Aim, Wait, Shoot: How the CACHESNIPER Technique Improves Unprivileged Cache Attacks

1st Samira Briongos
NEC Laboratories Europe
samira.briongos@neclab.eu

2nd Ida Bruhns
Universität zu Lübeck
ida.bruhns@uni-luebeck.de

3rd Pedro Malagón
Universidad Politécnica de Madrid
malagon@die.upm.es

4th Thomas Eisenbarth
Universität zu Lübeck
thomas.eisenbarth@uni-luebeck.de

5th José M. Moya
Universidad Politécnica de Madrid
josem@die.upm.es

2021 IEEE European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1491-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/EUROSP51992.2021.00051

***Abstract*—Microarchitectural side channel attacks have been very prominent in security research over the last few years. Caches proved to be an outstanding side channel, as they provide high resolution and generic cross-core leakage. All major cryptographic libraries provide countermeasures to hinder key extraction via cross-core cache attacks by now. In this paper, we analyze implementations protected by prefetch-based countermeasures aimed at preventing well-known cache attacks, and highlight the circumstances causing them to remain vulnerable. Further, we craft a novel attack technique that precisely synchronizes the attacking and the victim processes, enabling the attacking process to evict the target data from the cache at the desired instants. One key improvement of our approach is that it provides unprivileged attackers with a method to remove specific data from the cache with a single memory access and in absence of shared memory by leveraging the transient capabilities of TSX and relying on the L3 replacement policy. We show the feasibility of our approach by extracting an RSA key from the latest wolfSSL library and an AES key from the T-Table and S-Box implementations included in OpenSSL with CACHESNIPER. Both libraries implement prefetch-based methods as a protection against cache attacks.**

1. Introduction

In the age of cloud computing and online services, multiple processes run simultaneously on shared hardware. For example, many tenants can run their virtual machines on a single host. The execution of any application interacts with the microarchitectural elements of the processor, changing their state. Since it is possible to measure and influence this microarchitectural state, it can be used as a side-channel to infer the secrets of a victim process [1]–[8].

Among all the microarchitectural elements that can be exploited to break security assumptions, such as the isolation between processes, the cache memory plays one of the most significant roles. It is a shared resource that provides fine-grained temporal and spatial information. Cache attacks have been very prominent in research over the last few years. They target cryptographic implementations, retrieving ECDSA, RSA and AES keys, and break the isolation between virtual machines (VMs) [9]–[17]. Other attacks infer keystrokes, spy on user behavior, steal SGX enclave keys and many more [18]–[20].

As a direct response to the threat imposed by microarchitectural attacks, many different countermeasures have been proposed. Preemptive countermeasures try to help in the hard task of designing leakage free code [21]–[24]. Hardware based countermeasures either design or take advantage of hardware features to avoid the leakage. Finally, detection based countermeasures accept that vulnerable applications exist and try to determine whether there is an attack going on by analyzing the state of the system [25]–[30]. In this paper, we will focus on the preemptive countermeasure of *prefetching*.

Prefetching or cache warming is used as a strategy to improve performance as well as preventing attackers from observing the cache state in cache attacks [12], [31]–[34]. For example, a prefetching strategy was included in the AES S-Box implementation of OpenSSL 1.0.0a and beyond: The S-Box is loaded into the cache before executing each round [35], [36]. If the attacker tries to retrieve information from the cache lines holding the S-Box during the execution of any intermediate round or after the encryption, she will only observe cache hits. Consequently, she would not be able to distinguish whether those accesses occurred due to the actual utilization of the line or due to the load in the prefetch stage [35].

While prefetching as a countermeasure against cache attacks may not be generally considered completely effective by researchers [37], it is still widely deployed in real-world applications. To the best of our knowledge, no previous work has proved that it can be completely bypassed by unprivileged cache attacks. Many prominent and current publications have either exploited the intra-core resource sharing of simultaneous multithreading (SMT) [8], [38]–[40] or the ability of privileged attackers to interrupt the execution of SGX enclaves to attack different cryptographic implementations [41]–[44]. While these scenarios match a cloud setting or a trusted execution environment threat model, these attackers are so powerful that several cryptographic libraries, including OpenSSL, now ignore them: The cost of protecting against them is exuberant and arguably not justified in scenarios

This research was supported by DFG (Projects 427774779 and 439797619) and by the Spanish MINECO under grant PID2019-110866RB-I00.

DOI 10.1109/EuroSP51992.2021.00051

where the attacker already controls the OS (which may simply be worse than the attacks) [45]. On the contrary, CACHESNIPER requires no elevated user privileges or system setup, and thus poses a real threat to running cryptographic service using state of the art libraries.

Contribution. In this work, we show that a classic user-level cache adversary can overcome a prefetch protection by solving four challenges: (1) The attacker needs to be able to *detect* when the victim is running the target algorithm. (2) She has to determine the time window between the detection of the target algorithm and the utilization of the target data. (3) At exactly the right instant, she needs to *evict* the target data from memory. (4) As in previous attacks, she then needs to *recover* the information about a potential access of the victim algorithm. Our work contributes the following:

- We analyze different methods to tackle these challenges and evaluate which work best by testing with a synthetic benchmark.
- We then use the outcomes of the analysis to craft CACHESNIPER, a novel attack technique that allows a user level attacker to leverage tiny windows of opportunity.
- CACHESNIPER achieves high precision by combining a TSX transaction to precisely determine the victim process's state, the corresponding abort handler to directly conduct the attack, and fast and accurate eviction techniques to get data from the cache.
- We show in realistic experimental setups the feasibility of side channel last-level cache attacks against prefetch-protected implementations by retrieving keys from AES and RSA implementations, both from real world libraries.

We demonstrate the success of CACHESNIPER by retrieving keys from protected AES and RSA implementations, both from real world libraries. This shows that last-level cache attacks against protected implementations are still possible even without special privileges. The code for CACHESNIPER and our benchmark experiments is available at https://github.com/greenlsi/CacheSniper.

Disclosure. We responsibly disclosed to both wolfSSL and OpenSSL on June 22nd and 23rd respectively. OpenSSL did not issue a CVE since CACHESNIPER falls outside their threat model [45], eventually the communication stopped after we analyzed their proposal of an alternative AES software implementation. WolfSSL immediately issued CVE-2020-15309 and proposed a fix for the vulnerability, which we tested and acknowledged. It will be part of the next wolfSSL release.

2. Preliminaries

This section introduces some basic concepts on cache memory, cache attacks and transactional execution that are of key importance in order to understand the proposed technique and its differences to previous approaches.

2.1. Cache architecture

Caches are small memory blocks located between the processor and the main memory, specially designed to reduce the gap between processor and memory throughput. Modern processors include caches that are hierarchically organized; low level caches (L1 and L2) are core private, smaller and closer to the processor (with reduced latency), whereas the last level cache (LLC or L3) is bigger and shared among all the cores. Intel processors traditionally have L3 inclusive caches, in order to simplify the implementation of cache coherency: All the data which is present in the private low-level caches has to be in the shared L3 cache.

Most modern processors include w-way set-associative caches; a trade-off between directly mapped caches, usually with high cache miss rates, and fully associative caches, with a very complex logic. The cache is organized into multiple sets (s), each of them containing w lines of usually 64 bytes of data. Many caches additionally group the sets into slices. The location of each memory block is derived from its physical address. The address bits are divided into offset (usually the lowest-order 6 bits used to locate data within a 64 byte line), index ($\log_2(s)$ consecutive bits starting from the offset bits that address the set) and tag (remaining bits that identify whether the data is cached). The slice number is computed by a hash function f, which usually depends on some fixed bits of the data. Slice selection mechanisms are often not public, and effort has gone into reverse engineering them [46].

There is a noticeable timing difference between data retrieved from the cache hierarchy (cache hit) and data that has to be fetched from main memory (cache miss). In the event of a cache miss, the new block has to be placed in the cache. In this case, the *replacement policy* decides which block is evicted from the cache set and thus, the location of the new block within the set. A good replacement policy is crucial for achieving good performance and most manufacturers do not publish the implementation of their replacement policies. In the case of Intel, their latest replacement policy is known as "Quad-Age LRU" [47]. There have been several efforts to gain more insight into the replacement policy of Intel processors [48]–[50] or to study eviction strategies in order to improve cache or memory fault injection attacks [51], [52]. Recently, more concrete details about the replacement policy of all the cache levels of modern Intel processors have been published [53]–[55]. These works highlight that Intel's LLC replacement policy is deterministic: The eviction candidate depends on the location of the data within the set and the accesses to the blocks in the LLC. If data is loaded from the L1 or L2 cache, the LLC eviction candidate does not change.

2.2. Cache attacks

Cache memory was first mentioned as a side channel in 1992 [56]. Since then, many different techniques have been developed: Osvik et al. proposed the widely known *Evict+Time* and *Prime+Probe* attacks, revealing the cache sets accessed by the victim, and Gullasch et al. and Yarom et al. developed a powerful attack that exploits shared memory, which was later named *Flush+Reload* [57]–[60]. From these attacks, the latter two are widely used. *Flush+Reload* is popular due to its high resolution and accuracy and *Prime+Probe* has very low requirements regarding the attack scenario.

The *Flush+Reload* technique requires shared memory, which means that the victim and attacker use the same data during their respective execution. This can be met by both using the same shared library, which is often the case for libraries shipped with the operating system. The attacker uses one instruction, such as `clflush` in Intel processors, to *flush* the desired lines from the cache, making sure the victim process needs to load them from memory to use them. She then waits for a certain period of time, giving the victim process time to execute. Then the attacker reloads the data, measuring the time this takes. If the victim process used the data, the reload time observed by the attacker will be short. This attack is easy to implement and provides precise information about the data the victim process uses at cache-line granularity. As for cryptographic implementations, this attack has successfully retrieved AES, RSA and ECDSA keys [12], [15], [59], [60]. After some successful attacks [12], [16], [61], [62] showed how to recover secret information from co-resident VMs, cloud vendors realized shared memory poses a security risk and disabled it on their machines.

