimal changes.

Table 3: Error rates (%) for ARL — 1× Arrow Lake — Fixed Frequency, No Prefetcher, Single Threshold

Model	Avg.	Min.	Q1	Med.	Q3	Max.
TU-FF	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.29
TU-FR	0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.59
TU-LFR	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.10
Core-AV-FF	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.24
Core-AV-FR	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.59
Core-AV-FF-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-FR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-LFR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-Addr-FF	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.24
Core-AV-Addr-FR	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.54

is still much noisier than alternatives. Figure 6 shows examples of Core-AV and Core-AV-Addr median and worst case histograms, which demonstrate the issue.

Here Flush+Flush is more accurate than Flush+Reload, with Load/Flush+Reload more accurate than both.

Overall, all three attacks are applicable to Intel server CPUs in single-socket systems (**RQ1**). Flush+Flush sees some small improvement with topology awareness, while Flush+Reload sees a significant one, going from 41% to 7.8% under Core-AV-Addr-FR. Choosing the best configuration then reduces to a negligible error rate. Topology awareness leads to major improvements for Flush+Reload on Intel server CPUs (**RQ2**).

4.3 Calibration of AMD CPUs

We now turn to AMD systems to evaluate whether the same trends hold. On AMD CPUs, the `rdtsc` instruction has a coarser granularity. From Zen 2 onwards, the `rdpru` instruction gives access to the APERF MSR, incrementing with clock cycles [7, 12]. We thus use it when available.

Figure 7 shows topology-unaware histograms, using the `rdpru` method for Flush+Reload and Flush+Flush, and `rdtsc` one for Flush+Flush, showing the benefits of the `rdpru` method. The error rates in Table 5 show that Flush+Reload and Flush+Flush both work, but with measurable error rates, while Load/Flush+Reload is much more reliable.

Overall, Flush+Flush works on all AMD systems, and on some micro-architecture, such as Zen 2, Flush+Flush is more reliable than Flush+Reload, *cf.* Appendix D. The accuracy of either method tends to be correlated, with some micro-architecture showing high error rates for both, or lower error rates for both, with the difference between the two primitives being smaller than the micro-architecture induced one.

We can thus conclude that AMD single-socket systems are susceptible to our 3 attack primitives (**RQ1**). In particular, AMD susceptibility to Flush+Flush is a new finding that goes

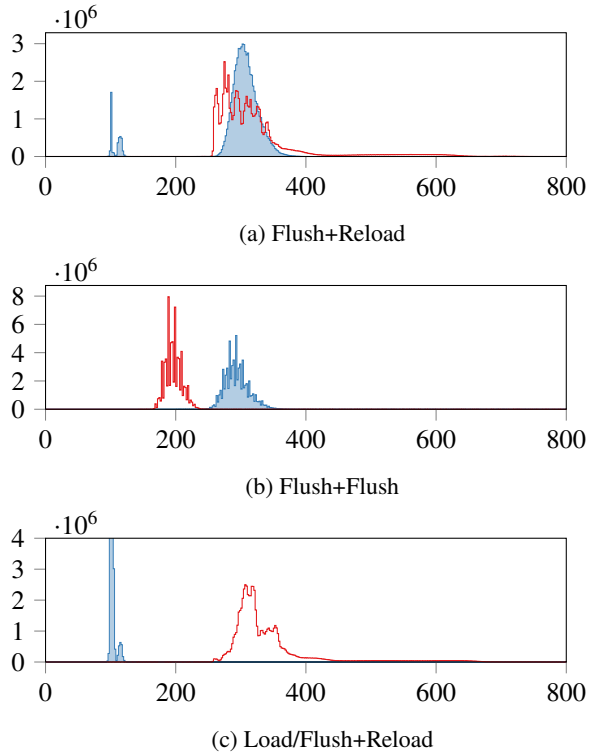
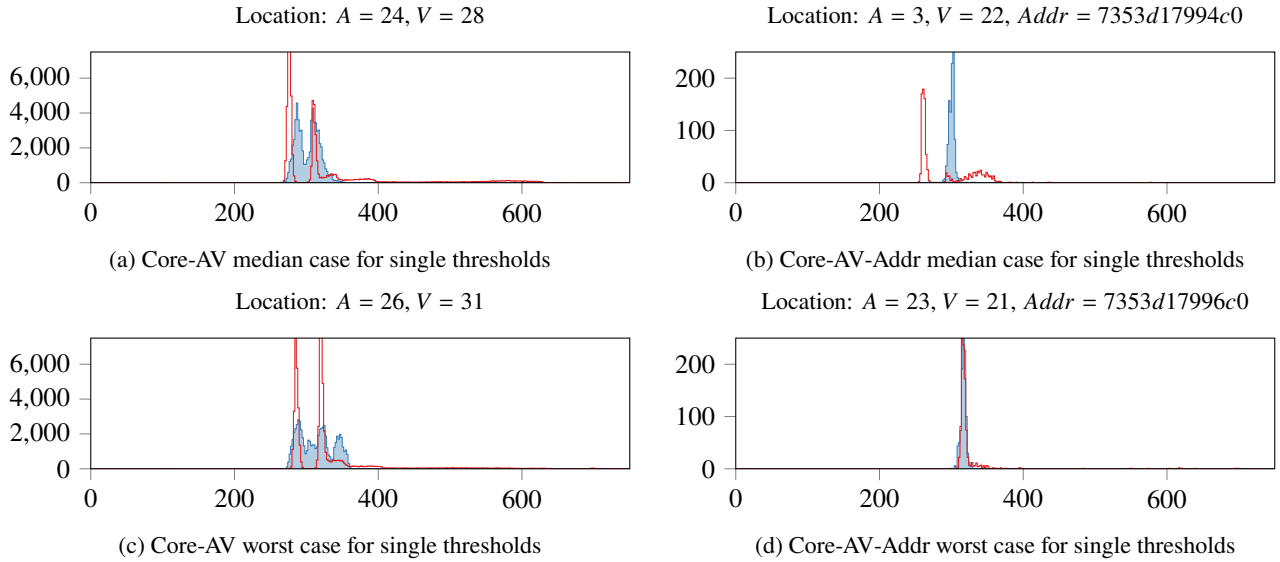


Figure 5: Flush+Reload, Flush+Flush and Load/Flush+Reload TU histograms, on EMR, (1× Emerald Rapids).

