RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks

ARTIFACT EVALUATED usenix ASSOCIATION PASSED

Samira Briongos[1], Pedro Malagón[1], José M. Moya[1] and Thomas Eisenbarth[2,3]

[1]Integrated Systems Laboratory, Universidad Politécnica de Madrid, Madrid, Spain
[2]University of Lübeck, Lübeck, Germany
[3]Worcester Polytechnic Institute, Worcester, MA, USA

Abstract

Caches have become the prime method for unintended information extraction across logical isolation boundaries. They are widely available on all major CPU platforms and, as a side channel, caches provide great resolution, making them the most convenient channel for Spectre and Meltdown. As a consequence, several methods to stop cache attacks by detecting them have been proposed. Detection is strongly aided by the fact that observing cache activity of co-resident processes is not possible without altering the cache state and thereby forcing evictions on the observed processes. In this work, we show that this widely held assumption is incorrect. Through clever usage of the cache replacement policy, it is possible to track cache accesses of a victim's process *without forcing evictions* on the victim's data. Hence, online detection mechanisms that rely on these evictions can be circumvented as they would not detect the introduced RELOAD+REFRESH attack. The attack requires a profound understanding of the cache replacement policy. We present a methodology to recover the replacement policy and apply it to the last five generations of Intel processors. We further show empirically that the performance of RELOAD+REFRESH on cryptographic implementations is comparable to that of other widely used cache attacks, while detection methods that rely on L3 cache events are successfully thwarted.

1 Introduction

The microarchitecture of modern CPUs shares resources among concurrent processes. This sharing may result in unintended information flows between concurrent processes. Microarchitectural attacks, which exploit these information flows, have received a lot of attention in academia, industry and, with Spectre and Meltdown [34, 39], even in the public news. The OS or the hypervisor in virtual environments provide strict logical isolation among processes to enable secure multi threading. Yet, a malicious process can intentionally create contention to gain information about co-resident processes. Exploitable hardware resources include the branch prediction unit [3–5], the DRAM [33, 50, 54] and the cache [7, 15, 22, 47, 48, 61]. Last level caches (LLC) provide very high temporal and spatial resolution to observe and track memory access patterns. As a consequence, any code that generates cache utilization patterns dependent on secret data is vulnerable. Cache attacks can trespass VM boundaries to infer secret keys from neighboring processes or VMs [23, 52], break security protocols [28, 53] or compromise the end users privacy [47], and they can leak information from within a victim memory address space [34] when combined with other techniques.

Cache and other microarchitectural attacks pose a great threat and consequently, different techniques have been proposed for their detection and/or mitigation [16]. Among these proposals, hardware countermeasures take years to integrate and deploy, may induce performance penalties and currently, we are not aware of any manufacturer that has implemented them. Other proposals are meant for cloud hypervisors [32, 37, 56] and require making small modifications to the kernel configuration. Similarly, to the best of our knowledge, no hypervisor implements them, presumably due to the overhead they entail.

As a result, the only solution that seems practical for users that want to protect themselves against this kind of threat, is to detect ongoing attacks and then react in some way. To this end, several proposals [10, 13, 36, 49, 64] use hardware performance counters (HPCs) to detect ongoing microarchitectural attacks. These counters are special registers available in all modern CPUs that monitor hardware events such as cache misses. Some of these proposals are able to detect even attacks that were specially designed to bypass other countermeasures [20]. The common assumption in these works is that the *attacker induces measurable effects* on the victim. We, on the contrary, demonstrate that it is possible to obtain information from the victim while keeping its data in the cache and, consequently, not significantly altering its behavior, thus making attack detection harder.

Our Contribution: We analyze the replacement policy of current Intel CPUs and identify a new strategy that allows an attacker to monitor cache set accesses without forcing evictions of the victim 's data, thereby creating a new and *stealthier* cache-based microarchitectural attack. To achieve this goal, we perform the first full reverse engineering of different replacement policies present in various generations of Intel Core processors. We propose a technique that can be extended to study replacement policies of other processors. Using this technique, we demonstrate that it is possible to accurately predict which element of the set will be replaced in case of a cache miss. Then, we show that it is possible to exploit these deterministic cache replacement policies to derive a sophisticated cache attack: RELOAD+REFRESH, which is able to monitor the memory accesses of the desired victim without generating LLC misses.

We analyze the covert channel that this attack creates, and demonstrate that it has similar performance to state-of-the-art attacks, with a slightly decreased temporal resolution. As a proof of concept, we demonstrate how RELOAD+REFRESH works by retrieving the key of a T-Table implementation of AES and attacking the square and multiply version of RSA. We verify that our attack has a negligible effect on LLC related events, which makes it stealthy for countermeasures monitoring the LLC behavior. Instead, the attack changes the behavior of L1/L2 caches. Thus, our work stresses the need for detection mechanisms to also consider such events. Which, in turn, highlights the hardness of the performance counters set selection to detect all possible cache attacks, including ours and possible future attacks. To sum up, this work:

- introduces a methodology to test different replacement policies in modern caches.
- uncovers the replacement policy currently implemented in modern Intel Core processor generations, from fourth to eighth generation.
- expands the understanding of modern caches and lays the basis for improving traditional cache attacks.
- presents RELOAD+REFRESH, a new attack that exploits Intel cache replacement policies to extract information referring to a victim memory accesses.
- shows that the proposed attack causes negligible cache misses on the victim, which renders it undetectable by state-of-the-art countermeasures.

2 Background and related work

2.1 Cache architecture

CPU caches are small banks of fast memory located between the CPU cores and the RAM. As they are placed on the CPU die and close to the cores, they have low access latencies and thus reduce memory access times observed by the processor, improving the overall performance. Modern processors include cache memories that are hierarchically organized; low level caches (L1 and L2) are core private, smaller and closer to the processor, whereas the last level cache (LLC or L3) is bigger and shared among all the cores. It is divided into slices interconnected by a ring bus. The physical address of each element determines its mapping to a slice by a complex addressing function [44].

Intel's processors traditionally have included L3 inclusive caches: all the data which is present in the private lower caches has to be in the shared L3 cache. This approach makes cache coherence much easier to implement. However, presumably due to cache attacks, the newest Intel Skylake Server micro architecture uses a non-inclusive Last Level Cache [24].

In most modern processors caches are W-way set-associative. The cache is organized into multiple sets (S), each of them containing W lines of usually 64 bytes of data. The set in which each line is placed is derived from its address. The address bits are divided into offset (lowest-order bits used to locate data within a line), index ($log_2(S)$ consecutive bits starting from the offset bits that address the set) and tag (remaining bits which identify if the data is cached).

2.2 Cache replacement policies

When the processor requests some data, it first tries to retrieve this data from the cache (it starts looking in the lowest levels up to the last level). In the event of a *cache hit*, the data is loaded from the cache. On the contrary, in the event of a *cache miss*, the data is retrieved from the main memory and it is also placed in the cache assuming that it will be re-used in the near future. If there is no free space in the cache set, the memory controller has to decide which element in the cache has to be evicted. Since the processor may stall for several cycles whenever there is a *cache miss*, the decision of which data is evicted and which data stays is crucial for the performance.

Many replacement policies are possible including, for example, FIFO (First in First Out), LRU (Least Recently Used) or its approximations such as NRU [55] (Not Recently Used), LFU (Least Frequently Used), CLOCK [29](keeps a circular list of the elements) or even pseudo-random replacement policies. Modern high-performance processors implement approximations to LRU, because a truly LRU policy is hard to implement, as it requires complex hardware to track each access.

LRU or pseudo-LRU policies have demonstrated to perform well in most situations. Nevertheless, LRU policy behaves poorly for memory-intensive workloads whose working set is bigger than the available cache size or for scans (bursts of one-time access requests). As a result, adaptive algorithms, which are capable to adapt themselves to changes in the workloads, have been proposed. In 2003, Megiddo el al. [45] proposed ARC (Adaptive Replacement Cache) a hybrid of LRU and LFU. One year later, Bansal et al. [9] presented their solution based on LFU and CLOCK, which they named CAR (Clock with Adaptive Replacement).

In 2007 Quereshi et al. [51] suggested that performance could be improved by changing the insertion policy while maintaining the eviction policy. LIP (LRU Insertion Policy) consists in inserting each new piece of data in the LRU position whereas BIP (Bimodal Insertion Policy) most of the times places the new data in the LRU position and sometimes (in-frequently) inserts it in the MRU position. In order to decide which of the two policies behaves better, they proposed a dynamic insertion policy (DIP). DIP chooses between LIP and BIP depending on which one incurs fewer misses.

In 2010, Jaleel et al. [31] proposed a cache replacement algorithm that makes use of Re-reference Interval Prediction (RRIP). By using 2 bits per cache line, RRIP predicts if a cache line is going to be re-referenced in the near future. In case of eviction, the line with the longest interval prediction will be selected. Analogously to Quereshi et al., they presented two different approaches: Static RRIP (SRRIP) which inserts each new block with an intermediate re-reference, and Bimodal RRIP (BRRIP) which inserts most blocks with a distant re-reference interval and sometimes with an intermediate re-reference interval. They also proposed using set dueling to decide which policy fits better for the running application (Dynamic RRIP or DRRIP).