Lacking shared memory or a flush instruction, e.g. when attacking from JavaScript [63] or in the aforementioned cloud scenario, an attacker can leverage *Prime+Probe* to extract sensitive information [9], [10]. *Prime+Probe* does not require special OS features, so it can be applied to virtually any system. As a preparation step for a *Prime+Probe* attack, the attacker needs to construct an eviction set (a group of w different addresses that map to one specific set in w-way set-associative caches). Constructing eviction sets and dealing with missing address information as well as slice selection mechanisms has been discussed extensively in the literature [9], [14], [17], [52]. Both the *Flush+Reload* and *Prime+Probe* techniques require precise timers, thus limiting the attacker's capacity to access these timers was considered a valid countermeasure. This notion was proven incorrect by Disselkoen et al., who designed a timer-less attack that exploits TSX to retrieve the same information as *Prime+Probe* attacks [64].

2.3. Transactional memory and Intel TSX

Intel TSX is an instruction set extension for x86 that supports Transactional memory and is available on several CPUs starting with the Haswell microarchitecture. Transactional memory enables optimistic execution of the transactional code regions specified by the programmer. The processor executes the specified sections assuming that there is no conflict with other threads or CPU cores, which might access or modify the same data. Transactional memory reduces the need of mutual exclusion mechanisms, using a local version of data and registering a hardware-based callback mechanism in case a conflict with other threads is detected. If the execution ends successfully, the processor commits all the changes as if they had occurred atomically, becoming visible to the remaining processes. Otherwise, the transaction is canceled, all memory changes are discarded and a callback function is called. This process is known as an *abort*, and the callback is known as an *abort handler*.

There are various reasons why a transaction may abort in Intel TSX, but we particularly focus on the cache

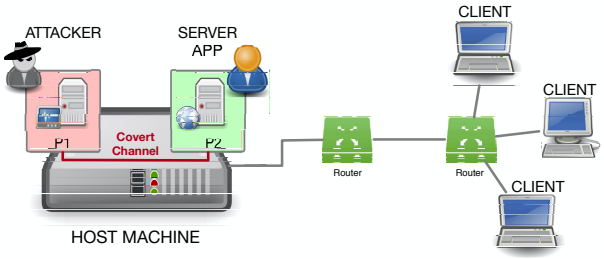

Figure 1. Diagram of the considered scenario for the attack against protected cryptographic implementations. The only connection between the attacker and the victim is the shared hardware.

related ones. Namely, a transaction aborts if data from its "write set" is evicted from the L1 cache or if data from its "read set" is evicted from the L3 cache [64], [65]. These properties of the transactions have been successfully exploited to carry cache attacks in cases where timers are not available [64] or to break kernel address space layout randomization (KASLR) without generating interrupts from the operating system [66], [67]. They have been leveraged to protect processes against cache attacks [68] or even to prevent data input modification and thus protect against the exploitation of double fetch bugs [69].

As a result of the discovery of the TSX Asynchronous Abort (TAA) vulnerability [4], [70], different mitigations were issued [71], [72]. TAA is similar to Microarchitectural Data Sampling (MDS) and affects the same buffers. It exposes data from either the current logical processor or from the sibling logical processor when certain loads speculatively pass that data to dependent operations while the asynchronous abort condition is pending in a transaction. The MDS mitigation helps address the TAA vulnerability, and starting with 8th and 9th Generations of Intel Core processors it is mitigated in hardware [72]. Not all CPUs are affected and therefore, not all of them need mitigation, if they do, two options are possible; either TSX is disabled or the CPU buffers are cleared. If TSX is disabled, root privileges are required to enable it [71], [72].

In this work, we further explore the capabilities of TSX for detection and synchronization of processes. This makes TSX an enabler for some cache attacks against implementations protected by prefetch-based countermeasures. Note that we focus on the LLC, thus clearing the buffers does not affect our approach.

3. Attacker scenario

The scenario we consider for the analysis of the effectiveness of the countermeasures and to carry out the different experiments is depicted in Figure 1. It tries to be as realistic as possible and includes the following agents:

Server Executes the target process (encryption/decryption) upon a request from a client. While this is a simplified version of a real server process, it is enough to create a realistic scenario for the attacks.

Client Sends requests to the server every 500μs plus a random time $\Delta \in [0..100\mu\text{s}]$. This process tries to emulate the behavior of a real network.

Attacker Monitors the cache to detect the exact times when the target process (an encryption process) is running so it can launch a precise attack.

```
1:
2: function VICTIM_FUNCTION
3:     ⋮
4:
5:
6:     load table[secret]
7: end function
```

```
1:              ▷ Eviction target T1
2: function VICTIM_FUNCTION
3:     ⋮
4:     load table
5:              ▷ Eviction target T2
6:     load table[secret]
7: end function
```

Figure 2. Example for (a) vulnerable and (b) protected algorithm

Our main assumption is that the attacker and the victim are using the same machine. The attacker has user-level access to the server and no special rights or privileges, She does not receive any input from the clients referring to the instants they make the requests. She cannot physically interfere with the machine or slow down or stop the victim process. As our attack targets the LLC, the attacker and victim may run on different physical cores.

4. Protected implementations as target

In this work we consider applications that prefetch data in the cache or implement always-load strategies as a countermeasure against cache attacks. The idea of both approaches is the same: The developer tries to ensure that an attacker cannot distinguish between data that is in the cache because the application loads it as a precaution and data that is in the cache because it is actually used.

4.1. Prefetch protected implementations

As explained in Subsection 2.2, an attacker in a cache side channel attack tries to infer secrets from observed cache access patterns. Prefetching helps to obliterate these patterns by loading data or instructions regardless of their actual utilization. When the time elapsed between prefetch and the vulnerable access is actually lower than the time required to measure the cache state, it makes virtually impossible for an attacker to discern whether the observed access is due to the prefetch or due to the (subsequent) secret-dependent use by the victim.

To illustrate this idea we provide a simple example scenario in Figure 2. The victim runs the function *victim_function*, which executes some operations before performing a secret-dependent access to a table. Assuming *table* spans several cache lines, the attacker can gain information about the secret by observing which cache lines are accessed by the victim process. In the protected version, the entire table is loaded into the cache in line 4 before the secret dependent access is performed in line 6. Thus, each cache line spanned by the table is accessed at least once, independently of the secret.

As a more realistic example, in the case of an unprotected AES T-Table implementation, the attacker could easily remove one line of the table from the cache, let the whole encryption run, and later test the cache. Since the probability of using that line is around 92%[1], the information about its actual utilization leaks information about the secret key. On the contrary, as stated by a Red Hat security blog the probability of not using one line of the S-Box implementation should be 10^{-20}. In the particular case of OpenSSL, such probability is indeed zero because the S-Box is fully prefetched before the execution of each round [73]. As a result, they later conclude that this implementation should not be vulnerable to cache attacks.

Note that the existence of this prefetch does not eliminate the existence of leaky patterns in the code. However, applications protected this way are still considered secure against user-level cache attacks for the following reasons:

- If there is no shared memory, the *Prime+Probe* attack will require to probe the whole set to evict the prefetched data. This operation sometimes takes longer than the execution of the protected application.
- If there is shared memory, a *Flush+Reload* attack monitoring the cache will detect the presence of the data in the cache with high accuracy, but it lacks a mechanism to infer whether this observation is due to the prefetch or to the actual utilization of the data.

Mainly, in both cases, the temporal resolution of the attacks is usually assumed to be too low to distinguish the prefetch access from the subsequent secret-dependent access. This is particularly true in the common synchronous attack scenario where the attacker evicts the observed target, then lets the victim run the encryption, and then accesses the observed target again to see if it is in the cache, or when the spy process continuously observes the cache seeking for consecutive high frequency accesses.

Since the protected code theoretically still leaks information, the objective of this work is to study and analyze whether a regular attacker can exploit that code, and under which circumstances. For this analysis we have designed a synthetic benchmark that tries to mimic one of these protected applications. We provide further details about this set of experiments in Section 5.

4.2. Attack challenges

In a cache attack against an implementation without prefetching, it is usually sufficient to remove the target data from the cache before the victim process starts executing, then checking for the target data in the cache after the victim process terminates. Measuring close to the end of the victim process will reduce noise, but the exact point of the observation is not critical. This attack scheme does not work for implementations that use prefetching.

To overcome the prefetch-based countermeasure, timing is everything. The attacker needs to evict the data precisely *after the prefetch*, but *before utilization*. This corresponds to between line 4 and line 6 in our example. Since we assume no synchronization between the victim and the attacker, the attacker does not know when the victim is running the targeted application, neither does she know when data is prefetched in the cache. Thus, in order to evict the data at exactly the right instant, the attacker needs to solve the following challenges:

1) Detect the victim's execution of the target algorithm.
2) Determine the state of the target after detection.
3) Calculate the remaining time until data is prefetched.
4) Evict the target data from the LLC at the desired instant.

To clarify this idea, we use the example given in Figure 2: The attacker would try to detect the execution

1. The probability of using a single line from the T-Table increases depending on the number of rounds the AES uses, which in turn depends on the key size and it is given as $(1-(16/256))^{Rounds*4}$.

of the function *victim_function*. Since the victim process continues to execute during the detection, a certain amount of time will have passed until the attacker actually knows the target function is running. To tackle the third challenge, she then needs to determine where in the target function the victim is, and thus how much time will pass until the prefetch in version (b) line 4. Last she needs to find a way to evict the data and retrieve the information.

5. Overcoming the challenges

To find out whether an attacker can gain any information from protected applications, we design a minimal worst-case application that implements the countermeasure. It is based on the example we have been using so far, so we continue the pseudocode in Figure 2 version (b). With this application, we can

- control the number of operations executed since the beginning of the application until the prefetch.
- decide whether the data is used after the prefetch and store that information.
- collect information about the exact instants when the process starts, prefetches the data and when it ends.

