Revisiting Microarchitectural Side-Channels

Miro Haller

School of Computer and Communication Sciences

Semester Project

May 2020

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Muhammed Fatih Balli
EPFL / LASEC

Abstract

Micro-architectural side-channels exploit observations on the internal state of (cryptographic) algorithms obtained by measuring side-effects such as contention on a shared resource. In this project, we focussed on cache side-channels, which were among the first practically exploited information leakages. We provide an overview of the extensive research on cache timing attacks and a more in-depth analysis of the wide-spread Prime+Probe technique. We find that due to the empirical approach on cache side-channels, the results are often tailored to specific software and hardware versions. However, we argue that the underlying root causes of side-channels are likely to persist over time because of their fundamental relation to the system's performance. Therefore, we revisit a classical chosen-plaintext attack on OpenSSL AES-CBC on contemporary hardware. We explain the challenges of implementing this attack in the presence of out-of-order execution, dynamic frequency scaling, hardware prefetching, line-fill buffers and other optimisations. Furthermore, we especially highlight the importance of an appropriate data structure to cope with the previous challenges while minimising cache side-effects of the measurement itself. Moreover, we contribute the library CacheSC that implements different variants of Prime+Probe on L1 and L2. CacheSC provides two methods to attack physically addressed caches. The first attack requires superuser privileges and translates virtual to physical addresses in user space by parsing the page map file. The second approach uses collision detection to build a cache attack data structure without requiring special privileges. Finally, we provide an initial review of the AES key scheduling algorithm as well as Argon2 and provide pointers to novel applications of cache side-channels.

Contents

Chapter 1

Introduction

Initially, cryptographers have developed cryptographic algorithms in purely abstract mathematical models. They analysed the specification, inputs, and outputs of those ciphers for structural weaknesses. However, in practice, the implementation of a cipher is not a black-box which transforms an input to an output as a mathematical function does. The computations have side-effects on the state of the implementing device, which can leak information on the internal state of the algorithm. Kocher presented in 1996 the first timing attacks against public key cryptosystems including Diffie-Hellman, RSA, and DSS [41]. Side-channel attacks are a very active field of research to this day. Additional to the field of timing attacks started by Kocher, researchers analyse the leakage due to power usage, electromagnetic signals, faults, and more. In this project, we focus on cache-based timing side-channels.

For a long time, CPU performance naturally increased due to the technical progress of increasing the transistor density predicted by Moore's law. However, as Herb Sutter stated in 2005 'the free lunch is over' [62]. Physical limitations such as heat dissipation, power consumption, and leakage made it more and more difficult to make single processors faster. Therefore, the industry turned to multi-core and hyper-threading architectures to increase performance, as well as various low-level optimisations. The introduced concurrency increased the processor's complexity significantly and led to new types of side-channel attacks such as Spectre [40] and Meltdown [45]. Those attacks break the isolation between processes by exploiting side-effects of speculative performance optimisations. There seems to be a fundamental trade-off between performance and side-channel leakage. Many mitigations have a severe impact on the processor speed and are thus somewhat reluctantly implemented in practice.

The previous historical overview provides good reasons to revisits microarchitectural side-channels. Processors have evolved significantly since the first timing attacks in 1996. However, it seems likely that contemporary systems are still vulnerable to older side-channel attacks due to the performance-security trade-off. Moreover, we look at the granularity and precision of observations on modern architectures. Therefore, the contributions of this project are the following:

1. We provide a literature overview of cache timing side-channel attacks.
2. We show how to perform cache side-channel attacks on contemporary hardware in the presence of out-of-order execution, concurrency, hardware prefetching, line fill buffers, and write buffers. We implement our findings in CacheSC, a library for cache side-

channels, that simplifies further research on the effect of cache-based observations on newer ciphers.

3. We reproduce one-round attack on OpenSSL, one of the first cache side-channel attacks, from Osvik et al. [51] on a Dell Latitude E6430.
4. We analyse the implementation of the AES key scheduling algorithm in OpenSSL-1.1.1f and show that some AES keys are weaker than others in terms of their cache footprint. Already trivial observations reduce the keyspace of more than 0.11% keys by two bits.
5. We evaluate the granularity of cache side-channel observations on Argon2d and provide ideas for further research.

1.1 Literature Review

This section provides a brief overview of the cache side-channel related literature. The goal of this review is to serve as an entry point in this field rather than to be a complete list.

Kocher started the field of timing attacks in 1996 by publishing the first side-channel attacks against public key cryptosystems, including Diffie-Hellman, RSA, and DSS [41]. Kelsey et al. mentioned the idea of using caches as side-channels when they introduced the notion of *side-channel cryptanalysis* [38]. Page expanded this idea and presented theoretical cache attacks in 2002 [52]. Moreover, Page described the trace-driven and time-driven cache attack types. While the first attacks assume a complete profile of the victim's cache activity, time-driven cache attacks consider a less powerful attacker that makes aggregated observations such as the total execution time of a cipher. Later, Osvik et al. introduced attacks based on the observation of cache sets [51], which is sometimes considered as the third type of cache attacks, called access-driven. Tsunoo et al. discussed the feasibility of cache attacks on ciphers that use many S-box lookups [64] in 2003 and implemented the first cache attacks on DES and 3-DES. They improved their results in 2006 by considering the cipher structure [65].

One of the first practical cache side-channel attacks were presented simultaneously by Bernstein [9] and Osvik et al. [51] in 2005. The latter introduced the time-driven Evict+Time and the access-driven Prime+Probe, which are two different approaches to practical cache attacks that we discuss in Section 2.2. We implement Prime+Probe on a modern processor, among other building on insights from the follow-up paper of Osvik, Tromer and Shamir, which includes a more in-depth discussion of their attacks against OpenSSL-0.9.8 and Linux's `dm-crypt` [63]. Bernstein described a time-driven remote attack which is a variation of Evict+Time. Bonneau et al. significantly improve on the performance of Bernstein's AES attack in 2006 by exploiting cache collisions [14]. Ashokkumar et al. presented an efficient access-driven AES key retrieval attack that only needs 6-7 plaintext-ciphertext blocks [8]. After Neve thoroughly analysed Bernstein's attack in his doctoral thesis [50] and raised questions about its practicality, Acıiçmez et al. presented a more realistic remote attack in 2007 [5], which works without resetting the cache since this would require access to the target platform.

Bertoni et al. [10] and Lauradoux [44] introduced the first trace-driven attacks on AES in 2005. Bertoni et al. exploit external collisions between processes by flushing S-box entries and detecting cache misses using a power trace analysis. Lauradoux presented an attack on the first round of AES using a combination of cipher-internal collision and cache attacks. Acıiçmez et al. improved on the previous two results by significantly reducing the required number of observations [4].

Gullasch et al. presented practical access-driven cache attacks on AES [25], which improved on the AES attacks of Osvik et al. [51] by using fewer observations and no plaintext information. Gullasch et al. achieve higher-granularity observations of cache hits and misses by exploiting the Completely Fair Scheduler (CFS) of modern Linux kernels. They gave rise to the new class of attacks called Flush+Reload. Yarom and Falkner extended the ideas of [25] and applied Flush+Reload to attack the last level cache (LLC) [73]. The LLC is shared between cores and thus enables a variety of cross-core and cross-VM attacks. Irazoqui et al. present a fast Flush+Reload attack recovering full OpenSSL AES keys in a realistic setting with co-located VMware VMs [34]. Gülmezoğlu et al. presented a fully asynchronous and faster cross-VM Flush+Reload attack exploiting memory deduplication of virtualised systems [26]. Yarom and Benger applied the Flush+Reload attack to recover ECDSA nonces form OpenSSL [72].

Other cross-VM attacks before the application of Flush+Reload include the following: Ristenpart et al. showed that it is possible to identify a target VM in a third-party cloud computing environment and obtain a co-located VM to mount cross-VM side-channel attacks [57]. Zhang et al. demonstrated the first fine-grained attack on a symmetric multiprocessing system. They implemented an access-driven side-channel attack to extract ElGamal decryption keys from victims using `libgcrypt` [74]. Weiß et al. applied Bernstein's remote attack to an embedded ARM-based platform with virtualisation [69]. The first that studied cache attacks on embedded systems were Bogdanov et al., who introduced a time-driven differential cache-collision timing attack [13]. Irazoqui et al. transferred Bernstein's attack to Xen and VMware VMs attacking various libraries, including OpenSSL and `libgcrypt` [6].

Although a substantial part of the research focused on AES, some also targeted RSA. Already the first side-channel attacks in 1996 by Kocher targeted public key cryptosystems [41]. Brumley and Boneh developed a practical remote timing attack against OpenSSL, recovering the private key by exploiting Montgomery reductions [16]. Acıiçmez et al. exploited the Montgomery multiplication of OpenSSL's RSA implementation as well, but they were the first to use evictions in the instruction cache to distinguish multiplication from squaring [3]. Chen et al. improved on the previous result with a trace-driven instruction cache timing attack [18]. Yarom and Falkner mounted a Flush+Reload attack on the RSA implementation of GnuPG [73].

Countermeasures against cache attacks include making the program's memory accesses independent from any secret data either in software ([15, 46, 42, 36]) or with dedicated hardware instructions such as AES-NI [24]. Other approaches ([19, 28, 55]) try to detect a malicious process by monitoring the hardware performance counters since the previous cache attacks cause significantly more cache hits and misses than usual. However, Gruss et al. [23] introduced a new attack called Flush+Flush, which evades such detection by not causing any cache misses and minimal cache hits. Their attack relies only on the execution time of the `clflush` instruction, which is slower when the data was cached. Ashokkumar et al. showed that the AES implementations of recent OpenSSL versions are still vulnerable to cache side-channels, despite only using 256-byte S-box tables.

Yarom published a microarchitectural side-channel toolkit called Mastik that implements Prime+Probe, Flush+Reload and Flush+Flush attacks [71]. We discuss in Section 2.3 that such a library is quickly outdated as cache side-channels are fragile due to their strong dependence on the low-level architecture.

We conclude this section by referring to the survey of microarchitectural side-channels by Qian, Yarom, Cock, and Heiser [22]. This survey features a well-arranged table with most known microarchitectural attacks, clearly categorised and including the targets and

methods of every attack. Moreover, the survey discusses current countermeasures in-depth as well as future directions and challenges.

1.2 Methodology

The approach to analyse microarchitectural side-channels differs from other fields of information security – especially from cryptography – in its empirical rather than mathematically rigorous approach. This different methodology is a consequence of both a complex environment and incomplete information. First, the information leakage typically results from complex interactions including temporal components or physical concomitants, which is challenging to model in an abstract analysis, and the main reason why those side-channels exist in the first place. Second, hardware manufacturers often do not entirely document the architectural details of their components in an effort to maintain a competitive advantage over their competitors. As a consequence of this complex setting, we find it useful to model the hardware as a black-box. We can then create hypotheses on the internal behaviour of specific components and design experiments to accept or refute those explanations.

This empirical methodology comes with the downside of a restricted area of application of any reverse-engineered results. The exact conclusions and developed tools are likely to be limited to particular hardware models and microcode versions. However, we expect that the side-channels' underlying causes to persist over a long time, because of their connection to performance. On the one hand, complex performance optimisations are the root of many side-channels, and on the other hand, proposed mitigations often have a severe performance impact. Therefore, side-channels are likely to persist because hardware manufacturers will continue to strive for faster products.

In conclusion, we need to take an empirical approach to analyse microarchitectural side-channels due to incomplete information and a complex execution environment. While the obtained results and written tools may not apply directly to other hardware or software versions, the observed underlying phenomenon is likely to persist. As a consequence, the experiments and results might need to be revised on architectural changes of even microcode updates. To ease this revision, we consider it vital to document the conducted experiments as well as the reasoning behind their structure in detail.

Chapter 2

Cache Side-Channels

Cache side-channels are one of the most well-known sources of information leakage. In this chapter, we start by briefly recalling some essential background knowledge on the cache architecture. Then, we introduce Prime+Probe, a fundamental approach to cache side-channel dating back to the pioneering paper 'Cache Attacks and Countermeasures: the Case of AES' [51] by Osvik, Shamir and Tromer. Afterwards, we discuss various practical challenges that occur when applying this attack on L1 caches of contemporary hardware. We will see that developments in modern computer architecture – such as multi-threading, out-of-order execution, prefetching and variable clock frequency – make it more difficult to obtain reliable side-channels.

2.1 Cache Architecture

While hardware tries to maintain the illusion that processes run in an isolated environment, most components inherently need to be shared for efficiency. For instance, the memory of programs is separated by the use of different virtual address spaces, preventing a malicious application from reading the data of a benign one. However, the physical addresses used underneath this abstraction can be shared between different processes, e.g. for common libraries. Caches are shared as well, which can leak detailed information about other processes running at the same time.

Table 1 lists the different cache levels on a Dell Latitude E6430. The L1 cache is the smallest and fastest memory buffer, and is usually the physically closest cache to the core. It is separated into a data cache (L1d) and an instruction cache (L1i). The last level cache, in our case L3, is shared with other processors[1]. The access times in Table 1 are empirical measurements for Ivy Bridge from [32], which seem to be plausible compared to the values from the Intel Optimization Manual [30] in the table 'Lookup Order and Load Latency' for Sandy Bridge. Note that the difference between an L1 hit and an L2 hit is only eight cycles. We will discuss in Section 2.3.2 how we can achieve time measurements that are precise enough to distinguish cache hits and misses. Figure 1 visualises standard cache terminology of a set-associative cache: cache lines of B bytes are the unit of data transfer; the cache consists of S sets, each having N slots for cache lines. When all slots are occupied, the cache replacement policy decides which line is evicted (i.e. removed from this level of cache while possibly remaining in higher cache levels). Note that the cache is of size $S \cdot N \cdot B$ bytes.

[1] On Ubuntu, the `lscpu` command provides various information about the devices' CPU architecture.

Cache	Size	Sets	Associativity	Shared	Access Latency
L1d	32 KiB	64	8	no	4-5 cycles
L1i	32 KiB	64	8	no	-
L2	256 KiB	512	8	no	12 cycles
L3	4096 KiB	4096	16	yes	~30 cycles

Table 1: Cache sizes on a Dell Latitude E6430 with Intel® Core™ i7-3520M CPU @ 2.90GHz.