Regarding Intel processors, their replacement policy is known as "Quad-Age LRU" [30] and it is undocumented. The first serious attempt to reveal the cache replacement policy of different processors was made by Abel et al. [1]. In their work, they were able to uncover the replacement policy of an Intel Atom D525 processor and to infer a pseudo-LRU policy in an Intel Core 2 Duo E6300 processor. They later complemented their original work [2] and found a model that explained the eviction policy in other machines (Intel Core 2 Duo E6750 and E8400). Later on, Wong [60] showed that Intel's Ivy Bridge processors indeed implement a dynamic insertion policy as suggested in previous proposals [31, 51]. He was able to identify the regions that apparently had a fixed policy by measuring the average latency of the accesses to arrays of different sizes and provided some test code. Such regions were similarly observed by us in our experiments (Figure 3). These works have in common that the authors perform different sequences of memory accesses, and use a mechanism to estimate/measure the number of misses and later compare their measurements with the expected misses. However, they did not explain which concrete element in the cache would be evicted in the event of a miss.

Gruss et al. [19] studied cache eviction strategies on recent Intel CPUs in order to replace the `clflush` instruction and build a remote Rowhammer attack. As they mention, their work is not strictly a reverse engineering of the replacement policy, rather they test access patterns to find the best eviction strategy. In a work concurrent to ours, Vila et al. [57] tried to evaluate the influence of the replacement policy when obtaining the eviction set. Their results also show that some processors include adaptive policies whereas others do not.

To the best of our knowledge, our work is the first one that provides a comprehensive description of the replacement policies implemented on modern Intel processors up to the point that we are able to accurately determine which element of the set would be evicted using the information about the sequence of accesses.

2.3 Cache attacks

Cache attacks monitor the utilization of the cache (the sequence of cache hits and misses) to retrieve information about a co-resident victim. Whenever the pattern of memory accesses of a security-critical piece of software depends on the actual value of sensible data, such as a secret key, this sensitive data can be deduced by an attacker and will no longer be private.

Traditionally, cache attacks have been grouped into three categories [16]: FLUSH+RELOAD, PRIME+PROBE and EVICT+TIME. From those, the FLUSH+RELOAD and the PRIME+PROBE attacks (and their variants) stand over the rest due to their higher resolution.

Both attacks target the LLC, selecting one memory location that is expected to be accessed by the victim process. They consist of three stages: `initialization` (the attacker prepares the cache somehow), `waiting` (the attacker waits while the victim executes) and `recovering` (the attacker checks the state of the cache to retrieve information about the victim).

2.3.1 FLUSH+RELOAD

This attack relies on the existence of shared memory. Thus, it requires memory deduplication to be enabled. Deduplication is an optimization technique designed to improve memory utilization by merging duplicate memory pages. Using the `clflush` instruction the attacker removes the target lines from the cache, then waits for the victim process to execute (or an equivalent estimated time) and finally measures the time it takes to reload the previously flushed data. Low reload times mean the victim has used the data.

It was first introduced in [22], and was later extended to target the LLC to retrieve cryptographic keys, TLS protocol session messages or keyboard keystrokes across VMs [21, 28, 61]. Further, Zhang et al. [65] showed that it was applicable in several commercial PaaS clouds.

Relying on the `clflush` instruction and with the same requirements as FLUSH+RELOAD, Gruss et al. [20] proposed the FLUSH+FLUSH attack. It was intended to be stealthy and bypass existing monitoring systems. This variant recovers the information by measuring the execution time of the `clflush` instruction instead of the reload time, thus avoiding direct cache accesses and, as a consequence, detection. However, some works [10, 36] consider its effect also on the victim's side and succeed in its detection.

2.3.2 PRIME+PROBE

Contrary to the FLUSH+RELOAD attack, PRIME+PROBE is agnostic to special OS features in the system. Therefore, it can be applied to virtually every system. Moreover, it can recover information from dynamically allocated data. To do so, the attacker first fills or primes the cache set in which the victim data will be placed with its own data (initialization stage). Then, he waits and finally probes the desired set looking for time variations that carry information about the victim activity.

This attack was first proposed for the L1 data cache in [48] and was later expanded to the L1 instruction cache [6]. These approaches required both victim and attacker to share the same core, which diminishes practicality. However, it has been recently shown to be applicable to LLC. Researchers have bypassed several difficulties to target the LLC, as retrieving its complex address mapping [25, 44, 62], and recovered cryptographic keys, keyboard typed keystrokes [15, 26, 38] or even a RSA key in the Amazon EC2 cloud [23].

In case a defense system tries to either restrict access to the timers [35, 42] or to generate noise that could hide timing information, cache attacks are less likely to succeed. The PRIME+ABORT attack [14] overcomes this difficulty. It exploits Intel's implementation of Hardware Transactional Memory (TSX) to retrieve the information about cache accesses. It first starts a transaction to prime the targeted set, waits and finally it may or may not receive and abort depending on whether the victim has or has not accessed this set.

2.4 Countermeasures

Researchers have tackled the problem of mitigating cache attacks from different perspectives. Several proposals suggest limiting the access to the shared resources that can be exploited to infer information about a victim by modifying the underlying hardware [41, 58]. System-level software approaches, on the other hand, require modification of the current cloud infrastructure or the Linux kernel. STEALTHMEM [32] uses private virtual pages that ensure the data located in them is not evicted from the cache and avoid mapping any other page with these private virtual pages. CATalyst [40] uses Intel Cache Allocation Technology (CAT), which is a technology that enables system administrators to control how cores allocate data into the LLC. CACHEBAR [66] designs a memory management subsystem that dynamically changes the number of lines per cache set that a security domain can occupy to defeat PRIME+PROBE attacks and changes the state of the pages to avoid FLUSH+RELOAD. As we have already stated, we are not aware of any CPU manufacturer, cloud provider or OS implementing them.

A different approach to protect sensitive applications is to specifically design them to be secure against side-channels (no memory accesses depend on private information). Developers can use specific tools [59, 63] to ensure the binary of such applications does not leak information, even if it is under attack. There are other tools, such as MASCAT [27], which use code analysis techniques to detect potential attacks before running a program. This kind of tools is effective before malware distribution or execution, but their effectiveness is reduced in cloud environments where the attacker does not need to infect the victim.

For these reasons, we believe that the only countermeasures that an attacker may have to face when trying to retrieve information from a victim, are detection based countermeasures that can be implemented at user level. Cache attacks exploit the side effects of running a program in certain hardware to gain information from it, and similarly, these countermeasures employ monitoring mechanisms to observe these effects. Detection systems can use time measurements [12], hardware performance counters [10, 13, 36, 64] or place data in transactional regions [18] defined with the Intel TSX instructions. These detection systems measure the effect of the last level cache misses on the victim or on both the victim and the attacker. As a consequence, an attack that does not generate cache misses on the victim's side would be undetectable by these systems.

Detection systems that use performance counters as a source of information to infer anomalies in the execution of a program, are limited by the number of counters that can be monitored simultaneously. This number varies between processors, but implies that such counters must be carefully selected. As our work shows, although the monitoring approach can still consider more counters, it is limited and can not be arbitrarily extended to detect upcoming attacks.

3 Retrieval of Intel cache eviction policies

This work focuses on the LLC. Since it is shared across cores, the attacks targeting the LLC are not limited to the situation in which the victim and the attacker share the same core. It is also possible to extract fine-grained information from the LLC and many researchers are concerned about the attacks targeting the LLC. Attacks that assume a pseudo LRU eviction policy such as PRIME+PROBE or EVICT+RELOAD can benefit from detailed knowledge of the eviction policy, and can also benefit one attacker wishing to carry out a "stealthy" attack that does not cause cache misses on the victim.

In order to study the eviction policy, we try to emulate the hardware in software. We ensure that we can fill one set of the cache with our own data, access that data and force a miss when desired, to observe which element of the set is evicted. Thus, we have constructed an eviction set (a group of w different addresses that map to one specific set in w-way set-associative caches) and what we call a conflicting set (a second eviction set that maps to exactly the same set and is composed of disjoint addresses). Previous works have retrieved the complex addressing function [25,44,62] or demonstrate how to create the aforementioned sets dynamically [15]. When the number of cores in our test systems is a power of 2, we compute the set and slice number using the hash function in [44] and use that information to construct the eviction and conflicting sets. In the remaining situations such sets were constructed following the procedure proposed by Liu et al. in [15] (Algorithm 1).

For all the experiments, we have enabled the use of hugepages in our systems. Note that the order of the accesses is important to deduce the eviction policy. We enforce this order using *lfence* instructions, which act as barriers that ensure all preceding load and store instructions have finished before any load or store instruction that follows *lfence*. We have observed that *mfence* does not always serialize the instruction stream, that is, it does not completely prevent out of order execution.