Once our test code runs, it executes a variable number of *xor* operations (line 3) before prefetching a table spanning four cache lines (line 4). This prefetch is followed by a single access, where a secret bit `secret` determines which of the four cache lines is accessed a second time (line 6). Note that this example shows the worst case scenario for the attacker: the secret related data is accessed immediately after the prefetch, so the size of the time window during which the attacker could evict data from the cache to gain information is minimal. Other scenarios with bigger tables and additional computation will increase this window and simplify the attack.

For the posterior analysis, we also collect timestamps at different points of the execution of this application. Namely before executing the first instruction of the function (line 2), before the prefetch (line 4), and once the function has completed its execution (line 7). This allows us to reason about which of the existing approaches is more accurate for detection, about the information referring to the state of the application we have after detection and about the requirements for the posterior eviction of the target that could allow exploitation of the leakage.

5.1. Detection approaches

We study *Flush+Reload*, *Prime+Probe* and a TSX-based approach to detect the execution of the victim's function *victim_function*. Once this is the case, *victim_function* will appear in the cache. Thus, we monitor the cache line that contains the call to the function itself. We call this the *target for detection*, or `T1`.

The *Flush+Reload* [60] technique is considered one of the most reliable sources of information in cache attacks, especially when compared to the noisier and slower *Prime+Probe* technique [9], [12]–[14]. The main reason is that, in the former case, observations are made on a shared memory block, whereas in the latter case, any other process running in the machine could force an eviction and the attacker would not be able to distinguish the origin of the eviction. When using the *Flush+Reload* technique we repeatedly flush `T1` from the cache, wait for around 20 cycles (this value was empirically determined for our scenario, values lower than 20 cycles reduce the detection rate) and then reload again. Once the reload time indicates `T1` has been retrieved from the cache instead of from the main memory, we know the victim has started the execution. We then take note of the detection time.

For both *Prime+Probe* and TSX-based detection, first of all we have to build an eviction set A mapping to the same sets as our detection target `T1`. We enabled hugepages to construct eviction sets as Liu et al. did in [9], although it is possible to use the reverse-engineered mapping function [46], [74] or a different approach that does not require the use of hugepages [52], [75]. The way eviction set are built does not have an impact on the detection. Assuming that hugepages are enabled in the server simplifies the construction of the eviction sets without changing the attacker model for the actual attack.

Due to the pseudo-LRU replacement policy implemented in Intel's L3 cache [53], [55], always accessing the elements in our eviction set in the same order ensures that we are always about to access the block that our application will evict as soon as it starts to run. This is easily implemented through pointer-chasing. It also means when using *Prime+Probe*, we can avoid accessing the whole set and just measure the access time after accessing one of the blocks in the eviction set, which is faster. Once we detect a cache miss we assume the victim has begun its execution and collect that timestamp for the posterior analysis. Since we accessed all blocks in the current set recently, this also allows us to estimate the ages of the blocks at this point.

Finally, for TSX-based detection we leverage the fact that, whenever a process is running inside a TSX transaction, the process can either be completed and commit the results or suffer an abort when suffering a L3 cache miss and rollback the computations. The *Prime+Abort* attack [64] deliberately causes conflicts in the L3 cache, leading to an abort of the attacker's process, to determine whether the victim process has used certain data. This results in a timer-free attack similar to *Prime+Probe*. We assume we have access to timers and use the abort as a signaling mechanism instead. We access all the elements in the eviction set during the transaction and, in case it aborts, we consider that the victim process has started and use the abort handler to store the timestamp.

In all cases, we collect information referring to 100,000 executions of the target process. We collect information on both sides, victim and attacker. The experiments were performed on a Intel core i5-7600K, the details of the platform are summarized in Table 2 ind the Appendix.

The correctly detected executions are: 99% for *Flush+Reload* 94,6% *Prime+Probe* and 97,3% for TSX. As previous researchers have shown, the *Flush+Reload* approach shows great accuracy [69], [76]. Also the results obtained for both TSX and *Prime+Probe* are good enough for detection purposes. Note that in the case of TSX the undetected executions are most likely due to the situations where the data is being loaded inside the transaction while the victim is already running.

While the detection capability is important, it is not the main requisite for carrying out the precise attack that

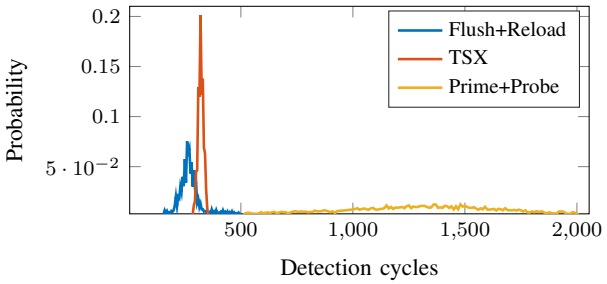

Figure 3. Histogram of the times elapsed between the beginning of the execution of the victim process that takes 780 cycles in mean to execute, until it is detected with the different approaches

is needed to infer secrets from an implementation with prefetching. To achieve accurate evictions, we additionally require the detection time to be almost constant. That means we also need to check the time elapsed between the start of the target execution and the detection and the time between the detection and the prefetch for the correctly detected processes. Figure 3 shows the histogram of the detection times compared with the beginning of the execution of the target. Our sample application runs in almost constant time (about 600 cycles). Even though the detection causes a cache miss, the execution time does not show great variance and stays around 780 cycles. Times elapsed between detection and prefetching show similar behavior: both distributions are complementary. The exception is the *Prime+Probe* approach: most of the times the detection time exceeds the execution time (780 cycles) and thus we could not evict our target later.

Thus by themselves, *Flush+Reload* or *Prime+Probe* have severe limitations. For instance *Flush+Reload* shows quite a lot of variation for detection, which means low reliability for the posterior eviction. The *Prime+Probe* results not only have a large variation but also too much delay. Apparently, since we are continuously reading from the cache, and accessing the eviction candidate, there are many race conditions between the target and the candidate. Even when detected in time, and although the attacker knows the state of the cache, she does not have any control over it as the victim executes further. As a result, she cannot accurately predict how long she has to wait to evict the target, because this time depends on the location of the target within the set, which in turn determines the number of memory accesses required to evict it. In comparison, TSX only shows a slightly smaller detection rate than *Flush+Reload*. We believe that during the "undetected" executions we were loading the data inside the transactional region concurrently or just after the execution of the victim process, or an unrelated process accidentally evicted our data.

Based on these measurements, we state that the TSX technique accurately informs about the execution of the victim and that the attacker receives the abort as soon as the conflict happens. As a result, we further explore the eviction approaches assuming we can use TSX for detection. The measured times include the time it takes the CPU to retrieve `T1` from the main memory. That is, the transaction aborts once the data has been effectively loaded into the cache and the attacker's data is removed. While a heavy system load leads to additional aborts that are not related to the execution of the victim application, it does not influence the detection time. The TSX-based detection approach by itself solves three of the challenges for the attacker, who this way knows with great accuracy the state of the target process when she detects it. She can additionally use measurements like the ones we performed here to profile the target application on a machine similar to the server. This way, the attacker can determine the time t she should ideally wait to achieve an eviction.

5.2. Eviction approaches

After the detection step, the attacker's code will be executing the abort handler with quite exact knowledge of the cache state. She also knows the wait time t from profiling the target application, and she can use the abort handler to manipulate the cache and gain information. The attacker's objective now is to evict the prefetched data or instructions after the prefetch but before they are used in the execution of the victim process. We call the prefetched data or instructions *target for eviction* or `T2`. In this section, we study two different approaches to evict `T2`. In the following, we introduce method 1 for when there is shared memory between the victim and the attacker, and method 2 when there is not.

As we will show, the methods differ in the way the cache set is filled during the transaction, in the value of the wait time t (even if the target is the same), in the way this target `T2` is evicted from the cache and finally in the way the information about the actual access to the data is inferred. The following subsections explain the particularities of both methods.

5.2.1. Method 1 – Remove `T2` by flushing. As we have already stated, this method relies on the existence of shared memory. Once the victim's access to `T1` is detected, the attacker waits for the wait time t and then evicts the eviction target `T2`. Since there is shared memory, `T2` is simply evicted by using the *clflush* instruction and the information is later retrieved by measuring the time it takes to read it (reload). This is a traditional *Flush+Reload* attack carried out by an abort handler, which allows for very precise timing of the flush instruction: If the prefetch is executed n cycles after detection, then the flush should be triggered at approximately $n - 40$, the minimum value for n being around 60 cycles. Note that since the detection target `T1` has to be loaded from main memory, it introduces some variation in the execution time of the victim and, as a result, this last eviction may not be as accurate as the detection. These values were determined empirically and may vary between machines.

5.2.2. Method 2 – Remove `T2` with memory accesses. If there is no shared memory or the data has to be retrieved from a non shared variable, it is still possible to achieve the desired eviction accurately. The attacker needs to leverage the replacement policy implemented in the LLC. The replacement policy described in previous works [53]–[55] shows that data is linearly inserted into the cache set and each block is assigned an age value that will change depending on accesses and evictions. In this work, the following properties are the most relevant:

- Data is evicted from the cache at age 3.

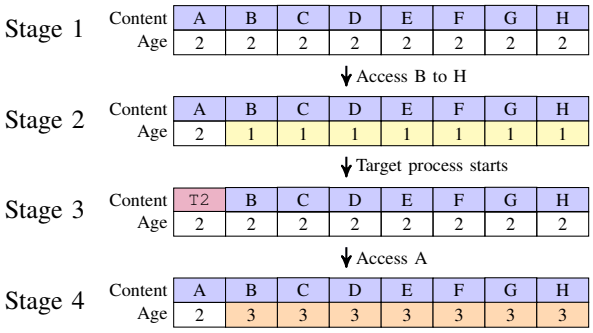

Figure 4. Process required to evict the data from the cache with just one access. This is required for method 2

- Only access to the LLC update the values of the ages of the elements in the LLC. The age of a block decreases when it is accessed or increases if necessary to select an eviction candidate.
- Once the data is inserted into the cache set, two different ages above age 0 can be distinguished.