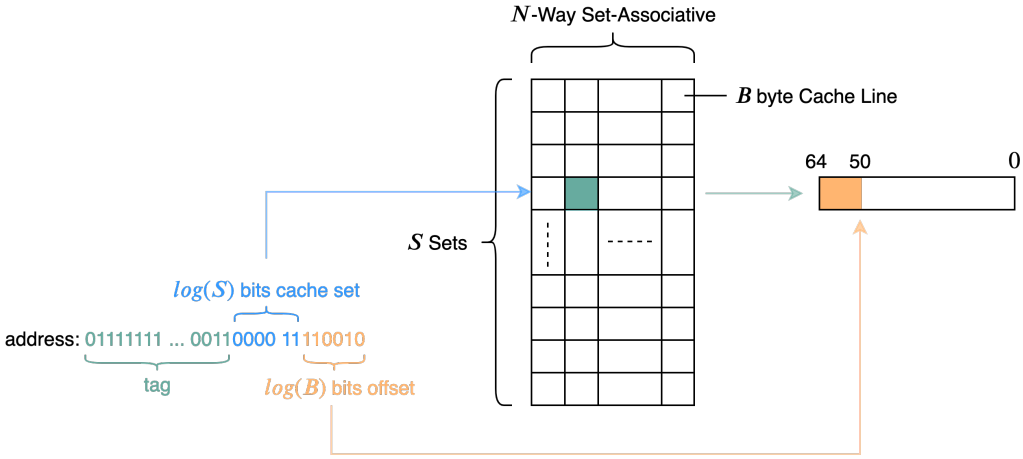


Figure 1: Visualisation of the cache terminology as well as a cache addressing example: for $S = 64$ and $B = 64$, the given memory address maps to set 3 and retrieves the bytes of this cache line at offset 50.

Furthermore, Figure 1 shows a concrete example of how the processor determines the set, offset, and tag of a memory address to search the cache and retrieve a line in case of a cache hit. Of course, the same computation determines in which cache sets some accessed data is placed. We should keep in mind that for accessing d bytes of data starting at some address t_0, the processor fetches all k cache lines that include some of the bytes at positions $t_0, t_0 + 1, \ldots, t_0 + (d - 1)$. More precisely, it fetches $t_1, t_1 + 1, \ldots, t_1 + (k \cdot B - 1)$ such that $t_1 \leq t_0$ is the first address smaller or equal to t_0 that is aligned to B bytes (i.e. at the start of a cache line) and $t_1 + (k \cdot B - 1) \geq t_0 + (d - 1)$. In other words, the processor transfers data on the granularity of cache lines and thus chooses a superset of cache lines that includes the requested data.

After covering the fundamental cache knowledge, we would additionally like to mention the following types of buffers that are both less well known and often sparsely documented:

- The *Line Fill Buffers* (LFB) are located between the L1 and the L2 cache and combine stores and loads between those caches for better performance. Moreover, the Intel Optimization Manual [30] describes that the LFB allocates an entry whenever an L1 cache miss occurs. When a subsequent load hits the same location, the processor knows that a previous operation already initialised the data fetching process. The LFB can also buffer data that already arrived from L2 until it is transferred to L1. According to the authors of RIDL [68], a paper introducing a class of speculative

execution attacks that mainly targets LFBs, there are micro-optimisations where the CPU (speculatively) serves a load from the LFB. Those optimisations are relevant for cache side-channel attacks as the time to serve entries from the LFB could be different than both an L1 hit and miss. There are ten LFB's on the Ivy Bridge microarchitecture of our test laptop.

- *Store Buffers* hold memory write requests until they are completed. When a store operation enters the re-order buffer (ROB), an entry is allocated in the store buffer until this operation exits the ROB, and the data is added to L1. *Store-to-load forwarding* is an optimisation where a subsequent load can directly use the data from the store buffer. Those optimisations again impact cache side-channel measurements as accessed data can reside in store buffers and thus can thwart an attack based on a specially prepared L1 cache (see Section 2.2). More information on store buffers and their optimisations can be found in the Intel Manual [30] and the Fallout paper [17], which is together with RIDL [68] part of the MDS attacks [48] that were independently discovered by various researchers. Fallout uses store buffers to implement Meltdown-like attacks on recent CPUs.
- *Load Buffers* function as a queue for memory loads: the CPU allocates a load buffer entry for every load operation it dispatches. There is little information on those buffers, but they seem to be a rather invisible architecture feature. The ZombieLoad paper [59] by Schwarz et al., which uses a combination of the all previously described buffers and faulting load operations to leak data, provides some further information.

2.2 Prime+Probe in Theory

Prime+Probe describes the general cache side-channel technique visualised in Figure 2. The adversary allocates a data structure of size $S \cdot N \cdot B$. During the so-called prime phase, he fills the entire L1 cache by accessing all entries of his data structure. When the victim process executes, all its memory accesses evict cache lines from L1. During the probe phase, the adversary re-accesses his data and measures for which cache sets the accesses took longer. In those sets, some of his cache lines were evicted, most likely by the victim, and re-fetching them is slower. We discuss in Section 2.3 that this measurement is noisy since the OS and other processes could cause evictions as well. Listing 1 shows the simplified pseudo-code for the prime phase and Listing 2 shows the probe phase. We make those implementation and various practical challenges more precise in Section 2.3.

```
1  function prime()
2    foreach cache line L in L1 do
3      read(L)
4    end
5  end
```

Listing 1: Pseudo-code of prime phase.

```
1  function probe()
2    // dictionary for timings
3    T = {}
4    foreach cache line L in L1 do
5      t_start = time()
6      read(L)
7      t_end = time()
8      T[L] = t_end − t_start
9    end
10   return T
11 end
```

Listing 2: Pseudo-code of probe phase.

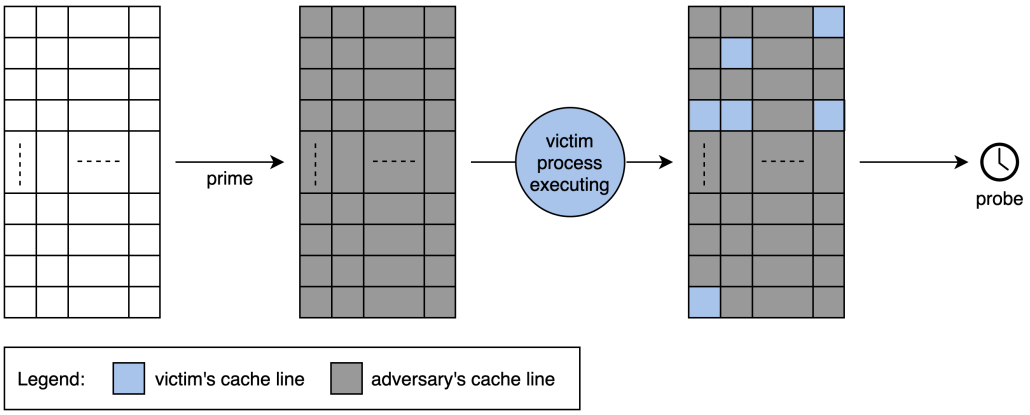


Figure 2: Visualisation of Prime+Probe: first, in the prime phase, the adversary fills the entire L1 cache with his data, then the victim process executes and evicts some cache lines. Afterwards, in the probe phase, the adversary times the accesses to his data and learns in which cache sets the victim process evicted some cache lines.

We would like to mention the following two alternative approaches to cache side-channels:

- Evict+Time was independently developed by Bernstein [9] and Osvik et al., where the latter coined this term together with their presentation of Prime+Probe. Evict+Time repeatedly times the victim process while evicting all cache lines of a specific set. When the adversary removed the victim's data, it has to be re-fetched, and the victim process is slower. Repeating this trial-and-error procedure, the adversary learns which cache set the targeted process uses. Therefore, unlike Prime+Probe, this approach times the victim process' run time and not directly the cache access times. This has the disadvantage that the attacker can only make a single observation (the duration of the access) and not a measurement for each cache set. This reduced observation granularity can increase the required number of samples substantially and make the attack less practical. Both Evict+Time and Prime+Probe need to have precise time measurements and to observe multiple executions with guessable input.
- Flush+Reload was introduced by Yarom et al. in [73] as an extension of another side-channel attack from Gullasch et al. [25]. Flush+Reload is an extension of Prime+Probe that targets the last level cache (LLC) by precisely evicting cache lines with `clflush`[2]. While this enables attacks across cores (even cross-VM [73]), it requires shared memory between processes (for example, due to memory de-duplication).

We decided to focus on Prime+Probe because it can achieve more accurate time measurements than Evict+Time while relying on fewer pre-conditions than Flush+Reload. Moreover, the LLC on current architectures provides the additional complication of using *complex addressing*, an undocumented feature requiring the reverse engineering of hash functions to deduce the cache mappings as described in [33, 47]. While countermeasures against Flush+Reload exist (e.g. to limit access to `clflush` or to prevent page sharing), it seems unlikely that the L1 cache will no longer be shared between processes due to the significant

[2] `clflush` is an x86 instruction that removes the cache line from the last level cache. It removes the cache line from all lower cache levels too, if the caches are inclusive (as it is the case with Ivy Bridge).

performance impact. Therefore, we expect Prime+Probe, or similar attacks focussing on smaller caches, to persist.

2.3 On the Challenges of Applying Prime+Probe to Contemporary Hardware

In this section, we discuss the challenges that we faced when implementing the Prime+Probe attack on a Dell Latitude E6430 with an Ivy Bridge processor. We started by using Mastik [71], a micro-architectural side-channel toolkit by Yarom that abstracts the low-level details of cache attacks. However, after this Prime+Probe implementation with Mastik did not produce the desired results, we wrote CacheSC, a library that specialises on Prime+Probe attacks for contemporary hardware. During this process, we discovered which features of modern processors we have to taken into account to perform successful cache side-channel attacks. In this section, we address the following questions:

- How should we approach the complexity of modern processors in general?
- Can we achieve the co-location of attack and victim processes on the same core in a multi-core environment?
- How can we filter the measurement independent cache footprint?
- How can we achieve precise timing information in the presence of out-of-order execution and dynamic frequency scaling?
- Which data structure for prime and probe avoids hardware prefetching while not polluting the cache itself when we traverse it?
- How does the compiler affect our side-channel library? Which compiler flags and C constructs can we use to avoid undesirable modifications?
- What are some possible sources of noise?

2.3.1 Simplicity Is Key

We started by trying to use Mastik to perform a cache attack against the lookup tables of AES-CBC encryption in OpenSSL-0.9.8. Mastik implements Prime+Probe as a program that does many back-to-back repetitions of prime and probe, reporting the results of each iteration. Therefore, running this concurrently to a victim process should leak which cache sets the victim accessed. However, we realised that this setting is too complicated to start with: first, in the long list of cache access time measurements, it is not straightforward how we can find the measurements that were precisely before and after the victim executed. Second, due to scheduling, we do not know whether the victim and attacker processes run on the same core in parallel. Third, it is a priori unknown, which cache sets the OpenSSL library call accesses; therefore, we do not know for which pattern we are looking.

We simplified our experiment step-by-step, starting by controlling when the victim process runs. We slightly extended Mastik to perform the prime and probe phases separately. Then, we forked the victim process from the attacker and set up inter-process communication using pipes. In other words, the attacker can execute prime, trigger the encryption of the victim and then perform probe after the victim reported that it finished executing. However, we observed that the two processes avoid each other by moving to different cores, which means that they no longer share the same cache. To prevent this, we set the CPU

affinity[3] for each process, which 'determines the set of CPUs on which it [the process] is eligible to run' [58]. To reduce the interference of other processes, we set the kernel parameter `isolcpus` [39], which isolates the CPU from the general scheduler. Only processes that explicitly set the CPU affinity run on this core. Moreover, we disabled hyperthreading as well, to prevent that two virtual cores share the same physical core (since the affinity targets virtual cores).

Therefore, we already bypassed a lot of sophisticated features of modern processors to start with simple measurements. However, we were still not able to correlate the cache set timing measurements with the key used by the victim for AES encryption (we explain this cache side-channel attack in detail in Section 2.5.3). Therefore, we decided to simplify the scenario further and refrain from using multiple processes. Moreover, we replaced the OpenSSL library call with a simple memory read, of which we know precisely in which cache set it causes an eviction.

Despite those simplifications, we were still not observing the expected evictions. However, we simplified the experiment to a point where we are able to identify further issues, which we discuss in the next sections.

> *Conclusion:* Side-channel attacks on modern computer architectures have various complex dependencies. Therefore, it is advisable to verify your expectations on simple examples, where you control as many extraneous variables as possible, before applying cache attacks in more realistic scenarios.

2.3.2 Precise Time Measurements

Table 1 shows that the difference between an L1 cache hit and miss is only eight cycles. Therefore, the measurement routine must produce reliable results on the precision of cycles in the presence of superscalar and out-of-order execution, with as little noise as possible. Before we improved the time measurement of Mastik, we saw random patterns of slower cache sets that were independent of the victim's memory accesses. We were able to remove this effect by using better fencing against previous and subsequent instructions.

According to Intel's white paper on 'How to Benchmark Code Execution Times' [54], the best way to serialize instructions is to use `cpuid` and `rdtsc` in the way we summarised in Listing 3. To understand why this is a smart way to measure time, we first describe the used instructions: `cpuid`, while intended for CPU identification, is the only instruction that, according to the Intel manual [29], 'can be executed at any privilege level to serialize instruction execution'. The instruction `rdtsc` 'reads the current value of the processor's timestamp counter' [29]. The `rdtscp` instruction additionally waits until all previous instructions have finished before sampling time. Now we proceed to explain Listing 3: the first `cpuid` ensures that all previous instructions have terminated before we read the current timestamp and store it. Otherwise, overlapping instructions can non-deterministically slow down the measured code, because depending on the execution state before starting the measurement, more or less additional instructions are measured. After the code we want to measure, we use `rdtscp` instead of `rdtsc` because then all measured operations have to finish before we read the timestamp a second time, instead of being executed in parallel. On the last line, we use `cpuid` again to prevent subsequent instructions from already starting to execute while the timestamp is read and thereby slowing `rdtscp` down. While `cpuid` can

[3]The CPU affinity can be set with the unprivileged `sched_setaffinity` function for the current process.

```
1 cpuid
2 rdtsc
3 /* store timestamp */
4 /* measured code */
5 rdtscp
6 /* store timestamp */
7 cpuid
```

Listing 3: Code benchmarking following the recommendation of Intel's benchmarking white paper [54].

have high variance, this does not propagate to our time measurement since we use it before and after we sample the timestamp.

On modern computer architectures, the processor timestamp poses another challenge: today, processors use dynamic frequency scaling to change their speed depending on the current utilisation. However, the timestamp normally uses a constant rate, which is signalled by the `constant_tsc` flag in `/proc/cpuinfo`. Therefore, the relative meaning of one cycle can differ depending on the actual processor frequency. We deactivated, in the spirit of Section 2.3.1, the dynamic frequency scaling by disabling Intel SpeedStep® and Intel TurboBoost® in the BIOS. Moreover, we used the `cpufreq` [20] tool to observe the frequency and, if possible, set a governor that does not change it. On the Dell Latitude E6430, despite the previously mentioned efforts, we still did not achieve an entirely fixed frequency. However, we observed that the 'performance' governor always increased the clock frequency to the maximum when we pinned a process to that core. Thus, we ensured some preliminary CPU intensive processing runs before we start the measurements so that the clock frequency is most likely fixed to the maximum value.