3.1 Design of the experiments

Algorithm 1 Test of the desired eviction policy

```
Input: Eviction_set, Conflicting_set
Output: Accuracy of the policy                          ▷ hits/trials
  function TESTPOLICY(eviction_set, conflicting_set)
    $hits = 0$;
    while $i \leq num\_experiments$ do
      $j = 0$,i++;
      $control\_array \leftarrow \{\}; address\_array \leftarrow \{\}$;
      initialize_set();   ▷ Fills address and control arrays
      $lim = random()$;
      while $j \leq lim$ do
        lfence; j++;
        $next\_data = eviction\_set[random()]$;
        measure time to read next_data;
        if $time \geq ll\_threshold$ then          ▷ LLC access
          $update(control\_array, next\_data)$;
      $conf\_element = conflicting\_set[random()]$;
      read($conf\_element$);                    ▷ Force miss
      candidate=$getEvictionCandidate()$;
      if ($testDataEvicted()$ ==candidate) then
        $hits$++;
    return $hits/num\_experiments$;
```

We have performed experiments in different machines, each of them including an Intel processor from different generations. Table 1 presents a summary of the machines employed in this work. It includes the processor name, its number of cores, the cache size and associativity and the OS running on each machine. We have started by studying the processors of the fourth generation, which have been a common victim of published PRIME+PROBE attacks. We have extended our analysis to cover processors from fourth to eighth generation.

Before conducting the experiments to disclose the eviction policy implemented in each of the used machines, we have performed some experiments intended to verify that no cached data is evicted in the event of a cache miss if there is free room in the set. The procedure is quite straightforward: for each of the sets, we first completely fill it with the data on its corresponding eviction set. Next, we randomly *flush* one of these lines to ensure there is free room in the set, and we access one of the lines in the conflicting set checking that it is indeed loaded from main memory (cache miss). Finally, we make sure that all the lines in the eviction set (except for the one evicted) still reside in the cache by measuring times when re-accessing them. As expected, in all cases the incoming data was loaded in replacement of the *flushed* line.

The procedure we propose to retrieve the replacement policy, compares the actual evolution of the data in each of the sets with its theoretical evolution defined by an eviction policy during the runtime. Algorithm 1 summarizes this procedure. Each of the policies that has been tested had to be manually defined. We have evaluated true LRU, Tree PLRU, CLOCK, NRU, Static and Bimodal RRIP, self-defined policies using four control bits, etc. among many other possible cache eviction policies. After multiple experiments, we conclude that the policy implemented on the processors corresponds to the policy which best matches the experimental observations.

Algorithm 1 tries to emulate by software the behavior of the hardware (of the cache). For this purpose, it uses two arrays of size W. On the one hand, *address_array* mimics the studied set, storing the memory addresses whose data is in the cache set. On the other hand, *control_array* contains the control bits used for deciding which address will be evicted in case of conflict. Additionally, we need to manually define one function that updates the content of the *address_array*, one function that updates the *control_array* and another one that provides the eviction candidate i.e. it returns the address of the element that will be evicted in case of conflict. These functions are defined based on the tested replacement policy.

Note that for all the experiments the *initialize_set()* function makes sure that the tested set is empty (by filling it and then flushing all the elements that it holds) and later fills this set with all the elements in the eviction set. That is, the *address_array* contains the set of addresses of the eviction set with their corresponding control bits initialized.

To set an example, we assume we want to test the NRU policy [55], which turns out to match the policy implemented in an Intel Xeon E5620 according to our experiments. According to its specification, NRU uses one bit per cache line, this bit is set whenever a cache line is accessed. If setting one bit implies that all the bits of a cache set will be equal to one, then all the bits (except for the one that has just being accessed) will be cleared. In case of conflict, NRU will remove from the cache one element whose control bit is equal to zero. Thus,

Table 1: Details of the machines used in this work to retrieve their Replacement Policies

Generation	Processor	Number of cores	Cache size	Associativity	OS
4th	i7-4790	4	8Mb	16	CentOS Linux 7
4th	i5-4460	4	6Mb	12	Kali Linux 2019.2
4th	i7-4770K	4	8Mb	16	Kali Linux 2019.2
4th	Xeon E3-1226	4	8Mb	16	CentOS Linux 7
5th	i3-5010U	2	3Mb	12	Ubuntu 14
5th	i5-5200U	2	3Mb	12	Kali Linux 2019.2
6th	i7-6700K	4	8Mb	16	Ubuntu 16
6th	i5-6400	4	6Mb	12	Kali Linux 2019.2
6th	i7-6567U	2	4Mb	16	Kali Linux 2019.2
7th	i5-7600K	4	6Mb	12	CentOS Linux 7
7th	i7-7700HQ	4	6Mb	12	Ubuntu 16
7th	i7-7700	4	8Mb	16	Kali Linux 2019.2
8th	i7-8650U	4	8Mb	16	Debian 9.5
8th	i5-8400	6	9Mb	12	Kali Linux 2019.2
8th	i7-8550U	4	8Mb	16	Kali Linux 2019.2

in our procedure, the control bits would be -1 (line empty), 0 (line not recently used), and 1 (line recently used). When a memory line is accessed, the *update* function first checks if its address is already included in the *address_array*. If it is not, our function will add it to the *address_array* and set the corresponding bit in the *control_array*. On the contrary, the function only updates the values of the *control_array*. The *getEvictionCandidate* function will return one array position whose control bit value is -1, or, if no control bit is equal to -1, one whose control bit is equal to 0. In case multiple addresses have control bits equal to -1 or to 0, the function will return the first address whose control bits are -1 or 0, that it encounters when traversing the *control_array* from the beginning. Finally, after forcing a cache miss, the *testDataEvicted()* checks if the element evicted is the predicted by the NRU policy (the output of *getEvictionCandidate*).

We have noticed that only accesses to the LLC update the values of the control bits of the accessed element. That is, if the data is located in L1 or L2 caches when requested (reload time lower than ll_threshold), we do not update the values in the *control_array*. Figure 1 shows the distinction between accesses to low and last level caches based on reload times observed in the i7-4790 machine and validated with performance counters. The value of the ll_threshold varies between the different machines and requires calibration.

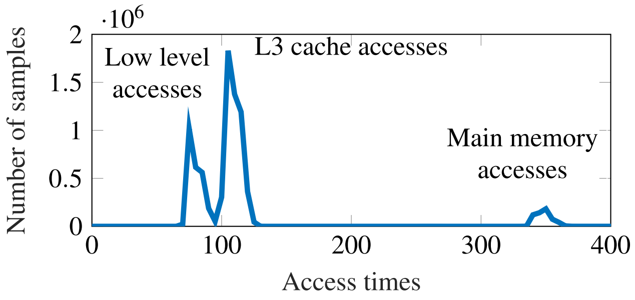


Figure 1: Distribution of the access times to different data. These times depend on which memory it was located.

3.2 Results

The outcomes of our experiments highlight some differences in the cache architecture of the machines, as also noticed in [14]. Traditionally, the number of slices of the cache used to be equal to the number of physical cores of the machine. This is true for the 4th and 5th generation processors. On the contrary, the newest ones have as many slices as virtual cores; that is, two times the number of physical cores. Cache sizes are similar, so they also differ in the number of sets per slice (2048 vs 1024).

Since several policies and previous works [60] suggest that different sets perform differently, we have repeated the experiment in Algorithm 1 for each of the sets in the last level cache. As a result, we have found out that apparently only the machines from the 4th and 5th generation implement *set dueling* to dynamically select the eviction policy. We conducted several further experiments intended for determining which sets implement a fixed policy and which others change their policy based on the number of hits and misses. Locating the sets with a fixed policy is interesting for various reasons: these sets will allow us to accurately determine the two different replacement policies, and they will allow favoring one policy over the other depending on our interests. This also means that monitoring one set belonging to the group of followers, gives information about which policy is currently operating.

The strategies for locating the sets included different access patterns that would lead to a different number of misses. For

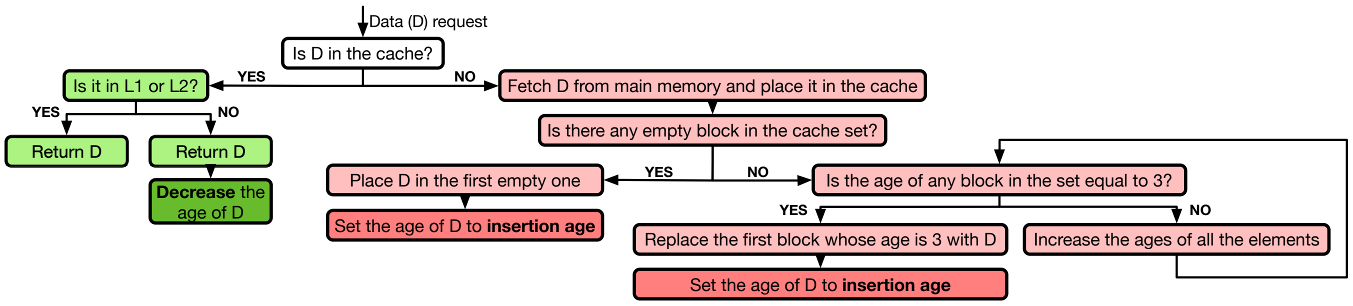

Data (D) request
Is D in the cache?
YES
NO
Is it in L1 or L2?
YES
NO
Return D
Return D
Decrease the age of D
Fetch D from main memory and place it in the cache
Is there any empty block in the cache set?
YES
NO
Place D in the first empty one
Set the age of D to insertion age
Is the age of any block in the set equal to 3?
YES
NO
Replace the first block whose age is 3 with D
Increase the ages of all the elements
Set the age of D to insertion age

Figure 2: Diagram that represents the process of data (D) retrieval whenever the processor makes a request. The blocks with green background represent a cache hit, whereas the blocks with red background represent a cache miss.