The attacker has to construct an eviction set B for `T2`. The point in the attack algorithm at which the elements of B are accessed and how they are accessed depend on two time windows: The first one is the time between `T1` and `T2`, while the second window is between the prefetch of `T2` and the utilization of it. The attacker needs to adapt her approach to the size of these windows. We consider that a large window is one that leaves enough time for an entire *Prime+Probe* cycle, while the span of small window is any time less than that.

The most interesting and challenging scenario is the one where the second window is small: It means the attacker has to reduce the time required for the eviction of `T2`. We manage to do so by manipulating the ages of the cache blocks. The method is based on the *Reload+Refresh* attack from [53]. While we use the same technique to age the cache elements, we do not use the known ages of the cache elements to infer a victim access, but to evict `T2` accurately with a single access. The entire process is depicted in Figure 4. The cache is prepared in stage 1 by accessing all elements (A-H) of the eviction set. The attacker then accesses elements B-H again in stage 2 to change their age. That makes A the eviction candidate. In stage 3, the prefetch of `T2` evicts A. `T2` instantly becomes the eviction candidate as it is the first entry with the highest age. Even if the victim uses it more than once, it will retrieve `T2` from the first and second level caches, not changing its age, so `T2` stays the eviction candidate. In stage 4, the attacker evicts `T2` by accessing A.

If the first window is large, stage 1 and stage 2 are performed before the transaction, to avoid detecting `T2` instead of `T1`. If `T1` and `T2` are the same, stage 1 and 2 are performed during the transaction. After the abort, the attacker waits for wait time t, when stage 3 should be present in the cache. She then evicts `T2`.

If we have a small first window in addition to the small second window, stage 1 and stage 2 are performed in the transaction. Note that all the code inside a transaction is executed transiently. While after an abort happens the operations executed are not visible for the program and the previous logical state is recovered, the microarchitectural changes remain: The blocks accessed during the transaction will remain in the cache, and they will keep their ages. After the abort, the attacker waits for wait time t, when stage 3 happens and `T2` is present in the cache.

If the second window is large, the size of the first window is not important and a regular *Prime+Probe* approach to evict `T2` from the cache will suffice.

As we did in the previous case, we have conducted some experiments to compute the minimum time elapsed between the execution of `T1` and `T2` in which the attacker is still able to achieve the accurate eviction with just one access. In this scenario or when `T1` and `T2` are the same, the cache state is assumed to be the one depicted in the stage 2. For this analysis, we have performed some experiments changing the number of operations executed before the prefetch in our test application to test the different times and evaluate the accuracy of the technique. Note that, even when `T2` has to be loaded from main memory for the prefetch and the execution time increases, the time between detection and prefetch only changes slightly.

In these experiments we started observing the leakage for times greater than 260 cycles, but the best results were obtained when these times are greater than 300 cycles. The main reason for the difference between this scenario and the one described in the method 1 is the fact that, in this case, the eviction is only possible when A has been effectively loaded from main memory (cache miss), whereas in the other it is only necessary to wait for the execution time of the flush. This also means that the attacker should access the block A at time $n - 280$ to achieve the eviction at time n.

When trying to force all the elements in the set to get the ages depicted in the stage 2 of Figure 4, we accessed the corresponding blocks B to H as a linked list (to avoid pipeline effects as much as possible). In theory, this way all the data would be retrieved from the LLC and, as a result, the ages of all the elements would be updated. To test this hypothesis, we measured the times it takes to read each of the blocks when accessing them. Surprisingly, the experimental results show that some of them were retrieved from the L1 cache. In fact, we only observed this behavior in a processor whose LLC is 12-way associative (Intel i5-7600K in Table 2), whereas the L1 and the L2 caches are 8-way and 4-way associative respectively. However, we performed the same test in a different processor (Intel i7-6700K) with a 16-way associative cache, and all the data was retrieved from the LLC. This is due to the L1 replacement policy as described in [54], [55]. Reordering the linked list with regard to the L1 replacement policy solved the problem. We include a more detailed explanation to ease the use of CACHESNIPER for other researchers in Appendix A.

5.3. Dealing with noise

The accuracy for detecting the execution of the victim and evicting `T2` from the cache determines the number of samples that the attacker needs to collect. In the detection, the attacker would see false positives if the transaction aborts from a different cause than the execution of the target. TSX gives information about the cause of each abort in the *eax* register. The attacker could use that information to discard some of the traces. However, we

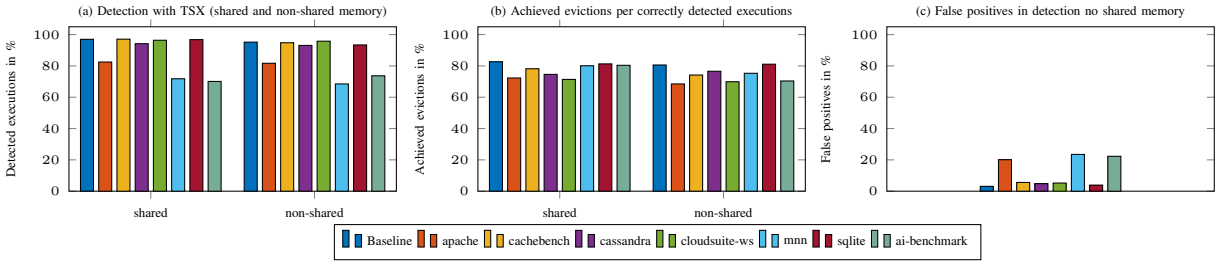

Figure 5. Detection, eviction and false positive rate when running the sample victim code in parallel to different benchmarks to generate noise.

enforced cache conflicts and observed the values of *eax*. Interestingly, the value of *eax* did not always correspond to L3 cache miss. That means by ignoring traces with the wrong value in *eax*, false positives can be lowered, but false negatives may be introduced.

The state of the system, and in particular the CPU load and its memory utilization has an effect on the accuracy of the detection and eviction phases. To evaluate the extend of this effect, we have conducted a series of experiments running different memory intensive benchmarks of the Phoronix benchmark suite [77] in parallel with an attack on our sample victim application described in Section 5 mimicking our client-server-scenario. We executed benchmarks that represent typical server loads, including mnn, apache, cachebench, cassandra, cloudsuite-ws, sqlite, and ai-benchmark. For the experiments we distinguish between the same two cases as in the shoot-phase: there is shared memory or not. We executed the victim process 100 000 times per benchmark. We measured the amount of correctly detected traces, and from these traces the accuracy of the posterior eviction. The results are summarized in Figure 5. All experiments with shared memory use one value of t, the ones without shared memory need another t. Both values are determined beforehand.

If there is shared memory, we only consider a trace valid if `T1` can be detected in the cache, in the abort handler. This means adding an extra step to the attack and implies an adaption of t, but it completely avoids false positives. Without shared memory, we have measured the access time to A at the time of forcing the eviction in the shoot phase (see Figure 7). If this time is low (A is in the cache), then the abort was not due to a relevant cache conflict, and the sample needs to be discarded. If A has been evicted, the attacker considers the sample as valid as she cannot distinguish if the eviction was caused by the victim or by an unrelated process.

As Figure 5(a) shows, detection is equally affected by noise in both scenarios. If there are many aborts the attacker loses detection accuracy. The three benchmarks (apache, mnn and ai) that have the gravest effect on detection and lead to false positives if there is no shared memory (Figure 5(c)) do not only use the memory heavily, but also try to use all the available CPU. These two conditions reduce the attacker capabilities. The eviction is also affected by the noise (Figure 5(b)), but comparatively less than the detection. Note that after detecting the execution of the victim and before the access, another process has to use exactly the same cache set to impede the eviction. As expected, access based evictions are affected by noise more than flush based evictions, but not significantly.

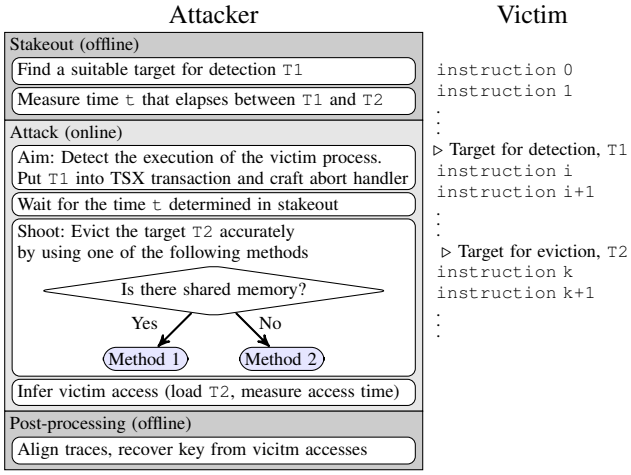

Figure 6. Overview of the offline and online attack steps. The online steps are repeated multiple times to collect sufficient traces for key recovery.

Further analysis of the false positives during the detection phase reveals that they happen in bursts. This is a known phenomenon for some workloads. If these bursts happen to use the same cache regions as the victim, the measurements get noisier. The attacker can identify these situations because the expected distributions of cache hits/misses changes. She can then do one of two things to still launch a successful attack: Take more measurements or wait for a short period, hoping the burst will pass as the noise generating process terminates.

6. CacheSniper: Crafting the attack

We use the knowledge gained from the previous analysis to build an attack. It includes a *stakeout* phase of offline preparation where the attacker has to analyze the victim process. The CacheSniper then *aims* at his target in the detection phase and waits for a good moment in the execution using the information from stakeout. At the right moment, the sniper *shoots* the correct entry from the cache, evicting it accurately. The attack overview including these steps is depicted in Figure 6.

The stakeout phase is similar to the profiling we performed in Section 5. The goals are to find an appropriate region of the code that can serve as target for detection `T1` and to determine the wait time t for the second attack step. The main requirement for `T1` is that it is executed long enough before the target `T2`. It should also not be used too often to avoid false positives. When `T1` has been

determined, the attacker finds the set and slice where it maps and builds the corresponding eviction set.

