Flushgeist: Cache Leaks from Beyond the Flush

Pepe Vila[1], Andreas Abel[2], Marco Guarnieri[1], Boris Köpf[3], and Jan Reineke[2]

[1]IMDEA Software Institute
[2]Saarland University
[3]Microsoft Research

Abstract

Flushing the cache, using instructions like `clflush` and `wbinvd`, is commonly proposed as a countermeasure against access-based cache attacks. In this report, we show that several Intel caches, specifically the L1 caches in some pre-Skylake processors and the L2 caches in some post-Broadwell processors, leak information even after being flushed through `clflush` and `wbinvd` instructions. That is, security-critical assumptions about the behavior of `clflush` and `wbinvd` instructions are incorrect, and countermeasures that rely on them should be revised.

1 Introduction

Caches are small, fast memories that bridge the latency gap between the CPU and the main memory. Caches are critical for performance: they speed up computation by storing recently accessed data and reducing the interaction with main memory [53].

Caches are also critical from a security perspective since they are often shared (both temporally and spatially) across security domains. This has inspired a multitude of covert [35,54] and side-channel attacks [5, 19, 38, 43, 55] in many different settings: web pages [15, 37, 50], OS processes [20, 21], mobile phones [18, 31], virtual machines [26, 34, 35], and trusted environments [6, 17, 36, 41, 57]. Caches also have a prominent role in recent transient-execution [30, 32, 44] and data-sampling [9, 40, 46] attacks, which use caches as high-bandwidth exfiltration channels.

Classic cache-based attacks, such as Prime+Probe [38] and Flush+Reload [55], leak information through the content of the cache, i.e., which memory blocks are accessed and stored in the cache. Researchers, however, also explored leaks through the so-called cache's *control state* [49], i.e., the cache's replacement policy metadata. Doychev et al. [12] show that the lookup table preloading countermeasure used in some AES implementations may leak information through the control state of a cache with a PLRU replacement policy. More recently, new attacks based on the cache's control state have been devised for high-bandwidth covert channels [54], bypassing Intel CAT's [24] isolation [29], and performing practical side-channel attacks in modern L3 caches [8].

To protect a victim program against access-based cache attackers (i.e., attackers that can interact with the shared cache before and after the victim's execution), it suffices preventing that the effects of the victim's execution on the cache cross the security domain boundary. Given the belief that cache flushing (using the `clflush` and `wbinvd` instructions) cleanses the cache from any history-dependent information, cache flushing on context switches has been thought to provide—at least on single-threaded cores—resistance against access-based attacks. As an example, several works [2–4, 7, 11, 16, 39, 48, 51, 58] propose using cache flushing (sometimes in combination with other countermeasures and optimizations) to prevent cache covert and side-channel attacks.

Cache flushing has also been used as part of mitigations against recent microarchitectural vulnerabilities like L1TF [44] or CacheOut [47]. For instance, Intel suggests that cache flushing (using `clflush` and `wbinvd` instructions) and finer-grained flushing instructions like `IA32_FLUSH_CMD` (supported by new microcode updates when the `L1D_FLUSH` processor flag is set) could be used to remove secrets from the L1D cache [22]. Furthermore, processors for which the `L1D_FLUSH` flag is enabled and which are affected by L1TF will automatically flush the L1D cache when executing the `RSM` instruction that exits System Management Mode (SMM). Similar mechanisms have been recommended for protecting virtual machines and SGX enclaves on context switches [22, 44].

Previous work by Ge et al. [13, 14] investigates the effectiveness of flushing operations and shows that information persists in some microarchitectural components (specifically in the instruction cache, branch target buffer, branch history table, and translation lookahead buffer) even after flushing. They also observe some *residual* leakage after flushing the data cache, and they associate it to the effects of data prefetchers.

In work concurrent to ours, Wistoff et al. [51] show that, on the Ariane RISC-V core [56], software solutions based on priming[1] are insufficient to mitigate cache covert channels. This is explained by the pseudo-random cache replacement policy implemented in Ariane.