After modifying Mastik's time measurements in the above-described manner, the random pattern disappeared, and all cache sets showed approximately the same mean access time. However, even those that we expected to be slower because of targeted evictions did not stand out. We discuss in the next section that the data structure and implementation of Mastik were suboptimal for our use case and concealed the evictions.

> *Conclusion:* Out-of-order and superscalar execution, as well as dynamic frequency scaling, must be taken into account to achieve reliable and precise time measurements with low noise. We presented the Intel recommended method for fencing against surrounding instructions.

2.3.3 Developing the Appropriate Data Structure

We noticed that Mastik's implementation stores the measurement results during the probe phase to an array in memory. Those writes could interfere with the measurements by bringing other data to L1. Hence, we might measure more evictions than only those caused by the victim process. We decided to initiate our library CacheSC and implement L1 cache side-channel attacks from scratch instead of adapting Mastik, to have a simpler implementation, specialised on L1 side-channels.

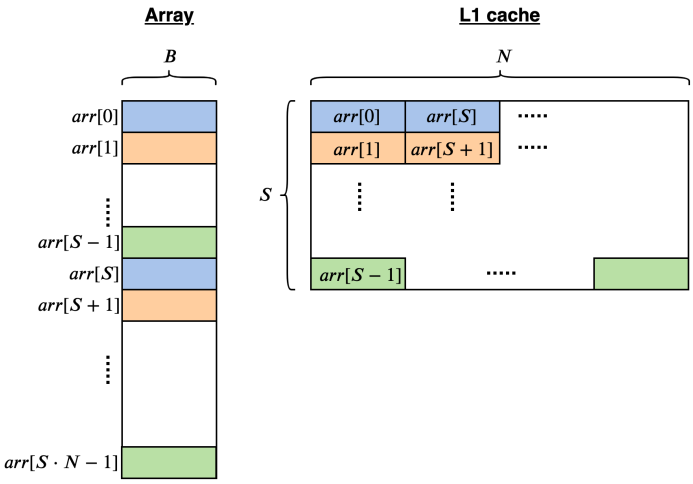


Figure 3: On the left, we have an array of $S \cdot N$ entries with B bytes each, filling the entire L1 cache on the right.

Figure 3 shows the initial idea of a data structure that fills L1. We use an array of the cache line sized `struct`'s shown in Listing 4. The array contains as many cache lines as necessary to fill the entire L1 cache. There are three additional requirements that we found to be desirable for a useful data structure:

1. *Minimise cache pollution:* as suggested in Osvik et al.'s extended and refined paper [63] of [51], we reserve a field in the `struct` for storing the access time measurement. This minimises the cache pollution by reusing the cache entry that we just loaded and measured instead of writing to a different memory location.

2. *Avoid hardware prefetching:* the naïve approach of linearly traversing the described array fails, which is likely caused by the prefetcher following the linear access pattern and guessing the next data blocks, which it then retrieves to L1 ahead of time. In other words, most measured data would be in L1, independent of whether the victim evicted it or not because this data would be prefetched before we measure its access time. To break the linear pattern, we create a doubly-linked list of cache lines in a randomised order.

3. *Reduce overhead:* the time measurement routine from Section 2.3.2 has a significant overhead due to `rdtscp` and the `mov`'s to save the timestamp[4]. When we measured the access time to each cache line separately, we were not able to measure evictions: Figure 8b (which we explain in more detail in Section 2.5.2) shows the average cache access time measurement for a cache set, where we measure each line individually and take the sum of the results for all lines in the same set. We see that the measurements look uniform; however, in Figure 8a the same experiment shows one cache set that stands out. We achieve this by performing a single time measurement for all cache lines in the same cache set. This reduces the overhead by a factor of N, since we no longer sum the overheads of each cache line measurement to obtain the overall cache set timing.

[4]On the Latitude E6430, this overhead was around 50 cycles.

```
1 typedef struct cacheline_t {
2   cacheline *next, *prev;
3
4   uint16_t cache_set, flags;
5   uint32_t time_msrmt;
6
7   char padding[] = /*pad to cache line size*/;
8 } cacheline;
```

Listing 4: Cache line struct.

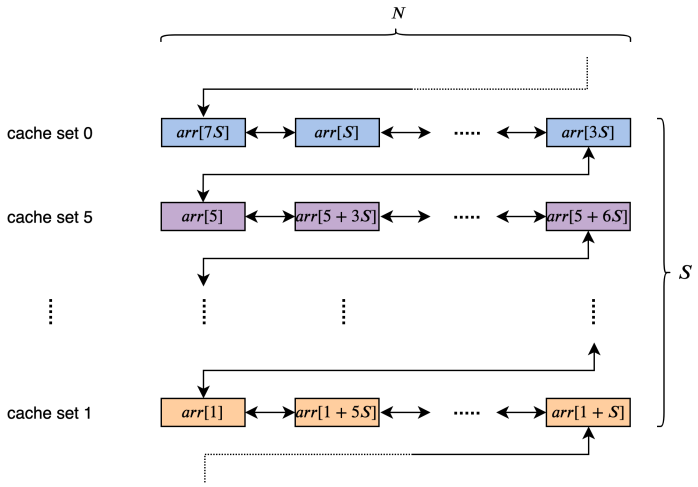


Figure 4: This is a concrete example, visualising our data structure used for Prime+Probe. The randomised doubly-linked list consists of S cache sets that are linked together in a random permutation, where each set contains N cache lines that are also randomly permuted.

Based on these requirements, we implement our data structure as follows: we use the Fisher-Yates shuffle algorithm to obtain a random permutation of cache sets. Each cache set itself is a doubly-linked list of a random permutation of the N cache lines of the array in Figure 3 that map to this set. Figure 4 visualises the final data structure, where the single elements are of the type shown in Listing 4.

Traversing our data structure consists of chasing pointers from one cache line to the next in a linked list. We now argue that this gives us more control over which cache lines are evicted in the presence of out-of-order execution and runtime optimisations. We can use this to either obtain more precise observations or amplify the measured effect, which we explain at the end of this paragraph. First, we argue why pointer chasing is beneficial. In essence, the processor cannot reorder our memory accesses during the probe phase, because we store the address of the next cache line in the previous one. Since the processor cannot predict the random order, it has to finish reading the previous cache line before it can access the next one. Of course, the execution engine cannot bypass the fencing explained in Section 2.3.2, but it could reorder the N accesses that we make within the single time measurement for a cache set. The order of those cache line accesses is relevant because it

can cause cache pollution depending on how prime and probe traverse the linked list. For example, say $\{A, B, C, D, E\}$ are cache lines that map to the same set in a 4-way associative cache. Assuming we start on an empty cache, accessing (A, B, C, D) in this order during priming fills the cache set. Now, assuming the victim accesses E, this most likely evicts A, since it is the least recently used entry (L1 uses a pseudo-LRU cache replacement policy, see also Section 2.3.5). When we access the cache lines during probing in the same order as in priming, then A would evict B, since it is the LRU line after A has been replaced by E. Next, we measure B, which now needs to be refetched and evicts C and this chain reaction continues for all blocks of the cache set. The other option that we have is to access the cache lines in the reverse order during probing, i.e. (D, C, B, A) and thus minimise the chance of such chain reactions. Now, when we access A, we already measured all other cache lines, so it does not influence the measurement when they are evicted. We conclude this paragraph by arguing that both scenarios can be desirable for an attacker. On the one hand, chain reactions mean that we cannot distinguish the number of evictions in a cache set. As soon as there is a single eviction, it triggers a chain reaction, and we measure N slow accesses. On the other hand, in scenarios where we only want to measure a single eviction, chain reactions can be used to amplify the time measurement since we measure N slow cache accesses instead of a single one.

We have, however, two important remarks on the described chain reactions. First, they likely make our measurements noisier. The measured time is more dependent on the interactions of the line fill buffers and the L1 cache since we access those cache sets immediately after evicting them. The Intel optimization manual [30] states that 'the L1 DCache can handle multiple outstanding cache misses and continue to service incoming stores and loads'. Therefore, a read operation could still be a cache hit even though the previous read evicts that entry from the cache, depending on the exact timing and line buffer state. Second, this chain reaction behaviour seems to depend on the micro-architecture. While we could not observe it on our test laptop, it was present on a newer Skylake processor.

> *Conclusion:* We fill the cache with a doubly-linked list of a random permutation of cache line sized entries to avoid interference caused by out-of-order execution and hardware prefetching while minimising the cache side-effect of our measurements.

2.3.4 The Compiler

We learned that it is essential to be aware of what modifications the compiler does, since our measurement code is sensitive to slight changes. We obtained the best results by using the optimisation level `-O1` of `gcc`, which may seem slightly counter-intuitive at first sight, because it is not the lowest level. However, we noticed that `-O0` has a large cache footprint, as it seldomly moves values to registers but inefficiently loads them from memory each time. This is undesirable, as it interferes with our cache measurements.

Furthermore, it is indispensable to inspect the assembly code of a compiled executable due to instruction reordering and smart register usage. For example, the compiler might realise that a victim process always accesses the same memory location in a loop of repeated measurements and decide to place this value in a register. This choice seriously affects the measurement: the victim does no longer evict any cache line, and thus we are observing another behaviour than intended. Without checking the assembly code, we run the risk of drawing the wrong conclusions.

Moreover, it is advisable to write the measurement-critical code (i.e. everything connected to the prime and probe phases, the time measurement and the victim's eviction) in a way that narrows down the compilers freedom. For this purpose, we directly inline assembly code with `asm volatile`. Additionally, we used `static inline` functions in our library to avoid the overhead of function calls as well as the associated register preservation movements.

Another potential issue is branch prediction, i.e. when the processor speculatively starts executing code. Mispredictions have a penalty because the speculatively executed instructions must be discarded. We have two options to prevent this from influencing our measurement: for small loops, we can consider to unroll them entirely and therefore remove the branch in the first place. However, we observed that this is not a viable solution for larger branches: there is more noise, also because the program becomes too large. In such cases, we can use the compiler flag `__builtin_expect` to optimise the compiled code for the measurement-critical branch decision. For example, when we have a loop over many measurements, we can use this flag to tell `gcc` to expect that the branch back to the start of this loop is usually taken. While the real effect of this flag depends on the exact scenario, on a high level, the compiler arranges the code such that the expected case is the one that will more likely be speculatively executed. Listing 5 shows the source code of an artificial example: to optimise for the if branch, the assembly code in Listing 6 is created, and the code in Listing 7 to favour the else branch. The difference is which branch body comes after the compare instruction because, assuming the branch predictor has no other information, it is more likely to execute the subsequent instructions speculatively since they are already available. Of course, in reality, this optimisation only makes sense for more computation-intensive branch bodies and depends on the surrounding code.

```
1 if(__builtin_expect(i == 0, α)) {
2   i = 1;
3 }
4 else {
5   i = 2;
6 }
```

Listing 5: `__builtin_expect` source code for the expected value $\alpha = (i == 0)$.

```
1 cmp %i, 0
2 jne 5
3 mov %i, 1
4 jmp 6
5 mov %i, 2
6 ret
```

Listing 6: `__builtin_expect` for $\alpha = 1$.

```
1 cmp %i, 0
2 je 5
3 mov %i, 2
4 jmp 6
5 mov %i, 1
6 ret
```

Listing 7: `__builtin_expect` for $\alpha = 0$.

Conclusion: The compiler's modifications can interfere with our measurements by reordering instructions or causing additional memory accesses. Therefore, we should inspect the assembly code of the executable regularly.

2.3.5 Remaining Noise Sources

After we discussed various challenges and our solutions to them in this section, we point out some remaining sources of noise: the cache replacement policy, write and line fill buffers.

First, the L1 cache only approximates the LRU cache replacement policy, because it is too expensive to keep track of all accesses accurately. According to the results of Abel and Reineke, who reverse-engineered cache replacement policies in [2] and [1], the Ivy Bridge processor of our Dell laptop uses a Tree-PLRU replacement policy. Since this is an approximation, we have no guarantee that the complete L1 is filled after the prime phase; some of our loads could have evicted other lines from our data structure instead of replacing older entries. The initial state of the cache is unknown, and `invd`, the only instruction to reset the cache, can only be executed in kernel space. Further research could try to determine how many prime phases are needed to make sure all cache lines are in L1, starting from a random initial cache state. However, we did not observe this effect to cause significant bias in our results, which might be because wrong evictions are rare or maybe the replacement policy's mistakes are uniformly distributed.

Second, one hypothesis is that the LFBs cause the four cycles variance that we see in most measurements: some reads could be served directly from the line fill buffers, instead of L2, and thus be faster. Those values might still reside in the LFB because they were requested shortly before or they were evicted in L1 but not yet written back to L2. Write buffers could have a similar effect: modified lines can still be stored in a write buffer and supplied to a subsequent load due to store-to-load forwarding. This optimisation might be faster than an L1 lookup. Connected to this, we observe that overwriting instead of reading a memory location can decrease the noise in some scenarios (or vice versa).

2.4 Lifting Prime+Probe to Physically Addressed Caches

In this section, we discuss how we can apply Prime+Probe on larger, physically indexed caches. The papers we reviewed in Section 1.1 all either target L1 with Prime+Probe or similar approaches, or they attack L3, often using `clflush`. However, we argue that applying Prime+Probe on L2 has advantages over both of those approaches. Moreover, we discuss how to build an attack data structure for physically indexed caches and contribute a proof of concept implementation. On the one hand, the disadvantage of attacks on L3 is that they are easier to mitigate than other attacks by enforcing a more stringent separation of processes (e.g. avoiding shared memory). On the other hand, L1 attacks might provide detailed observations, but the size of the first level cache is limited. A program with many or large memory accesses (such as Argon2, see Section 2.5.5) can quickly fill the entire cache. In that case, Prime+Probe on L1 does no longer provide useful measurements, as all attacker data is evicted. Therefore, it is beneficial to apply Prime+Probe on the larger L2. Moreover, the penalty of a cache miss in L2 is typically larger than one in L1 (for our test laptop, we see in Table 1 that the first has a penalty of 18 cycles, while the L1 miss penalty is only eight cycles). Both L1 and L2 attacks require to be co-located with the victim process. However, processes running on the same operating system can achieve this by spawning enough processes for all CPUs and pin them to fixed processors. The challenging part of targeting L2 is that it is often physically indexed. This means that the cache set in L2 is determined based on the physical rather than virtual address. Userspace programs are usually oblivious to this translation. Therefore, it is not straightforward for the attacker to build a data structure that fills L2 with the correct number of lines per set.