In addition, the attacker has to determine or estimate the wait time t for the second attack step. This can be done, for example, by instrumenting the victim code and, as we will show later, t can be changed dynamically depending on the expected hit/misses ratio and the observations, which also means this ratio can be used for finding a reasonable value of t. The stakeout can thus be performed offline, although ideally it should be performed on a machine similar to the target machine.

Note that the location of `T1` and `T2` within the cache is determined once the victim process is executed. In case there is shared memory, the attacker could determine the set where the victim code maps using its own data. If not, profiling the cache while running the victim code is required. This has been discussed in many *Prime+Probe* attacks, and the usual approach works fine here. The information from the aborts can also be used to find the cache set of interest [64].

Assuming that the attacker has been able to measure the time it takes the process to execute the code between the point that serves for detection (`T1`) and the point at which the attacker can gain information from the evicted data (`T2`), she has to consider the time it takes to execute the *clflush* on `T2` or to evict the block out of the cache memory by accessing it. That is, the waiting time is then determined by subtracting these times from the measured time in the instrumented code during the stakeout phase.

If the attacker cannot determine the waiting time t in advance or the target system does not behave the same way as the profiling system, she can still determine t online if she knows some characteristics of the process that she can observe. For example, around 1% of cache misses will be ideally observed if we hit exactly the last round of the AES encryption or around 50% of the bits are expected to be 1 in an RSA secret key. In this case, the value for t can be retrieved automatically by analyzing the number of hits and misses observed when inferring the victim accesses (line 10 in Figure 7) and modifying its value accordingly. Even further, the attacker can define an initial value for t and update or adapt it dynamically based on the comparison between the actual and the expected observations at the cost of an increase in the number of samples required to derive the secret information. Recall that this estimation of the value t is valid while the server with the victim code is running.

For example, back to the algorithm shown in Figure 2, we would expect to observe 50% of cache hits if the eviction is accurately achieved. If the eviction is not achieved at the right instant, we see a different ratio. If we force the eviction before the prefetch, then we only see cache hits. If, on the contrary we force the eviction after the data has been actually used, we only see cache misses. Thus, we define an observation window of size w to compute the statistics. Once we have collected the w samples, we observe the hit/misses ratio and either increase or decrease t. The value of t gets stable after some iterations (depending on how bad was the original estimation, and the size of the window). The best scenario turned out to be the one where the value t is properly selected, and only minor adjustments are made on runtime. To do so, we increased the size of the observation window.

```
Input: Address(T2),
    Eviction_set A,                          ▷ Eviction set for T1
    t                          ▷ Waiting time determined in stakeout
Output: X̄0                          ▷ Information about the access

 1: function START_TRANSACTION()
 2:     fill_cache_set(A);
 3:                          ▷ Aim: Abort handler detects the access
 4:     function ON_ABORT()
 5:         time_interrupt=timestamp()+t;
 6:         while(timestamp() ≤ time_interrupt) {};
 7:         evict_from_cache(T2)                          ▷ Shoot
 8:
 9:         Wait until encryption ends
10:         infer_victim_access_to(T2)
11:         if has_accessed(T2) then
12:             X̄0 [t] = 1;                              ▷ Data used
13:         else
14:             X̄0 [t] = 0;
15:         end if
16:     end function
17:     return X̄0;
18: end function
```

Figure 7. Generic attack pseudocode for the TSX-based detection scenario, eviction using method 2, case 2.

The pseudocode in Figure 7 shows the online phase procedure. We generate an eviction set for `T1`, then use it as input to our algorithm. The other inputs are the address of `T2` and the waiting time t which has been determined during the offline preparation.

7. Practical evaluation

In this section we explain how we recover AES and RSA secret keys, demonstrating that not only the implemented countermeasures to prevent cross-core LLC attacks can be circumvented, but also that some previous approaches can benefit from the accurate eviction of the data. All the experiments were performed in the machine described in Table 2. Note that the replacement policy on which this attack relies is implemented in the Intel Core processors starting from the 6th generation.

7.1. CacheSniper against AES

AES [78] is a commonly used symmetric block cipher that operates with data in blocks of 16-bytes. It consists of different operations (*AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns*) that are repeated each round. To speed up execution, tables with values that are used repeatedly are precomputed in many implementations. We analyzed the well-known T-Table implementation and an S-Box implementation. The T-Table is available in OpenSSL 1.0.2k when compiled with the *no-asm* flag, and it is also available in newer OpenSSL versions. If the *no-asm* flag is not used, then the S-Box implementation is used instead. Note that this particular version is the one included by default in our CentOS system, and is shared among all the processes. We have checked that CentOS uses the native OpenSSL implementation for AES.

When OpenSSL is called from the command line, it will use AES-NI for encryption. However, if the C API of OpenSSL is used, a call to *AES_encrypt* would call

TABLE 1. PARAMETERS USED FOR THE ATTACK OF OPENSSL'S PREFETCH-PROTECTED T-TABLE AND S-BOX IMPLEMENTATIONS

Parameter	T-Table	S-Box
Detection target `T1`	AES_encrypt	S-Box (Prefetch in first round)
Eviction target `T2`	Tei[0]	S-Box (After prefetch in last round)
Samples required method 1	300	$\approx$ 500000
Samples required method 2	360	$\approx$ 500000

the S-Box implementation until version 1.0.2k, while a developer wishing to use the default AES-NI instructions (as in the command line) has to use a different instruction to execute the encryption. During our communication with OpenSSL they informed us that they have removed the S-Box from the latest 1.1.1 version. However, we found that in this case the C API calls instead use the even more vulnerable T-Table implementation.

T-Table-based implementation. The T-Table implementation uses four tables (T-Table) with pre-computed values of the *SubBytes*, *ShiftRows* and *MixColumns* operations. That is, it transforms the aforementioned operations into look up operations in order to improve the performance of the encryption and decryption processes. The accesses to the tables are key dependent and not all of them are used during the encryption process. This fact has been exploited multiple times to recover the keys [12]–[14], [62].

All of these approaches assume that the attacker monitors the cache before and after the execution of the victim process. As a result the target cache line, which holds 16 T-Table values, can be used at any round of the encryption. This approach generates false positives, ant the attacks requires mores samples until eventually one of the key candidates can be clearly distinguished from the others. With our CACHESNIPER technique we can accurately hit the last round and recover the secret key faster. The information we collect is more likely to refer exclusively to the last round. Indeed, using the same approach to compute the key that other works suggest [13], we can reduce the samples required to retrieve the whole key from 3000 to 300. The settings for this attack are described in Table 1.

S-Box-based implementation. The S-Box software implementation of AES replaced the previously used T-Table implementation in many cases, not only in the case of OpenSSL. The S-Box is the table that holds the data for the *SubBytes* operation, concretely 256 byte values. As opposed to the T-Table implementation, the S-Box implementation does not merge different operations into one. The S-Box is used 16 times each round. Considering a cache line size of 64 bytes, such table uses 4 cache lines. If we compute the probability of not accessing one of these cache lines and assume a key size of 128 bits, and as a result 10 rounds, such probability is almost equal to 0, as shown in Equation 1 and observed in [73]. As a consequence, an attacker observing the cache before and after the encryption process will not gain sufficient information from that observation.

$$\Pr[\text{no access S-Box in encryption}] = \left(1 - \frac{64}{256}\right)^{10*16} \simeq 0 \quad (1)$$

$$\Pr[\text{no access S-Box in round}] = \left(1 - \frac{64}{256}\right)^{16} = 0.01 \quad (2)$$

In contrast, Equation 2 shows that observing each round individually would give a 0.01 chance of not accessing one of the lines. To enforce this probability to become 0 and relying on an attacker model that cannot interrupt the process after the execution of each instruction, the countermeasure of prefetching was applied: The OpenSSL S-Box implementation includes a prefetch stage before each of the rounds. Since the 256 bytes of the S-Box table map to 4 different cache lines, the encryption process only has to read 4 values to ensure the whole S-Box table is loaded into the cache memory. When the S-Box implementation performs a key-dependent memory access, the data will always be in the cache. As a result, traditional approaches of side channel attacks cannot be used to extract information from this implementation. Irazoqui et al. analyzed the OpenSSL implementation in 2017 with a tool for leakage detection and came to the same conclusion, declaring it leakage free [79].

Nevertheless, this implementation is still vulnerable, since the attacker can accurately time the evictions to happen in the tiny time windows between the prefetch and utilization of the target data (`T2`)[2]. The CACHESNIPER attack, bypasses this way the prefetch and allows an attacker to observe information referring to the last round of the encryption, even in the absence of shared memory. We can thus perform a cross-core cache attack that recovers the secret key of this protected implementation.

In the last round, the output of the S-Box is xored with the corresponding round key to get the ciphertext. In order to retrieve the secret key in all scenarios (method 1 and 2) we use the information referring to the accesses to the S-Box retrieved during the attack phase (output of Figure 7) and assume the ciphertext to be known by the attacker. This is obviously repeated many times with different ciphertexts.

We use the *non-access* approach described in [13]. They use information from cache misses, meaning they track when the victim did *not* load the data into the cache. In this particular implementation, the S-Box is accessed 16 times during the last round, and even if we are able to accurately get the information referring to the last round exclusively, we would not know which of the 16 accesses was responsible for this access. On the contrary, if we determine that an element has not been accessed it means none of the operations in the last round has used it. As a consequence, we xor each byte of the ciphertext with the 64 values of the S-Box held in the cache line ($k_i = C_i \oplus$ S-Box [0 to 63]). None of these values could be the secret key. When observing multiple different encryptions, the key can be inferred by method of elimination.