We complement these results and demonstrate that cache flushing does *not* cleanse the cache from history-dependent information in several Intel processors. Even though cache flushing effectively eliminates leaks through the content of the cache (i.e., *which* memory blocks are stored in it), it does *not* prevent leaks through metadata, specifically through the state of the cache replacement policy (i.e., *how* memory blocks are accessed).

Our results imply that flushing instructions are insufficient to completely defeat access-based cache attacks in some Intel CPUs.

2 Cache Control States Survive Flushing

Prior work [1, 49] independently reports on nondeterministic behaviors, on several Intel processors, after flushing caches with `wbinvd` and `clflush` instructions. Specifically, nondeterministic behavior after flushing has been observed in the L1 caches of Sandy Bridge, Ivy Bridge, Haswell, and Broadwell CPUs, and in the L2 caches of Skylake, Kaby Lake, and Coffee Lake CPUs.

Our hypothesis is that the nondeterminism is due to the persistent control state of the cache replacement policy. Specifically:

- *the control state is not modified after a flush operation* whereas the lines in the cache are invalidated;
- *insertion of new blocks in the cache does not fully override the control state whenever there are invalid lines*.

If our hypothesis holds, then the common assumption that instructions like `clflush` and `wbinvd` cleanse all information from the cache is incorrect. This would also indicate that using flush operations to prevent access-based attacks is insufficient.

Our approach to validate the aforementioned hypothesis is the following: (1) First, we fill the cache set with associativity many known memory blocks $I_0 \dots I_{n-1}$; we call the resulting state s_0. (2) Then, we perform additional cache hits to bring the cache control state into an arbitrary known state s_i. (3) Finally, we invalidate the cache contents (by executing `wbinvd` or several `clflush` instructions) and refill the cache with different memory blocks $I'_0 \dots I'_{n-1}$; we call the resulting state s'_i. If our hypothesis is *true*, and the control state withstands flush operations, the control state s'_i will depend on the previous control state s_i.

[1] RISC-V cache management still lacks standard flushing mechanisms.

3 Validating the Hypothesis

In this section, we empirically evaluate our hypothesis that information from the cache's control state survives the flush operations on 11 different Intel processors from different generations.

We start by introducing the tools and setup of our evaluation. We continue by discussing two in-depth examples targeting the L1 PLRU cache from an i7-4790 CPU, and the L2 Quad-age LRU cache from an i5-6500. We conclude our evaluation by presenting a summary of all our findings.

3.1 Tools and Setup

In our evaluation, we use two tools to interact with caches: *CacheQuery* [49] and the *nanoBench Cache Analyzer* [1]. Both tools provide a clean interface and low-noise environment for probing caches, liberating the user from dealing with intricate details such as the virtual-to-physical memory mapping, cache slicing, set indexing, cache filtering, and other sources of interferences or measurement noise, thus enabling a "civilized" interaction with an individual cache set.

Namely, users can specify a cache set (say: set 63 in the L2 cache) and a pattern of memory accesses (say: "$I_0\ I_1\ I_2\ I_0\ I_1\ I_2$"), and they receive as output a sequence (say: `Miss Miss Miss Hit Hit Hit`) representing the hits and misses produced by a sequence of memory loads to addresses that are mapped into the specified cache set and that follow the specified pattern.

Example For instance, we can identify the Least Recently Used (LRU) block of a cache containing blocks "$I_0\ \dots\ I_7$" as follows: we cause an eviction by accessing "I_8" (a block not in the cache) and then check whether accessing block "I_j", where $0 \leq j \leq 7$, produces a hit or a miss. Note that one would need to reset the cache state each time and access "$I_8\ I_j$" for all $0 \leq j \leq 7$ to determine which block has been evicted. If accessing "$I_8\ I_1$" causes a cache miss for "I_1", this shows that "I_1" has been evicted by "I_8" and thus must have been the LRU block prior to the access to I_8.