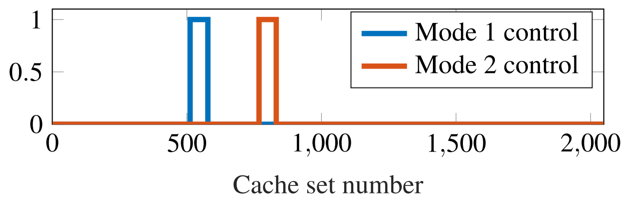


Figure 3: Location of the sets controlling the eviction policy within a slice of 2048 sets. Mode 1 (blue) and mode 2 (red).

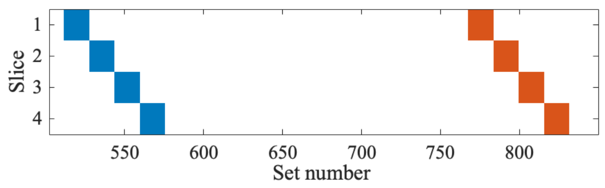


Figure 4: Detailed representation of the sets with fixed policy within each of the slices for the i7-4790 machine.

example, we have simulated bursts by accessing the eviction set in an ordered way, then the whole conflicting set, and finally re-accessing the eviction set. The observed number of misses depends on the policy. Pseudo LRU policies evict all the data in the eviction set after accessing the elements in the conflicting set. Whereas other policies intended for good performance in these situations (burst accesses to memory) cause fewer misses. As a result, we have located two regions composed of 64 cache sets in each slice that control each policy as did Wong before [60]. Figure 3 represents all the sets of a cache slice with the control regions. The region coloured in blue controls the policy 1, and the region coloured in red controls the policy 2. Except for the Xeon machine, where these regions are located in sets 1024-1088 and 1280-1344, the remaining machines are consistent with Figure 3.

Not all the sets within the aforementioned regions implement a fixed policy. Particularly, only one of the sets in each slice controls one policy. This fact was observed and discovered after multiple experiments with different patterns. The sets with a fixed policy for each of the slices are depicted in figure 4. In processors with two slices, these control sets also alternate between slices. As a result each slice has 32 control sets. To obtain the actual control sets within the slice, it is important to test the sets and slices without order, otherwise it may seem that some sets have a fixed policy and they do not.

The policy we will uncover is the one implemented in the L3 cache. The policies implemented in the L1 and L2 caches can be different (actually, in L1 is different). We have been able to uncover a policy that seems to explain the observed evictions. In fact, over 97% of the evictions have been correctly predicted in all cases [1], and it is likely that the errors were due to noise.

Although we have observed differences between generations and some machines implement set dueling, the decision of which data is going to be evicted is the same in all cases. The replacement policy is always the same; what changes is the insertion policy. Due to space limitations and to avoid creating confusion, we only include here the description of the policies revealed by our experiments as the ones implemented in the Intel processors. Assuming that the policy is named Quad-Age LRU, in the following we refer to ages instead of control bits. Figure 2 represents the procedure followed to retrieve a piece of data when requested by the processor. It summarizes the replacement policy and our observations. If the data is retrieved from the LLC, the controller decreases the age of the requested element when giving it to the processor. If there is a cache miss and one element has to be evicted, the replacement policy will select the oldest one.

Intel's processors use two bits to represent the age of the elements in the cache. Consequently, the maximum age is three. In the case that there are multiple blocks whose age is three, the evicted one is the first one the processor finds. The cache behaves somehow like an array of data, and when searching for a block of data placed on it, the controller always starts from the same location, which would be equivalent to index 0 in an array. We have observed that when all the elements in

[1]These results refer to the sets with fixed policy in the machines that implement set dueling. The remaining sets were tested once the two policies were known, and we checked they followed one of them.

a set reach age 0, the age of all of them is incremented so the processor is still able to track the accesses.

As we have already stated, the machines used in our experiments only differ in the insertion age; that is, the initial value for the age of a cache line when it is first loaded into the set or when it is reloaded after a cache miss. Particularly, the processors from 4th and 5th generations that implement *set dueling*, insert the elements with age 2 in one of the cases and with age 3 in the other. We denote each of these situations or working modes as mode 1 and mode 2, respectively. The remaining processors (6th, 7th and 8th generations) always insert the blocks with age 2, which is equivalent to the mode 1 in the previous generations.

In order to help the reader to understand how the cache works, figure 5 shows an example of how the contents of a cache set are updated with each access according to each policy. When the processor requests the line "d", there is an empty block in the set, so "d" is placed in that set and it gets age 2 (Mode 1) or age 3 (Mode 2). In mode 1, the eviction candidate is now "a" because it is the only one with age 3, whereas in mode 2 the eviction candidate is "d" as it has age 3 and is on the left of "a". The processor then requests "d", so its age decreases from 2 to 1 in both cases. Accessing "g" causes a miss. The aforementioned eviction candidates will be replaced with "g", and its age will be set to 2 or 3 respectively. Eventually, when the processor requests "a", it will cause a miss in mode 1 (it was evicted on the previous step) and a hit in mode 2, so it will decrease its age. Note that in this example, we assume that all the requests are directly made to the last level cache.

4 RELOAD+REFRESH

If any kind of sharing mechanism is implemented, an attacker knowing the eviction policy can place some data that the victim is likely to use in the cache (the *target*) and in the desired position among the set. Since the position of the blocks and their ages (which in turn depend on the sequence of memory accesses) determine the exact eviction candidate, the attacker can force the *target* to be the eviction candidate. If the victim uses the *target* it will no longer be the eviction candidate, because its age decreases with the access. The attacker can force a miss and check afterwards if the *target* is still in the cache. If it is, the attacker retrieves the desired information, that is, the victim has used the data whereas victim has loaded the data from the cache without suffering any cache misses (no attack trace). This is the main idea of the RELOAD+REFRESH attack.

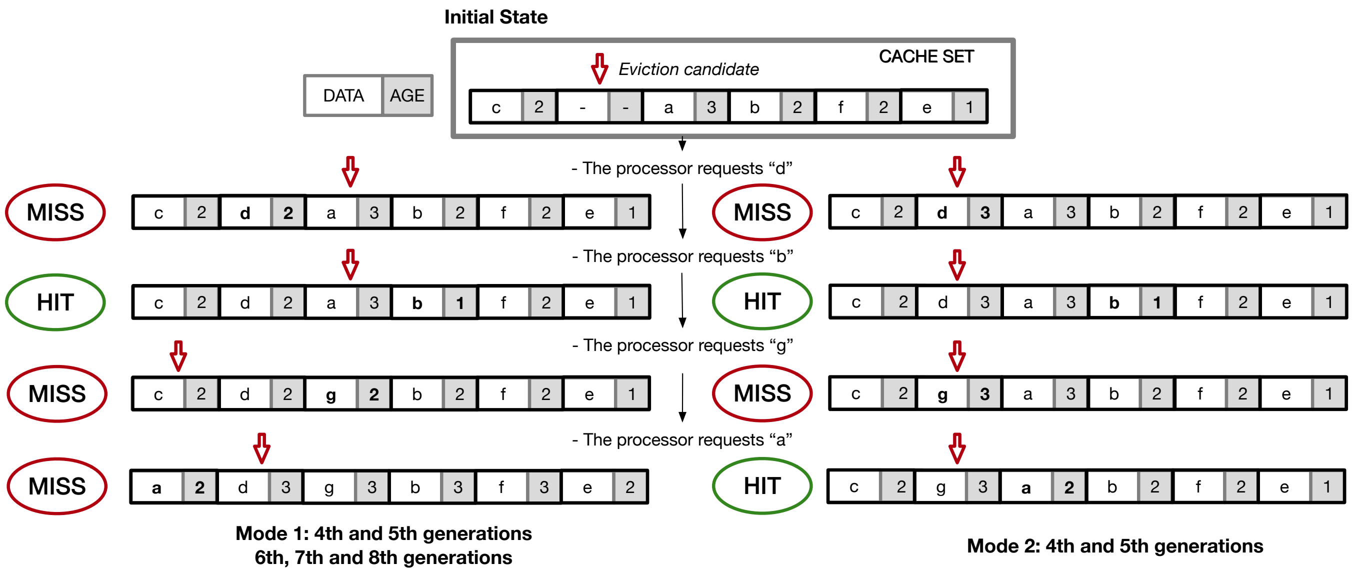
OSs implement mechanisms such as Kernel Same-page Merging (KSM) in Linux [8] that improve memory utilization by merging multiple copies of identical memory pages into one. This feature was originally designed for virtual environments where multiple VMs are likely to place the same data in memory, and was later included in the OSs. Although most cloud providers have disabled it, it is still enabled in multiple OSs. When enabled, the attacker using the RELOAD+REFRESH technique needs some reverse engineering to retrieve the address he wants to monitor, and he also needs to find an eviction set that maps to the same set as this address.

