UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

MemJam+Abort: Combining two microarchitectural side-channel attacks

MemJam+Abort: Kombination zweier mikroarchitektonischer Seitenkanalangriffe

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Jonas Sebastian Sander

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Jan Wichelmann

Lübeck, den 15. November 2018

Abstract

Microarchitectural side channel attacks exploit particular behaviours of processors to collect security-relevant information. A well-studied subfamily is the class of cache attacks, which exploit the properties of modern cache hierarchies. This work summarizes the fundamental properties of modern cache hierarchies and briefly explains the principle of virtual memory to subsequently discuss some essential representatives of the family of cache attacks. We also implement the attacks Prime+Abort and MemJam as well as an algorithm to determine the eviction sets needed for the prime phase of cache attacks. We then analyse whether it is possible to add a temporal resolution to the MemJam attack by combining it with Prime+Abort (to enable attacks on asymmetric encryptions). While we were not able to fully implement a new attack, we can present some intermediate results and implementations that support the development of an attack in the future.

Kurzfassung

Mikroarchitektonische Seitenkanalangriffe nutzen bestimmte Verhaltensweisen von Prozessoren aus, um sicherheitsrelevante Daten zu sammeln. Eine gut untersuchte Unterfamilie dieser Angriffsklasse sind die sogenannten Cache Angriffe, welche die Eigenschaften moderner Cache Hierarchien ausnutzen. Diese Arbeit führt in die grundlegenden Eigenschaften moderner Cache Hierarchien ein und erläutert kurz das Prinzip des virtuellen Speichers, um anschließend einige wichtige Vertreter der Familie der Cache Angriffe vorzustellen. Wir implementieren zudem die Angriffe Prime+Abort und MemJam sowie einen Algorithmus mit dem aktuelle Cache Angriffe, die für die sogenannte Prime Phase benötigten eviction sets bestimmen. Anschließend analysieren wir, ob es möglich ist, den MemJam Angriff durch Kombination mit Prime+Abort um eine zeitliche Auflösung zu ergänzen (um Angriffe auf asymmetrische Verschlüsselungen zu ermöglichen). Während es uns nicht möglich war einen neuen Angriff vollumfänglich zu implementieren, können wir einige Zwischenergebnisse und Implementierungen präsentieren, die die Entwicklung eines Angriffs in der Zukunft unterstützen.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 15. November 2018

Acknowledgements

At this point, I would like to thank Professor Eisenbarth and Jan Wichelmann for their great patience and excellent support. Both assisted me with advice and practical support throughout the entire project and often dissolved my incomprehension with detailed explanations and illustrative panel drawings. From time to time the unknown development in C and x86 assembly and the numerous index orgies challenged me a lot, but mostly Jan could help with handy tips. I would also like to thank all the other members of the Institute for IT Security. I felt very welcome and comfortable. To listen to numerous exciting lectures, when the Institut members talked about their work, was a real Highlight. In particular, I would also like to thank Moritz Krebbel and Daniel Moghimi for their support by finding eviction sets. I thank my cousins Malte and Tobias Sander for their many comments on my English. And last but not least I have to thank my parents, whom I can always rely on. They not only supported me during my bachelor thesis, but throughout all my studies.

Contents

1 Introduction

The "classical" cryptanalysis treats cryptographic primitives as abstract mathematical objects that transform an input, possibly by using a key, into an output. The level of security of a cryptographic algorithm is made accordingly dependent on its mathematical properties and the key length. Typically it is assumed that the corresponding implementations are ideal black boxes whose internal operations cannot be observed or disturbed from the outside. For a cryptographic system to be secure, the utilized secret key must not be disclosed in any way. Since cryptographic algorithms have been verified for years by many experts to be resistant against classical cryptanalysis, attackers try to exploit vulnerabilities in the hardware and the system (the black box) in which the cryptographic schemes are implemented. They use information such as energy consumption and electromagnetic emanation, that leak through the implementation during protocol execution and are not taken into consideration in classical security models. This so-called side-channel information correlates with the internal state of the cryptographic implementation and this way also with the secret key. As a result, the attackers can significantly weaken or even by-pass the theoretical security of security solutions entirely [ZF05].
Kocher developed the class of attacks described above in 1996 [Koc96, KJJ99]. These side-channel attacks are a large and currently very active field of cryptographic research. At this point, we will briefly introduce some subfamilies of the side channel attacks.

- Timing Attack: These attacks take advantage of the fact that the runtime of cryptographic algorithms and their operations depend to a certain extent on the secret keys. In this way, it is possible for the attacker to conclude the secret key with simple runtime measurements [ZF05]. Kocher [Koc96] introduced the idea of Timing Attacks, Brumley and Boneh demonstrated the practicability of remote timing attacks against OpenSSL in 2005 [BB05].

- Fault Attack: This class of attacks takes advantage of the fact that external influences such as manipulated supply voltage, photon irradiation or heat can induce errors in the executions performed by hardware that lead to failure of the system security [BDL97, Til11].