Table 4: Error rates (%) for EMR — 1× Emerald Rapids — Fixed Frequency, No Prefetcher. When results differ, -ST and -DT distinguish (single) thresholds and ranges.

Model	Avg.	Min.	Q1	Med.	Q3	Max.
TU-FF	0.05	< 0.01	< 0.01	< 0.01	< 0.01	50.05
TU-FR-ST	41.34	9.13	25.54	38.82	50.00	100.00
TU-FR-DT	33.67	1.27	18.80	29.88	42.24	99.80
TU-LFR	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	2.44
Core-AV-FF	0.02	< 0.01	< 0.01	< 0.01	< 0.01	45.90
Core-AV-FR-DT	19.74	< 0.01	6.93	16.89	28.12	90.09
Core-AV-FF-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-FR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.05
Core-AV-LFR-S-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-Addr-FF-ST	0.01	< 0.01	< 0.01	< 0.01	< 0.01	23.78
Core-AV-Addr-FF-DT	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	3.12
Core-AV-Addr-FR-DT	7.79	< 0.01	1.66	5.86	11.91	43.41
Core-AV-Addr-FR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01



(a) Core-AV median case for single thresholds

(b) Core-AV-Addr median case for single thresholds

(c) Core-AV worst case for single thresholds

(d) Core-AV-Addr worst case for single thresholds

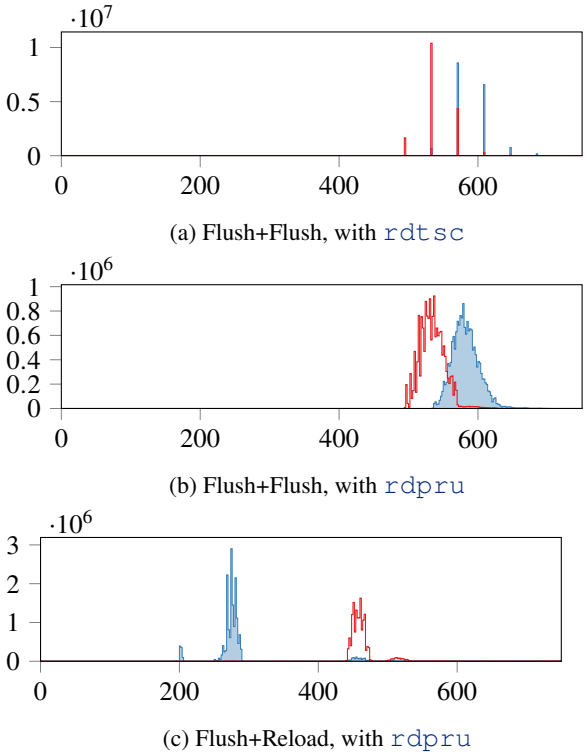
Figure 6: Flush+Reload worst and median case histograms on `EMR`, (1× Emerald Rapids), for two attacker models.

Table 5: Error rates (%) for `Zen5` — 1× Zen 5 (Granite Ridge) — Fixed Frequency, No Prefetcher, Single Threshold

Model	Avg.	Min.	Q1	Med.	Q3	Max.
TU-FF	8.67	< 0.01	< 0.01	0.05	0.63	50.34
TU-FR	3.19	< 0.01	< 0.01	0.10	0.20	50.10
TU-LFR	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.15
Core-AV-FF	5.54	< 0.01	< 0.01	0.05	0.20	50.24
Core-AV-FR	3.14	< 0.01	< 0.01	0.10	0.20	61.04
Addr-FF	0.98	< 0.01	< 0.01	< 0.01	0.05	43.21
Addr-FR	3.19	< 0.01	< 0.01	0.10	0.20	50.10
Core-AV-FF-Best	0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.15
Core-AV-FR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.10
Core-AV-LFR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-Addr-FF	0.38	< 0.01	< 0.01	< 0.01	0.05	27.59
Core-AV-Addr-FR	3.01	< 0.01	< 0.01	0.10	0.20	49.76
Core-AV-Addr-FF-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-Addr-FR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01

contrary to what was usually assumed. Interestingly, Flush+Flush sees a much larger improvement with topology awareness than Flush+Reload, and it goes from being noisier to being significantly more accurate. There is thus a significant benefit to topology awareness (**RQ2**).

Overall, we can conclude that all three primitives are applicable on modern x86 single-socket systems (**RQ1**), and that topology awareness can often bring benefits, with variations across micro-architectures (**RQ2**).



(a) Flush+Flush, with `rdtsc`

(b) Flush+Flush, with `rdpru`

(c) Flush+Reload, with `rdpru`

Figure 7: Flush+Flush and Flush+Reload TU histograms on `Zen5` (1× Granite Ridge)

Multi-socket systems: Moving on to multi-socket systems, we first verify assumptions about the symmetries of the system used in the covert channel benchmark, and then discuss results for two-socket Intel and AMD systems and a four-socket Intel system. Table 12 and our online supplement (*cf.* Appendix C) present results on 23 additional such machines.

4.4 Are Numa Systems Symmetric?

One key assumption for Experiment 2, the Covert Channel Benchmark, is that the system behavior is symmetric with respect to Numa nodes, *i.e.*, swapping all Numa-nodes (A, V, and M) leaves the behavior unchanged.

We illustrate this with Figure 2, showing histograms on a dual-socket Sapphire Rapids system (2023). We plot both the topology-unaware histogram (Figure 2a) and the 8 Numa-AVM histograms (Figure 2b). The topology-unaware histogram exhibits quite a complex behavior, with several large and wide humps. This behavior is not surprising: it is expected for Numa to cause large variations of latencies for memory operations. In Figure 2b, we see that changing the memory Numa-node (M) doesn't change reload hit timings, as expected for cache hits. Meanwhile, it is a major source of variability for misses: e.g., with $A = V$, a reload is much slower if $A \neq M$ than when $A = M = V$, which must thus be accounted for in attacks. As such, the behavior is complex, makes topology-unaware classifiers ineffective, and motivates our topology-aware approach.