3.2 Example 1: L1's Tree-based PLRU

As confirmed by prior research [1, 49], many Intel CPUs implement a tree-based Pseudo-LRU replacement policy in their L1 caches.

Replacement Policy Tree-based PLRU is a well known approximation of the LRU policy. It consists of a binary tree where each cache line is a leaf, and each internal node has a bit of control, where 0 represents an arrow pointing to the left child and 1 an arrow pointing to the right child. See Figure 1 for an example of a PLRU tree.

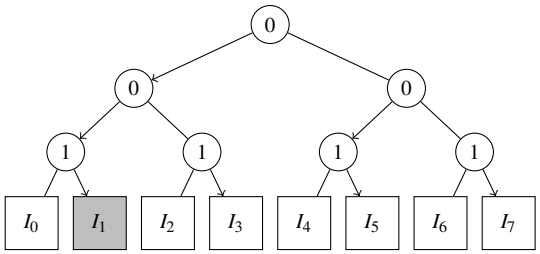


Figure 1: Cache state s_1 of an 8-way PLRU cache set after filling an empty (invalidated) cache with blocks I_0 … I_7 and accessing a synchronizing sequence "$I_0\ I_2\ I_4\ I_6$". Arrows point to the LRU block, in this case I_1. On an empty cache, new block are inserted from left to right.

Upon a cache hit, all the ancestors of the accessed cache line update their arrows to point towards the opposite direction of the leaf. Upon a cache miss, the block to replace is identified by following the arrows from the root node. When a new block is inserted, the arrows are updated as in the hit operation.

As a tree with n leafs has $n-1$ internal nodes, corresponding to the PLRU bits in our example, the total number of control states for PLRU is 2^{n-1}, where n is the associativity or ways of the cache. Thus, for an 8-way PLRU cache we have 128 different control states.

Experiment We now proceed to test our hypothesis, as described in Section 2, for the L1 cache of an i7-4790 processor.

First, we fill the cache with blocks I_0 … I_7, which on an empty cache are inserted from left to right, and we bring the cache's control state to any desired state, for example, to state s_2 by further accessing the sequence "$I_0\ I_2\ I_4\ I_6\ I_4\ I_0$" (see Figure 2).

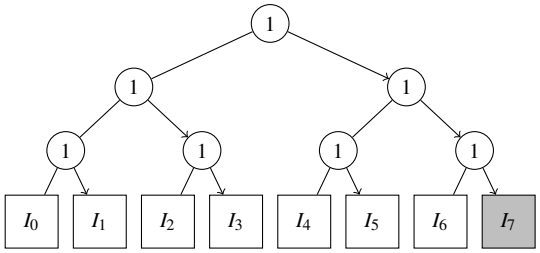


Figure 2: Cache state s_2 of PLRU cache after marking I_7 as the LRU block, by accessing sequence "$I_0\ I_2\ I_4\ I_6\ I_4\ I_0$".

Then, we continue by invalidating the cache with a `wbvind` instruction, and refilling the cache with associativity many new memory blocks I_8 … I_{15}, which again are inserted from left to right.

At this point, we validate our hypothesis by comparing the resulting control state s_2' (see Figure 3) with s_2 (see Figure 2).

For that, we rely on CacheQuery to probe the cache

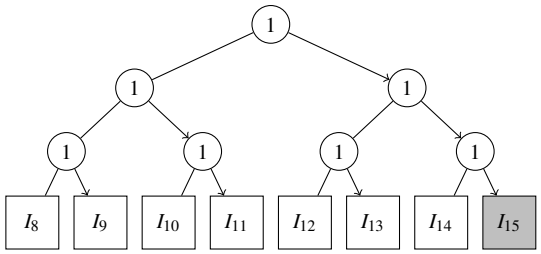


Figure 3: Resulting cache state s_2' after invalidating state s_2 and refilling the cache with new blocks I_8 … I_{15}. On an empty cache blocks are inserted from left to right.

state and we verify that the eviction order of s_2''s blocks is "$I_{15}\ I_{11}\ I_{13}\ I_9\ I_{14}\ I_{10}\ I_{12}\ I_8$", which uniquely corresponds to a control state identical to that of s_2.