- Power Analysis Attack: This type of attack is only possible on implementations in hardware. It was also introduced by Kocher in 1999. It takes advantage of the fact

that the energy consumption of a cryptographic device can correlate with its operations and parameters [KJJ99].

- EM Attack: The hardware components of computers often generate electromagnetic emanation in their calculations. An attacker can measure this emaniation and possibly draw conclusions about the performed operations and the processed data [ZF05].

- Low-Bandwidth Acoustic Attack: This attack exploits the fact that many computers generate high pitched noises during their computations due to vibrations in the electronic components. This results in a side channel that can be used to extract security-critical information. Genkin et al. [GST14] demonstrate that a mobile phone placed next to the victim's computer is sufficient to carry out such measurements. Besides, they show a similar attack that exploits the electronic potential of a computer housing that can be picked up by touching it with one's hand.

- Meltdown and Spectre: The publication of these two attack classes has met with great media interest in the recent past. This is because they exploit one of the most significant IT security vulnerabilities for a long time and are not based on a software bug, but on a design flaw in the most common processor microarchitectures (x86, x64, ARM), which can be exploited via side channel attacks. Both attack families make considerable use of the speculative execution processor feature to gain access to the entire address space of the victim, including all security-critical data stored in it [KGG+18, LSG+18].

- Cache Attack: These attacks exploit the microarchitectural structure of modern CPUs. For this reason, they are also often called microarchitecture attacks. Modern CPUs have caches, fast, small buffer memories that are located between main memory and CPU and are intended to speed up access to requested data. If a required address is in the cache, it is called a cache hit. Otherwise, it is called a cache miss. Cache hits lead to fast access times, while cache misses lead to significantly slower access times, because the requested data must be retrieved from a lower and slower level of the memory hierarchy to the CPU. In a cache attack, an attacker tries to exploit this characteristic of the CPU architecture. To do this, he prepares the cache and then measures the access times to specific addresses. In this way, an attacker can build a side channel based on runtime differences [Hu92]. Unlike timing attacks, the runtime differences in cache attacks are not directly based on conditional branches in cryptographic implementations. The runtime differences are rather based on the properties of the microarchitecture of the CPU, and by preparing the cache, the at-

tacker can then force runtime differences that allow conclusions about the secret key. With Prime+Abort there is also an attack that exploits the cache without running time measurements. Instead of time dependencies, Intel's implementation of hardware transactional memory called Intel Transactional Synchronization Extensions (Intel TSX) is exploited [DKPT17, gui18].

The project developed during the thesis aims to combine the attacks Prime+Abort and MemJam. MemJam achieves intra cache level resolution and creates time dependencies in otherwise constant-time implementations. However, only full encryptions can be measured with the attack, and thus only implementations of symmetric encryptions can be attacked. With AES and DES, the attack of some S-Box[1] indices allows tracing back the entire key statistically, since a large part of the key is also included in the last encryption round. In many asymmetric implementations, such as RSA, statistical conclusions of this kind are not as easy to reach because the individual key parts are processed one after another. As a result, it is possible to conclude some key bits, but not a large part of the key as in symmetrical ciphers. By combining MemJam with Prime+Abort, we can determine which key block is currently processed and apply the MemJam attack to each block. In this way, significantly more key bits can be exposed than with previous MemJam attacks. The work is organised as follows. In Chapter 2 we introduce the background and present the related work. In particular, we will discuss the cache architecture of current Intel processors. Furthermore, the general structure of cache attacks is presented to introduce the attacks Flush+Reload, Prime+Probe, Prime+Abort and MemJam. We close the chapter with a section that explains the determination of eviction sets. In the third Chapter we discuss a combination of MemJam and Prime+Abort and point out several problems. In the end we present a conclusion with details about the success and failure of the MemJam+Abort, point out open problems and provide an overall summary.

[1] According to Claude Shannon confusion and diffusion are the two basic concepts with which strong ciphers can be realized. Confusion obscures the relationship between key and ciphertext and is realized in AES by using a substitution table called S-Box [Sha49, Pub01].

2 Background and Related Work

2.1 Cache Architecture

The rapidly increasing transistor density and clock speed of modern CPUs lead to ever stricter demands on the speed of memory requests. Today's processors use combinations of different caches and advanced caching techniques to meet their requirements. These structures of relatively small memories are called the cache architecture of a CPU. There are a wide variety of architectures that meet a range of requirements. The architecture of the current eighth generation Intel processors is described below. Where possible, more exact numbers and descriptions refer to the processor of the type used in the thesis' project: Core i7-8650U. Unlike many other eighth-generation Intel processors, it is not based on Coffee Lake microarchitecture, rather on the improved Kaby Lake architecture called Kaby Lake Refresh [Wik].
Cache architectures are based on the principle of locality. The principle is made on two assumptions [PH14]:

- The temporal locality, which assumes that addresses in memory are more likely to be reaccessed if their last access happened recently. Cache architectures, therefore, keep recently used data as close as possible to the CPU in the lower cache levels.