We use Figure 6 to depict the stages of the attack and the possible "states" of the cache set. The attacker first inserts the target address into the cache and then all the elements in the eviction set, except one, which will be used to force an eviction. By the time the attacker has finished filling the cache with data, the target address will be in level 3 cache. The number of ways in low level caches is lower than the number of ways in the L3 cache, and since the L3 cache is inclusive, it will remove the target address from the low level caches when loading the last elements of the eviction set. Even if the victim and the attacker are located in the same core, an access of the victim to the target address will update its age, so the attacker would be able to retrieve this information.

The data is placed in such a way that the target becomes the eviction candidate. The attacker then waits for the victim to access the target. If it does, the element inserted in the second place turns into the oldest one, and thus into the eviction candidate. If it does not, the eviction candidate is still the target address. The attacker then reads the element of the eviction set (ev_{W-1}) that remains out of the cache, forcing this way a conflict in the cache set, and the eviction of the candidate. As a consequence, when reading (RELOAD) the target address again, the attacker will know if the victim has used the data (low reload time) or not (high reload time). The state of the cache has to be reverted to the initial one, so all the elements get the same age again (REFRESH). The element ev_{W-1} is forced out of the cache, so it could be used to create a new conflict on the next iteration.

When the cache policy is working in mode 2, each element is inserted with age 3. In this case, steps 1 to 5 are equivalent. However, step 6 changes depending on whether the victim is allocated in the same core as the attacker or not. When not, the other elements have age 3 and the target is the eviction candidate, so there is no need to refresh the data for the attack. On the other hand, when they are on the same core, the attacker needs to remove the target from the low level caches by refreshing the other elements in the cache set. Note that in this situation, the attacker could target the low level caches. The RELOAD time reveals if both victim and attacker are sharing the same core or not.

Additionally, the mode 2 policy enables a detectable fast cross core cache attack that does not require shared memory. Once the cache set is filled with the attacker's data, all the elements get age 3 and the eviction candidate is now the first element inserted by the attacker. If the victim uses the expected data, the eviction candidate will be replaced. Even if the victim uses the data multiple times, its age will not change, since it will be fetched from the low level caches. Then, the



Initial State
CACHE SET
Eviction candidate
DATA
AGE
- The processor requests "d"
- The processor requests "b"
- The processor requests "g"
- The processor requests "a"
MISS
HIT
MISS
MISS
MISS
HIT
MISS
HIT
Mode 1: 4th and 5th generations 6th, 7th and 8th generations
Mode 2: 4th and 5th generations

Figure 5: Sequence of data accesses in a cache set updating their content and their associated ages for the two observed policies. Mode 1 of the 4th and 5th generations behaves exactly the same as the 6th, 7th and 8th generations. The red arrow points the eviction candidate, that is, the data that would be evicted in case of cache miss.

attacker only has to access the first element (eviction candidate) to check whether the victim has or has not accessed the target data. Note that with this access the attacker replaces the victim's data (because it became the eviction candidate when loaded with age 3) so it is equivalent to the REFRESH. If, on the contrary, the victim does not use the data, the attacker's data will still be in the cache. The attacker will then flush and reload this data to ensure it gets age 3 again.

Algorithm 2 Reload function

```
Input: Eviction_set, Target_address
Output: Reload time
  function RELOAD(Target_address,eviction_set)
      "rdtsc";
      "lfence";
      $read(eviction\_set[w-1]);$          ▷ Forces a miss
      "lfence";
      $flush(eviction\_set[w-1]);$
      "lfence";
      $read(Target\_address);$
      $flush(Target\_address);$
      "lfence";
      $read(Target\_address);$          ▷ Reload on first position
      "lfence";
      "rdtsc";
      $read(eviction\_set[0]);$
      return $time\_reload;$
```

Algorithms 2 and 3 summarize the steps of the RELOAD+REFRESH attack when the insertion age is two (newest Intel generations or mode 1 in oldest generations). The cache set is filled with the target address plus $W-1$ elements of the eviction set during initialization. Then, the attacker waits for the victim to run the code. Later, he performs the RELOAD and REFRESH steps. The RELOAD step gives information about the victim accesses and the REFRESH step gets the set ready to retrieve information from the victim. When initializing the set, we first fill the set, then flush the whole set and finally reload the data again to ensure the insertion order and that the cache state is known by us.

In the RELOAD function it is not necessary to flush the Target_address unless it has not been used by the victim. The same assumption is true for the conflicting address or the element $W-1$ of the eviction set, which would have to be flushed only in that situation. However, to avoid *if* conditions in the code, we have chosen to implement the RELOAD function this way. Low reload times mean the data was used by the victim, whereas high reload times mean it was not.

The REFRESH function is meant for a 12 way set. Since the target and the first element of the eviction set have been loaded in the RELOAD step, the REFRESH function only has to access the remaining 10 elements of the set. To avoid out of order execution and ensure the order, which in turn ensures the ages of the elements in the eviction set are updated, such elements have to be accessed as a linked list (one element contains the address of the following one). Thus, this function is similar to the probe function in [15] except for the fact that it loads W-2 elements of the linked list. Additionally, the *refresh time* can be used to detect if any other process is also using that set.

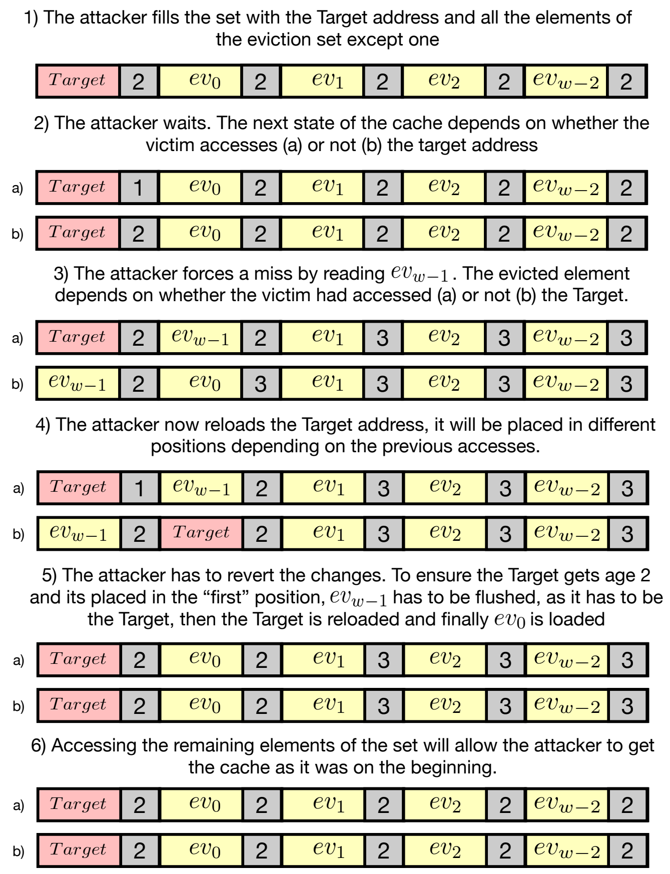

1) The attacker fills the set with the Target address and all the elements of the eviction set except one
Target 2 ev_0 2 ev_1 2 ev_2 2 ev_{w-2} 2
2) The attacker waits. The next state of the cache depends on whether the victim accesses (a) or not (b) the target address
a) Target 1 ev_0 2 ev_1 2 ev_2 2 ev_{w-2} 2
b) Target 2 ev_0 2 ev_1 2 ev_2 2 ev_{w-2} 2
3) The attacker forces a miss by reading ev_{w-1}. The evicted element depends on whether the victim had accessed (a) or not (b) the Target.
a) Target 2 ev_{w-1} 2 ev_1 3 ev_2 3 ev_{w-2} 3
b) ev_{w-1} 2 ev_0 3 ev_1 3 ev_2 3 ev_{w-2} 3
4) The attacker now reloads the Target address, it will be placed in different positions depending on the previous accesses.
a) Target 1 ev_{w-1} 2 ev_1 3 ev_2 3 ev_{w-2} 3
b) ev_{w-1} 2 Target 2 ev_1 3 ev_2 3 ev_{w-2} 3
5) The attacker has to revert the changes. To ensure the Target gets age 2 and its placed in the "first" position, ev_{w-1} has to be flushed, as it has to be the Target, then the Target is reloaded and finally ev_0 is loaded
a) Target 2 ev_0 2 ev_1 3 ev_2 3 ev_{w-2} 3
b) Target 2 ev_0 2 ev_1 3 ev_2 3 ev_{w-2} 3
6) Accessing the remaining elements of the set will allow the attacker to get the cache as it was on the beginning.
a) Target 2 ev_0 2 ev_1 2 ev_2 2 ev_{w-2} 2
b) Target 2 ev_0 2 ev_1 2 ev_2 2 ev_{w-2} 2

Figure 6: Sequence of possible cache set states during the attack for the mode 1 or the newest generations, starting with all elements in the set with age 2.

4.1 Noise tolerance

The proposed attack relies on the order in which the elements are inserted into the cache set to both avoid misses on the victim side and to learn information about the data that has been accessed. If other processes are running and using data that maps to the same cache slice (introducing noise), the efficiency of the attack can be lessened and also some detection mechanisms can be triggered.