After repeating the experiment for all possible states, we conclude that our hypothesis holds and that the control state survives both flushing and the insertion of new blocks.

Note that we observe the same effect when replacing the `wbinvd` instruction with a sequence of `clflush` instructions for each block I_0 … I_7, or when writing `1` to `IA32_FLUSH_CMD` MSR.

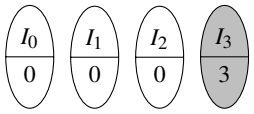
3.3 Example 2: L2's Quad-age LRU

Modern Intel CPUs have an L2 cache of associativity 4 with an undocumented replacement policy that has only recently been reverse engineered [1, 49].

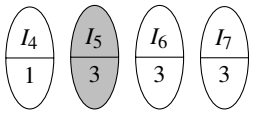
Replacement Policy This policy is an instance of Quad-Age LRU [23], called *New1* in [49] and QLRU_H00_M1_R2_U1 in [1]. The policy keeps two bits of control state (or metadata) per line, which can be interpreted as associating one of four possible ages with each line (0 to 3), hence the name. See Figure 4a for an example.

Upon a cache hit, the age of the accessed block is set to 0. Upon a cache miss, the first line, starting from the left, with age 3 is replaced, and the new block is inserted in this line with an initial age of 1. Observe that empty (or invalidated) lines are filled from right to left, before considering blocks with age 3. After each memory access, the ages of the lines are normalized to ensure the presence of an age-3 cache line: As long as there is no age-3 cache line, the ages of cache lines, except for the updated (or inserted) one, are incremented by 1.

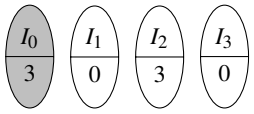
Interestingly, after normalization any valid state has: (1) at least one age-3 cache line; and (2) after a hit or miss, at least one cache line with age 0 or 1. Thus, for a 4-way cache, we can count the total number of valid states—on a filled cache set—as $4^4 - 3^4 - 2^4 + 1 = 160$, where: 4^4 is the size of the complete state space, 3^4 is the number of states without any age-3 line , 2^4 is the number of states without ages 1 and 0, and finally there is 1 control state, $\{2,2,2,2\}$, that we subtracted twice.



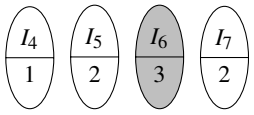
(a) Cache state s_1 on a 4-ways *New1* cache set after filling an empty (or invalidated) cache with blocks I_3 $\ldots I_0$ and further accessing sequence "I_0 I_1 I_2 I_0 I_1".



(b) Resulting cache state s'_1 after invalidating state s_1 and refilling the cache with new blocks I_7 $\ldots$ I_4. On an empty cache blocks are inserted from right to left.



(c) Cache state s_2 on a 4-ways *New1* cache set after filling an empty (or invalidated) cache with blocks I_3 $\ldots I_0$ and further accessing sequence "I_0 I_1 I_2 I_0 I_1 I_3 I_1".



(d) Resulting cache state s'_2 after invalidating state s_2 and refilling the cache with new blocks I_7 $\ldots$ I_4. On an empty cache blocks are inserted from right to left.

Figure 4: List of cache states for the experiment on L2's Quad-age LRU variants.

Experiment To validate our hypothesis that the control state survives cache-flushing operations, we perform an experiment, similar to the one described in Section 3.2, for the L2 cache of a Core i5-6500 processor.

First, we fill the cache with blocks I_3 $\ldots$ I_0, which on an empty cache are inserted from right to left, and bring the cache's control state to a specific state, for example, to s_1 by further accessing the sequence "I_0 I_1 I_2 I_0 I_1" (see Figure 4a).