- The spatial locality, which assumes that addresses that are located close to recently accessed addresses are more likely to be queried again than addresses that are located further away. Processors take this assumption into account by also loading speculatively neighbored data of currently accessed addresses into the cache.

To follow the principles above, cache architectures often have several hierarchically arranged cache levels. The smaller the cache level number, the closer is the memory to the CPU and the faster and smaller is the memory. At each cache level, the memory is either uniquely partitioned to each CPU core or shared across all cores. The present processor has three cache levels, with each processor core having its own L1 and L2 cache, while the last cache level (L3 or LLC) is shared across all cores. Besides, the first cache level is divided into an instruction cache (L1I) and a data cache (L1D), while the other two cache levels are not divided [Wik]. Figure 2.1 shows a graphical illustration of the cache hierarchy and Table 2.1 shows all relevant data of our target processor.

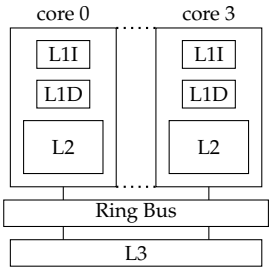

Figure 2.1: The schematical structure of the cache architecture of our victim processor.

	L1I or L1D*	**L2***	**L3**
Size	32 KiB	256 KiB	8 MiB
Associativity	8	4	16
Sets	64 Sets	1024 Sets	1024 Sets
Line Size	64 B	64 B	64 B
Slices	-	-	8

Table 2.1: The relevant cache parameters of our victim processor Intel Core i7-8650U . (*)The data for the L1 and L2 cache refer to individual cores. For example, the L2 cache has 4*1024 sets.

The smallest unit of information that can or cannot be stored in a cache is called a *block* or *line*. If the CPU requests a line and that is in the cache, it is called a *cache hit*, otherwise it is called a *cache miss*. As already described in the introduction, cache misses lead to significant time delays, because the requested line must be transferred from a lower level of the memory hierarchy to the level at which the miss occurred [PH14].
In principle there are two schemes to keep cache and deeper memory levels consistent:

- *Write-through:* With each update in the cache the next level of the memory hierarchy is also updated. A write buffer that holds the data until it is written back to the main memory can speed up the process.

- *Write-back:* A modified line is only written back to the main memory when it is removed from the cache. A write-back buffer can be used to speed up the caching of new lines. Alternatively, a store buffer can also be used.

Like all Intel processors based on the current microarchitectures Haswell, Broadwell, Skylake, Kaby Lake and Coffee Lake, this processor uses the write-back policy at all cache levels [Wik].
There are mainly three ways in which a cache level can be organised. These are *directly mapped*, *set associative* and *fully associative* caches, whereby the first and the latter are in principle only special cases of the second. With direct mapped caches, each line can only fill one location in the cache depending on its address (set associativity with sets of size 1). With set associative caches, the cache is divided into sets. A line can take any position in a particular set, depending on its address. Fully associative caches are like set associative caches with only one set. Each line can be cached anywhere, regardless of its address [PH14]. Each cache level of the present processor is set associative. The numbers of sets and the sizes of the sets (called *associativity*) can be found in Table 2.1.

In the following, we will look at how the L3 cache of the present processor is addressed. The six least significant bits, the so-called *line offset*, determine the position of the addressed data byte within a line. The next ten bits following the line offset, called *set index*, describe to which cache set the data is assigned. All other bits belong to the so-called *tag*, which is stored in the cache in addition to each line and is used to determine the addresses and their data within a set uniquely. Up to here, the description corresponds to the general way of addressing a set associative cache. Additional current Intel processors divide the L3 cache into so-called *slices*, in order to enable parallel access by multicore CPUs. The number of slices corresponds mostly either to the number of physical cores or logical cores. The processor used below has eight slices. The assignment of the lines to the slices is done via an undocumented hash function, which uses all bits of the physical address except for the line offset, to distribute the accesses uniformly over all slices [LYG+15]. In the past, these hash functions were reverse-engineered several times [IES15b, IGI+16, YGL+15, MLSN+15, KAGPJ16]. Reverse-engineering will not be necessary for the attacks described later. This makes the attack more flexible for different Intel processor versions and generations. Figure 2.2 gives a rough overview of the entire structure of the L3 cache of this processor.

Another important feature of the L3 cache for cache attacks is that every cache line in lower cache levels must also be in the L3 cache, so the L3 cache is called to be *inclusive*. This means that any line accessed by a core must not only be placed in the private L1 cache but also in the shared L3 cache. On the other hand, if a line is evicted from the L3 cache, this line must also be invalidated in all private L1 and L2 caches [PH14].