As mentioned before, the refresh step can reveal such situations. Then, the attacker can slightly change the approach. Assuming that only one address is being used by the noise-generating process, the attacker can easily handle noise, avoid detection and still gain information about the victim. The trick to deal with noise is placing the target on a different place within the set (the second place in this example). In case somebody else uses any data mapping to that set, the replaced data belongs to the attacker; specifically it is the data placed in first place in the set. When the attacker forces a miss, the eviction candidate will be either the target address (if the victim did not use it) or the element inserted in third place (the victim did use the target data). The attacker can gain information about the victim by reloading the target address, and he must begin by refreshing the third element of the eviction set and finish with the first one which will evict the "noise" from the cache, so the age of all the blocks is set to 2 again.

Algorithm 3 Refresh function

```
Input: Eviction_set
Output: Refresh time
  function REFRESH(Eviction_set)
      volatile unsigned int time;
      asm __volatile__(
        " lfence \n"
        " rdtsc \n"
        " movl %%eax, %%esi \n"
        " movq 8(%1), %%rdi \n"          ▷ Eviction_set[1]
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"
        " movq (%%rdi), %%rdi \n"     ▷ Eviction_set[w-2]
        " lfence \n"
        " rdtsc \n"
        " subl %%esi, %%eax \n"     ▷ Time value on %eax
      );
      return time_refresh;
```

5 Comparison with previous approaches

5.1 Covert channel

In order to study the resolution of the proposed technique and to characterize it, as well as to compare it with previous approaches (FLUSH+RELOAD and PRIME+PROBE) we construct a covert channel between two processes (referred as *sender* and *receiver*) in a similar way as previous works have done [15, 20, 43]. [2]

The *sender* transmits a 1 by accessing a memory location from a shared library and a 0 by not-accessing it. Once the memory location is accessed, he waits for a fixed time and reads that data again. The receiver monitors the cache utilization using each of the aforementioned techniques and determines whether a 1 or a 0 was transmitted. That is, whether the victim has used the data or not. The *sender* and the *receiver* are not synchronized.

[2]The source code for this test can be found at https://github.com/greenlsi/reload_refresh

In each of the experiments executed, the *sender* reads the target memory location once during each fixed window of time. That is, it accesses one memory location (sends 1) and then waits (transmits 0) for a fixed time before the following access. We consider as true positives when the *sender* accesses a piece of data and the *receiver* detects that access. Similarly, true negatives are non-accesses that are classified as 0. In some situations, the processor appears to be performing other tasks that do not allow the retrieval of information. Since we do not get these samples, we cannot classify them and we do not consider them for evaluation.

The PRIME+PROBE attack can be conducted following different approaches. We do not use the zig-zag pattern that was intended to avoid changes in the replacement policy [15]. Accessing the elements in a cache set this way increases the number of false positives since it sometimes fails to remove the data from the cache. We access the eviction set of size W always in the same order, and the elements are accessed as a linked list. If the initial state of the cache is known, this means that at most we need 2 probes to evict the data from the cache, in the case when the access to the target happens in the middle of a probing stage. We have also tested the proposal of Gruss et al. [20] with the configuration parameters S=W, C=2 and D=2. This approach is faster than accessing the elements in the eviction set as a linked list and thus, presents better time resolution. In scenarios where victim and attacker do not interfere with each other (such as the attack against AES in section 5.2), the eviction rate of this approach is around 99%. However, in a different scenario where interference is possible, as in the case of the attack against RSA (section 5.3) or when the interval between monitored accesses is low, the number of false positives slightly increases with this approach. In any case, both approaches yield to comparable results. We include in this and the following subsections, results referring to the PRIME+PROBE attack when the eviction set is accessed as a linked list.

The results of these experiments in terms of the F-Score for each fixed time window are presented in Table 2. These experiments were performed in the i5-7600K machine (Table 1). The statistics for each waiting time are computed for 50000 windows. As a result, the number of samples collected for each experiment is different. Note that when the waiting time between samples is low, both PRIME+PROBE and RELOAD+REFRESH are not able to distinguish between 1 and 0. PRIME+PROBE presents a slightly better resolution in our test system. Note that, in this case, we sometimes do not get two samples for each window (access and idle), we do not consider as false positives the samples classified as 1 in that window.

Even when RELOAD+REFRESH has lower resolution than other attacks, it can be used to retrieve secret keys of cryptographic implementations. We demonstrate this statement and replicate two published attacks: one against the T-Table implementation of AES (section 5.2) and one against

Table 2: F-Score for the different attacks when the *sender* accesses the data at different and fixed intervals (ns). R+R stands for RELOAD+REFRESH F+R for FLUSH+RELOAD and P+P for PRIME+PROBE

Times >	50000	10000	1000	750	500	250
R+R	**0.988**	**0.975**	**0.925**	**0.684**	**-**	**-**
F+R	0.999	0.995	0.996	0.991	0.989	0.981
P+P	0.934	0.911	0.873	0.716	0.548	-

the square and multiply exponentiation implementation included in RSA (section 5.3). Although both implementations have been replaced by new ones, we use them for comparison.

5.2 Attacking AES

The T-Table implementation used to be a popular software implementation of AES. While still available, this implementation is not the default option when compiling the OpenSSL library due to its susceptibility to microarchitectural attacks. This implementation replaces the *SubBytes*, *ShiftRows* and *MixColumns* operations with table lookups (memory accesses) and XOR operations. Since the accesses to the T-Tables depend on the secret key, an attacker monitoring just one line of each T-Table is able to recover the full AES key.

Our scenario is similar to the one described by Irazoqui et al. [7], which was later replicated by Briongos et al. [11]. They focused on retrieving information about the last round of the AES encryption process, in which the ciphertext is obtained by performing one XOR operation between an element contained in the tables and the secret key. As the content of the tables is publicly available from the source code, they obtained the secret final round key by xoring the table content hold in the cache line, with the ciphertext.

Besides performing the attack against the AES T-Table implementation (OpenSSL 1.0.1f compiled with gcc and the no-asm and no-hw flags) using the RELOAD+REFRESH (R+R) technique, we have performed the same attack using the FLUSH+RELOAD (F+R) and PRIME+PROBE (P+P) techniques, to provide a fair comparison regarding the number of traces required to obtain the key. In order to retrieve the whole key, the attacker has to monitor at least one line of each T-Table. The attacker can monitor from one up to four lines at a time. For this comparison, we monitor one table at a time.

Table 3 shows the results for each of the approaches. In this scenario, the attacker performs one operation, then the victim performs the encryption, and finally the attacker retrieves the information about the victim. That is, the victim and the attacker do not interfere with each other while doing the different operations. To obtain the key we use cache misses [11], so false positives are measured misses when the victim used the data in the T-Table. We repeated each experiment until we have recovered the key 1000 times.

Table 3: Mean number of samples required to retrieve each four byte group of the whole AES key when monitoring one line per encryption, and the corresponding F-Score.

Attack	**R+R**	F+R	P+P
Samples	**3800**	3500	3900
F-Score	**0.98**	0.99	0.97

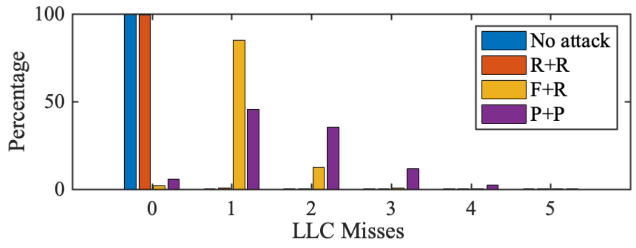


Figure 7: Distribution of the number of misses induced in the victim process by the different attacks, and with no attack. Each includes 1 million of samples

5.2.1 Measurement of LLC misses

RELOAD+REFRESH is able to retrieve an AES key with a negligible impact on the victim process. We compare the number of L3 cache misses that the victim suffers per encryption performed, for all the attacks and for normal executions. We use the PAPI software interface [46] to read the counters referring to the victim process. PAPI allows us to insert one instruction just before, and another one just after the encryption process ending to read the L3 cache misses counter, which is mainly the information used so far for cache attack detection [10, 13, 36, 64]. Figure 7 shows the resulting distribution of the number of misses the victim sees for each attack and for the normal execution of the encryption.

As implied by Figure 7, our attack cannot be distinguished from the normal performance of the AES encryption process by measuring the number of L3 cache misses. As we did for the analysis of the covert channel, when performing the PRIME+PROBE attack against AES, we access the data in the same order every time. The reason is that in previous experiments that we have conducted, the eviction rate we achieved with the zig-zag pattern was below 80% using just one probe per measurement.