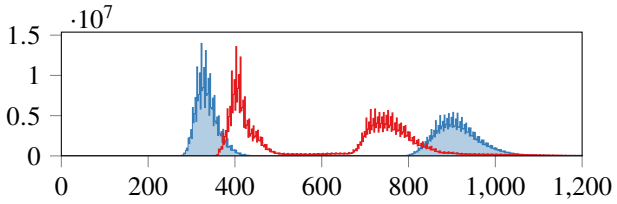
As seen in Figure 2b, exchanging all Numa nodes, e.g., from $A = 0, V = 1, M = 1$ to $A = 1, V = 0, M = 0$, results in similar histograms. It is thus possible to explore all behaviors of the system while considering only $M = 0$, which Section 5 relies on to accelerate the run-time of the experiments.

The hump width, and the overlap between hits and misses, notably for $(A, V, M) \in \{(0, 1, 0), (1, 0, 1)\}$, hampers Flush+Reload attacks, and motivates a fine-grained topology approach, *i.e.*, one considering more than simply Numa-nodes.

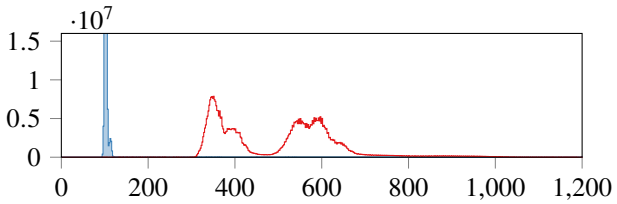
4.5 Calibration of Two-socket Intel Systems

We have already seen Flush+Reload histograms for `esterel41` in Figure 2. Figure 8 presents, as a complement, the Flush+Flush and Load/Flush+Reload topology-unaware histograms. Table 6 shows the resulting error rates under different attacker models. While all three primitives' error rates can be reduced satisfyingly under the Numa-M-Core-AV-Best attacker model, Flush+Flush is usually slightly better than Flush+Reload. Models without choice remain exposed to excessively noisy worst cases. Load/Flush+Reload, unsurprisingly, is much more reliable.

Thus, we conclude that on Intel two socket systems all three primitives are applicable (**RQ1**), and that topology awareness leads to significant improvements (**RQ2**)



(a) Flush+Flush



(b) Load/Flush+Reload

Figure 8: Flush+Flush and Load/Flush+Reload TU histograms on `esterel41` (2× Sapphire Rapids)

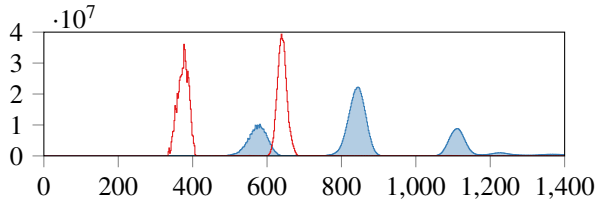
Table 6: Error rates (%) for `esterel41` — 2× Sapphire Rapids — Fixed Frequency, No Prefetcher

Model	Avg.	Min.	Q1	Med.	Q3	Max.
TU-FF-DT	7.69	< 0.01	0.05	1.27	11.91	100.00
TU-FR-DT	31.86	< 0.01	6.05	33.40	50.00	100.00
TU-LFR-ST	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	23.05
Numa-M-Core-AV-FF-ST	3.80	< 0.01	< 0.01	< 0.01	2.25	78.71
Numa-M-Core-AV-FR-ST	16.48	< 0.01	< 0.01	1.51	35.60	100.00
Numa-M-Core-AV-LFR-ST	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	6.98
Numa-M-Core-AV-FF-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Numa-M-Core-AV-FR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Numa-M-Core-AV-LFR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Numa-M-Core-AV-Addr-FF	1.53	< 0.01	< 0.01	< 0.01	0.15	41.06
Numa-M-Core-AV-Addr-FR	6.57	< 0.01	< 0.01	0.49	8.94	47.75

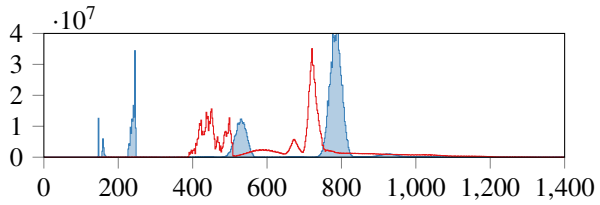
4.6 Calibration of Two-socket AMD Systems

For multi-socket AMD systems, Figure 9 shows the three topology-unaware histograms and Table 7 shows a subset of the error predictions, where the parameters make a difference. Load/Flush+Reload is the most accurate, unsurprisingly. Topology awareness improves significantly the accuracy of Flush+Reload and Flush+Flush, with the average error rate respectively going from 18% and 12% in topology unaware, down to 2.1% and 0.12% with Numa-M-Core-AV, and 0.9% and 0.03% under Numa-M-Core-AV. Adding the ability to choose, with the Numa-M-Core-AV-Best model, is enough to make the error rate negligible. Here, ranges (-DT) are also more accurate than thresholds (-ST) for Flush+Reload.

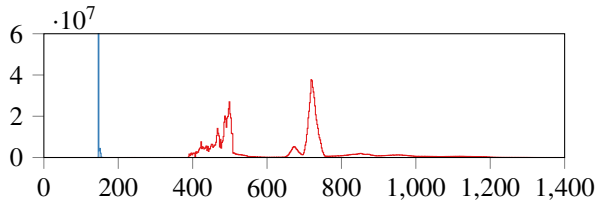
We conclude that Flush+Flush works on multi-socket AMD systems, and is worth using instead of Flush+Reload, while Load/Flush+Reload is the best covert channel (**RQ1**). Topology awareness is, again, needed for accurate attacks (**RQ2**).



(a) Flush+Flush, with `rdpru`



(b) Flush+Reload, with `rdpru`



(c) Load/Flush+Reload, with `rdpru`

Figure 9: Flush+Flush and Load/Flush+Reload TU histograms on `musa` (2× Zen 4 (Genoa))

4.7 Calibration of a Four-socket Intel System

We also tested the 4× Skylake SP `yeti` machine. The results were overall similar to the two-socket machines, with however an extra bump in the topology-unaware histogram, *cf.* Figure 10. Those extra modes arise from the fact that the victim V, attacker A, and target memory controller M can be in distinct sockets, which is not possible with only two sockets. This complex behavior underlines the need for a topology-aware approach, as distinguishing all the topology configurations gives histograms that are far more tractable for classification.