As already described, all cache levels of the present processor are set associative, and each cache set can hold a certain number of lines determined by the associativity. If the CPU assigns a new line to a set of valid lines, one line must be evicted. The *eviction policy* is the scheme by which a processor evicts lines from the cache. The processor removes the line from the cache that has not been accessed for the longest time and in this way exploits the temporal locality. This eviction policy is called *least recently used* (LRU). Since tracking this policy accurately would be very costly, many processors use an approximation called pseudoLRU. Often it is not documented for the current Intel processors which eviction policy they use. This also applies to the present processor. The following procedure assumes that the used processor implements a pseudoLRU policy.

Caches have a significant influence on the performance of CPUs, which is noticeable with every memory access. For this reason, processor manufacturers are rapidly driving the development of fast cache systems and regularly introducing new technologies. The high speed at which new features are introduced makes a comprehensible evaluation of microarchitectural security impossible. As a consequence, well researched cryptographic

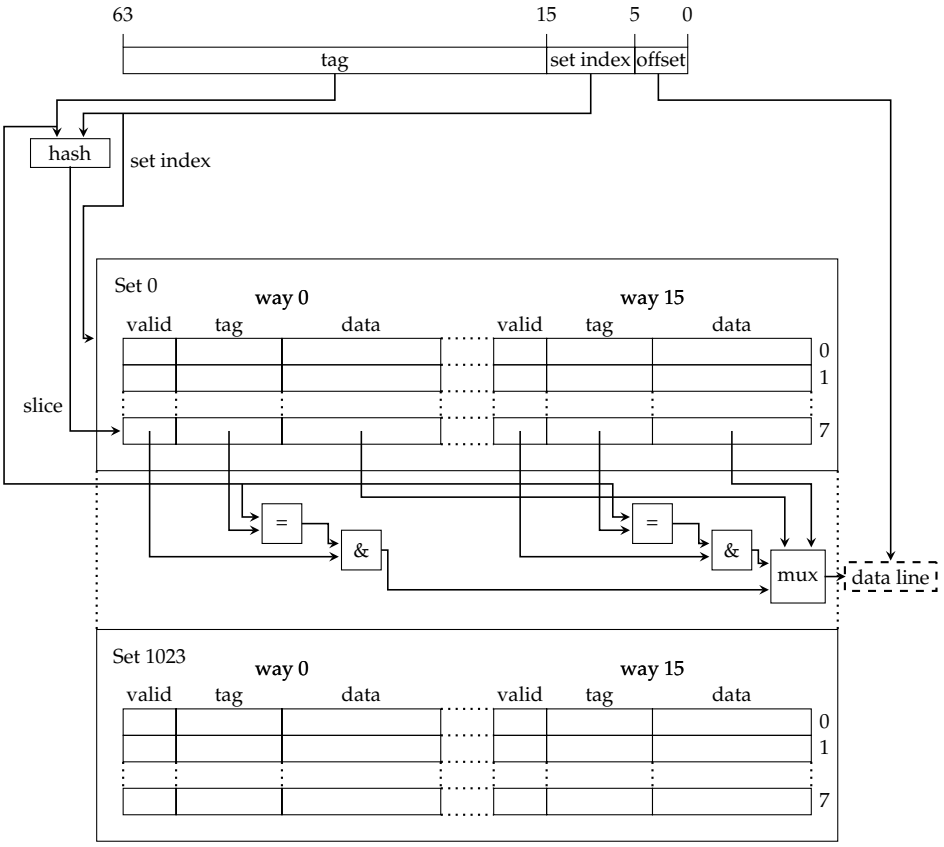

Figure 2.2: The graphic gives a rough overview of the cache of our victim processor and its addressing. It can be seen how the requested cache line is determined based on the physical addressing of the cache.

implementations can become insecure with the introduction of the next processor generation, or worse when discovering a previously unobserved behavior. In section 2.3 we will explore such unwanted and exploitable behavior.

2.2 Virtual Memory

In addition to the cache architecture of the CPU, the memory hierarchy of a computer usually has two further levels consisting of the *main memory* and a mass storage such as HDDs or SSDs. The main memory acts as a kind of cache for the mass storage. This technique is called *virtual memory*. Primarily, two goals are pursued with the use of virtual memory [PH14].

First, the use of virtual memory ensures that various programs such as virtual machines can securely share the *physical address space* of the main memory. This means that each

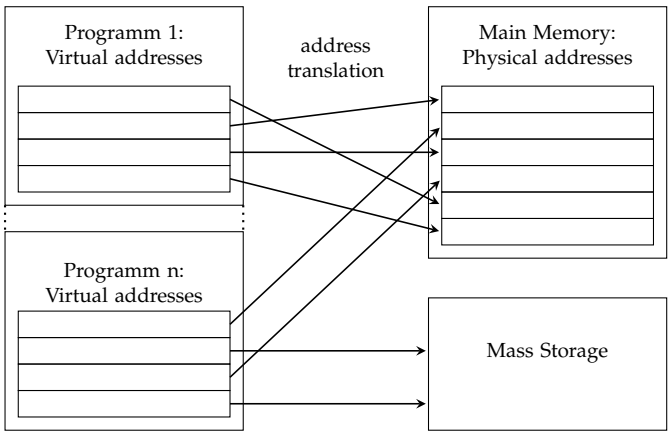

Figure 2.3: This figure shows the relationships between physical and virtual addresses through address translation. The technique of virtual memory enables each program to gets its own contiguous virtual address space.

program can read and write only the physical addresses assigned to it.