Additionally, we use the *rdtsc* instruction to measure the time it takes to complete each encryption and show the results in Figure 8. The differences observed in Figure 8 between the normal encryption and the RELOAD+REFRESH approach are not significant, especially when compared with the other attacks. The mean encryption time when there is no attack is 595 cycles, whereas it increases up to 623 cycles when attacked with the RELOAD+REFRESH technique. This time difference exists because, when suffering the RELOAD+REFRESH attack, the victim has to load the data

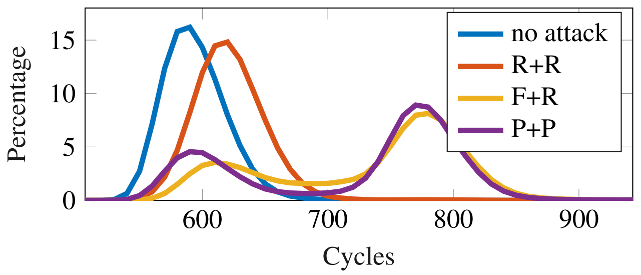


Figure 8: Distribution of the encryption times in different situations. Each distribution includes 1 million of samples.

(if used) from the L3 cache instead of loading it from the L1 or L2 caches.

5.3 Attacking RSA

RSA is the most widely used public key crypto system for data encryption as well as for digital signatures. Its security is based on the practical difficulty of the factorization of the product of two large prime numbers. RSA involves a public key (used for encryption) and a private key (used for decryption). There are many algorithms suitable for computing the modular exponentiation required for both encryption and decryption. In this work we focus on the *square and multiply* exponentiation algorithm [17] as Yarom et al. did [61]. As in the case of AES, this implementation has been replaced by implementations with no key-dependent memory accesses that attempt to achieve constant execution times.

Square and multiply computes $x = b^e \bmod m$ as a sequence of Square and Multiply operations that depend on the bits of the exponent e. If the bit happens to be a 1, then the Square operation is followed by a Multiply operation. If the bit is a 0, only a Square operation is executed. As a consequence, retrieving the sequence of operations executed means recovering the exponent; that is, the key.

As a difference with the attack against AES, we monitor instructions instead of data. Additionally, an attack against RSA needs to have enough time resolution to correctly retrieve the sequence of operations. As we did before, we performed the attack using our stealthy technique as well as using the FLUSH+RELOAD and PRIME+PROBE techniques.

The targeted crypto library is *libgcrypt* version 1.5.0, which includes the aforementioned square and multiply implementation. The key length in our experiments was 2048 bits, and we collected information for 1000 decryptions per attack. When attacking RSA, it is possible to monitor all the functions implied in the exponentiation or just one. When monitoring all the instructions, the attacker is able to reconstruct the sequence of observations. If the attacker monitors only one instruction, he has to use the differences of times between occurrences of the monitored event to retrieve the key. We only monitor the Multiply operation.

Figure 9 compares part of a trace retrieved using the RELOAD+REFRESH approach with the real execution of a RSA decryption operation (we collect timestamps). The trace corresponding to the real sequence of squares and multiplies is represented as blue bars with different values: 800 means a Square was executed and 700 it was a Multiply. The slight misalignment between the two traces occurs because the RSA execution *timestamp* is collected after each exponent bit has been processed, and the *timestamp* of the attack samples after the reload operation has finished.

The results of our experiments are summarized in table 4. As in the case of the characterization of the covert channel, we do not classify as false positive or false negative the samples that are lost, that is, not collected in time. This situation happens for about 1-2% of the samples. Since we try to detect Multiply operations, false positives refer to the situation in which a Multiply was detected but not executed. The accuracy is given as the number of correctly classified samples (True positives+True negatives) divided by the number of collected samples during the RSA decryption.

Table 4: Percentage of samples correctly retrieved and false positives generated by each approach when attacking RSA.

Attack	**R+R**	F+R	P+P
Accuracy	**96.1%**	98.6 %	95.4%
F-Score	**0.952**	0.99	0.945

5.3.1 Measurement of LLC misses

We have monitored the number of cache misses detected when executing a complete RSA decryption. When trying different keys, we have observed that the distributions change not only depending on the attack, but on the secret key. For this reason, the total amount of misses per encryption cannot be used to detect ongoing attacks, thus the cache misses have to be measured concurrently with the execution of the decryption.

We have also monitored the victim LLC misses periodically. We have collected samples for 1000 complete RSA decryptions in each of the scenarios with a sampling rate of 100 μs. The results obtained in this case show a varying number of misses during the initialization steps. During this initialization, considering exclusively the number of misses caused in the different scenarios, is not possible to distinguish between attacks and the normal operation. Later on, the number of misses gets stable and tends to zero during the normal operation. Similarly, this trend can be observed during the RELOAD+REFRESH attacks. On the contrary, both FLUSH+RELOAD and PRIME+PROBE cause a noticeable amount of misses. The concrete mean values of the misses are presented in Table 5.

Since detection mechanisms such as CacheShield [10], define a region in with some misses are tolerated to avoid false positives, and only cache misses are considered, our attack will not trigger an alarm. Figure 10 shows the section of the decryption process in which the number of misses has become stable for the different scenarios.

The RELOAD+REFRESH approach (as well as the other attacks) are not synchronized with the decryption operation, as a result, there are situations in which both victim and attacker can try to access the target date simultaneously. If the victim tries to execute the Multiply operation when the attacker is flushing and reloading the mentioned line, the victim may get a miss. Therefore, a few misses can be observed in Figure 10 for our approach.

6 Detection evaluation

RELOAD+REFRESH causes a negligible amount of LLC misses on the victim process. Thus, existing detection techniques would fail to detect the attack unless adapted. Our attack highlights a problem that has not been considered before in performance-counter-based detection systems: the selection of counters is a hard problem because it is unknown if future attacks could similarly evade the concrete selection of counters of such systems. Besides, the number of available counters that can be read in parallel in each platform, is limited. As a consequence, detection systems cannot be arbitrarily expanded to deal with future attacks.

With the aim of quantifying the effect that our proposal has on the victim, and in order to provide some insights about which counters should a detection system consider to deal with RELOAD+REFRESH, we have periodically monitored different counters when executing the attacks against AES and RSA and analyzed the outcomes. We have used PAPI to collect such information. The sampling rate was set to 100 μs. Given that not all the counters can be read in parallel, we have repeated the experiments multiple times. We have merged the results when the sampling intervals were in a range defined by the expected value $\pm$ a 10% of this sampling value. In the particular case of RSA, the samples that refer to the beginning of the execution have been removed (we focus on the stable part). Finally, we have randomly selected 10000 samples per algorithm and attack to conduct the analysis. The results of the analysis are summarized in Table 5. Note that L2 cache misses report the same value as L3 accesses, and similarly L1 misses are L2 accesses, so only one of these values is included in the Table.

As it can be inferred from the Table 5, a single counter referring to L3 misses or accesses cannot be used to distinguish attacks and the normal operation for both target algorithms. In the particular case of L3 accesses, it could be used for RSA but not for AES. However, the L2 instruction misses counter, could distinguish between attacks and non-attacks for both algorithms. Note that if the sampling rate of the attack is reduced, the number of L2 misses would similarly be reduced. As a solution, this value could be normalized with respect to the total number of instructions executed.

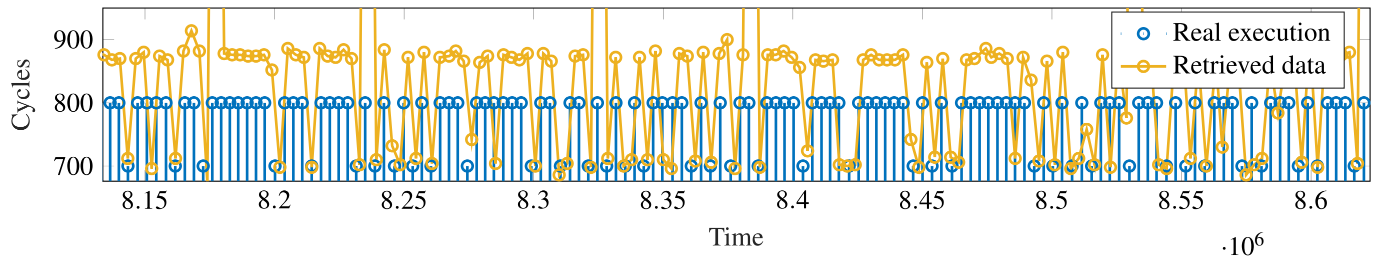


Figure 9: Example of a retrieved trace referred to an execution of a RSA decryption. The blue bars represent the real execution of squares (points equal to 800) and multiplies (700). The yellow line represents the information retrieved, low reload times mean detection of the Multiply execution.

Table 5: Mean and variance of the different counters collected during the execution of the attacks against AES and RSA. The results were obtained using 10000 samples collected each 100 μs for each scenario.