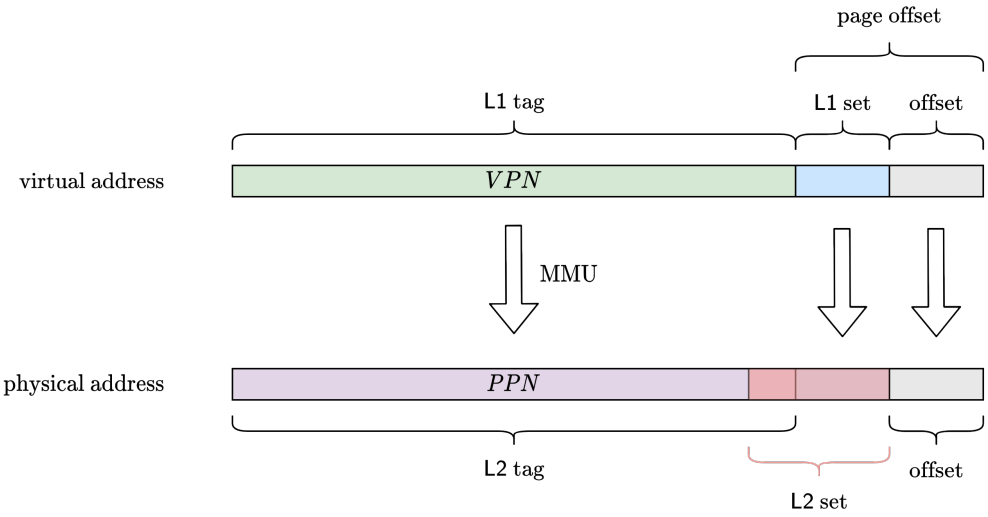


Figure 5: Comparison of virtual and physical addressing on a common choice of L1 and L2 addressing. While the L1 set is defined from the virtual address, the L2 set requires part of the physical page number (PPN). Therefore, indexing in L2 has to wait until the memory management unit (MMU) has translated the virtual page number (VPN) to the PPN.

2.4.1 Physical Addressing

Before we discuss how we apply Prime+Probe to L2, we recall the difference between virtually and physically indexed caches. Figure 5 visualises the common choice of indexing, implemented on our Ivy Bridge processors. L1 is indexed only based on the virtual address to be accessible without having to wait on the address translation. Since physical memory is operated on page granularity, the last part of virtual and physical addresses, called the page offset, is the same. Our memory is byte-addressable, so depending on the size of cache lines, we have a particular offset into cache lines. L1 makes sure to use not more than the remaining bits in the page offset. For example, on our architecture, we have 64-bit cache lines, and thus six bits offset. Moreover, we have 4 KiB pages, i.e. 12 bits page offset. There are 64 sets in L1, which use precisely the remaining six bits of the page offset. However, larger caches often prefer to use physical addressing because they use more sets than the page offset limitation would allow. Additionally, they can tolerate waiting on the address translation, as they have slower access times. On our architecture, L2 has 512 sets and therefore an index of nine bits, six bits are known from the virtual address, but the remaining three bits require the physical address. In other words, we know the set of L2 modulo 64 (this corresponds to the L1 set), but eight different L2 sets map to the same L1 set.

2.4.2 Building a Data Structure for a Physically Indexed Cache

In this section, we present a privileged as well as an unprivileged approach to building the Prime+Probe data structure described in Section 2.3.3. Both strategies follow the high-level procedure shown in Figure 6. To simplify the subsequent formulation, we assume a virtually indexed L1 with S_1 sets and a physically indexed L2 with S_2 sets and N_2 ways. However, the described procedure works for any physically indexed cache, where the sets can be classified

by the bits in the page offset excluding the cache line offset bits (see Figure 5) instead of the L1 sets.

The main idea is to allocate memory pages, and then detect whether or not the cache lines covered by those pages should be included in our data structure. If we already have N_2 cache lines in an L2 set, we schedule that page to be later freed. This delayed freeing avoids that we repeatedly obtain the same page during allocation. Otherwise, when we still have less than N_2 lines, we add all cache lines of the page to a temporary cache data structure and continue allocating pages until this data structure fills L2. The collision detection and how we can identify cache lines in the same sets depends on the available privileges:

- A *privileged* attacker can use the `pagemap` [53] interface exported by the kernel to `/proc/pid/pagemap` to translate virtual to physical addresses. The attacker counts how many cache lines are in which physical set (which he can read of the physical address), and uses this to decide whether or not add a newly allocated page to the data structure. The cache lines in the same L2 can be directly identified from the address. However, this translation requires the `CAP_SYS_ADMIN` capability since Linux 4.0 (this was added as part of the Rowhammer [60] mitigation).
- The *unprivileged* version is more challenging to implement. We maintain a linked list of cache lines that map to the same L1 set, which we know from the virtual address. However, we do not know the exact L2 set of those cache lines. Then, we append the cache lines of a newly allocated page tentatively to their respective L1 set list in our temporary data structure. Next, we use Prime+Probe to detect collisions in L2. If we have $N_2 + 1$ cache lines in the same set in L2, they evict one cache line from L2 during prime, and we can measure an access time of L3 during probe. The challenge in practice is to make this detection reliable. Assuming we identified a collision, we can use it to find the other cache lines in the same set in L2, by testing which $N_2 + 1$ cache lines in the temporary structure cause the collision.

At this point, we successfully allocated a data structure which fills L2 with exactly S_2 sets, each with N_2 cache lines. We can build the data structure of Section 2.3.3 in the same way as for L1. Note that although we can identify which cache lines map to the same L2 set, we only know the specific L1 set. It depends on the application whether or not this is a disadvantage. For example, one attacker against Argon2 in Section 2.5.5 uses every 16th set in the data structure for Prime+Probe, e.g. $T = \{7, 23, \ldots, 503\}$. It does not affect the observation that we do not know the specific L2 sets, because the constructed data structure contains a permutation of the mentioned physical sets in T as the sets are accurate modulo 64.

2.5 Cache Side-Channel Applications

We first start by explaining the data post-processing steps we make to improve the timing side-channel data and continue with the artificial example of detecting a single cache line read in a controlled environment. After that, we mount a Prime+Probe attack against AES-CBC encryption of OpenSSL-0.9.8 and show that we can recover the upper half of any key byte by filtering the key-independent cache footprint. Since the difference between the cache set that was accessed due to the key byte and the others is small, we could no longer leak this information in a more complex example which used two processes and inter-process communication. Furthermore, we provide an initial analysis of vulnerability of the OpenSSL key schedule as well as Argon2d to cache side-channels. We investigate the granularity of

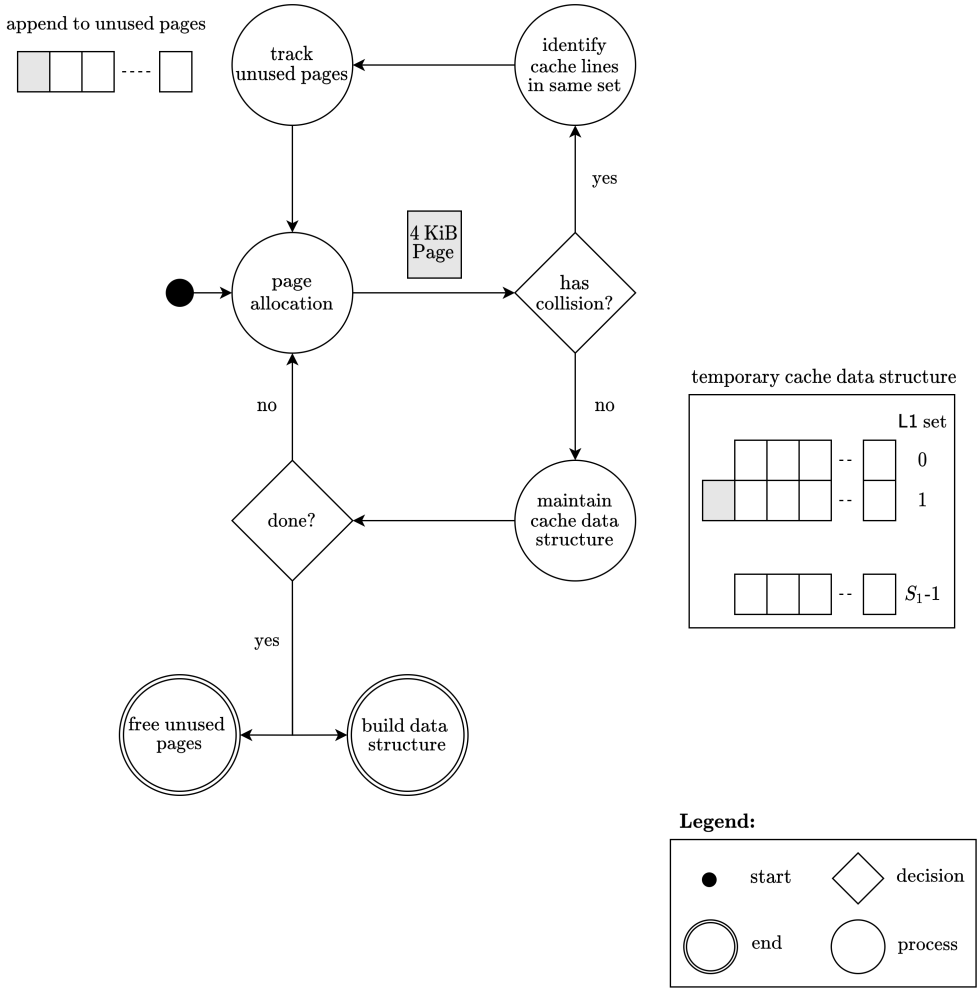


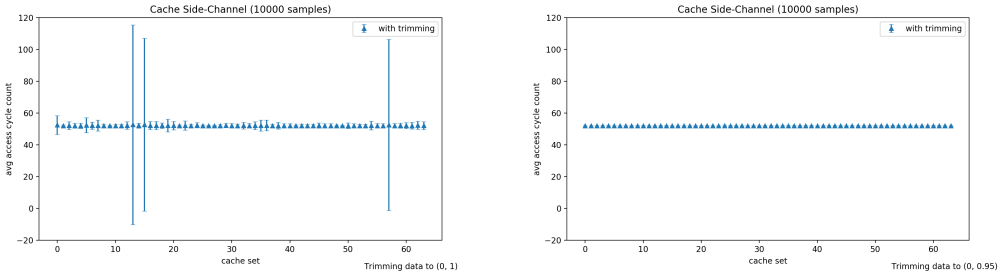
Figure 6: Overview of the cache structure building process for physically addressed caches.

cache observations and provide pointers for further research on novel applications of cache attacks on those targets.

2.5.1 Data Visualisation and Post-Processing

Generally, we often display the cache set on the x-axis and the (normalised) cycle access time on the y-axis. We perform multiple samples of the same measurement to cope with noise. For each of them, we display the mean (as a triangle) and the standard deviation (as a bar). Furthermore, we often use two post-processing steps to reveal the slower cache access time: trimming and normalisation.

For trimming, we observe that outliers usually only increase the measurement. For example, interrupts, port contention or branch mispredictions delay the timed operation. Therefore, the idea of trimming is to remove the slowest measurements because they are most likely outliers and not part of the behaviour that we want to measure. Moreover, the



(a) Without trimming (b) With trimming

Figure 7: Comparison of two plots of the same cache access time measurements with and without trimming. We see that the high variance, e.g. of the cache sets 13, 15 and 57, is due to outliers which are part of the slowest 5% of all measurements.

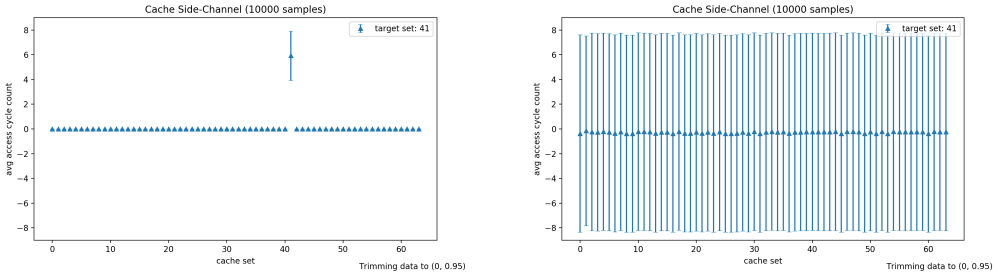
normal mean and standard deviation are both sensitive to high outliers. Figure 7 shows that trimming stabilises the observations drastically: it reduces the standard deviation, for example for set 13, from over 60 cycles to almost zero. We state the interval of used data in the bottom right corner: e.g. in Figure 7b, we filter the top 5% of the data when it is ordered by increasing number of cycles.

Figure 10 shows the value of normalisation on the case of a single AES library call. The characteristic pattern in Figure 10a is due to all cache evictions by the library code. This pattern mainly includes 'normal' behaviour which is independent of the key. To focus on key-dependent differences, we can normalise the data by subtracting the mean values of this expected pattern. Section 2.5.3 gives more information on this specific AES scenario.

2.5.2 Single Cache Line Eviction

In this section, we present a straightforward proof of concept of our cache side-channel implementation in CacheSC on an artificial example. We perform a Prime+Probe attack using the CacheSC library and its data structure (see Section 2.3). Between priming and probing, we access a single memory location, which evicts one cache line from some cache set in L1. Figure 8a shows that the measurements identify this eviction: the mean access time of set 41 is six cycles slower on average than the other set timings. We observe that only the measurement of the targeted cache set has a variance of four. In fact, from investigating the raw timings, we see that in approximately half of the cases accessing the targeted set is only four cycles slower than accessing L1; in the other cases, it is the expected difference of eight cycles[5]. It is difficult to pinpoint the source of this behaviour since the processor architecture is not fully transparent, and the behaviour does not occur consistently. Further research could consider the performance monitoring counters (described in the Intel developers manual volume 3B [31] and an official online reference [56]) to obtain more information on the processor's operations. It would be especially interesting to investigate the difference between a case where no variance occurs and one where it is four cycles.

[5]We describe the cache access times of our test laptop in Table 1



(a) Combined access time measurement for all cache lines in one set.

(b) Sum of individual access time measurements of all cache lines for each set.

Figure 8: Normalised cache access time measurements with a single eviction in cache set 41.

2.5.3 AES-CBC in OpenSSL

In this section, we start by briefly recalling AES encryption based on T-tables. Since those tables are stored in memory, and the table lookups depend on the encrypted plaintext as well as the key, they are vulnerable to cache side-channel attacks. We show a chosen-plaintext attack, similar to the one-round attack from Osvik et al. [51], to recover half of any key byte of the AES-CBC encryption in OpenSSL-0.9.8. Note that [51] performed the Evict+Time attack, while we apply the Prime+Probe version. Before we present the measurement results, we discuss some practical challenges that we have to overcome to perform a successful attack. Moreover, we analyse the theoretical and practical accuracy of this attack.

Notation

To simplify the following explanations, we introduce some notations and definitions:

- $[\alpha) := \{0, 1, \dots \alpha - 1\}$ for any $\alpha \in \mathbb{N}$.
- $\mathbb{B} = \{0,1\}^8$ denotes the set of bytes.
- For some matrix $M \in \mathbb{A}^{n \times m}$ over a set $\mathbb{A}$, let $M_{i,j} \in \mathbb{A}$ denote the element in row i' and column j' and $M_{:,j'} \in \mathbb{A}^n$ denotes the j'-th column, where $i' = i \bmod n, j' = j \bmod m$. Therefore, all indices are always considered to be modulo the respective dimension of the matrix.
- For some bitstring $q = q_{k-1}||q_{k-2}|| \dots ||q_0$ of k bits $q_i \in \{0,1\}, i \in [k)$, we define the substring $q_{[b,a]} = q_b||q_{b-1}|| \dots ||q_a$ for $0 \leq a \leq b < k$.