The other reason for using virtual memory is that many programs do not fit into the main memory. By using virtual memory, operating systems manage automatically the two memory levels of main and mass storage, so that inactive program parts are stored in the mass storage (swap space), and active program parts remain in the main memory.

Since the requirements of a program for the size of its main memory change dynamically, virtual memory assigns each program to its own contiguous *virtual address space*. Virtual memory uses a technique called *address translation* to map the physical addresses of the main memory to the virtual addresses of the programs, threrefore isolating the virtual address spaces of the individual programs from each other. Figure 2.3 shows a schematic representation of the relationship between virtual and physical address space.

Unlike caches, the smallest memory size to be processed is called a *page* [Den70]. *Page faults* occur if a requested page is not in the main memory. Usually, the size of a page is 4 KiB. Most current CPU architectures also support larger pages. In Linux, these are called *Huge Pages*, in BSD *Super Pages* and in Windows *Large Pages*. Since the project environment runs Linux, the following refers to Huge Pages with a size of 2 MiB [Deb].

When using virtual memory, each address gets a *page number* and a *page offset*. The page number determines the position of the page in the main memory, and the page offset determines the position of the addressed byte within the page. For some cache attacks, a so-called *eviction set* must be determined. It is necessary to find addresses that match the same cache set. The cache of the present processor is *physically indexed*, so the physical address determines in which set a line is allocated. First of all, it is not clear which virtual addresses have the same set index in their associated physical addresses. Because the

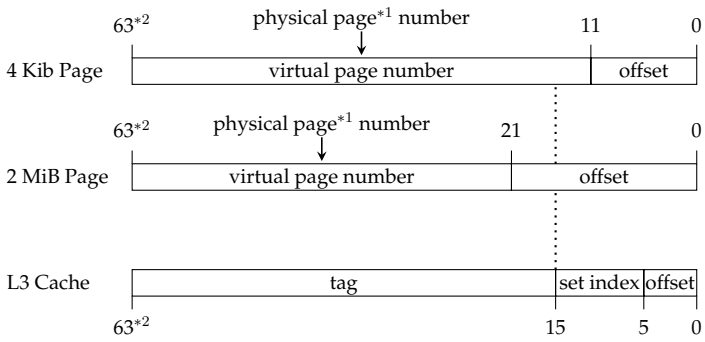

Figure 2.4: The graphic shows the process of address translation in the context of addressing the L3 cache. *1Physical pages are often called frames in the literature. *2This number is not accurate. Often the possible 64-bit address space in x86-64 instruction set architectures is not used for either virtual or physical addresses. Besides, the virtual address space of current machines is often larger than the physical address space [Hen17, PH14].

page offset remains constant during address translation, the address bits in this interval are identical for physical and virtual addresses. If the page offset is large enough, the set index can be read from the virtual address. Figure 2.4 shows a graphical representation of the address translation for 4 KiB pages and 2 MiB huge pages in the context of addressing the L3 cache. As one can see, 2 MiB pages are sufficient to determine the cache set to a virtual address as described in [LYG+15, PH14]. A detailed description of determining the eviction sets follows in subsection 2.3.6. The mapping of virtual to physical addresses is stored individually for each program in so-called page tables.

2.3 Cache attacks and their variants

Cache attacks are a well-researched subfamily of side channel attacks. There exist several attacks and the four most common are: Prime+Probe (with Prime+Abort), Flush+Reload, Evict+Time and MemJam. Due to their comparatively high resolution and distribution in the literature, we will only consider the variants Prime+Probe, Flush+Reload and MemJam in this chapter.

2.3.1 General Structure

Disselkoen et al. [DKPT17] classify attacks (at a high-level perspective) into a *pre-attack portion* and an *active portion*. This subdivision enables a quick and precise description of attacks and is also made in the subsequent sections.

The pre-attack portion collects important architecture and runtime specific information. The active portion that uses this information to analyse the victim's memory access consists of three phases:

1. The *initialisation phase* prepares the cache in a certain way,
2. the *waiting phase* gives the victim the possibility to request the observed memory addresses,
3. and the *measurement phase* performs measurements to determine whether the cache state has changed in a way that would imply memory accesses to the observed memory addresses.