The overall conclusion does not change either, all three primitives are applicable (**RQ1**), with topology awareness improving significantly the attack's accuracy **RQ2**.

Conclusion On both single and multi-socket systems, we have demonstrated the applicability of Flush+Reload, Flush+Flush, and Load/Flush+Reload (**RQ1**). Depending on the micro-architecture, and the attacker model, there can be variation about which of Flush+Flush and Flush+Reload is the most accurate. However, Flush+Flush often ends up having an edge over Flush+Reload, while Load/Flush+Reload covert channels are always the most accurate.

We also observe that overall topology-aware approaches usually manage to obtain negligible error rates for at least one of Flush+Reload or Flush+Flush, and generally provide significant accuracy improvements (**RQ2**).

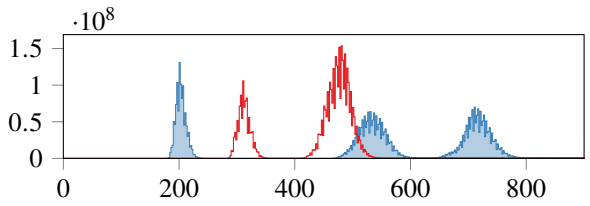
Table 7: Error rates (%) for `musa` — 2× Zen 4 (Genoa) — Fixed Frequency, No Prefetcher

Model	Avg.	Min.	Q1	Med.	Q3	Max.
TU-FF	12.66	< 0.01	< 0.01	< 0.01	49.85	51.51
TU-FR-DT	18.01	< 0.01	6.25	8.89	22.31	90.48
TU-LFR	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.34
NUMA-M-Core-AV-FF-ST	0.12	< 0.01	< 0.01	< 0.01	< 0.01	50.05
NUMA-M-Core-AV-FR-ST	7.61	< 0.01	4.64	6.79	9.18	87.55
NUMA-M-Core-AV-FR-DT	2.10	< 0.01	0.20	0.68	1.95	68.80
NUMA-M-Core-AV-FF-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
NUMA-M-Core-AV-FR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
NUMA-M-Core-AV-LFR-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
NUMA-M-Core-AV-Addr-FF-ST	0.03	< 0.01	< 0.01	< 0.01	< 0.01	49.56
NUMA-M-Core-AV-Addr-FR-ST	6.55	< 0.01	3.86	6.54	8.94	48.39
NUMA-M-Core-AV-Addr-FR-DT	0.93	< 0.01	0.05	0.39	0.93	47.36

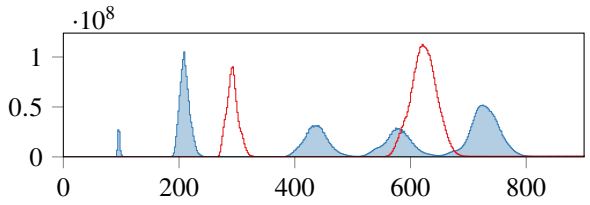
5 Results for Experiment 2: Covert Channel

In this section, we answer **RQ3**, "What covert-channel true-capacity can be achieved using the best trade-off between accuracy and attack speed?". We ran our generic covert channel benchmark on three different attacker models, topology-unaware **TU**, topology-aware **TA** (NUMA-M-Core-AV-Addr), and the best choice topology-aware **Best-TA** (NUMA-M-Core-AV-Addr-Best). We report in Appendix D.2 the full results, including standard deviations, for the five machines we discuss here. For each, we report p, the error rate (lower is better), C, the bandwidth, and T, the *true capacity*, *cf.* Section 3.4.

Table 8 presents the results on the single socket `Zen5` system. Flush+Flush displays a steady decrease in error rate using more detailed models. However, Flush+Reload and Load/Flush+Reload have error rates below the standard deviation, and see an uptick in the TA model. To determine



(a) Flush+Flush



(b) Flush+Reload

Figure 10: Flush+Reload and Flush+Flush TU histograms for `yeti` (4× Skylake SP)

Table 8: Covert Channel Results for `Zen5` — 1× Zen 5 (Granite Ridge) — Fixed Frequency, No Prefetcher

Primitive	Model	p (%)	C (Mbit/s)	T (Mbit/s)	σp (%)	σC (Mbit/s)	σT (Mbit/s)
FF	TU	24.58	2.22	0.46	5.19	0.03	0.17
	TA	19.73	2.21	0.64	2.71	0.05	0.16
	Best-TA	0.67	2.07	1.95	0.00	0.00	0.00
FR	TU	0.56	2.30	2.23	2.24	0.02	0.24
	TA	1.12	2.29	2.13	2.04	0.02	0.26
	Best-TA	0.00	2.27	2.27	0.00	0.00	0.00
LFR	TU	0.04	4.89	4.87	0.02	0.10	0.11
	TA	0.05	4.89	4.86	0.15	0.09	0.10
	Best-TA	0.00	4.98	4.98	0.00	0.00	0.00

whether this is a spurious uptick or if this is a real effect, more samples would be required.

Overall here, we see that Flush+Reload performs better than Flush+Flush, and that Load/Flush+Reload performs much better than either. A Load/Flush+Reload channel has an 4.9 Mbit/s bandwidth, with only marginal improvement from topology awareness. Meanwhile, topology awareness improves significantly Flush+Flush, which sees its bandwidth go from 0.46 Mbit/s to 1.95 Mbit/s, thanks to a division by 36 of the error rate.

Table 9 shows a sample of results from the 4 other machines in Appendix D.2. It shows that Load/Flush+Reload is undoubtedly the best covert channel, and that Flush+Reload and Flush+Flush benefit from topology awareness.

More generally, we observe on our data set that Load/Flush+Reload has a much higher bandwidth (> 1.5×) than Flush+Flush and Flush+Reload. It usually does not require topology awareness, while such awareness can improve Flush+Reload and Flush+Flush significantly.

On AMD machines, Flush+Flush is usually worse than Flush+Reload, while on Intel machines Flush+Flush performs better. The observed error rates tend to be above the prediction, which is an optimistic bound, but do vary in a consistent way overall. Machines with very high predicted error rates also have high error rates (and even impossible to exploit topology-unaware Flush+Reload for `EMR`), while machines where we expect low error rates produce low error rates (e.g., `ARL`).

It is worth noting that the lowest error rates reported tend to be below the uncertainty of the measure, given the number of samples we have, which sometimes explain a slight increase in error rate when increasing the model quality.