AES Encryption with T-Tables

In practice, AES encryption is often implemented with lookup tables to increase performance, although it could be performed solely with logic and algebraic operations. Daemen and Rijmen already showed in the AES proposal of 1999 [35] how to combine the SubBytes, ShiftRows and MixColumns transformations into four tables Te_0, Te_1, Te_2, and Te_3. Each of those so called T-tables contains 256 four byte entries. The last round is special as it omits the MixColumns transformation. OpenSSL uses yet another table Te_4 for the final round, where all 16 lookups are performed in Te_4.

We now briefly describe the 10-round AES-CBC encryption of a single 128-bit plaintext block $P = p_0||p_1||\dots p_{15}$ with a 128-bit key $k = k_0||k_1||\dots||k_{15}$, for the bytes $p_i, k_j \in \mathbb{B}$ for $i, j \in [16)$. There is a key scheduling algorithm, which we discuss in more detail in Section 2.5.4, that generates key matrices $K^{(r)} \in \mathbb{B}^{4\times4}, r \in [11)$ for every round plus one additional key which is XORed to the output of the last round. Below, we use $A^{(r)} \in \mathbb{B}^{4\times4}$ to denote the input to the T-tables in round $r \in [10)$, i.e. $A^{(r)}$ stores the encryption state after the key addition. $A^{(0)}$ is initialised with $A^{(0)}_{i,j} = p_l \oplus k_l$ for $l = i + 4j$ and $i, j \in [4)$. We start with the first 9 rounds, which are all performing the same operations. The T-tables are used as follows to transform $A^{(r)}$ to $A^{(r+1)}$ (the input of the next round $r + 1$):

$$A^{(r+1)}_{:,j} = \mathsf{Te}_0[A^{(r)}_{0,j}] \oplus \mathsf{Te}_1[A^{(r)}_{1,j+1}] \oplus \mathsf{Te}_2[A^{(r)}_{2,j+2}] \oplus \mathsf{Te}_3[A^{(r)}_{3,j+3}] \oplus K^{(r+1)}_{:,j}$$

In the last round, the final output block $c_0||c_1||\dots||c_{15}$ for $c_i \in \mathbb{B}, i \in [16)$ is produced as follows, for $C \in \mathbb{B}^{4\times4}$ with $C_{i,j} = c_{i+4j}$ and $i, j \in [4)$:

$$C_{:,j} = \left(\mathsf{Te}_4[A^{(9)}_{0,j}]_0, \mathsf{Te}_4[A^{(9)}_{1,j+1}]_1, \mathsf{Te}_4[A^{(9)}_{2,j+2}]_2, \mathsf{Te}_4[A^{(9)}_{3,j+3}]_3\right)^T \oplus K^{(10)}_{:,j}$$

Note that $\mathsf{Te}_4[A_{i,j}]_t \in \mathbb{B}$ for any $i, j, t \in [4)$ is the t-th component of the vector $\mathsf{Te}_4[A_{i,j}] \in \mathbb{B}^4$.

Chosen-Plaintext Attack

In this section, we explain how we can recover half of any byte of the key by making many observations of the encryption of maliciously chosen plaintexts with an unknown key. Since we control the input of the cipher, we fix the IV without loss of generality to zero for all encryptions. We could simply remove the effect of a non-zero IV because it is public knowledge.

In the first round of AES, as we discussed in Section 2.5.3, we have $A^{(0)}_{i,j} = p_l \oplus k_l$ for $l = i + 4j$. Therefore, the first 16 table accesses directly depend on the key bytes k_l and the plaintext bytes p_l as follows:

$$\mathsf{Te}_t[p_l \oplus k_l], \text{ for } l = t + 4s, \text{ and } t, s \in [4)$$

Since we consider T-table entries of four bytes, the table lookups access the following memory locations for some $t, s \in [4)$:

$$\tau_t + 4 \cdot (p_l \oplus k_l)$$

where τ_t is the base pointer of table Te_t. Figure 1 showed that $q_{[11,6)}$ determines the cache set of an address q. We deduce that $(p_l \oplus k_l)_{[7,4]}$, the leftmost four bits of the byte $(p_l \oplus k_l)$, added to the cache set offset of the base address τ_t define to which cache set the lookups map:

$$\begin{aligned}(\tau_t + 4 \cdot (p_l \oplus k_l))_{[11,6]} &= (\tau_t)_{[11,6]} + (4 \cdot (p_l \oplus k_l))_{[11,6]} \\ &= (\tau_t)_{[11,6]} + (p_l \oplus k_l)_{[9,4]} \\ &= (\tau_t)_{[11,6]} + (p_l \oplus k_l)_{[7,4]}\end{aligned}$$

Consequently, when we know the plaintext byte p_l, and we can leak the accessed cache set, we can compute $(k_l)_{[7:4]}$, i.e. the leftmost four bits of the key byte k_l.

The simplest scenario to exploit the above key leakages in practice is to target only one key byte at a time. Because when we measure multiple bytes at once, we observe the eviction of the same number of cache lines, but we do not know which cache set maps to

which key byte. Therefore, to focus on k_l, we fix p_l over all measurements, while we choose the other plaintext bytes uniformly at random. We now argue that all table accesses, except $\mathsf{Te}_t[p_l \oplus k_l]$, are uniformly distributed. First, we observe that in the first round $\mathsf{Te}_t[p_{l'} \oplus k_{l'}]$ for $l' \neq l$ is a uniformly random access since $p_{l'}$ is random and therefore also the result of the XOR. It remains to show that all accesses in the subsequent rounds are random as well. For this purpose, we argue in the unoptimised (but of course equivalent) version of AES, that the state after round one is random. In the first step, the invertible per-byte transformation SubBytes is applied to the plaintext (which is the initial state). It trivially preserves the uniform distribution of the all bytes, except for the fixed one, which is just renamed. The following ShiftRows operation only reorders bytes in the block. Thus, it moves the fixed byte to a different location, but otherwise, ShiftRows preserves the uniform distribution. Next, *MixColumn* updates each byte with a combination the elements of its column (the bytes are organised in a 4×4 matrix). Therefore, also the fixed byte is combined with three random ones and thus results in a uniformly random byte. After this step, all bytes are uniformly random. XORing this result with the static round key still produces a block of uniformly random bytes. Therefore, all table accesses of the subsequent rounds are random as they are computed based on a random state. We have thus shown that only the lookup $\mathsf{Te}_t[p_l \oplus k_l]$ is fixed. Consequently, only the cache set $(\tau_t)_{[11,6]} + (p_l \oplus k_l)_{[7,4]}$ is always evicted and, therefore, has a slower time measurement than the other sets when averaged over enough samples.

Osvik et al. showed in [51] how the presented chosen-plaintext attack can be extended to work in an asynchronous scenario, where the attacker does not control the encryption algorithm nor knows the plaintexts. He can continuously run Prime+Probe and detect the start of the encryption by monitoring the cache usage. Furthermore, he can use first-order statistics about the plaintexts (e.g. the letter frequency in English texts) and correlate them with guesses for the key bytes.

There are multiple countermeasures against the described attacks: replacing the memory accesses by logical operations, use data oblivious access pattern or disable cache sharing to only name a few. However, they often have severe performance penalties and are seldomly implemented in practice for this reason. A viable countermeasure is the use of hardware implementations, in case they are available.

Statistical Analysis of the Chosen-Plaintext Attack Advantage

We continue by analysing how many cycles the access time of a cache set is slower when it has a fixed access in the first round compared to the other cache sets. This influences, together with the noise, how many measurements we need to identify the targeted cache set and consequently, how practical this attack is. The 10-round AES-CBC encryption of one block has a total of 160 table lookups because every round performs 16 lookups. We assume the slightly simplified L1 cache layout of the T-tables shown in Figure 9: every table has 256 entries of four bytes; thus, a cache line contains 16 entries and a table covers 16 different cache sets. We assume that the T-tables are successive in memory, which is reasonable as the tables are often hardcoded. Under this assumption, Te_0 and Te_4 are the only tables that map to the same cache sets, as shown in Figure 9.

We hereafter use $s \in [S)$ to refer to the cache set with number s, where we number all cache sets from 0 to $S - 1$[6], and we refer to s as a cache set for simplicity. We introduce the following three functions to simplify the subsequent calculations:

[6]Recall from Figure 3 that we use S to denote the total number of cache sets.

- $\mathsf{Cover}(\mathsf{Te}) \subseteq [S)$ is the subset of the cache sets covered by the table Te. Note that we have $|\mathsf{Cover}(\mathsf{Te})| = 16$ for all T-tables of OpenSSL.
- $\mathsf{Tab}(s) = \{\mathsf{Te} \mid s \in \mathsf{Cover}(\mathsf{Te})\}$ is the set of tables that map to cache set $s \in [S)$. We note that $|\mathsf{Tab}(s)| \in \{1, 2\}$ for our assumed table alignment (shown Figure 9).
- $\mathsf{Lookups}(\mathsf{Te})$ is the number of lookups in Te during the complete encryption.

Moreover, we introduce the following random variables:

- Let F_{s,Te^*} denote the event that table Te^* has a fixed access to cache set $s \in [S)$. Note that we should have $s \in \mathsf{Cover}(\mathsf{Te}^*)$, otherwise Te^* could not access s, and this random variable would not make sense.
- Let $X_{s,\mathsf{Te}}$ be the indicator variable that is one if and only if a lookup in table Te evicts a cache line in $s \in [S)$ during the encryption.
- Let Y_s be the random variable counting the number of evicted lines in $s \in [S)$. We have:
$$Y_s = \sum_{\mathsf{Te} \in \mathsf{Tab}(s)} X_{s,\mathsf{Te}}$$

We proceed to calculate the expected number of accessed cache lines in some cache set $s \in [S)$. For this purpose, we first calculate the probability $\Pr[X_{s,\mathsf{Te}} = 1 \mid F_{s',\mathsf{Te}^*}]$ of an eviction by table Te in s conditioned on a fixed access to some cache set $s' \in [S)$ by table Te^*. There are two trivial cases: first, when Te has no entries in s, it does not evict anything, and second, when Te has a fixed access to s, we always have $X_{s,\mathsf{Te}} = 1$. For the other cases, we observe that a cache line of Te is brought to $\mathsf{L1}$ as soon as at least one element in this line was accessed. Therefore, in case of no fixed accesses, we have $X_{s,\mathsf{Te}} = 0$ when all lookups mapped to sets in $\mathsf{Cover}(\mathsf{Te}) \setminus \{s\}$. This happens with the probability $(15/16)^{\mathsf{Lookups}(\mathsf{Te})}$ when $s \in \mathsf{Cover}(\mathsf{Te})$ because all table lookups have to map to the 15 cache sets other than s that store data of Te. The fixed access in our scenario slightly changes this probability when it is to another cache set of the same table: in that case, we have one lookup less that should not map to s. The evictions of different tables in the same cache set are independent under the assumption of a perfect LRU replacement policy, as long as there are not more tables mapping to s than the associativity N of the cache, i.e. $|\mathsf{Tab}(s)| \leq N$. Therefore, the variables $X_{s,\mathsf{Te}}$ and X_{s,Te^*} are independent for different tables, and a fixed access in $\mathsf{Te}^* \neq \mathsf{Te}$ has no influence on $X_{s,\mathsf{Te}}$. In summary, we have the following probability:

$$\Pr[X_{s,\mathsf{Te}} = 1 \mid F_{s',\mathsf{Te}^*}] = \begin{cases} 0 & \text{if } s \notin \mathsf{Cover}(\mathsf{Te}) \\ 1 - \left(\frac{15}{16}\right)^{\mathsf{Lookups}(\mathsf{Te})} & \text{if } s \in \mathsf{Cover}(\mathsf{Te}) \wedge \mathsf{Te} \neq \mathsf{Te}^* \\ 1 - \left(\frac{15}{16}\right)^{\mathsf{Lookups}(\mathsf{Te})-1} & \text{if } s \in \mathsf{Cover}(\mathsf{Te}) \wedge \mathsf{Te} = \mathsf{Te}^* \wedge s \neq s' \\ 1 & \text{if } s \in \mathsf{Cover}(\mathsf{Te}) \wedge \mathsf{Te} = \mathsf{Te}^* \wedge s = s' \end{cases}$$

Note that the case distinction is complete, i.e. it covers all possible scenarios. For the sake of brevity, we introduce the following variables:

$$\Gamma_{\mathsf{Te}} = 1 - \left(\frac{15}{16}\right)^{\mathsf{Lookups}(\mathsf{Te})}, \; \hat{\Gamma}_{\mathsf{Te}} = 1 - \left(\frac{15}{16}\right)^{\mathsf{Lookups}(\mathsf{Te})-1}$$

Since $X_{s,\mathsf{Te}}$ is an indicator variable, we have $\mathsf{Exp}[X_{s,\mathsf{Te}} \mid F_{s,\mathsf{Te}^*}] = \Pr[X_{s,\mathsf{Te}} = 1 \mid F_{s,\mathsf{Te}^*}]$. Therefore, we get by the linearity of the expected value:

$$\mathsf{Exp}[Y_s \mid F_{s,\mathsf{Te}^*}] = \sum_{\mathsf{Te} \in \mathsf{Tab}(s)} \mathsf{Exp}[X_{s,\mathsf{Te}} \mid F_{s,\mathsf{Te}^*}] = 1 + \sum_{\substack{\mathsf{Te} \in \mathsf{Tab}(s) \\ \mathsf{Te} \neq \mathsf{Te}^*}} \Gamma_{\mathsf{Te}}$$

Note that one cache line is always brought to $\mathsf{L1}$ due to the fixed lookup in Te^*. Moreover, note that from F_{s,Te^*} it follows that at least $s \in \mathsf{Cover}(\mathsf{Te}^*)$. When no table maps to s, i.e. $\mathsf{Tab}(s) = \emptyset$, we of course have $Y_s = 0$, but in our case $\forall s \in [S).\ \mathsf{Tab}(s) \neq 0$.