2.3.2 Flush+Reload

Flush+Reload [GBK11, YF14] monitors accesses to a specific memory line. This results in a high accurate attack which is less prone to false positives. Furthermore, the attack is successful across cores due to the properties of the L3 cache. The attack can monitor access to instructions and data. Due to the content-based page sharing technique, also called memory deduplication, which is used by many virtual machine monitors, the attack also works across VMs. The attack operates in the following way:

- The active portion:
  1. Initialization phase: The attacker uses Intel's `CLFLUSH` instruction to evict the observed address out of the cache.
  2. Waiting phase.
  3. Measurement phase: The attacker reaccesses the observed address and measures the time for the access operation. If the access is fast, the attacker can conclude that another process has accessed the address because it has been reloaded into the cache. If the access was slow, consequently the address was not accessed.
- The pre-attack portion: Before the attacker initiates the active portion of the attack, he must determine a timing threshold and an exact target address in the virtual address space, which maps to the physical address the attacker wants to monitor.
  1. Time threshold: The attacker must determine the timing of a fast and a slow access in the measurement phase of the active portion.
  2. Determine the target address: There exist two procedures which were introduced by Yarom and Falkner [YF14]. Both are limited to shared memory, so Flush+Reload can only be applied to addresses in shared memory.

With Flush+Flush [GMWM16] and Evict+Reload [GSM15], there are also two variants of Flush+Reload. The first variant exploits the fact that the `CLFLUSH` command itself leads to temporal dependencies and thus makes it possible to combine the measurement and initialisation phases. The latter variant uses a prime phase instead of the `CLFLUSH` command (just like Prime+Probe) to evict the cache line. So this variant avoids the `CLFLUSH` command.

2.3.3 Prime+Probe

Prime+Probe does not rely on shared memory. The first forms of Prime+Probe were targeting the L1 cache [Per05, OST06]. Recent work extended the attack on the L3 cache of Intel processors, allowing attacks over several cores [IES15a, KAGPJ16, LYG+15]. Like all L3 cache attacks, Prime+Probe can monitor access to both instructions and data. Also, L3 Prime+Probe works across VMs. The attack monitors a single cache set and detects accesses from any other process including the operating system. The attack proceeds in the following phases:

- The active portion:
  1. Initialization phase (called *prime*): The attacker accesses enough cache lines from the cache set to be monitored such that this cache set is only filled with his data.
  2. Waiting phase.
  3. Measurement phase (called *probe*): The attacker reloads the data that he already accessed during the initialisation phase and observes how long this operation takes. If the victim has not accessed any data in the observed cache set, the operation is fast. However, if the victim has accessed data in the cache set, the operation takes longer because of cache misses.
- The pre-attack portion: Before the attacker initiates the active portion of the attack, he must determine a timing threshold and an eviction set.
  1. Timing threshold: The attacker must determine the timing of a fast and a slow access is in the measurement phase of the active portion.
  2. Eviction set: The attacker must determine a set of cache line addresses that fill an entire cache set (See subsection 2.3.6).

Typically, several runs of this attack are performed sequentially. In this case, the probe phase can serve as the prime phase for the following attack cycle.

2.3.4 Prime+Abort(-L3)

Prime+Abort is a relatively new attack introduced in 2017 by Disselkoen et al. [DKPT17]. Last-level cache attacks are typically based on precise timer operations for the measurement phase (see subsections 2.3.2 and 2.3.3). Many defence strategies against these attacks are based on limiting access to precise timers. Prime+Abort undermines these defence strategies by using Intel's TSX hardware instead of timers and completely avoids a measurement phase and the determination of a time threshold. Apart from that, this attack beats Prime+Probe both in its accuracy, as well as in its efficiency and maximum detection speed while producing less false positives [DKPT17]. Before the attack process is described below, we give a brief overview of how Intel's TSX works.

Intel Transactional Synchronization Extensions (Intel TSX)

Multithreaded applications use the increasing number of cores in modern processors to work as efficiently as possible. Programmers must use synchronisation mechanisms to manage the shared data of such applications. Typically, serialisation mechanisms are used that protect critical sections with locks. The use of locks during serialisation is pursued quite conservatively in practice. This procedure limits parallel execution more than necessary and therefore has a negative effect on the performance of the application [gui18].
Unlike the traditional locks described above, Intel's TSX [2] allows multiple threads to run in parallel, which are aborted when a conflict occurs. With the commands `XBEGIN` and `XEND` TSX allows to divide the program code into transactions. For each transaction TSX guarantees either completeness, which means that all memory changes made during the transaction will be visible atomically to all other processors and cores, or that the transaction will be aborted, which means that all changes caused by the transaction will be undone. If the transaction is terminated, a fallback routine is executed, which is also specified by the developer [gui18].
A TSX transaction may also terminate for some reasons without a memory conflict occurring. The reason for the termination of a transaction is reported by TSX using the `EAX` register. Table 2.2 lists the most important termination reasons from the Intel Software Developer's Manual [gui18]. The last two abort reasons were communicated by Dyce et al. [DHKL15] and exploited in [DKPT17] using Prime+Abort.
The attack described here exploits the fact that a transaction is aborted if a cache line that was read during the transaction is deleted from the L3 cache (Reason 9). With this abort criterium in Intel's TSX, one can use the fallback routine described above to construct a hardware callback that notifies the attacker as soon as a corresponding cache line has been

[2] TSX is a Hardware implementation of transactional memory. See also [HEM93, Sco13].