Overall, this validates that Load/Flush+Reload is an effective covert channel, with speeds around 5 Mbit/s on `ARL` and `Zen5`, and above 2.5 Mbit/s on multi-socket systems. For the other attack primitives, topology awareness can improve the accuracy, but may reduce the raw bandwidth. Consequently,

Table 9: Other results for covert channels

Machine	Attack	Attacker Model	p (%)	C (Mbit/s)	T (Mbit/s)
`ARL` Arrow Lake	FF	TU	0.00	3.20	3.20
	FR	TU	0.01	2.73	2.73
	LFR	TU	0.01	5.43	5.42
`EMR` Emerald Rapids	FF	TU	0.06	2.07	2.06
	FR	TU	55.47	1.68	−−
	FR	TA	34.97	1.74	0.42
	FR	Best-TA	0.00	2.20	2.20
	LFR	TU	0.19	3.28	3.22
`esterel41` 2× Sapphire Rapids	FF	TU	26.67	1.32	0.65
	FF	Best-TA	0.67	2.07	1.95
	FR	TU	30.89	1.47	0.51
	FR	Best-TA	0.00	2.27	2.27
	LFR	TU	0.19	2.54	2.49
`musa` 2× Zen 4 (Genoa)	FF	TU	43.67	1.13	0.03
	FF	Best-TA	42.33	1.88	0.03
	FR	TU	36.85	1.11	0.13
	FR	Best-TA	0.00	1.88	1.88
	LFR	TU	0.20	2.13	2.08
	LFR	Best-TA	0.00	2.92	2.92

depending on the machine, the tradeoff may differ. Some level of topology awareness can be needed to make attack viable, but the model with the best predicted error rate isn't necessarily the one with the highest true capacity.

One further source of noise, is that the transmission uses several cache lines in parallel, to maximize the bandwidth. This could interfere with the individual latencies, and contribute to the degradation in accuracy.

We can conclude regarding **RQ3** that the tradeoffs vary depending on the system, but that it is usually possible to achieve a bandwidth in the Mbit/s range with at least one of Flush+Flush and Flush+Reload with suitable topology awareness, and that Load/Flush+Reload will usually be significantly better than either of these, by a factor of ×1.5 or even ×2.

6 Discussion

6.1 Limitations

While our study spans a wide range of architectures, several limitations remain. First, the measured covert-channel bandwidths do not always match our timing-based predictions, possibly due to limited sampling in some configurations. Additional measurements would be needed to refine error-rate

estimates. Second, our calibration process tends to keep a DRAM row open, which may lower observed miss latencies. It also executes on a system with very little memory traffic, which results in less accurate results under higher memory traffic settings, such as our multi-cache line covert channel. Future work should develop sampling methods that mitigate these biases. Finally, the applicability of topology-aware calibration must be considered in the context of the attacker model. In native scenarios, an attacker can often control its own NUMA node or core placement and influence the victim through the scheduler. In contrast, such control may be limited or unavailable in other threat models, including virtualized environments, cloud systems, or web browsers.

6.2 Related Work

Didier and Maurice [5] introduced a slice-aware calibration method for the Flush+Flush attack on Intel client CPUs. Their work observed that the execution time of `clflush` depends on the targeted cache slice of the LLC, which in Intel architectures is partitioned into per-core slices. This slice is determined by hashing the physical address, which is not accessible to the attacker. When the hashing function is linear, *i.e.*, when the number of slices is a power of two [13], within each virtual page, the equivalence class of addresses that map to the same slice is easily accessible, which is used by Didier and Maurice [5]. This approach was practical on Intel CPUs, where the hashing function is linear, but less so on architectures where this assumption does not hold, such as AMD systems with undocumented slice mappings.

In the case of client Intel CPUs, Didier and Maurice's topology-aware method [5] reduces the error rate of Flush+Flush to less than 0.01%, from 20% previously, while also tripling its bandwidth. Their findings highlight the importance of incorporating architectural and topological factors into attack calibration, a principle we extend and generalize in our work to multi-socket and AMD systems. We cover generalize to a much larger range of micro-architecture, instead of only Intel client CPUs whose number of core is a power of two, and also account for the extra factors induced by NUMA in multi-socket system.

6.3 Future Work

In some cases, using a range instead of a threshold to classify attack outcomes leads to better accuracy. Consequently, there could be room to evaluate whether more elaborate classification strategies than a single threshold are relevant. Rauscher et al. [18] introduced metrics to quantify the quality of cache attack primitives, however, these metrics are only suitable for topology-unaware approaches, using a single global threshold. Future work could adapt such metrics to more complex classification strategies and apply them to complex systems where topology awareness is required. Our data also show complex behavior of the cache hierarchy on AMD CPUs, in particular those with multiple compute dies. There is thus room for further reverse engineering of AMD's cache hierarchy and topology, as it is not as well known as Intel's [1, 4].

This work focused on x86 architectures, however, complex CPUs using the ARM and RISC-V ISAs have started shipping. As such, a similar assessment of the benefit of the topology awareness on attacks would be valuable, applying to Evict+Reload and Prime+Probe. On RISC-V [2], it has been shown Flush+Flush and Flush+Reload could also be applicable on CPUs implementing the *Zicbom* extension.

Additionally, it would also be worth checking how Evict+Reload, Prime+Probe and attacks based on `prefetchw` [9] behave on more recent and multi-socket systems. While it would be unexpected for Prime+Probe to be effective in systems that share neither a last-level cache nor a cache-coherence directory, such as multi-socket systems, this assumption should be verified.

7 Conclusion and Further Work

We have evaluated the performance of Flush+Flush and Flush+Reload on different 36 systems, both from AMD and Intel, including a large spread of micro-architectures, and in particular recent ones. We showed that Flush+Flush and Flush+Reload were widely applicable to all these machines, including multi-socket and AMD systems, but that accurate attacks may require topology awareness. In particular, on NUMA-system, NUMA-node rebalancing must be avoided to get reliable attacks, and NUMA-awareness is critical to accurate attacks. In addition, we presented Load/Flush+Reload, a new covert-channel primitive that achieves high accuracy and 1.5× true capacity improvement over Flush+Reload and Flush+Flush. Overall, topology awareness is essential for reliable cache attacks on contemporary x86 architectures.