Then, we continue by invalidating the cache with a `wbinvd` instruction, and by refilling the cache with associativity many new memory blocks I_7 $\ldots$ I_4, which again are inserted from right to left.

Unfortunately, in this case, comparing s'_1 (see Figure 4b) and s_1 (see Figure 4a) is not enough for validating our hypothesis, because the insertion of new blocks modifies the control state. We later provide a precise description of this behavior.

Instead, we verify that two different initial control states s_1 and s_2 (see Figure 4c) result in two different control states s'_1 and s'_2 (see Figure 4d), causing different eviction orders. Namely, we use CacheQuery to test that s'_1 first evicts I_1 and s'_2 first evicts I_2.

After repeating the experiment for all possible states, we conclude that our hypothesis holds and that the control state partially survives the flushing and the insertion of new blocks.

Note that we observe the same effect[2] when replacing the `wbinvd` instruction with a sequence of `clflush` instructions for each block I_0 $\ldots$ I_3.

Reverse Engineering the Insertion Logic In order to reverse engineer the insertion logic when invalid blocks are present, we obtain all the resulting control states, after invalidating and refilling the cache, from the 160 possible initial control states.

For this, we set the cache set into a given control state, invalidate the cache set, insert associativity many new blocks, and probe the cache until we uniquely identify its resulting control state. Note that since the probing is destructive, we often require to redo all these steps.

The probing works as follows: (1) Start with the complete set of states; (2) Perform a random memory access and eliminate all the states that are inconsistent with the observation (i.e., hit or miss) according to the replacement policy; (3) Repeat step 2 until we are left with a single possible state.

Once we obtained the mapping from the 160 *pre-flush* control states to the *post-refill* control states, we manually inferred the following rules that are fully consistent with the mapping:

- Invalidation does not reset the cache control state;
- A new block's age is set to 1 if its cache line's age was previously greater than 0, and kept to 0 otherwise;
- Normalization occurs as described earlier, and does not discriminate between valid and invalid lines.

Interestingly, with the simple refilling we use (i.e., "I_7 $\ldots$ I_4"), the set of *leaked* states is reduced to a subset of only 11 different control states. We do not explore whether more complicated insertions—with interleaved hits—can increase the size of this subset, and, therefore, increase the leakage.

3.4 Experimental Results

In this section we test how the `wbinvd` and `clflush` operations affect the cache's control state in all the cache levels of 11 different Intel processors from different generations. For each of the processors and cache levels, we performed tests similar to those explained in Sections 3.2–3.3.

[2]Invalidation with `IA32_FLUSH_CMD` is available only for L1 caches.

We quantify how much information survives the flush operation. For this, let the random variable S be a uniform distribution of initial control states, and the random variable O be the control states after a flush operation. The mutual information $I(S;O) = H(S) - H(S|O)$ captures how many bits are leaked.

For L1's PLRU, $H(S) = \log_2 128 = 7$ and $H(S|O) = 0$, given that we are able to uniquely identify all the initial states. Hence we find that the leakage is 7 bits.

For L2's *New1 (QLRU_H00_M1_R2_U1)*, $H(S) = \log_2 160 = 7.32$ and $H(S|O)$ requires more fine grained information about the joint probability distribution, which we are able to obtain from the 160 transitions. We find that the leakage is 3.17 bits.

Observe also that if there is only one observation (i.e., the resulting state after a flush is always the same) for any initial state, then $H(S|O) = H(S)$ and therefore the mutual information $I(S;O)$ is 0, i.e., there is no leakage.