2. While analyzing the shared library included in Ubuntu 16.04 or CentOS 7.6 (OpenSSL 1.0.2g), we observed an additional protection. The OpenSSL implementation of AES has four different copies of the S-Boxes. If there are, for example, two processes using the library at the same time, each of them will use a different table. This does however not hinder the attack

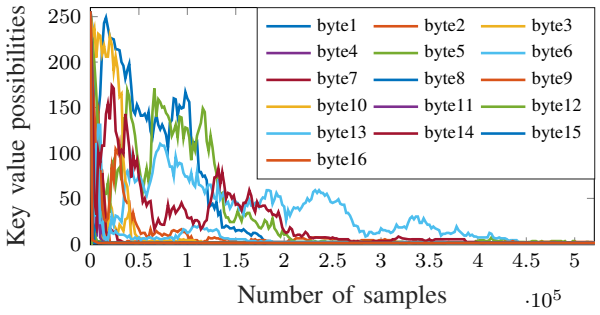

Figure 8. Key candidates for each of the bytes of the key of the S-Box AES implementation of OpenSSL retrieved using the TSX-based detection and method 2.

In our test system the last round takes around 40-50 cycles to execute, with a variance of around 15 cycles. Thus, even if we interrupt the victim at the exactly chosen time, which we estimated in the preparatory step, we may not manage to evict `T2`. Even a slight variation in the interruption instantly leads to a different number of observed accesses. Since we are assuming that the observed data refers only to the last round, evicting once the processor has executed at least one instruction after the prefetch leads to some false positives affecting some of the bytes. Even worse, we see false positives if we hit the cache after the encryption process has ended.

To maximize our success chances despite the varying execution times, we adopt the strategy of dynamically updating the value of t described in Section 6. Instead of using the probability of 1% as the expected one, we allow up to 7% of cache misses. This way we try to ensure that the observed cache misses actually happen in the round. This value was determined empirically on our machine, by selecting different probability values and running multiple experiments with the value of t adapted dynamically according to that probability and window sizes of 10 000 observations.

For this implementation, we can use the content of one line of the S-Box as `T1`, since the data is prefetched in every round. Both method 1 (assumes shared memory) and method 2 (does not require shared memory) are able to successfully retrieve the entire secret AES key with a minimum number of samples of about 500 000. Even with the false positives introduced by all the variances, we get enough information through the 500 000 samples. Our results show that it is more likely to evict the data in the middle of the execution of the last round than just at the beginning. This means some bytes are recovered faster than others, as it can be observed in Figure 8.

Figure 8 shows half of the key bytes has been completely leaked with less than 100 000 samples. Focusing on the different bytes, Figure 8 shows that the initial four bytes are obtained with 10 000 samples, a relatively small amount. Around 12 of the 16 bytes are already known with 200 000 samples. Retrieving the last 4 bytes of the key is the hardest part, and it requires 300 000 more samples. This shows how difficult it is to evict the data in between the execution of the prefetch and the subsequent access in the last round. While completely brute-forcing these 4 bytes takes 2^{32} trials, the information already collected (with 200000 samples) reduces the key space to around

Input: base b, modulo m, exponent $e = (e_{n-1}...e_0)_2$
Output: $b^e \pmod m$
1: init(R);
2: **for** i from $n-1$ downto 0 **do**
3: mul($R[0], R[1], R[e_i]$)); ▷ Load R[0] and R[1]
4: modRed($R[e_i]$);
5: ▷ R[2] is a temp variable that avoids the leakage of $R[e_i]$
6: sqr($R[2], R[2]$);
7: modRed($R[2]$);
8: **end for**
9: **return** R;

Figure 9. wolfSSL exponentiation implementation

2^{15} options, making a complete search of the remaining key space faster and stealthier than continuing the attack.

The number of samples required to retrieve the secret key slightly varies between executions and depends on the selected t even if we use the adaptive approach. The location of the S-Box in the cache is also important as some sets are noisier than others.

7.2. CacheSniper against RSA

RSA is the most widely used public key cryptographic algorithm. It considers a public key (n, e) where n is the product of two prime numbers p and q that remain secret, and a private key (p, q, d) where $d \equiv e^{-1} \pmod{(p-1)(q-1)}$. Only the encryption and decryption operations are relevant to understand the attack. For a message m, the ciphertext c is obtained as $c = m^e \pmod n$ and it is recovered with an analogous operation $m = c^d \pmod n$. The decryption, which is the exponentiation operation using the secret key d, is the attack target.

There are multiple ways of implementing this exponentiation [80], [81]. We will focus particularly on the square-and-multiply exponentiation, since the wolfSSL implementation is based on it. The square-and-multiply approach scans the bits of the secret exponent d, performing a square operation independently of the value of the scanned bit, and a multiplication if such bit is equal to 1. Thus, an attacker monitoring these operations can retrieve the sequence of bits of the exponent.

The modular exponentiation executed for the RSA decryption operations in the wolfSSL implementation is a variation of this well-known square-and-multiply algorithm. It is shown in Figure 9. The countermeasures wolfSSL has deployed to protect this implementation are to always perform the square and the multiply operations for each bit of the exponent (lines 3 and 6) and to load the two possible values of the secret bit related parameter R (R[0] and R[1]) into the cache, so they prevent an attacker from distinguishing which one (0 or 1) was actually used during execution of the multiplication function (line 3). For the square operation, they even initialize a temp variable R[2] to hide accesses to (R[0] and R[1]). These are clearly to prevent cache attacks, which can be seen in the source code comments and the release notes [82], [83].

Despite the always-load countermeasure, there are two possible windows to retrieve the secret information. Firstly, at the end of the multiplication operation in line 3 only the result referring to the actually used bit is stored. In that copy process, one of the two possible values

is loaded while the other one remains untouched. This leaks the key bit. The second window is even bigger, because of the reduce operation in line 4 that only uses the information of the actual key bit value, not taking the precaution of loading both values. This means that this function could be even vulnerable to a traditional cache attack, although the synchronization between the attacker and the victim process would be a challenge.

Based on the code wolfSSL provides for the tests, we generated different secret keys of 2048 bits, and embedded them in our server application in such a way that it decrypts the received data by calling to the wolfSSL RSA decrypt operation. We can observe the leakage by monitoring accesses to one of the two array entries R[0] or R[1] (our `T2`), since the accesses to each of them to depend on the key bit value (0 or 1). During the execution of the multiply function, they are both loaded into the memory, but at the end of this function they perform a copy operation which only accesses the required value. That is, an attacker can, for example, remove R[0] from the cache before the execution of the copy operation and check it afterwards. This operation takes around 70-80 cycles in our system, which is enough for the observation.

Attacking this implementation is eased by the reduce operation executed after the multiply operation. This function only loads the correct value of R[e_i], so either R[0] or R[1]. The execution of the reduce operation takes about 2300 cycles in our test system. This time even allows the execution of a complete probe cycle, so we do not have to be so precise evicting the data when targeting it.

We used both the multiply and the reduce operations for the detection `T1` and later evict R[0](`T2`). Note that, while the functions are shared, R[0] and R[1] are not, so method 2 is required. The attacker has to profile the application to determine t and to find the cache set in which R[0] is loaded. Since our scenario is a continuously running process in a server, the location of R[0] does not change. The task of profiling is eased with the help of the detection of the multiply function. In our posterior experiments we assume that the attacker already knows in which set R[0] is located.

There are other differences to the approach taken in the attack against the S-Box (method 2, case 1). Loading the data of a whole eviction set conflicting with R[0] in the transactional region to achieve a very accurate eviction leads to false positives in detection. This is due to other elements being loaded into the cache set during the large time window of 2300 cycles. This large window also means we do not require such high accuracy, so we can just load some blocks in the transaction to reduce the time it takes later to evict R[0]. This avoids loading the whole eviction set during the transaction. After the detection, only the remaining blocks of the eviction set need to be accessed in the abort handler to retrieve the information about the access (inference step in Figure 7, line 10).

The retrieved key bits depend on both the accuracy of the detection and on the ability of the attacker to remove the data from the cache during the execution of the leaky parts of the code. The mean time between the execution of two multiply operations is about 24000 cycles. That time seems to be "constant" and it is enough for carrying the detection, eviction and retrieving the data. We collected information for the execution of 100 RSA decryptions. Our attack correctly detected 96.8% of the multiply operations introducing 1.3% of false positives. From those correctly detected operations, the information referring to the access to R[0] featured 91% of true positives rate and 87.2% of false negative rate, namely a precision of 87.6%.

Note that no further processing of the results was done. Since we get quite exact timestamps from the TSX aborts, trace alignment becomes fairly easy. For the same reason, some of the retrieved samples that do not match the expected temporal pattern can be discarded to improve the accuracy. Finally, the decision about the correct value of the secret bits of the exponent can be made based on the information retrieved from various traces [84].

8. Countermeasures

The presented attack is feasible due to the fact that code with secret dependent access patterns exist. Even if data is prefetched in the cache, there is a short interval between that prefetch and the actual utilization of the data (as short as the execution of a single instruction) in which an attacker can evict it. Therefore, the attacker has the possibility to observe such accesses and retrieve the secret information. In order to prevent this leakage, these susceptible windows must be removed from the source code and the code should be redesigned. In order to help developers to find leakages in their code, there are tools that detect these leakages [21], [85]. As mentioned above, these tools need to be handled with care, as the very OpenSSL implementation attacked in this work was declared leakage free after such an analysis [79].

In particular, efficient and constant-time implementations of AES are possible by using the bit-slicing technique [86]. Alternatively, each S-Box lookup could access all four cache lines and choose the correct lookup value via arithmetic, eliminating cache line leakage. OpenSSL also provides a constant-time AES software implementation based on bit-slicing, which needs to be selected via the *-DOPENSSL_AES_CONST_TIME* flag. However it would be good to have widely used APIs call secure implementations, since many deployed applications will update the library, but keep the API calls. This gives developers a false sense of security.

The wolfSSL RSA implementation can be repaired by loading the leaky data into the cache in the two vulnerable functions or use of a temporary value, which is in sync with the currently implemented countermeasures in other parts of the code. Note that the fix will prevent exploitation through the LLC, while it may still be able to retrieve some information in the powerful SGX scenario.

There are some other approaches intended to defeat cache attacks [87]. Hardware based countermeasures that prevent cache attacks by means of new cache designs [88] or applying hardware modifications [89], [90], can be effective for the presented attack. However, they are not available yet and some of them are not expected to be implemented soon. Similarly, techniques that allocate the victim and the attacker data in different and mutually exclusive cache sets [91] would prevent this attack.