Acknowledgments

Multi-socket experiments used the Grid'5000 test-bed [3], supported by a scientific interest group hosted by Inria, including CNRS, RENATER, several universities, and other organizations (*c.f.* `https://www.grid5000.fr`).

This project started at IRISA, with support from the French DGA, and was then pursued at Saarland University. This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020415).

We would also like to thank Stefan GAST for the measurements ont the `lab32` (Zen4c) machine, and Jan REINEKE for his advice and thorough feedback.

References

[1] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2020.

[2] Cédrick Austa, Jan Tobias Mühlberg, and Jean-Michel Dricot. Systematic assessment of cache timing vulnerabilities on RISC-V processors. In *30th European Symposium on Research in Computer Security (ESORICS)*, 2025.

[3] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In *Communications in Computer and Information Science*. 2013.

[4] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects. In *31st USENIX Security Symposium*, 2022.

[5] Guillaume Didier and Clémentine Maurice. Calibration Done Right: Noiseless Flush+Flush Attacks. In *DIMVA*, 2021.

[6] Guillaume Didier, Clémentine Maurice, Antoine Geimer, and Walid J. Ghandour. Characterizing Prefetchers using CacheObserver. In *SBAC-PAD*, 2022.

[7] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: exploiting the scheduler queue contention side channel. In *44th IEEE Symposium on Security and Privacy, SP*, 2023.

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

[9] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (S&P)*, 2022.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 6th edition edition, 2019.

[11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2025.

[12] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD prefetch attacks through power and time. In *31st USENIX Security Symposium, USENIX Security*, 2022.

[13] 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.

[14] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2nd edition edition, 2020.

[15] Hamed Okhravi, Stanley Bak, and Samuel T. King. Design, implementation and evaluation of covert channel attacks. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, 2010.

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

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

[18] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A systematic evaluation of novel and existing cache side channels. In *NDSS*. The Internet Society, 2025.

[19] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. Port contention goes portable: Port contention side channels in web browsers. In *ASIA CCS*, 2022.

[20] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys*, 2021.

[21] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[22] Yuval Yarom and Katrina Falkner. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium*, 2014.

A Artifact

We provide, for review purposes, an anonymized version of our experimental code on Anonymous 4 Open Science[7].

B List of All Machines

Table 11 lists the 36 machines we used in this study.

C Detailed Results

Table 12 includes a summary of the average predicted error rates under the TU, TA and Best-TA attacker models, for all three attack primitives, using simple thresholds.

We also provide an online supplement, available for review on Zenodo via this link[8], which contains the full results, for calibration and covert-channel benchmark, under all attacker models, for all 36 machines, and with both fixed frequency and variable frequency settings.

We also intend to make available the experimental results files (> 10 GiB) and the rust library needed to deserialize them.

D Additional Results

D.1 Single-socket Zen 2 (Matisse)

To further illustration Section 4.3, we include in Table 10 results on a second client, single die, single-socket AMD system.

D.2 Covert Channel Results

We include the full table of results for the 5 machines discussed in Section 5, as Tables 13 to 17. Further results for the whole data set of 36 can be found in the online supplement (*cf.* Appendix C).

Table 10: Error rates (%) for `Zen2` — 1× Zen 2 (Matisse) — Fixed Frequency, No Prefetcher

Model	Avg.	Min.	Q1	Med.	Q3	Max.
TU-FF	0.74	< 0.01	< 0.01	< 0.01	1.12	12.55
TU-FR	16.97	< 0.01	0.05	2.44	38.28	84.33
TU-LFR	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.05
Core-AV-FF	0.52	< 0.01	< 0.01	< 0.01	0.24	9.47
Core-AV-FR	12.79	< 0.01	< 0.01	0.05	21.04	84.72
Addr-FF	0.61	< 0.01	< 0.01	< 0.01	0.83	11.18
Addr-FR-ST	10.81	< 0.01	< 0.01	0.10	3.52	52.64
Core-AV-FF-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.15
Core-AV-FR-Best-ST	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-LFR-Best-ST	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Core-AV-Addr-FF-DT	0.09	< 0.01	< 0.01	< 0.01	< 0.01	4.10
Core-AV-Addr-FR-DT	5.07	< 0.01	< 0.01	< 0.01	1.12	48.44
Core-AV-Addr-FF-Best	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01

[7] https://anonymous.4open.science/r/flush-based-cache-attacks-modern-x86-0DD2/

[8] https://zenodo.org/records/17485253?token=eyJhbGciOiJIUzUxMiIsImlhdCI6MTc2MTgzNDEyOCwiZXhwIjoxNzcyMzIzMTk5fQ.eyJpZCI6IjVhMDAyMmVkLTA5ZmEtNDlmZi04MGY4LTcyYjQ3NGVkYTEzZSIsImRhdGEiOnt9LCJyYW5kb20iOiI3NTQ3NWRmN2JjNDQ4ZmJmYjE0NzUyMmQ1OTU1YzE4NyJ9.Sjz0pZfNR69wbthozAlBlDiI18t9dEGSv5Pqd3jU9t19EsuyijdiPs7EGM89V8L-0DiJoSrHwN3c02xoBRd3jw