1. Internal buffer for tracking the transaction state overflows.
2. Execution of `XABORT`, `CPUID`, `PAUSE`, `ENCLS` or `ENCLU` Instruction
3. Execution of system calls (e.g. `fopen()`)
4. Interrupts
5. Exceed implementation specific depth for nesting transactions
6. Access violation and page faults (implementation specific)
7. False dependencies
8. A cache line in the write set of the transaction (written during the transaction) is evicted from the L1 cache
9. A cache line in the read set of the transaction (read during the transaction) is evicted from the L3 cache

Table 2.2: Reasons for transactional aborts in Intel's TSX

accessed by another process [DKPT17].

Attack Procedure

As already described, the attack does not require a measurement phase. Memory requests by the victim are not registered by time operations, but directly by hardware callbacks via TSX. This means that the attacker can wait for the victim's memory access to happen and can avoid splitting the attack into intervals at which he suspects the memory accesses. This prevents the attacker from missing the victim's memory accesses due to an incorrect subdivision. Consequently, the attack has a higher resolution [DKPT17].
The attack has a disadvantage compared to Prime+Probe and Flush+Reload. It cannot monitor multiple targets, i.e. cache sets or addresses, simultaneously and distinguish between their accesses. Prime+Abort works across cores and can detect access to instructions and data. The attack goes through these steps [DKPT17]:

- The active portion:
    1. Initialization phase (called "prime"): The attacker opens a TSX transaction and accesses enough cache lines from the cache set to be monitored such that this cache set is only filled with his own data.
    2. Waiting phase: The attacker waits until the transaction from the prime phase is terminated (with the correct termination criterion). With the aborted transaction, the attacker can now conclude that an address in the monitored cache set has been accessed. The attacker gets the same information as during a Prime+Probe attack.

- The pre-attack portion: The attacker must determine an eviction set as in the regular Prime+Probe attack. A time threshold is not required.

2.3.5 MemJam

MemJam was introduced in 2017 by Moghimi, Eisenbarth and Sunar and later further researched in collaboration with Jan Wichelmann. It uses false dependency of memory read-after-writes and is the first intra cache level attack that can be applied to all major Intel processors [MWES18]. The attack is based on 4K Aliasing which is explained in the following section. At this point, we should also mention the CacheBleed attack, which was developed by Darom, Genkin and Heninger and also exploits false dependencies [YGH17].

4K Aliasing

Aliasing occurs when two objects have the same name [PH14]. In 4K Aliasing, this ambiguity occurs due to the virtual addressing of the L1 cache when two virtual addresses may reference the same physical address [MWES18].
Using 4 KiB Pages the physical and virtual addresses share the last twelve bits (see figure 2.4). This means that two virtual addresses which match in the last twelve bits, can refer to the same physical address. Simultaneous reading and writing of these addresses is not possible and slows down both the reading and writing process, because an address translation must be performed before the dependency of the two accesses can be resolved. This simultaneous access occurs with out-of-order execution on two different virtual processor cores that share a physical core (Intel's implementation of this technique known as *Simultaneous Multithreading* is called *Hyper-Threading*.) [MWES18, gui18].
Because our test environment supports the x86-64 instruction set as well as the x86-32 instruction set, the processor works using the latter instruction set with a word size of 32 bits or 4 bytes. From this, it follows that not the last 12 bits of the addresses must agree, but rather the bits 2 to 11 to cause the described dependency [MWES18].
Moghimi, Eisenbarth and Sunar describe in [MWES18] a series of experiments to analyse the memory dependencies of virtual processor cores. They use two threads running on the same physical core but different virtual cores. Both threads perform memory operations, and one thread (the future victim thread) also performs time measurements. The experiments show that read-after-write false dependencies with simultaneous access to addresses of the same cache line but different line offsets cause a two cycle penalty, while access to addresses of the same cache line and the same line offset cause a ten cycle penalty. Thus the attack channel achieves intra cache line resolution. The attacker can observe the

address bits 2 to 11, where the bits 2 to 5 describe the intra cache line leakage. Besides, the experiments showed that the attacker must write continuously on the victim's address in order to maintain the effectiveness of the attack channel [MWES18].
The attack described here uses the read-after-write false dependencies and the associated timing behaviour to open a side channel on the cache. The attacker intentionally creates such dependencies by comparing the last twelve bits of his address with the last twelve bits of the monitored address of the victim [MWES18].