For all other cache sets $s' \in [S) \setminus \{s\}$, we have the following expected value:

$$\mathsf{Exp}[Y_s \mid F_{s',\mathsf{Te}^*}] = \begin{cases} \sum_{\mathsf{Te} \in \mathsf{Tab}(s)} \Gamma_{\mathsf{Te}} & \text{if } s \notin \mathsf{Cover}(\mathsf{Te}^*) \\ \hat{\Gamma}_{\mathsf{Te}^*} + \sum_{\substack{\mathsf{Te} \in \mathsf{Tab}(s) \\ \mathsf{Te} \neq \mathsf{Te}^*}} \Gamma_{\mathsf{Te}} & \text{otherwise} \end{cases}$$

Because if $s \in \mathsf{Cover}(\mathsf{Te}^*)$, then we have the expected value $\mathsf{Exp}[X_{s,\mathsf{Te}^*} \mid F_{s',\mathsf{Te}^*}] = \hat{\Gamma}_{\mathsf{Te}^*}$ since this corresponds to the case $s \in \mathsf{Cover}(\mathsf{Te}) \wedge \mathsf{Te} = \mathsf{Te}^* \wedge s \neq s'$.

Therefore, we can now calculate the expected difference between a targeted cache set t and another cache set $t' \neq t$ with no fixed access:

$$\mathsf{Exp}[Y_t \mid F_{t,\mathsf{Te}^*}] - \mathsf{Exp}[Y_{t'} \mid F_{t,\mathsf{Te}^*}] = \begin{cases} 1 - \Gamma_{\mathsf{Te}^*} & \text{if } t' \notin \mathsf{Cover}(\mathsf{Te}^*) \\ 1 - \hat{\Gamma}_{\mathsf{Te}^*} & \text{otherwise} \end{cases}$$

For the concrete case of OpenSSL, the fixed access is in Te_j for $j \in [4)$, since OpenSSL only uses those four tables in the first round of encryption. All of them are used in nine rounds with four lookups each, so we have $\mathsf{Lookups}(\mathsf{Te}_j) = 36$. Thus, for a fixed access of Te_j to the cache set $t \in [S)$, we obtain a distinguishing advantage of $1 - \Gamma_{\mathsf{Te}_j} = (15/16)^{36} \approx 0.098$ over the 48 cache sets $[S) \setminus \mathsf{Cover}(\mathsf{Te}_j)$. Moreover, compared to the cache sets in $\mathsf{Cover}(\mathsf{Te}_j) \setminus \{t\}$, we have an advantage of $1 - \hat{\Gamma}_{\mathsf{Te}_j} = (15/16)^{35} \approx 0.104$. Because each cache line eviction in $\mathsf{L1}$ causes eight additional load cycles (see Table 1), we can expect the targeted cache set t to be around 0.8 cycles slower than all other cache sets.

We make two observations on the above calculations: first, we assumed a perfect LRU replacement policy, however, Section 2.3.5 mentions that real $\mathsf{L1}$ caches only approximate the LRU policy. Nevertheless, also the Tree-PLRU policy of our Dell Latitude E6430 guarantees that the most recently used cache line is never evicted. Thus, the above calculations are perfectly justified for OpenSSL on our test laptop, since two successively accessed cache lines never evict each other. Second, the normalisation explained in Section 2.5.1 performs the subtraction $\mathsf{Exp}[Y_s \mid F_{s',\mathsf{Te}^*}] - \mathsf{Exp}[Y_s \mid F_{s'',\mathsf{Te}_\perp}]$ for $s, s', s'' \in [S)$ and $s \neq s''$ and a non-existent table $\mathsf{Te}_\perp$. Because with this slight abuse of notation, $F_{s'',\mathsf{Te}_\perp}$ corresponds to the event of no fixed access by any table, which is what the baseline measurements do. Note that without this subtraction, the 16 cache sets in which up to two cache lines can be evicted would have a higher access time than the ones to which only a single table maps.

In summary, the intuition why a targeted cache set is only 0.8 cycles slower instead of the eight cycles difference between an $\mathsf{L1}$ and $\mathsf{L2}$ access is the following: There are 16 table entries in the same cache line, and this line is cached as soon as one of these entries is accessed. The only difference between a cache set with a fixed access to one without is that for the latter does not access (and thus evict) the cache line in 10% of the cases. Therefore, we only have a tenth of the distinguishing advantage between $\mathsf{L1}$ and $\mathsf{L2}$.

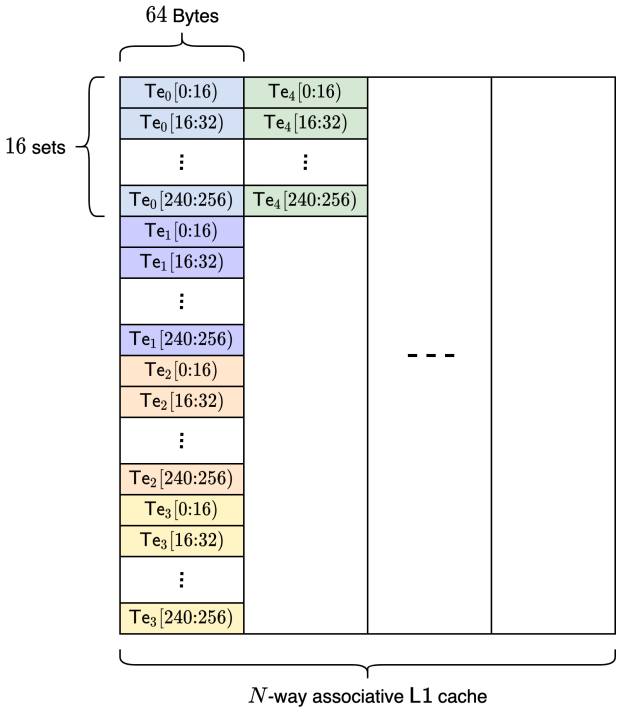
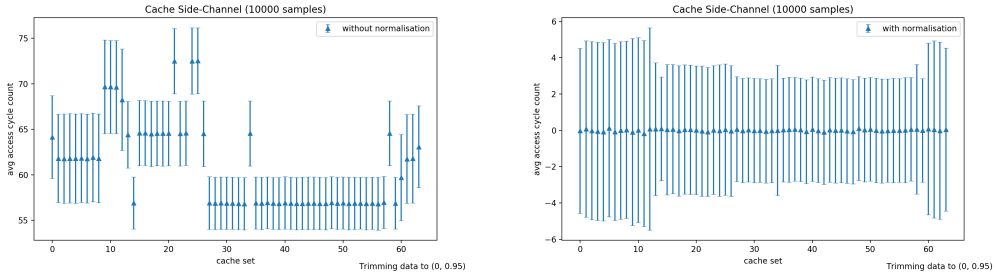


Figure 9: This shows the simplified L1 cache layout of consecutive AES T-tables Te_0 to Te_4. We use the notation $T[a{:}b)$ as a shorthand for the entries $T[a], T[a{+}1], \ldots, T[b{-}1]$. 16 entries fill one cache line and one table covers 16 cache sets. Note that we simplified this figure in two ways: first, we assume that all tables are allocated consecutively, which is plausible for OpenSSL-0.9.8 as they are hardcoded. Second, the cache lines are placed in an any of the N slots (this depends on the state of the cache replacement policy implementation).

Practical Challenges when Attacking OpenSSL

The first very fundamental challenge is to find the executed AES-CBC code for a library call. While this seems to be a simple task at first glance, there are multiple complications: First, OpenSSL is an extensive library and tracing API calls to the implementation needs many steps, involving functions that are only generated at compile time from macros. Second, which code is executed also depends on runtime decisions: the library checks at runtime whether the AES instruction set AES-NI is supported. As stated in Intel's introduction [21], this hardware implementation of AES mitigates cache side-channel attacks 'by running in data-independent time and not using tables'. For testing purposes, we can deactivate AES-NI for OpenSSL by either using the environment variable `OPENSSL_ia32cap`, removing the `aesni_intel` kernel module or using BIOS options (when available). However, we recommend tracing a library call with a debugger to make sure that AES-NI is not used. Third, there are multiple implementations of the same encryption: on the one hand, the executed code depends on engines[7], which can extend an algorithm or implement an alternative one. On the other hand, there are multiple implementations (e.g. different optimisations) of the same function and compiler flags decide which of those is used.

[7] Starting from OpenSSL version 3.0.0, providers will replace engines.



(a) Without normalisation (b) With normalisation

Figure 10: Comparison of the same cache access time measurements of ten rounds of AES-CBC encryption. The characteristic pattern is due to all cache evictions caused by this library call, independently of the key byte. They are stable over many measurements and can thus be removed by subtracting this 'normal' behaviour from the measurements.

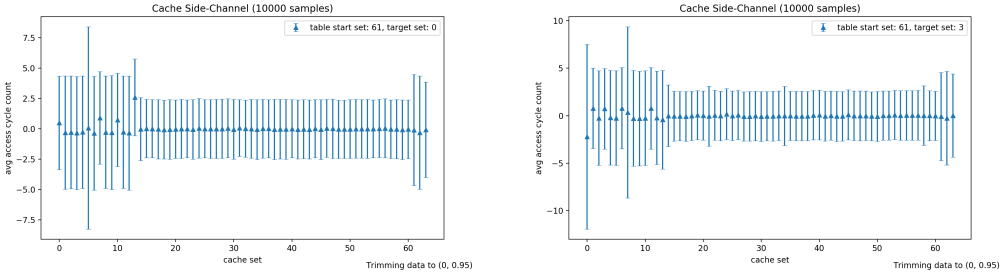


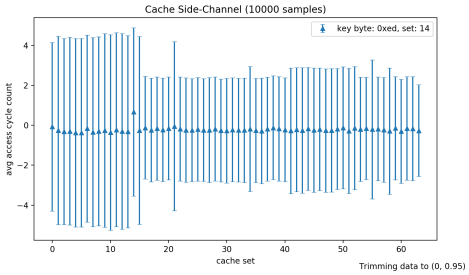
Figure 11: Detecting the Te_0 table offset by targeting all four lookups in round one and performing multiple measurements.

The second challenge is the cache footprint of all key-independent memory accesses of the library call. To cope with this, we use the normalisation that we already mentioned in Section 2.5.1, shown in Figure 10. We obtain key-independent baseline measurements by averaging the access cycle counts over many encryptions with random keys. It is vital to have very low noise measurements because otherwise, the cache footprint of a library call would not be stable across samples. This might increase the variance to a point where we can no longer leak the key-dependent cache accesses.

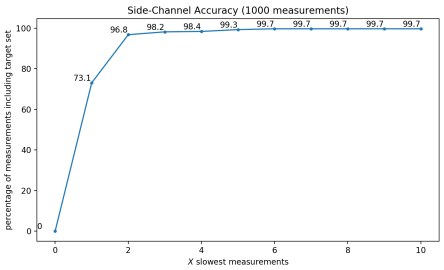
Lastly, the address space layout randomisation causes the OpenSSL library to be loaded to different addresses on each execution. Since it depends on the table offset to which cache set an entry maps, we need to know the table's address to leak information about the key. For simplicity, we patched the library to export this address. However, it is possible to detect this offset: Figure 11 shows the result of targeting all four bytes that have a lookup in Te_0 in the first round at the same time. When we combine enough of those measurements, all cache sets of the table stand out and we learn the offset.

Measurement Results

After we have introduced the chosen-plaintext attack on AES-CBC encryption using T-tables, we present the results of this attack implemented with our cache side-channel library



(a) Average cycle count per cache set. We see that accessing the target set 14 is significantly slower than the other sets' timing.



(b) Percentage of measurements where the targeted set is among the X slowest measurements.

Figure 12: Cache-side channel attack on AES-CBC of OpenSSL-0.9.8.

CacheSC.

Figure 12a shows the chosen-plaintext attack targeting set 14 and averaged over 10000 samples. Together with the table offset, which was conveniently aligned to the L1 cache in this execution, the set directly leaks the upper half of the targeted key byte 0xED, i.e. $0\text{xE} = 14$. We see that the mean access cycle measurement of the targeted set is nearly one cycle slower, matching the calculation of Section 2.5.3.

Since the expected difference of 0.8 cycles is quite a small, we evaluate the accuracy of this side-channel in the presence of noise. Figure 12b shows for which percentage of measurements the targeted set was among the X slowest measurements, where $X = 1, 2, \ldots, 10$. We see that for 76.5% of all measurements, the targeted set is the slowest one when we average over 10000 samples for each measurement. We conclude that the attack is therefore very accurate.

2.5.4 OpenSSL Key Scheduling

In this section, we briefly evaluate the cache side-channel leakage of OpenSSL's implementation of the AES key scheduling algorithm. We first provide a theoretical abstraction of the key expansion used in OpenSSL, before we show that some AES keys are weaker than others in the sense that leaking their memory accesses reduces the space of possible keys significantly.

Key Scheduling Algorithm

The key scheduling algorithm in OpenSSL is implemented with a single T-table Te_4. It makes 40 lookups in Te_4 to expand the key for 10-round AES-CBC encryption. We describe the expansion of a 128-bit user-supplied key $k = k_0||k_1||\ldots||k_{15}$ for key bytes $k_i \in \mathbb{B}, i \in [16)$[8]. Let $K^{(i)} \in \mathbb{B}^{4\times 4}, i \in [11)$ be the key matrix. The first ten rounds of encryption start with XORing the key matrix to the current state of the encryption. The last key $K^{(10)}$ is XORed to the output of the tenth round to produce the final ciphertext. For the first round of the key schedule, $K^{(0)}$ is initialised with k as follows: $K^{(0)}_{i,j} = k_{i+4j}$ for $i, j \in [4)$. Each round

[8]The notation used in this section was introduced in Section 2.5.3.

$r \in [10)$ computes the matrix $K^{(r+1)}$ recursively:

$$K^{(r+1)}_{:,0} = \left(\mathsf{Te}_4[K^{(r)}_{1,3}])_0, (\mathsf{Te}_4[K^{(r)}_{2,3}])_1, (\mathsf{Te}_4[K^{(r)}_{3,3}])_2, (\mathsf{Te}_4[K^{(r)}_{0,3}])_3\right)^T \oplus rcon_r$$
$$K^{(r+1)}_{:,j} = K^{(r)}_{:,j} \oplus K^{(r+1)}_{:,j-1}, \text{ for } j = 1, 2, 3$$

where $rcon_r \in \mathbb{B}^4$ is the constant for round r defined in the AES proposal [35]. Also note that $\mathsf{Te}_4[K^{(r)}_{i,j}]_t \in \mathbb{B}$ for any $i, j, t \in [4)$ is the t-th component of the vector $\mathsf{Te}_4[K^{(r)}_{i,j}] \in \mathbb{B}^4$.

Weak AES Keys

In this section, we first make a statistical analysis to show that some AES keys are weaker than others in terms of their cache footprint. We then continue to analyse how this reduces the search space of those weak keys.

Before we start, we first highlight an essential difference compared to the encryption attack. For the key scheduling algorithm, we do not have a second parameter, corresponding to the plaintext of the encryption, that we can manipulate. Therefore, we cannot focus on single bytes of the key while randomising the impact of the other bytes: we always observe the same cache footprint for the same key.