Table 11: Machines used in this work

Name	Processors	μ-arch	N	C/T	OS
abacus25	2× AMD EPYC 7413	Zen 3 (Milan)	2	2× 24/48	Deb. 11
chiclet	2× AMD EPYC 7301	Zen 1 (Naples)	2	2× 16/32	Deb. 11
chifflot	2× Intel Xeon Gold 6126	Skylake SP	2	2× 12/24	Deb. 11
chirop	2× Intel Xeon Platinum 8358	Ice Lake SP	2	2× 32/64	Deb. 11
dahu	2× Intel Xeon Gold 6130	Skylake SP	2	2× 16/32	Deb. 11
econome	2× Intel Xeon E5-2660	Sandy Bridge EP	2	2× 8/16	Deb. 11
ecotype	2× Intel Xeon E5-2630L v4	Broadwell EP	2	2× 10/20	Deb. 11
esterel41	2× Intel Xeon Gold 6426Y	Sapphire Rapids	2	2× 16/32	Deb. 11
graffiti	2× Intel Xeon Silver 4110	Skylake SP	2	2× 8/16	Deb. 11
grappe	2× Intel Xeon Gold 5218R	Cascade Lake SP	2	2× 20/40	Deb. 11
grele	2× Intel Xeon E5-2650 v4	Broadwell EP	2	2× 12/24	Deb. 11
grue	2× AMD EPYC 7351	Zen 1 (Naples)	8	2× 16/32	Deb. 11
kinovis	2× Intel Xeon Gold 6442Y	Sapphire Rapids	2	2× 24/48	Deb. 11
mercantour2	2× Intel Xeon E5-2650 v2	Ivy Bridge EP	2	2× 8/16	Deb. 11
montcalm	2× Intel Xeon Silver 4314	Ice Lake SP	2	2× 16/32	Deb. 11
musa	2× AMD EPYC 9254	Zen 4 (Genoa)	2	2× 24/48	Deb. 11
nova	2× Intel Xeon E5-2620 v4	Broadwell EP	2	2× 8/16	Deb. 11
paradoxe	2× Intel Xeon Gold 5320	Ice Lake SP	2	2× 26/52	Deb. 11
parasilo	2× Intel Xeon E5-2630 v3	Haswell EP	2	2× 8/16	Deb. 11
petitprince	2× Intel Xeon E5-2630L	Sandy Bridge EP	2	2× 6/12	Deb. 11
roazhon5	2× Intel Xeon E5-2660 v3	Haswell EP	2	2× 10/20	Deb. 11
sagittaire	2× AMD Opteron 250	K8	2	2× 1/1	Deb. 11
servan	2× AMD EPYC 7352	Zen 2 (Rome)	2	2× 24/48	Deb. 11
troll	2× Intel Xeon Gold 5218	Cascade Lake SP	2	2× 16/32	Deb. 11
uvb	2× Intel Xeon X5670	Westmere EP	2	2× 6/12	Deb. 11
yeti	4× Intel Xeon Gold 6130	Skylake SP	4	4× 16/32	Deb. 11
ARL	Intel Core Ultra 7 265K	Arrow Lake	1	8/8P+12/12E	Ub. 24.10
CFL	Intel Core i7-8700K	Coffee Lake	1	6/12	Ub. 24.04
EMR	Intel Xeon Silver 4514Y	Emerald Rapids	1	16/32	Ub. 24.10
lab32	AMD EPYC 8024P	Zen 4c (Sienna)	1	8/16 (2× 4/8)	Ub. 22.04
Ripper	AMD Threadripper 5995WX	Zen 3 (Chagall)	1	64/128 (8× 8/16)	Ub. 24.04
Zen2	AMD Ryzen 7 3700X	Zen 2 (Matisse)	1	8/16	Ub. 20.04
Zen3	AMD Ryzen 5 5600X	Zen 3 (Vermeer)	1	6/12	Ub. 20.04
Zen4	AMD Ryzen 5 7600X	Zen 4 (Raphael)	1	6/12	Ub. 22.04
Zen5	AMD Ryzen 7 9700X	Zen 5 (Granite Ridge)	1	8/16	Ub. 24.10
ZenP	AMD Ryzen 5 2600	Zen+ (Pinnacle Ridge)	1	6/12	Ub. 18.04

Ripper and lab32 are multi-die modules, whose number of dies and of Core/Threads per die are indicated between brackets after te total number of Core/Thread in the system.

Table 12: Summary of results for all 36 machines, with fixed frequency and single-threshold classifier

Machine	Flush+Flush			Flush+Reload			Load/Flush+Reload		
	TU	TA	Best-TA	TU	TA	Best-TA	TU	TA	Best-TA
abacus25	15.74	0.39	< 0.01	22.54	3.96	< 0.01	< 0.01	< 0.01	< 0.01
chiclet	6.02	< 0.01	< 0.01	16.69	0.86	< 0.01	< 0.01	< 0.01	< 0.01
chifflot	25.00	1.00	< 0.01	25.01	6.10	< 0.01	< 0.01	< 0.01	< 0.01
chirop	25.01	1.11	< 0.01	27.71	2.72	< 0.01	< 0.01	< 0.01	< 0.01
dahu	25.00	1.06	< 0.01	25.00	7.39	< 0.01	< 0.01	< 0.01	< 0.01
econome	25.00	< 0.01	< 0.01	24.35	0.29	< 0.01	< 0.01	< 0.01	< 0.01
ecotype	25.01	2.16	< 0.01	25.02	11.16	< 0.01	< 0.01	< 0.01	< 0.01
esterel41	27.55	1.53	< 0.01	34.15	6.57	< 0.01	< 0.01	< 0.01	< 0.01
graffiti	25.00	1.26	< 0.01	25.01	8.56	< 0.01	< 0.01	< 0.01	< 0.01
grappe	25.00	1.15	< 0.01	25.00	2.07	< 0.01	< 0.01	< 0.01	< 0.01
grele	25.00	1.61	< 0.01	25.04	11.55	< 0.01	< 0.01	< 0.01	< 0.01
grue	6.17	< 0.01	< 0.01	16.62	0.93	< 0.01	< 0.01	< 0.01	< 0.01
kinovis	27.66	2.29	< 0.01	34.40	8.08	< 0.01	< 0.01	< 0.01	< 0.01
mercantour2	25.00	11.56	< 0.01	24.02	10.86	< 0.01	< 0.01	< 0.01	< 0.01
montcalm	25.00	1.19	< 0.01	25.52	0.99	< 0.01	< 0.01	< 0.01	< 0.01
musa	12.66	0.03	< 0.01	24.25	6.55	< 0.01	< 0.01	< 0.01	< 0.01
nova	25.00	1.89	< 0.01	25.01	10.05	< 0.01	< 0.01	< 0.01	< 0.01
paradoxe	25.00	1.14	< 0.01	25.62	0.64	< 0.01	< 0.01	< 0.01	< 0.01
parasilo	25.00	< 0.01	< 0.01	24.18	0.31	< 0.01	< 0.01	< 0.01	< 0.01
petitprince	25.00	< 0.01	< 0.01	20.29	1.44	< 0.01	< 0.01	< 0.01	< 0.01
roazhon5	25.00	< 0.01	< 0.01	25.00	0.78	< 0.01	< 0.01	< 0.01	< 0.01
sagittaire	25.00	24.98	< 0.01	38.90	37.37	3.91	< 0.01	< 0.01	< 0.01
servan	35.65	6.09	< 0.01	37.96	9.48	< 0.01	< 0.01	< 0.01	< 0.01
troll	25.00	1.27	< 0.01	25.00	2.32	< 0.01	< 0.01	< 0.01	< 0.01
uvb	3.38	0.02	< 0.01	24.31	1.04	< 0.01	< 0.01	< 0.01	< 0.01
yeti	19.19	1.66	< 0.01	33.78	1.46	< 0.01	< 0.01	< 0.01	< 0.01
ARL	< 0.01	< 0.01	< 0.01	0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
CFL	18.73	14.46	< 0.01	0.04	5.12	< 0.01	< 0.01	< 0.01	< 0.01
EMR	0.05	0.02	< 0.01	41.34	0.01	< 0.01	< 0.01	< 0.01	< 0.01
lab32	0.75	< 0.01	< 0.01	25.16	1.96	< 0.01	< 0.01	< 0.01	< 0.01
Ripper	0.34	0.25	< 0.01	10.20	0.06	< 0.01	< 0.01	< 0.01	< 0.01
Zen2	0.74	0.52	< 0.01	16.97	0.26	< 0.01	< 0.01	< 0.01	< 0.01
Zen3	4.38	2.84	< 0.01	0.03	0.01	< 0.01	< 0.01	< 0.01	< 0.01
Zen4	0.87	0.32	0.01	0.08	0.04	< 0.01	< 0.01	< 0.01	< 0.01
Zen5	8.67	5.54	0.01	3.19	0.38	< 0.01	< 0.01	< 0.01	< 0.01
ZenP	10.69	7.79	0.25	28.95	7.51	< 0.01	< 0.01	< 0.01	< 0.01