CPU	Cache level	Assoc.	Leakage (bits)
Core i5-750 (Nehalem)	L1	8	0
	L2	8	0
	L3	16	0
Core i5-650 (Westmere)	L1	8	0
	L2	8	0
	L3	16	0
Core i7-2600 (Sandy Bridge)	L1	8	7
	L2	8	0
	L3	16	0
Core i5-3470 (Ivy Bridge)	L1	8	7
	L2	8	0
	L3	12	0
Core i7-4790 (Haswell)	L1	8	7
	L2	8	0
	L3	16	0
Core i5-5200U (Broadwell)	L1	8	7
	L2	8	0
	L3	12	0
Core i5-6500 (Skylake)	L1	8	0
	L2	4	3.17
	L3	12	0
Core i7-8550U (Kaby Lake)	L1	8	0
	L2	4	3.17
	L3	16	0
Core i7-8700K (Coffee Lake)	L1	8	0
	L2	4	3.17
	L3	16	0
Core i3-8121U) (Cannon Lake)	L1	8	0
	L2	4	0
	L3	16	0
Core i5-1035G1) (Ice Lake)	L1	12	0
	L2	8	0
	L3	12	0

Table 1: List of evaluated processors and cache levels. The leakage in bits correspond to the mutual information $I(S;O)$.

Table 1 reports all our findings, which we summarize here:

- For pre-Skylake processors (except for Nehalem and Westmere), the control state of L1 caches persists even after flushing operations (`wbinvd`, sequences of `clflush`, and writing `1` to the `IA32_FLUSH_CMD` MSR).
- For processors between Skylake and Coffee Lake, the control state of L2 caches persists even after flushing operations (through `wbinvd` and sequences of `clflush`).
- According to our experiments, it seems that the control state is wiped out on flushing for the most recent CPU families, like Cannon Lake and Ice Lake.

4 Discussion

In this section, we briefly discuss possible implications of, and possible solutions to, Flushgeist.

Access-based Attacks Access-based cache attackers[3] monitor their own cache activity to infer activity from a victim, namely, which cache lines or cache sets the victim accessed. While this provides a powerful primitive, it only exploits one dimension: *what* data is accessed. Knowledge about the control state provides access to a new dimension: *how* data is accessed.

While observing the control state is more difficult, it is possible for attackers with low-level control of the system [36, 45], and some recent works show that it is also possible in more common environments [8, 29, 54].

However, these attack were only possible for an adversary that shared memory, and hence the cache lines and control state, with the victim. Flushgeist breaks this assumption and enables an access-based attacker to leak the control state of the cache in non-shared memory scenarios. This is possible by flushing and refilling the cache with the attacker's own content, before probing the state.

An interesting open question is to quantify how much Flushgeist augments the attacker's extraction [10], i.e., the amount of information that an active attacker can extract from the cache state.

Cache Partitioning Cache partitioning [27] was initially proposed to improve predictability by reducing cache contention. Since then, several works [13, 28, 33, 42] have proposed the use of cache partitioning to also mitigate cache leakage. For instance, mechanisms like Intel's CAT [24] allow partitioning a cache's ways. In contrast, mechanisms like page coloring split the cache sets among untrusted parties. While Intel's CAT is known to leak through the cache replacement policy [29], for page coloring the question remains open. Thus, we like to point to modern adaptive policies, as described in [49, 52], as promising candidates for answering in the positive the question on cross cache set information leakage.

[3] We dismiss more powerful trace-based attackers, since general security guidelines already recommend to disable hyper-threading for sensitive computations.

Countermeasures Software mitigations would involve replacing (or extending) flush instructions on context switches with accesses to reset sequences—that bring the cache into a fix control state. Unfortunately, this requires knowledge of the specific cache content (to cause hits), or an even higher overhead due to some additional misses.

5 Conclusion

We evaluate the behavior of cache flushing instructions on several Intel processors and conclude that they do not properly cleanse all the information stored in the cache. Specifically, we show that in some caches the control state survives, allowing information leakage beyond cache flushes. We point out that countermeasures relying solely on flush instructions should be revised.

Disclosure

We first discussed our observations about the cache control state surviving flush operations in November 2019.

We reported our findings to Intel's PSIRT team on the 13th of April 2020, after having confirmed and understood the source of leakage.