	Cycles	**L3 misses**	**L3 accesses**	**L3 reads**	**L2 instruction misses**	**L2 accesses**
AES Normal	152000±750	0±0.1	714±62	697±63	31±7	2050±74
AES R+R	**148000±634**	**0±0.1**	**713±60**	**702±61**	**212±15**	**2035±114**
AES F+R	184000±3000	10±2	676±65	676±66	186±12	2000±92
AES P+P	158000±1200	19±3	452±140	451±139	209±18	1810±95
RSA Normal	336000±11000	14±26	100 ±206	99±204	7±12	139±316
RSA R+R	**374000±38000**	**26±14**	**233±114**	**231±113**	**60±23**	**308±163**
RSA F+R	364000±50000	127±38	200±110	199±108	97±54	285±240
RSA P+P	363000±55000	112±61	177±159	174±156	78±83	211±222

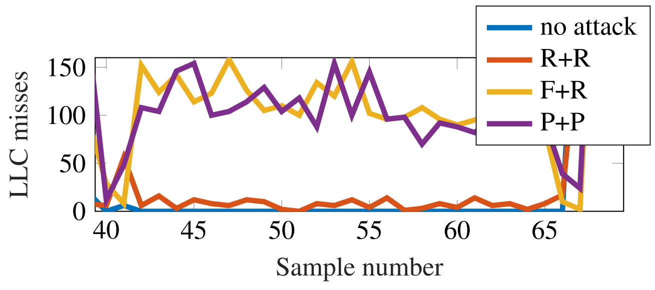


Figure 10: Detail of a trace of misses measured each 100 μs for each of the approaches.

We can conclude that RELOAD+REFRESH changes the performance of the system in an observable way in the low level caches only. Consequently, counters referring to the LLC are not enough to detect RELOAD+REFRESH. Then, the assumption of previous detection mechanisms [10,13,64] that LLC misses or accesses reveal the attacks does not hold for RELOAD+REFRESH. Existing detection systems thus need to be adapted or re-trained to include additional information about low level cache events if they want to be able to detect it. However, relying on low level cache events to detect the attack can be tricky, since it is unknown how benign applications that share the machine with the victim affect it. Therefore, further analysis must be conducted to build a reliable detection system.

7 Discussion of the results

The absence of randomness in the replacement algorithm makes it possible to accurately determine which of the elements located in a cache set will be evicted in case of conflict. Also, the accurate timers included in Intel processors, altogether with the *cflush* instruction, allow to trace accesses to the different caches and to force the cache lines to have the desired ages. We exploit these facts to run RELOAD+REFRESH. In turn, the fact that RELOAD+REFRESH works as expected, confirms some of our results about the replacement policy.

RELOAD+REFRESH is just one way to exploit the eviction policy assuming some kind of memory sharing mechanism enabled. In the case that the victim and the attacker do not share memory, our attack can be prevented. It could be prevented as well with some other general countermeasures against cache attacks that limit the sharing of resources. However, as mentioned in Section 4, RELOAD+REFRESH can be adapted to work in the absence of shared memory. We did not further explore this attack variant, as it requires to keep the replacement policy in Mode 2, which is also not available on the newest Intel processors.

The knowledge of the eviction policy enables the usage of a different access pattern to gain the information about the victim and to ensure that its data is really evicted from the cache, reducing the amount of false positives. Thus, PRIME+PROBE attacks, EVICT+RELOAD attacks or any attack requiring to evict some data from the cache can benefit from our results. For instance, the PROBE step can, in some cases, be reduced to just one access to the eviction candidate.

8 Conclusion

This work presented a thorough analysis of cache replacement policies implemented in Intel processors covering from 4th to 8th generations. To this end, we have developed a methodology that allows us to test the accuracy of different policies by comparing the data that each policy selects as the eviction candidate with the data truly evicted after forcing a miss.

The RELOAD+REFRESH attack builds on this deep understanding of the platforms replacement policy to stealthily exploit cache accesses to extract information about a victim. We have demonstrated the feasibility of our approach by targeting AES and RSA and retrieving as much information as we can retrieve with other state-of-the-art cache attacks. Additionally, we have monitored the victim while running these attacks to confirm that our attack causes a negligible amount of last level cache misses, rendering it impossible to detect with current countermeasures. Similarly, we show that events in the L1/L2 caches can reveal the attack and should be considered in detection systems. RELOAD+REFRESH underlines a flaw on such systems; they are limited and they do not scale.

These results are not only useful for broadening the understanding of modern CPU caches and their performance, but also for improving previous attacks and eviction strategies. Our work also demonstrates that new detection countermeasures have to be designed in order to protect users against RELOAD+REFRESH.

Acknowledgment

We thank our anonymous reviewers and our shepherd Daniel Gruss for their valuable comments and constructive feedback. This work was in part supported by DFG under Grant No. 427774779 and by the EU (FEDER), the Spanish Ministry of Economy and Competitiveness, under contracts AYA2015-65973-C3-3-R and RTC-2017-6090-3.

References

[1] A. Abel and J. Reineke. Measurement-based modeling of the cache replacement policy. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 65–74, April 2013.

[2] A. Abel and J. Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–142, March 2014.

[3] Onur Aciiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, Cryptography and Coding'07, pages 185–203, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'07, pages 225–242, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[6] Onur Acıiçmez and Werner Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and its Demonstration on OpenSSL. In *Topics in Cryptology–CT-RSA 2008*, pages 256–273. Springer, 2008.

[7] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-vm attack on AES. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 299–319, 2014.

[8] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *OLS '09: Proceedings of the Linux Symposium*, pages 19–28, July 2009.

[9] Sorav Bansal and Dharmendra S. Modha. Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[10] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 224–235, New York, NY, USA, 2018. ACM.

[11] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M. Moya. Cache misses and the recovery of the full aes 256 key. *Applied Sciences*, 9(5), 2019.

[12] Samira Briongos, Pedro Malagón, José L. Risco-Martín, and José M. Moya. Modeling side-channel cache attacks on aes. In *Proceedings of the Summer Computer Simulation Conference*, SCSC '16, pages 37:1–37:8, San Diego, CA, USA, 2016. Society for Computer Simulation International.

[13] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162 – 1174, 2016.

[14] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, Vancouver, BC, 2017. USENIX Association.

[15] Fangfei Liu and Yuval Yarom and Qian Ge and 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, Washington, DC, USA, 2015. IEEE Computer Society.

[16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, Apr 2018.

[17] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, April 1998.

[18] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 300–321, Berlin, Heidelberg, 2016. Springer-Verlag.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[21] 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, Washington, D.C., 2015. USENIX Association.

[22] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[23] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, 2016.

[24] Intel. Intel® 64 and ia-32 architectures optimization reference manual (section 2.1.1.2), 2017. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[25] G. Irazoqui, T. Eisenbarth, and B. Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design (DSD)*, volume 00, pages 629–636, Aug. 2015.

[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 *36th IEEE Symposium on Security and Privacy (S&P 2015)*, pages 591–604, 2015.

[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Preventing microarchitectural attacks before distribution. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 377–388, New York, NY, USA, 2018. ACM.

[28] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 85–96, New York, NY, USA, 2015. ACM.

[29] F J. Corbato. A paging experiment with the multics system. page 20, 07 1968.

[30] S. Jahagirdar, V. George, I. Sodhi, and R. Wells. Power management of the third generation intel core micro architecture formerly codenamed ivy bridge. In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–49, Aug 2012.

[31] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[32] 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, Bellevue, WA, 2012. USENIX.

[33] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, June 2014.

[34] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[35] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 463–480, Austin, TX, 2016. USENIX Association.

[36] Yusuf Kulah, Berkay Dincer, Cemal Yilmaz, and Erkay Savas. Spydetector: An approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, Jun 2018.

[37] Peng Li, Debin Gao, and Michael K Reiter. Stopwatch: a cloud architecture for timing channel mitigation. *ACM Transactions on Information and System Security (TISSEC)*, 17(2):8, 2014.

[38] 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, Austin, TX, USA, August 10-12, 2016.*, pages 549–564, 2016.

[39] 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 *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[40] 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, March 2016.

[41] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 203–215, Washington, DC, USA, 2014. IEEE Computer Society.

[42] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

[43] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, pages 46–64, Berlin, Heidelberg, 2015. Springer-Verlag.

[44] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 48–65, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[45] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[46] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[47] 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 *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[49] Mathias Payer. Hexpads: A platform to detect "stealth" attacks. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings*, pages 138–154, Cham, 2016. Springer International Publishing.

[50] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, 2016. USENIX Association.

[51] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[52] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212, 2009.

[53] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom. The 9 lives of bleichenbacher's cat: New cache attacks on tls implementations. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 967–984.

[54] Mark Seaborn. Exploiting the dram rowhammer bug to gain kernel privileges, March 2015. `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`.

[55] Inc. Sun Microsystems. Ultrasparc t2 supplement to the ultrasparc architecture 2007. Draft D1.4.3, Sep 2007.

[56] 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, San Diego, CA, August 2014. USENIX Association.

[57] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54. IEEE, 2019.

[58] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[59] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 161–173, New York, NY, USA, 2018. ACM.

[60] Henry Wong. Intel Ivy Bridge cache replacement policy, jan 2013.

[61] 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, 2014.

[62] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.

[63] Andreas Zankl, Johann Heyszl, and Georg Sigl. Automated detection of instruction cache leaks in modular exponentiation software. In Kerstin Lemke-Rust and Michael Tunstall, editors, *Smart Card Research and Advanced Applications: 15th International Conference, CARDIS 2016, Cannes, France, November 7–9, 2016, Revised Selected Papers*, pages 228–244, Cham, 2017. Springer International Publishing.

[64] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. *CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds*, pages 118–140. Springer International Publishing, Cham, 2016.

[65] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 990–1003, New York, NY, USA, 2014. ACM.

[66] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 871–882, New York, NY, USA, 2016. ACM.