We reuse the notation from Section 2.5.3 to analyse the probability that a table Te covering $k = |\mathsf{Cover}(\mathsf{Te})|$ sets and having $n = |\mathsf{Lookups}(\mathsf{Te})|$ lookups does not evict cache lines in all k cache sets. We model this problem as throwing n distinguishable balls uniformly at random into k bins. Note that this makes the idealised assumption that the AES keys chosen by the user are distributed uniformly at random. Let $C_{n,k}$ be the random variable that counts the number of empty bins. Thus, we want to compute $\Pr[C_{n,k} > 0]$, the probability that not all bins are occupied. We calculate this using the Stirling partition number $\mathsf{S}(n,k)$ [9], which counts the number of ways in which we can partition a set of n elements into k pairwise disjoint and non-empty sets. Given such a partition of the n balls, we have $k!$ different mappings of those k subsets to our k bins. Since each subset is non-empty, every bin has at least one ball. Finally, we know that the total number of possible assignments of n balls to k bins is k^n, where each ball chooses one of the k bins. In conclusion, we obtain the following probability for $\Pr[C_{n,k} > 0] = 1 - \Pr[C_{n,k} = 0]$ by dividing the number of cases with k non-empty bins by the total number of cases:

$$\Pr[C_{n,k} > 0] = 1 - \frac{\mathsf{S}(n,k) \cdot k!}{k^n}$$

For the 10-round OpenSSL key scheduling algorithm, we have $n = 40$ lookups in Te_4 and $k = |\mathsf{Cover}(\mathsf{Te}_4)| = 16$. Therefore, we get $\Pr[C_{40,16} > 0] \approx 0.76$. This means that approximately 76% of the keys do not fetch all 16 sets of table Te_4 to L1. In the following, we call those keys 'weak'.

We continue by showing how the search space of such weak keys is trivially reduced only due to the table accesses in the first round. Further research could investigate if the relations of the other nine rounds can be used to reduce the keyspace even more. Section 2.5.4 shows

[9] The Stirling partition number, also known as the Stirling number of the second kind, is defined as follows:

$$\mathsf{S}(n,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^i\binom{k}{i}(k-i)^n$$

Cache Evicitons $(k-i)$	Weak Keys	Keyspace
15	75.7%	77.2%
14	34.8%	58.6%
13	9.1%	43.6%
12	1.3%	31.6%
11	0.11%	22.3%
10	0.005%	15.3%

Table 2: Summary of how many weak keys exist that evict at most the given number of sets during key scheduling and thereby reduce the keyspace. The percentages are relative to all 2^{128} possible keys. Note that for 0.11% of the keys, we reduce the keyspace by 22.3%. Although this might seem to be much, it actually corresponds to an approximate security loss of two bits, as $2^{128}/2^{126} = 25\%$.

that the initial table lookups directly depend on the last four bytes of the user-supplied key. The table accesses $\mathsf{Te}_4[k_{12+i}]$ for $i \in [4)$ leak the upper four bits of k_{12+i} because the table entries are four bytes and thus the accessed memory location is $\tau_4 + 4 \cdot k_{12+i}$, where τ_4 is the base pointer of table Te_4. As we show in detail in Section 2.5.3, the cache set to which this address maps is $(\tau_4)_{[11,6]} + (k_{12+i})_{[7,4]}$. Thus the accessed cache sets leak $(k_{12+i})_{[7,4]}$. For the weak keys, we directly follow that there are only 15 possible values for k_{12+i} because there is at least one cache line that was never accessed. Therefore, we can reduce the keyspace to $15^4 \cdot 2^{112}$ possible keys, which corresponds to 77.2% of the original 2^{128} possible keys.

We can generalise this analysis for keys that access at most $k-i$ cache sets. We derive the probability that at least i bins are empty by summing all cases where the n balls fill exactly $k-i$ bins:

$$\Pr[C_{n,k} > i] = 1 - \frac{\sum_{j=0}^{i} \mathsf{S}(n, k-i)\binom{k}{k-i}(k-i)!}{k^n}$$

Because there are $\binom{k}{k-i}$ ways to choose $k-i$ bins (the rest of the formula follows the same reasoning as before). Table 2 summarises the results for weak keys.

Conclusion

The analysed implementation of the AES key schedule is present in both OpenSSL-0.9.8 as well as in the latest stable version OpenSSL-1.1.1f. However, the practical impact of the presented attack is limited for three reasons. First, this vulnerable key scheduling implementation is probably not often used in practice due to the different AES implementations in OpenSSL and especially the wide-spread hardware support for AES-NI. Second, the adversary needs precise observations of many key scheduling executions for the same key to obtain a precise cache footprint, which requires him to run an unprivileged process on the same core. Third, the reduced keyspace for the provided examples of reasonably likely weak keys is still of the order of 2^{126}, which is still far from being computable[10].

Nevertheless, we encourage further research to look into the relations for the last nine rounds, because we have only leveraged the implications of the cache observations on the first round of the key scheduling algorithm. Since the lookups of later rounds depend on the key bytes as well, it is possible that the keyspace can be further reduced.

[10] On April 16th 2020, the bitcoin network computed approximately $118 \cdot 10^{18}$ hashes per second [12], which corresponds to $2^{66.7}$ operations per second or $2^{91.6}$ per year.

2.5.5 Argon2

Argon2 [11] is a memory-hard password hash function, designed by Biryukov, Dinu, and Khovratovich, that won the Password Hashing Competition (PHC) in July 2015 [70]. This new hash function aims to protect low-entropy secrets, such as passwords, from dictionary attacks. The threat of password cracking increased over the last years, as the computation of hash functions got faster due to Moore's law and dedicated hardware. Memory-hard functions aim to mitigate this problem by requiring a substantial amount of memory and having significantly higher computation costs if less memory is available. Argon2 specifies two variants, Argon2d and Argon2i:

- 'Argon2d is faster and uses data-depending memory access, which makes it suitable for cryptocurrencies and applications with no threats from side-channel timing attacks' [11].
- 'Argon2i uses data-independent memory access, which is preferred for password hashing and password-based key derivation' [11].

The above definitions already indicate that Argon2d does not entirely resist side-channel attacks. In this section, we take a first look at the granularity of cache timing observations and provide pointers for further research on such timing attacks. Investigating the practicality of cache side-channels is relevant, as we found real-world usages of Argon2d for password hashing. For example, the open-source [61] password manager KeePass [37] uses Argon2d in its latest version KeePass 2.45 from May 5th, 2020. Without understanding Argon2d's susceptibility to side-channel attacks, we cannot evaluate the security risk for such services.

Background

Before we discuss the cache side-channel granularity, we briefly recall the parts of Argon2's specification that are relevant for the subsequent sections. The high-level overview in this section is intertwined with Figure 13, which is Biryukov et al.'s visualisation of a single-pass through memory by Argon2 [11].

Argon2 follows the extract-then-expand paradigm [43] by first using a plain hash function H[11] to extract entropy from the message, nonce, and other parameters. Overall, Argon2 uses m 1024-byte blocks of memory, organised in a matrix B of p rows and $q = \lfloor m/p \rfloor$ columns. The rows also referred to as *lanes*, correspond to the tunable number of threads. Argon2 uses the previously extracted information to initialise the first blocks in memory. It then makes t passes over memory, where it calculates $B_{i,j}$ for $i \in [p), j \in [q)$ using a compression function G on $B_{i,j-1}$ and another referenced block $B_{i',j'}$. Argon2d calculates the indices i', j' based on $B_{i,j-1}$, which makes it memory-dependant. In contrast, Argon2i uses the 2-round compression function G^2 in counter mode on inputs including the position – but never data – of the current block (slice, lane, and pass number) to select the referenced block. This referencing follows a set of rules that uses the further partition of B into l slices of q/l columns (see Figure 13). A *segment* is the intersection between slices and lanes and contains q/l blocks that are all processed by the same thread. Since Argon2 computes those segments in parallel, a block cannot reference any block from another segment in the same slice. However, it can reference other blocks in the same segment (except the previous block, as $B_{i,j-1}$ is already the first argument of G) and all blocks in segments of different slices, if they were already initialised. Finally, after t iterations over memory, Argon2 computes the

[11] Argon2 uses Blake2b for this hash function H.

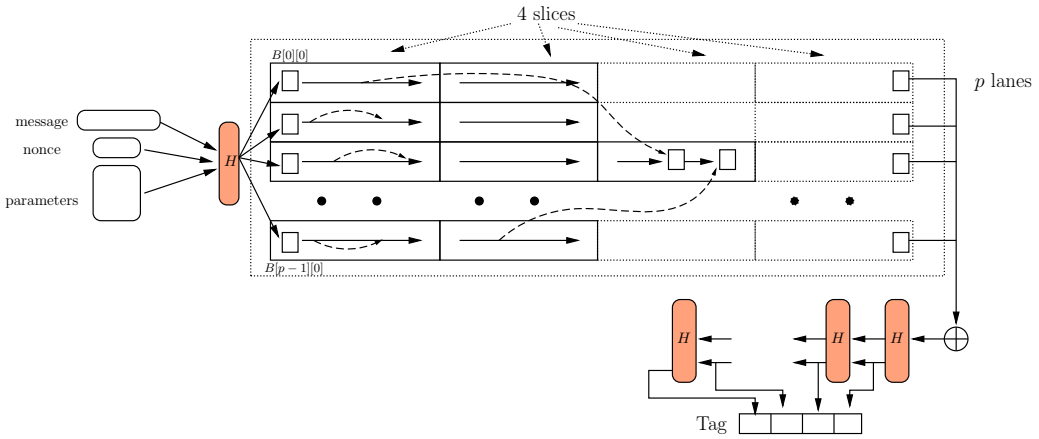


Figure 13: This is Figure 2 from the Argon2 paper [11]. It shows a single pass of Argon2 over the m 1024-byte memory blocks organised in p lanes and $l = 4$ slices. The intersection of lanes and slices is called segment, and contains $\left\lfloor \frac{m}{p \cdot l} \right\rfloor$ blocks.

output tag by the iterative application of H on the XOR of the blocks in the last column $B_{:,q-1}$.

We consider in the rest of this section the following parameters, used in the reference implementation of Argon2 [7]:

- $p = 1$, i.e. using a single thread and thus no parallelism.
- $m = 2^{16}$, i.e. a total memory usage of $2^{16} * 1024$ B= 64 MiB.
- $l = 4$ slices.
- $t = 2$, corresponding to initialising all blocks and then performing another pass over all of them.

Cache Side-Channel Granularity

In this section, we evaluate the granularity of cache side-channel observations on Argon2d. We consider two processes, an attacker and a victim, running on the same CPU. The attacker process $\mathcal{A}$ asynchronously performs many iterations of Prime+Probe, while the victim process $\mathcal{V}$ concurrently calculates an Argon2 hash. We define the observation granularity as the number of blocks that $\mathcal{V}$ accesses before the next Prime+Probe of $\mathcal{A}$. Figure 14 visualises how this scenario depends on the scheduler:

1. Both processes are ready to execute. Ideally, $\mathcal{A}$ is already running before $\mathcal{V}$ to measure the first Argon2 block accesses.
2. The scheduler allows $\mathcal{A}$ to execute, and $\mathcal{A}$ fills the caches with its data by performing $C^{\mathcal{A}}_{\mathsf{P+P}}$ iterations of Prime+Probe.
3. The scheduler preempts $\mathcal{A}$ and runs $\mathcal{V}$ instead. $\mathcal{V}$ accesses $C^{\mathcal{V}}_{blocks}$ blocks during Argon2's internal passes through memory. The referenced blocks create an access pattern depending on the read blocks.

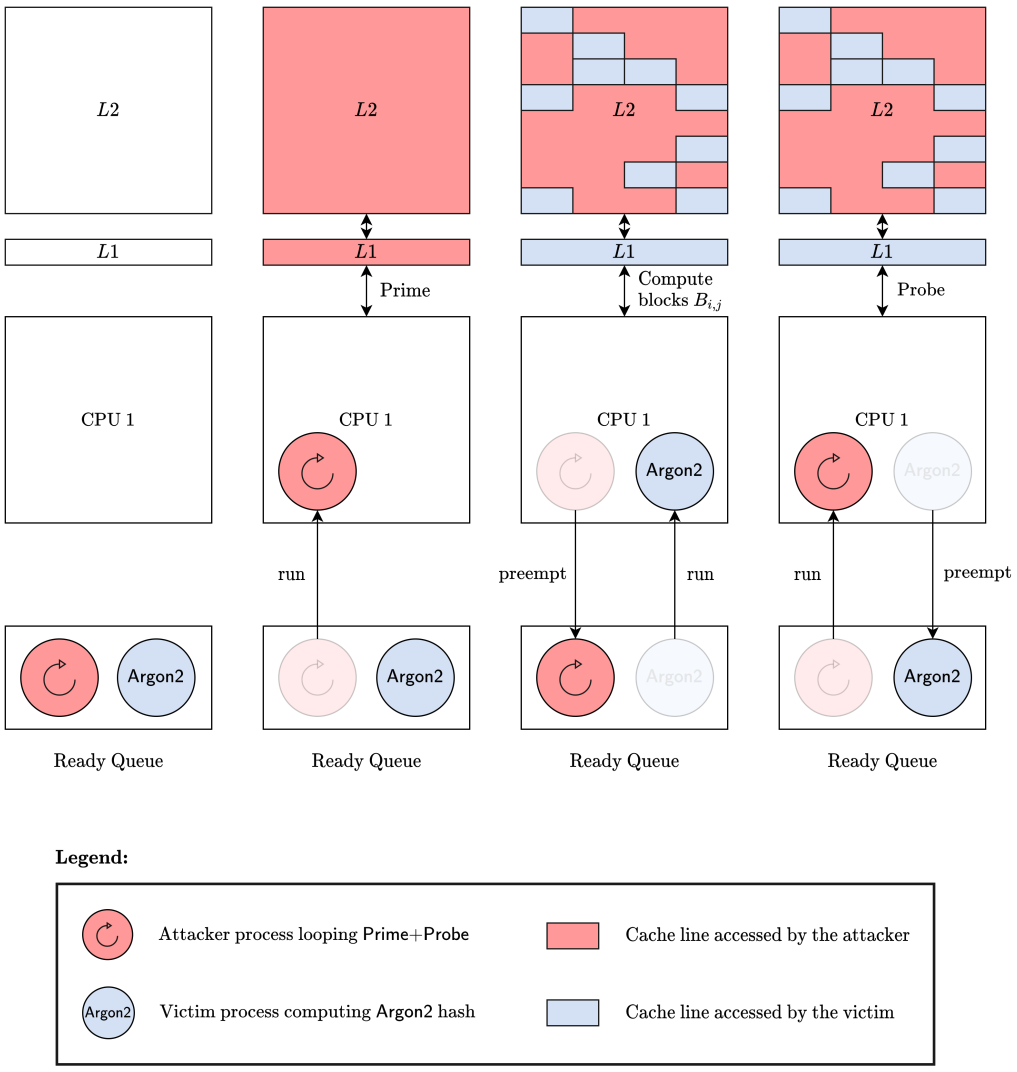


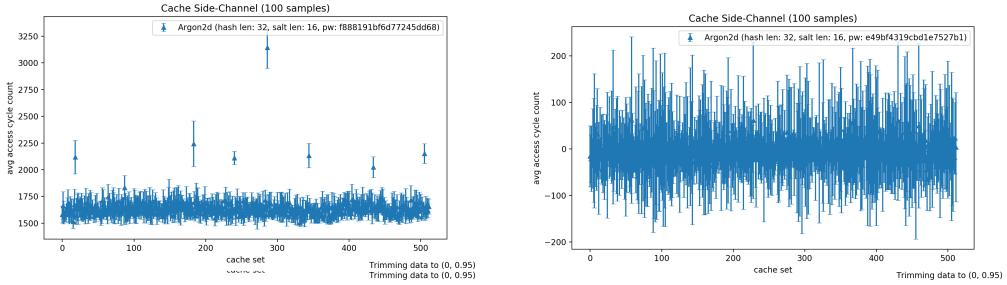
Figure 14: Visualisation of the cache side-channel granularity, depending on the scheduling of an attacker process that performs Prime+Probe in an endless loop, and a victim process calculating an Argon2 hash.