Table 13: Covert Channel Results for `ARL` — 1× Arrow Lake — Fixed Frequency, No Prefetcher

Model	Primitive	p (%)	C (Mbit/s)	T (Mbit/s)	σp (%)	σC (Mbit/s)	σT (Mbit/s)
TU	FF	0.00	3.20	3.20	0.05	0.15	0.16
	FR	0.01	2.73	2.73	0.13	0.20	0.21
	LFR	0.01	5.43	5.42	0.14	0.65	0.66
TA	FF	0.24	3.20	3.16	1.12	0.15	0.24
	FR	0.19	2.75	2.72	0.93	0.20	0.25
	LFR	0.10	5.45	5.39	0.10	0.64	0.65
Best-TA	FF	0.00	3.58	3.58	0.00	0.00	0.00
	FR	0.00	2.64	2.64	0.00	0.00	0.00
	LFR	0.00	5.05	5.05	0.00	0.00	0.00

Table 14: Covert Channel Results for `EMR` — 1× Emerald Rapids — Fixed Frequency, No Prefetcher

Model	Primitive	p (%)	C (Mbit/s)	T (Mbit/s)	σp (%)	σC (Mbit/s)	σT (Mbit/s)
TU	FF	0.06	2.07	2.06	0.59	0.07	0.08
	FR	55.47	1.68	−−	9.54	0.10	−−
	LFR	0.19	3.28	3.22	0.08	0.19	0.20
TA	FF	0.64	2.07	1.99	1.79	0.07	0.21
	FR	34.97	1.74	0.42	21.97	0.09	0.51
	LFR	0.19	3.24	3.18	0.13	0.18	0.19
Best-TA	FF	0.05	2.03	2.02	0.00	0.00	0.00
	FR	0.00	2.20	2.20	0.00	0.00	0.00
	LFR	0.10	3.20	3.16	0.00	0.00	0.00

Table 15: Covert Channel Results for `Zen5` — 1× Zen 5 (Granite Ridge) — Fixed Frequency, No Prefetcher

Model	Primitive	p (%)	C (Mbit/s)	T (Mbit/s)	σp (%)	σC (Mbit/s)	σT (Mbit/s)
TU	FF	24.58	2.22	0.46	5.19	0.03	0.17
	FR	0.56	2.30	2.23	2.24	0.02	0.24
	LFR	0.04	4.89	4.87	0.02	0.10	0.11
TA	FF	19.73	2.21	0.64	2.71	0.05	0.16
	FR	1.12	2.29	2.13	2.04	0.02	0.26
	LFR	0.05	4.89	4.86	0.15	0.09	0.10
Best-TA	FF	0.67	2.07	1.95	0.00	0.00	0.00
	FR	0.00	2.27	2.27	0.00	0.00	0.00
	LFR	0.00	4.98	4.98	0.00	0.00	0.00

Table 16: Covert Channel Results for `esterel41` — 2× Sapphire Rapids — Fixed Frequency, No Prefetcher

Model	Primitive	p (%)	C (Mbit/s)	T (Mbit/s)	σp (%)	σC (Mbit/s)	σT (Mbit/s)
TU	FF	26.67	1.32	0.65	23.95	0.29	0.70
	FR	30.89	1.47	0.51	20.59	0.39	0.63
	LFR	0.19	2.54	2.49	0.08	0.81	0.80
TA	FF	5.53	1.32	1.05	10.22	0.30	0.47
	FR	23.34	1.50	0.70	21.42	0.39	0.73
	LFR	0.18	2.56	2.51	0.08	0.84	0.83
Best-TA	FF	0.00	1.28	1.28	0.00	0.00	0.00
	FR	0.00	2.40	2.40	0.00	0.00	0.00
	LFR	0.05	1.67	1.66	0.00	0.00	0.00

Table 17: Covert Channel Results for `musa` — 2× Zen 4 (Genoa) — Fixed Frequency, No Prefetcher

Model	Primitive	p (%)	C (Mbit/s)	T (Mbit/s)	σp (%)	σC (Mbit/s)	σT (Mbit/s)
TU	FF	43.67	1.13	0.03	7.09	0.29	0.05
	FR	36.85	1.11	0.13	15.84	0.28	0.17
	LFR	0.20	2.13	2.08	0.08	0.65	0.65
TA	FF	40.84	1.14	0.05	6.88	0.29	0.07
	FR	23.46	1.13	0.40	14.87	0.28	0.50
	LFR	0.19	2.13	2.09	0.15	0.66	0.66
Best-TA	FF	42.33	1.88	0.03	0.00	0.00	0.00
	FR	0.00	1.88	1.88	0.00	0.00	0.00
	LFR	0.00	2.92	2.92	0.00	0.00	0.00