The TSX-based defense cloak suggests to perform the entire encryption within a transaction, which would then abort in case of a cache eviction [68], preventing the leakage and CACHESNIPER. When detecting the eviction

of the prefetched data this method stops the process and restarts it. However, this method is not widely adopted since it is prone to many false positives, due to spontaneous aborts. Frequent restarts introduce a large overhead and open the door to denial of service attacks. Besides, Cloak does not prevent an attack on the L1 cache, if the prefetched data does not belong to a write set.

Finally, detection-based countermeasures monitor the execution of the algorithms they aim to protect. They collect information about execution times or from performance counters (i.e. cache misses or accesses) to detect changes in the execution trace, which could imply an attack [26]–[28], [30]. CacheSniper was not designed to be stealthy and it generates cache misses on the victim algorithm. However, as we demonstrated by attacking the T-table implementation, CacheSniper can also improve the efficiency of existing attacks, seriously limiting the capability of the detection-based countermeasures to trigger the alarm on time.

9. Related work

The AES T-Table implementation has probably been (and still is) one of the most widely attacked implementations [12]–[14], [92]–[94]. Usually either the first or the last round are targeted, since these rounds only perform an XOR operation between some data and the secret key. Other approaches, such as targeting a deeper layer implementation of T-Table AES used only to encrypt seeds for the pseudo random number generator in AES, are possible but much less popular [95]. It was replaced by an S-Box implementation, which after suffering some attacks was protected with a prefetch. So far, cache-level attackers have only been able to observe cache accesses in very controlled scenarios. They have launched up to 200 spy threads whose execution is controlled with timers and their respective interruption routines to ensure the victim can only execute a few instructions [35], or taken advantage of the powerful adversarial scenario given by SGX. Moghimi et al. [96] assumed a much more powerful adversary with full OS control targeting SGX. They frequently interrupt the victim process and monitor the entire L1 data cache, observing various samples per round. This allows them to distinguish the prefetching stages from the normal operations of the round. Our approach on the contrary does not need to frequently interrupt the victim, works across cores and donly requires the user-level privileges commonly assumed for cache attacks.

Just as AES, RSA has been a target of side channel attacks for many years [97]. Since the execution time is considerably longer than that of AES, prefetching is usually employed in the form of always load/always execute strategies. This in combination with careful code design, such as the Montgomery reduction with constant execution flow, are supposed to ensure protection against cache attacks. However, there are subtleties out of the control of the programmer (e.g. a JIT interpreter that treats an if and else branch differently) that still enable attacks [63], or changes in the attack (e.g the exploitation of cache-bank conflicts) that also recover the key from protected implementations [11].

TSX is the main enabler for our approach. Several other attacks use TSX to improve on signal-to-noise ratio, many of them targeting KASLR [66], [67]. There are various suggestions for address mappings that are not vulnerable to these attacks [98]–[100]. Regarding TSX-based attacks targeting cryptographic operations, the *Prime+Abort* attack assumes an attacker with the same privilege level as ours, but attacks an unprotected implementation [64]. One of the goals of *Prime+Abort* is to demonstrate that removing timers is not a sufficient countermeasure against side channel attacks. It detects data usage by the victim, which is not enough information for targeting prefetched implementations that always load that data. While CACHESNIPER also builds on the fact that a transaction aborts if data used during the transaction is evicted from the LLC, we additionally show that the time between the eviction and the abort triggering is almost constant and infer the exact execution state of the victim. We further use the abort handler to carry the attack whereas it is used as an oracle in the *Prime+Abort* attack. Other works have exploited asynchronous aborts to leak part of an RSA key [42] or to leak data [101] assuming powerful attackers and in concrete scenarios.

10. Conclusion

Writing truly constant-time code is difficult and can result in significant performance penalties. Prefetch-based countermeasures ensure that the cache access profile of a full execution of a protected implementation is secret independent, at the cost of a minimal performance impact caused by few unnecessary reads. But for high-resolution attackers, it is possible to evict data between prefetch and the subsequent sensitive access, restoring the leaky behavior of the target implementation.

In this work we show that such powerful attacks are not restricted to cache attackers with root privileges that can conveniently interrupt the target process at will. Instead we show how user-level cache attackers – by carefully preparing caches and optimizing their timing with respect to the victim process – can achieve an unprecedented temporal accuracy. Existing cryptographic code relying solely on the inability to achieve this accuracy has to be rewritten, since we show that data that has just been prefetched cannot be assumed to still reside in the cache when used, even if used immediately.

Our work quantifies the achievable resolution for carefully designed attack code in several different scenarios. All of the discussed scenarios require user-level privileges only and work in a cross-core setting. Nevertheless they can overcome the prefetch countermeasure in either case. Our analysis reveals that TSX is a powerful mechanism for an attacker to synchronize her execution with the execution of the victim process. With TSX, the attacker sees an abort whenever the victim access a target memory location. This fact, combined with its transient capabilities and the knowledge of the replacement policies of Intel processors, can be leveraged to achieve cache evictions at the desired instants, i.e. after the prefetch, even in the absence of shared memory.

We demonstrate the feasibility of our approach by retrieving the secret key of the T-Table and S-Box AES implementations of OpenSSL and the secret bits from the exponentiation of the wolfSSL RSA implementation.

References

[1] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 973–990.

[3] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative load hazards boost rowhammer and cache attacks," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 621–637.

[4] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[5] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[6] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. 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.

[7] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 785800.

[8] S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2019, pp. 88–105. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2019.00087

[9] 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*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.43

[10] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud," IACR Cryptology ePrint Archive, Tech. Rep., 2015.

[11] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, Jun 2017. [Online]. Available: https://doi.org/10.1007/s13389-017-0152-y

[12] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. 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*, 2014, pp. 299–319.

[13] S. Briongos, P. Malagón, J.-M. de Goyeneche, and J. M. Moya, "Cache misses and the recovery of the full aes 256 key," *Applied Sciences*, vol. 9, no. 5, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/5/944

[14] G. Irazoqui, T. Eisenbarth, and B. 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)*, 2015, pp. 591–604.

[15] Y. Yarom and N. Benger, "Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack," *IACR Cryptology ePrint Archive*, vol. 2014, p. 140, 2014.

[16] T. Ristenpart, E. Tromer, H. Shacham, and S. 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*, 2009, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687

[17] M. S. İnci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, B. Gierlichs and A. Y. Poschmann, Eds., 2016.

[18] D. X. Song, D. A. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh." in *USENIX Security Symposium*, vol. 2001, 2001.

[19] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. 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*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1406–1418. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813708

[20] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[21] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 161–173. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274741

[22] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. 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)*. Washington, D.C.: USENIX, 2013, pp. 431–446.

[23] R. L. Brotzman, S. Liu, D. Zhang, G. Tan, and M. T. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 505–521, 2018.

[24] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Mascat: Preventing microarchitectural attacks before distribution," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: ACM, 2018, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/3176258.3176316

[25] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr 2018.

[26] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162 – 1174, 2016.

[27] S. Briongos, P. Malagón, J. L. Risco-Martín, and J. M. Moya, "Modeling side-channel cache attacks on aes," in *Proceedings of the Summer Computer Simulation Conference*, ser. SCSC '16. San Diego, CA, USA: Society for Computer Simulation International, 2016, pp. 37:1–37:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=3015574.3015611

[28] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, "Cacheshield: Detecting cache attacks through self-observation," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: ACM, 2018, pp. 224–235. [Online]. Available: http://doi.acm.org/10.1145/3176258.3176320

[29] B. Gulmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, "Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning," 2019.

[30] T. Zhang, Y. Zhang, and R. B. Lee, *CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds*. Cham: Springer International Publishing, 2016, pp. 118–140. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-45719-2_6

[31] A. Fuchs and R. B. Lee, "Disruptive prefetching: impact on side-channel attacks and cache designs," in *Proceedings of the 8th ACM International Systems and Storage Conference*, 2015, pp. 1–12.

[32] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge aes against cache-based software side channel vulnerabilities." *IACR Cryptology ePrint Archive*, vol. 2006, p. 52, 01 2006.

[33] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Architecting against software cache-based side-channel attacks," *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1276–1288, 2013.

[34] D. Page, "Defending against cache-based side-channel attacks," *Information Security Technical Report*, vol. 8, no. 1, pp. 30 – 44, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1363412703001043

[35] A. C., B. Roy, B. S. V. Mandarapu, and B. Menezes, "s-box implementation of aes is not side channel resistant," *Journal of Hardware and Systems Security*, vol. 4, 12 2019.

[36] A. Polyakov, "Commit message: Agressively prefetch s-box in sse codepatch, . . . ," https://github.com/openssl/openssl/commit/fc92414273bc30deee51bf1c99abe4b5802f55fb#diff-c0dcd6713547b63fc56ce9716bf52bd9, 2006.

[37] J. Kong, O. Aciicmez, J. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 393–404.

[38] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[39] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 785–800.

[40] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[41] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[42] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[43] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "Copycat: Controlled instruction-level attacks on enclaves for maximal key extraction," *arXiv preprint arXiv:2002.08437*, 2020.

[44] A. Cabrera Aldaya and B. B. Brumley, "When one vulnerable primitive turns viral: Novel single-trace attacks on ecdsa and rsa," *TCHES*, vol. 2020, Mar. 2020.

[45] O. S. foundation, "Security policy," https://www.openssl.org/policies/secpolicy.html, 2012, [Online; accessed 26-June-2020].

[46] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. 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*, ser. RAID 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 48–65.

[47] 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)*, Aug 2012, pp. 1–49.

[48] 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)*, April 2013, pp. 65–74.

[49] ——, "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)*, March 2014, pp. 141–142.

[50] H. Wong, "Intel Ivy Bridge cache replacement policy," jan 2013. [Online]. Available: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/

[51] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[52] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 39–54.