4. The scheduler interrupts $\mathcal{V}$ and schedules $\mathcal{A}$ again. $\mathcal{A}$ continues to execute Prime+Probe and can observe the block access pattern of $\mathcal{V}$ during the first probe phase.
5. This procedure is repeated from point 2 until $\mathcal{V}$ finishes its computation and terminates.

We know from the previous section that Argon2 accesses memory at the granularity of 1024-byte blocks, which corresponds on our Dell Latitude E6430 to 16 cache lines of 64 bytes in 16 different cache sets. Since our L1 is only 32 KiB, $\mathcal{V}$ likely replaces the entire L1 with Argon2 blocks. Therefore, we let $\mathcal{A}$ perform Prime+Probe on L2. We measured $C^{\mathcal{A}}_{\mathsf{P+P}}$ and $C^{\mathcal{V}}_{blocks}$ on our test laptop by printing the current timestamp counter for both processes. For

Ratio of Targeted Sets	$\mu_{C^{\mathcal{A}}_{\mathsf{P+P}}}$	$\sigma_{C^{\mathcal{A}}_{\mathsf{P+P}}}$	$\mu_{C^{\mathcal{V}}_{blocks}}$	$\sigma_{C^{\mathcal{V}}_{blocks}}$	μ_{msrmts}
512/512	34.4	2.3	6517.7	1460.9	20.1
32/512	595.6	41.9	6520.9	1445.6	20.1

Table 3: Empirical measurement of $C^{\mathcal{A}}_{\mathsf{P+P}}$ and $C^{\mathcal{V}}_{blocks}$ on a Dell Latitude E6430 for two attackers: the first performs prime on the entire L2, the second only on every 16th line (as each block spans 16 lines). For both numbers, we give the mean μ and the standard deviation σ averaged over 100 consecutive hashes. Moreover, we give the average number of useful Prime+Probe measurements μ_{msrmts}, where an observation is useful if it occurred after $\mathcal{V}$ accessed new blocks.



(a) not normalised (b) normalised

Figure 15: This figure shows the cache observations of an entire Argon2 hash for the parameters described in Section 2.5.5.

$\mathcal{A}$, we sample the time before prime and after probe, and for $\mathcal{V}$ we sample it on each call to the `index_alpha` function (which is used for to calculate the number of possible reference blocks in the same lane). Table 3 shows the mean and standard deviation for both $C^{\mathcal{A}}_{\mathsf{P+P}}$ and $C^{\mathcal{V}}_{blocks}$, and two different attackers. While the first attacker performs Prime+Probe to fill the entire L2 cache, the second attacker is slightly more sophisticated and only targets every 16th set, because Argon2's blocks cover 16 sets. Therefore, when $\mathcal{V}$ accesses a block, it evicts cache lines in all 16 sets covered by that block and thus, observing this eviction in a single set is sufficient. We see that once $\mathcal{V}$ is scheduled, it accesses approximately 6520 blocks before it is dispatched. This number is independent of the attackers Prime+Probe strategy, as it only depends on how much consecutive run time the process $\mathcal{V}$ is granted. Although the second attacker performs around 17 times more Prime+Probe measurements, it cannot observe the state of L2 after fewer block accesses.

Figure 15 shows the result of performing prime, then computing the complete Argon2 hash of a random password, before observing the L2 cache state with probe. It seems challenging to distinguish different passwords on those high-level observations, especially since they have significant noise.

After we have now established an understanding of the accuracy of cache side-channel observations, we discuss possible further research directions in the next section.

Further Research

The results of Section 2.5.5 suggest that applying Prime+Probe attacks on Argon2d faces the following problem: our L2 can store a total of 4096 different cache lines, but $\mathcal{V}$ accesses approximately $6520 \cdot 16$ lines. There are two main directions that further research could investigate:

1. Understand the Argon2 block access pattern: how likely can we distinguish the access patterns of different passwords? How does this distinguishing advantage decrease when more blocks are accessed? Those questions are similar to the OpenSSL key schedule analysis in Section 2.5.4, as it depends on the number of collisions that Argon2's block accesses cause in the same set of L2, i.e. on the number of Argon2 blocks that evict other Argon2 blocks instead of the attacker's data.
2. How can we reduce the number of Argon2 blocks that $\mathcal{V}$ accesses while it is scheduled? We suggest a few possible approaches to achieve this:
   - 2.1. The attacker could use multiple attack processes on the same core to slow down the victim process. Depending on the scheduler, this could reduce the time slice that $\mathcal{V}$ is running on the CPU, and thus, the number of blocks it accesses. However, there is probably a minimum time slice that a process is allowed to run so that the switching overhead does not outweigh the scheduled process' progress. Therefore, we would need to understand how many Argon2 blocks the victim can be processed in that time slice. Note that the second attacker of Table 3 was able to perform hundreds of Prime+Probe measurements, so it can tolerate a considerable slowdown since we only need one prime before $\mathcal{V}$ runs and one probe directly afterwards to obtain useful measurements.
   - 2.2. In the scenario of privileged attackers, such as an untrusted operating system, the Programmable Interrupt Controller (PIC) could be used to interrupt the victim frequently. Similar attacks have been applied to SGX enclaves [27, 66, 67, 49].
   - 2.3. An unprivileged approach to slow down $\mathcal{V}$ could try to evict all pages of $\mathcal{V}$ from L3, and ideally, also the TLB[12] entries to make $\mathcal{V}$'s memory accesses slower.
   - 2.4. In highly parallelised environments, $p \gg 1$ threads can be executed on separate cores. Since the Argon2 blocks are organised in a matrix with p rows, the number of different Argon2 blocks used on a single core decreases for a constant m, as each thread cannot reference the Argon2 blocks in the same slice (column).

Alternatively, we could perform the entire attack on L3. Although this has additional challenges (shared between cores, complex addressing), it is larger and could provide more useful access patterns for the same number of Argon2 block accesses than L2.

Assuming one of the previous approaches achieves to distinguish the Argon2 block access patterns of different passwords, this could be used to thwart the memory-hardness of Argon2. We can observe the memory access pattern of passwords during a dictionary attack and possibly avoid substantial computational cost by aborting a hash computation as soon as the access pattern does not match. Table 3 suggests that even with untampered scheduling, we have approximately ten Prime+Probe observations for every pass through memory (since in our test with $t = 2$, two passes had 20 useful Prime+Probe measurements on average). In the best case, when we can distinguish two passwords already after the first observation, which could reduce the computational costs for a password cracker by a factor of $10 \cdot t$.

[12]The Translation Lookaside Buffer caches virtual to physical address translations, and thus reduces the time to access memory.

Chapter 3

Conclusion

It is worth revisiting micro-architectural side-channels, and cache attacks in particular, for two reasons. First, they often persist in new designs due to their fundamental trade-off with performance. Second, the results often need to be revised for new software and hardware versions, as they are often tailored to specific versions as a consequence of the necessary empirical approach taken to investigate micro-architectural side-channels. We port the OpenSSL AES-CBC chosen-plaintext attack of Osvik et al. [51] – one of the first practical cache side-channel attacks – to contemporary hardware in the presence of various new optimisations. We explain the appropriate data structure for cache attacks and the reasoning behind its design. Moreover, port the Prime+Probe to physically indexed caches. We contribute the library CacheSC that implements Prime+Probe on virtually indexed caches, as well as unprivileged or privileged attacks on physically indexed caches.

Furthermore, we perform an initial review of novel side-channel applications on the AES key schedule and Argon2d. We show that only using the side-channel information on the first round of the AES key schedule, we can, for instance, identify $1.7 \cdot 10^{34}$ weak keys. We can detect those keys based on their memory access pattern and reduce the key search space by three bits. Further research could improve this result by using observations on the other key scheduling rounds. Finally, we argue that precise cache side-channel observations on Argon2d could be used for more efficient password cracking, bypassing the parameterisable number of passes over memory. Given the cache access pattern of a known target password, an attacker can abort Argon2d as soon as he detects a differing access pattern. We make the first step towards this goal and analyse the cache side-channel granularity of observations on Argon2d hashes. We observe on our Intel Ivy Bridge processor that a victim running Argon2 accesses approximately 25 times the number of cache lines in L2. We argue that this is due to the consecutive time slice that the scheduler allows the victim to run. Therefore, further research could try to reduce the number of blocks that the victim accesses, e.g. by slowing the process down.

Bibliography

[1] Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. pages 65–74, 04 2013.

[2] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM.

[3] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.

[4] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes (short paper). In *International Conference on Information and Communications Security*, pages 112–121. Springer, 2006.

[5] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the AES. In *Cryptographers' track at the RSA conference*, pages 271–286. Springer, 2007.

[6] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-vm attacks on xen and vmware are possible! *IACR Cryptology ePrint Archive*, 2014:248, 2014.

[7] Argon2. https://github.com/p-h-c/phc-winner-argon2. Visited on 2020-05-24.

[8] C Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. Highly efficient algorithms for AES key retrieval in cache access attacks. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 261–275. IEEE, 2016.

[9] Daniel J. Bernstein. Cache-timing attacks on AES. 2005.

[10] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II*, volume 1, pages 586–591. IEEE, 2005.

[11] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy*, pages 292–302. IEEE, 2016.

[12] Blockchain Charts. https://www.blockchain.com/charts/hash-rate. Visited on 2020-04-25.

[13] Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. Differential cache-collision timing attacks on AES with applications to embedded CPUs. In *Cryptographers' Track at the RSA Conference*, pages 235–251. Springer, 2010.

[14] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[15] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006(52), 2006.

[16] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

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

[18] Caisen Chen, Tao Wang, Yingzhan Kou, Xiaocen Chen, and Xiong Li. Improvement of trace-driven I-Cache timing attack on the RSA algorithm. *Journal of Systems and Software*, 86(1):100–107, 2013.

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

[20] cpufreq-set. https://linux.die.net/man/1/cpufreq-set. Visited on 2020-03-22.

[21] Gael Hofemeier and Robert Chesebrough. *Introduction to Intel® AES-NI and Intel® Secure Key Instructions*. 2012.

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

[23] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[24] Shay Gueron. Advanced encryption standard (AES) instructions set. *Intel, http://www.ferretronix.com/unigroup/intel_aes_ni/aes-instructions-set_wp.pdf, accessed 03.05.2020*, 25, 2008.

[25] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, May 2011.

[26] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on AES. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer, 2015.

[27] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 299–312, 2017.

[28] Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters–CPU Hardware Performance Counters for Security. *Black Hat Briefings*, 2015.

[29] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2*. Number 325383-060US. September 2016.

[30] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-042b. September 2019.

[31] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B*, chapter 18-19. Number 253669-071US. October 2019.

[32] Intel Ivy Bridge. https://www.7-cpu.com/cpu/IvyBridge.html. Visited on 2020-03-21.

[33] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. pages 629–636, 08 2015.

[34] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. pages 299–319, 09 2014.

[35] Vincent Rijmen Joan Daemen. AES Proposal: Rijndael, version 2, AES submission. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf, 1999.

[36] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–17. Springer, 2009.

[37] KeePass Password Safe. https://keepass.info/index.html. Visited on 2020-05-24.

[38] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In *European Symposium on Research in Computer Security*, pages 97–110. Springer, 1998.

[39] The kernel development community. The kernel's command-line parameters. https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html. Visited on 2020-03-22.

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

[41] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[42] Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In *Cryptographers' Track at the RSA Conference*, pages 187–202. Springer, 2008.

[43] Hugo Krawczyk. On extract-then-expand key derivation functions and an hmac-based kdf. *Draft available at http://www. ee. technion. ac. il/hugo/kdf*, 2008.

[44] Cédric Lauradoux. Collision attacks on processors with cache and countermeasures. In Christopher Wulf, Stefan Lucks, and Po-Wah Yau, editors, *WEWoRC 2005 – Western European Workshop on Research in Cryptology*, pages 76–85, Bonn, 2005. Gesellschaft für Informatik e.V.

[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[46] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on Intel Core2 processor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 121–134. Springer, 2007.

[47] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[48] MDS Attacks: Microarchitectural Data Sampling. https://mdsattacks.com/. Visited on 2020-04-22.

[49] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[50] Michael Neve. Cache-based vulnerabilities and spam analysis. *Doctor thesis, UCL*, 2006.

[51] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. January 2005.

[52] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 01 2002.

[53] pagemap, from the userspace perspective. https://www.kernel.org/doc/Documentation/vm/pagemap.txt. Visited on 2020-05-27.

[54] Gabriele Paoloni. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf, 2010-09.

[55] Mathias Payer. HexPADS: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[56] Performance Monitoring Events. https://download.01.org/perfmon/index/. Visited on 2020-04-01.

[57] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[58] sched_setaffinity. https://linux.die.net/man/2/sched_setaffinity. Visited on 2020-03-22.

[59] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15:71, 2015.

[61] SourceForge KeePass. https://sourceforge.net/projects/keepass/files/KeePass%202.x/2.45/KeePass-2.45-Source.zip/download. Visited on 2020-05-24.

[62] Herb Sutter. The Free Lunch Is Over A Fundamental Turn Toward Concurrency in Software. 2013.

[63] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23:37–71, 07 2010.

[64] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 62–76. Springer, 2003.

[65] Yukiyasu Tsunoo, Etsuko Tsujihara, Maki Shigeri, Hiroyasu Kubo, and Kazuhiko Minematsu. Improving cache attacks by considering cipher structure. *International Journal of Information Security*, 5(3):166–176, 2006.

[66] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[67] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195, 2018.

[68] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 05 2019.

[69] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on AES in virtualization environments. In *International Conference on Financial Cryptography and Data Security*, pages 314–328. Springer, 2012.

[70] Jos Wetzels. Open sesame: The password hashing competition and argon2. *arXiv preprint arXiv:1602.03097*, 2016.

[71] Yuval Yarom. Mastik: A Micro-Architectural Side-Channel Toolkit. https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf, 2016.

[72] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[73] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[74] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012.