Intel responded to us on the 19th of May 2020, concluding that the issue does not pose more risks than traditional cache side-channels, and thus recommending their best practice guidelines against side-channel attacks [25].

References

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

[2] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124. Springer, 2010.

[3] Taha Atahan Akyildiz, Can Berk Guzgeren, Cemal Yilmaz, and Erkay Savas. MeltdownDetector: A Runtime Approach for Detecting Meltdown Attacks. Cryptology ePrint Archive, Report 2019/613, 2019.

[4] Mohammad-Mahdi Bazm, Marc Lacoste, Mario Südholt, and Jean-Marc Menaud. Side channels in the cloud: Isolation challenges, attacks, and countermeasures. Url: `https://hal.inria.fr/hal-01591808`, March 2017.

[5] Daniel J. Bernstein. Cache-timing attacks on AES, 2004.

[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, August 2017.

[7] Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *CoRR*, abs/1506.00189, 2015.

[8] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[10] Pablo Cañones, Boris Köpf, and Jan Reineke. Security analysis of cache replacement policies. In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 189–209. Springer, 2017.

[11] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1441–1458. USENIX Association, August 2018.

[12] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 431–446. USENIX Association, 2013.

[13] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing OS abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. ACM, 2019.

[14] Qian Ge, Yuval Yarom, and Gernot Heiser. Do hardware cache flushing operations actually meet our expectations? *CoRR*, abs/1612.04474, 2016.

[15] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *Applied Cryptography and Network Security*, pages 83–102. Springer, 2018.

[16] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 163–170. IEEE, 2013.

[17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17. ACM, 2017.

[18] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why cache attacks on ARM are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091. USENIX Association, 2017.

[19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[20] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912. USENIX Association, 2015.

[21] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 191–205. IEEE, 2013.

[22] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault.

[23] Intel. Power management of the third generation Intel Core micro architecture formerly codenamed Ivy Bridge. Hot Chips, 2012.

[24] Intel. Are noisy neighbors in your data center keeping you up at night, 2017.

[25] Intel. Guidelines for mitigating timing side channels against cryptographic implementations, 2019.

[26] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 591–604. IEEE, 2015.

[27] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, November 1992.

[28] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204. USENIX Association, 2012.

[29] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 974–987, 2018.

[30] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[31] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564. USENIX Association, 2016.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 973–990. USENIX Association, 2018.

[33] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, 2016.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622. IEEE, 2015.

[35] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, NDSS. The Internet Society, February 2017.

[36] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of

cache attacks. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 69–90. Springer, 2017.

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

[38] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20. Springer-Verlag, 2006.

[39] Maria Méndez Real. *Spatial Isolation against Logical Cache-based Side-Channel Attacks in Many-Core Architectures*. PhD thesis, University of Southern Brittany, Morbihan, France, 2017.

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

[41] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[42] J. Shi, X. Song, H. Chen, and B. Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 194–199, 2011.

[43] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(1):37–71, January 2010.

[44] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[45] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 4:1–4:6. ACM, 2017.

[46] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[47] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. `https://cacheoutattack.com/`, 2020.

[48] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-VM side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 687–702, 2014.

[49] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köepf. CacheQuery: Learning Replacement Policies from Hardware Caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–532. ACM, 2020.

[50] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In *40th IEEE Symposium on Security and Privacy (S& P '19)*, pages 695–710. IEEE, 2019.

[51] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core, 2020.

[52] Henry Wong. Intel Ivy Bridge Cache Replacement Policy. Url: `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`, January 2013.

[53] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[54] Wenjie Xiong and Jakub Szefer. Leaking information through cache LRU states. *CoRR*, abs/1905.08348, 2019.

[55] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732. USENIX Association, 2014.

[56] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, November 2019.

[57] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache side-channel information leakage from the secure world on ARM devices. Cryptology ePrint Archive, Report 2016/980, 2016.

[58] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 827–838. ACM, 2013.