[53] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth, "Reload+refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos

[54] A. Abel and J. Reineke, "uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 673–686. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304062

[55] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf, "Cachequery: Learning replacement policies from hardware caches," *arXiv preprint arXiv:1912.09770*, 2019.

[56] W. Hu, "Lattice scheduling and covert channels," in *IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1992, pp. 52–61. [Online]. Available: http://dx.doi.org/10.1109/RISP.1992.213271

[57] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *Journal of Computer Security*, vol. 8, no. 2/3, pp. 141–158, 2000. [Online]. Available: http://content.iospress.com/articles/journal-of-computer-security/jcs133

[58] D. A. Osvik, A. Shamir, and E. 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*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/11605805_1

[59] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 490–505. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.22

[60] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[61] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Know thy neighbor: Crypto library detection in cloud," *PoPETs*, vol. 2015, no. 1, pp. 25–40, 2015. [Online]. Available: http://www.degruyter.com/view/j/popets.2015.1.issue-1/popets-2015-0003/popets-2015-0003.xml

[62] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 897–912. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[63] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *International Conference on Applied Cryptography and Network Security*. Springer, 2018, pp. 83–102.

[64] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+abort: A timer-free high-precision l3 cache attack using intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 51–67. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[65] D. Dice, T. Harris, A. Kogan, and Y. Lev, "The influence of malloc placement on TSX hardware transactional memory," *CoRR*, vol. abs/1504.04640, 2015. [Online]. Available: http://arxiv.org/abs/1504.04640

[66] R. Wojtczuk, "Tsx improves timing attacks against kaslr," https://bromiumlabs.wordpress.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/, 2014, [Online; accessed 03-Nov-2020].

[67] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 380392. [Online]. Available: https://doi.org/10.1145/2976749.2978321

[68] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 217–233. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

[69] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 587600. [Online]. Available: https://doi.org/10.1145/3196494.3196508

[70] Intel®, "Intel® transactional synchronization extensions (intel® tsx) asynchronous abort," https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort, 2019, [Online; accessed 5-Nov-2020].

[71] T. kernel development community, "Taa - tsx asynchronous abort," https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/tsx_async_abort.html, 2019, [Online; accessed 5-Nov-2020].

[72] Intel®, "Side channel vulnerabilities: Microarchitectural data sampling and transactional asynchronous abort," https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html, 2019, [Online; accessed 5-Nov-2020].

[73] R. H. S. Blog, "It's all a question of time - aes timing attacks on openssl," July 2014, https://access.redhat.com/blogs/766093/posts/1976303.

[74] 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)*, vol. 00, Aug. 2015, pp. 629–636. [Online]. Available: doi.ieeecomputersociety.org/10.1109/DSD.2015.56

[75] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "{SPOILER}: Speculative load hazards boost rowhammer and cache attacks," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 621–637.

[76] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[77] M. Larabel and M. Tippett, "Phoronix test suite," http://www.phoronix-test-suite.com/, 2008.

[78] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, ser. Information Security and Cryptography. Springer, 2002. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-04722-4

[79] G. Irazoqui, K. Cong, X. Guo, H. Khattri, A. K. Kanuparthi, T. Eisenbarth, and B. Sunar, "Did we learn from LLC side channel attacks? A cache leakage detection tool for crypto libraries," *CoRR*, vol. abs/1709.01552, 2017. [Online]. Available: http://arxiv.org/abs/1709.01552

[80] C. Ko, "Analysis of sliding window techniques for exponentiation," *Computers & Mathematics with Applications*, vol. 30, no. 10, pp. 17 – 24, 1995.

[81] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, no. 1, pp. 129–146, Apr. 1998.

[82] J. Barthelmeh, "wolfssl (formerly cyassl) release 3.10.0," https://github.com/wolfSSL/wolfssl/releases/tag/v3.10.0-stable, 2016.

[83] T. Ouska, "Commit message: switch timing resistant exptmod to use temp for square instead of leaking key bit to cache monitor," https://github.com/wolfSSL/wolfssl/commit/6ef9e79ff5ccd2b96fdfed404ada872fd29514be, 2016.

[84] G. D. Micheli and N. Heninger, "Recovering cryptographic keys from partial information, by example," Cryptology ePrint Archive, Report 2020/1506, 2020, https://eprint.iacr.org/2020/1506.

[85] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA – differential address trace analysis: Finding address-based side-channels in binaries," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 603–620. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser

[86] E. Käsper and P. Schwabe, "Faster and timing-attack resistant aes-gcm," in *International Workshop on Cryptographic Hardware and Embedded Systems — CHES*. Springer, 2009, pp. 1–17.

[87] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr 2018. [Online]. Available: https://doi.org/10.1007/s13389-016-0141-6

[88] T. Kim, M. Peinado, and G. 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)*. Bellevue, WA: USENIX, 2012, pp. 189–204. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kim

[89] F. Liu and R. B. Lee, "Random fill cache architecture," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 203–215. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.28

[90] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 494–505. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250723

[91] 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)*, March 2016, pp. 406–418.

[92] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security Symposium*, 2015, pp. 897–912. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[93] B. Gülmezoglu, M. S. Inci, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+reload attack on AES," in *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, 2015, pp. 111–126. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21476-4_8

[94] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 549–564. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[95] S. Cohney, A. Kwong, S. Paz, D. Genkin, N. Heninger, E. Ronen, and Y. Yarom, "Pseudorandom black swans: Cache attacks on ctr_drbg," Cryptology ePrint Archive, Report 2019/996, 2019, https://eprint.iacr.org/2019/996.

[96] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, pp. 69–90.

[97] M. Mushtaq, M. A. Mukhtar, V. Lapotre, M. K. Bhatti, and G. Gogniat, "Winter is here! a decade of cache-based side-channel attacks, detection & mitigation for rsa," *Information Systems*, vol. 92, p. 101524, 2020.

[98] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "Kaslr: Break it, fix it, repeat," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 481493. [Online]. Available: https://doi.org/10.1145/3320269.3384747

[99] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 161–176.

[100] D. Gens, O. Arias, D. Sullivan, C. Liebchen, Y. Jin, and A.-R. Sadeghi, "Lazarus: Practical side-channel resilient kernel-space randomization," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds. Cham: Springer International Publishing, 2017, pp. 238–258.

[101] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-leak forwarding: Leaking data on meltdown-resistant cpus," *CoRR*, vol. abs/1905.05725, 2019. [Online]. Available: http://arxiv.org/abs/1905.05725

Appendix A. Influence of L1 replacement policy

We have observed differences in the results between 12-way-associative caches and 16-way-associative caches. These are due to the L1 replacement policy. In the first case, just after the 12 elements of the set have been placed in the cache, 4 of them are exclusively present in the LLC, and the L1 cache will have suffered 4 misses. It will keep the 4 last accessed blocks, but it will not necessarily have evicted the first 4 accessed. On the contrary, when using a 16-way-associative cache, the L1 cache will have suffered 8 misses. This means it is likely that the 8 elements that were first accessed only reside in the LLC. We observed that in the first case the block we call B is in the L1 cache whereas it is not in the second case.

The replacement policy defines which elements are replaced and the dependence of the eviction order with the actual access to the data. Indeed, our observations can be explained by the tree-based pseudo-LRU replacement policy implemented in the L1 cache [54], [55].

For clarification, the tree-based replacement policy is represented in Figure 10. Starting from the root node, it selects each of the branches depending on the intermediate values of the nodes. In the example, the eviction candidate is marked in red. Since the root node contains a 1, it selects the right branch. The value of the child node is 0, so it selects the left branch. Finally, the last node has a 1, so it points to the element at the right, F in the example. Note that the blocks of memory in the cache set (A to H) are ordered. According to our experiments when the cache set is completely empty, the elements are inserted linearly in the first free block they find regardless of the actual values of the nodes. Once the set is completely filled with data, the apparent value of all the nodes seems to be 0. If an element in the cache is either accessed or replaced, the values of the nodes that pointed to it are switched. For instance, in the example, if F is accessed the nodes will switch from 101 to 010, and D would become the new eviction candidate.

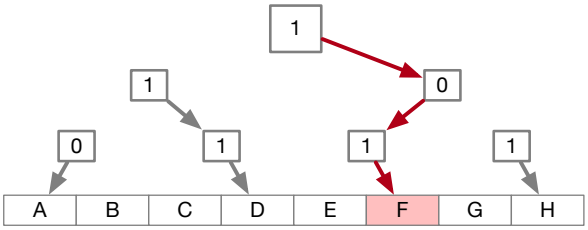

Figure 10. Tree structure that controls the Pseudo-LRU replacement policy of L1 and L2 caches. The eviction candidate is highlighted

TABLE 2. EXPERIMENTAL PLATFORM DETAILS.

Processor	Intel core i5-7600K
Cores	4
Frequency	3.8 GHz
Inclusive LLC	Yes
LLC slices	8
LLC size	6 MB
LLC ways	12
L1 size	32 KB
L1 ways	8

This replacement policy explains why B is in the L1 cache after reading the whole LLC eviction set (12 elements). For this reason, B cannot be the first element to be accessed because its age would not change. Based on this replacement policy, we have prepared a linked list of addresses to access all but one elements of the eviction set. The order of this list ensures that all of them are in the LLC only, where their age can be manipulated.

Appendix B. Technical experiment data

The following remarks should help fellow researchers to reproduce our results.

B.1. Experimental platform

All experiments were conducted on the same machine, the details of which are listed in Table 2.

B.2. wolfSSL setup

In order to analyze the wolfSSL implementation of the exponentiation, we compiled the latest version at the time of writing this paper (version 4.4.0) with the *–enable-debug –enable-keygen* flags in order to be able to keep

the symbols after the installation and to generate RSA keys. Note that we only used the *–enable-debug* flag to verify the execution path with `gdb`. It is not necessary otherwise. Later, we ran the tests included in the library itself to analyze their RSA implementation and found that the exponentiation is still vulnerable despite the steps taken to remove side channel vulnerabilities.