Attack procedure

MemJam uses read-after-write false dependencies to create time dependencies over the described side channel even in constant time implementations. The measured time dependencies are then exploited by a correlation attack [MWES18]. The attack takes place in the following steps:

- The active portion:
  1. Initialization phase: The attacker starts a process (same physical core as the victim process) in which he writes to a particular address continuously. This address matches the virtual memory offset, on which the victim performs security-related read operations, in the last twelve bits.
  2. Waiting/Measurement phase: Now the attacker asks the victim to execute encryption on a given plain text. The attacker records pairs of ciphertext and corresponding execution times. Higher times mean more accesses to the observed memory offset. The attacker repeats this step to continue collecting ciphertext-time pairs for the later following correlation attack.
  3. Analysis phase: Subsequently, the recorded ciphertext-time pairs are exploited using a correlation attack.
- The pre-attack portion:
  1. The attacker must find out on which virtual core the victim process is executed and with which other virtual core it shares the physical core.
  2. The attacker has to determine at which address the security-critical data he wants to observe is located.

2.3.6 Finding Eviction Sets

As described in the previous paragraphs, during the prime phase of the attacks Prime+Probe and Prime+Abort eviction sets are required. An eviction set consists of a set

of virtual addresses whose associated cache lines are located in the same cache set. The eviction set has exactly one address for each possible position (way) in the cache set. With a n-associative cache, each eviction set, therefore, consists of n addresses. With the L3 cache, an eviction set covers precisely one cache slice instead of an entire cache set.

Determining an eviction set for the L1 cache is trivial. As Table 2.1 shows, the L1 cache consists of 64 cache sets per core while the line size is 64 bytes. Consequently, the set index and the line offset of the L1 cache each require 6 bits. As shown in Figure 2.4, the address translation of 4 KiB pages keeps exactly an offset of 12 bits constant. By comparing the page offset, one can find virtual addresses whose associated data all reside in the same cache set.

The L3 cache has 1024 sets, so its set index requires 10 bits. Since the line size does not change compared to the L1 cache, the line offset also requires 6 bits. The Bits 6-15 of the physical address, therefore, determines the cache set. Consequently, when using 4 KiB pages, a simple comparison of the page offsets of the virtual addresses is no longer sufficient to determine the eviction set (see Figure 2.4). Besides this hurdle, cache slicing of the L3 cache also makes it difficult to find addresses for the eviction set. As mentioned before, an eviction set for the L3 cache does not cover an entire cache set but only a cache slice. We introduce the algorithm of Mastik [Yuv] for creating eviction sets, which overcomes the hurdles of physically indexed caches and cache slicing. The idea of this algorithm was introduced initially by [LYG+15].

To avoid the problem of physical indexing, we will proceed like [LYG+15, IES15b, DKPT17] in the following. As depicted in Figure 2.4 the cache index of the L3 cache can easily be read from the page offset if 2 MiB huge pages are used instead of 4 KiB pages. Because huge pages play a major role in the performance[3] of many applications, they are available on many systems. In the thesis' project, the libhugetlbfs library [lib] is used to access Huge Pages.

The addresses that match in both the set index and the line offset will be called *set-aligned*. Algorithm 1 determines a set of eviction sets, called *evictionGroup*. To determine whether an eviction set evicts the current line there are two methods. One is the Prime+Probe method shown in algorithm 2, which is also used in Mastik, and the other one is the Prime+Abort method introduced in [DKPT17] and shown in algorithm 3.

Both the method presented in algorithm 2 and the method presented in algorithm 3 were implemented in the thesis' project and were successfully used on the present CPU to determine eviction sets. In the following, we will only work with the Prime+Abort method, which proved to be more precise, practical and faster. The idea of the algorithm is based

[3] Huge Pages use the Translation Lookaside Buffer more efficiently because they need comparatively fewer entries for the same memory areas.

Algorithm 1: Determines an *evictionGroup* for a given cache set.

```
1  Input: A set of set-aligned cache lines called lines
2  Output: An evictionGroup for lines
3  evictionGroup ← {}
4  evictionSet ← {}
5  while lines not empty do
6      repeat                                                  // 1. Step
7          line ← random member of lines
8          remove line from lines
9          if evictionSet evicts line then          // algorithm 2 or algorithm 3
10             c ← line
11             break
12         add line to evictionSet
13     until forever
14     foreach member in evictionSet do                        // 2. Step
15         remove member from evictionSet
16         if evictionSet evicts c then             // algorithm 2 or algorithm 3
17             add member back to lines
18         else
19             add member back to evictionSet
20     foreach line in lines do                              // Cleaning loop
21         if evictionSet evicts line then          // algorithm 2 or algorithm 3
22             remove line from lines
23     add evictionSet to evictionGroup
24     evictionSet ← {}
25 return evictionGroup
```

on abort reason 9 from Table 2.2. If after accessing the whole eviction set followed by accessing our target line a TSX abort follows reliably, we can conclude that the transaction cannot hold the lines together in its read set. This means that more cache lines were accessed than fit into a cache slice (associativity +1) [DKPT17].

Algorithm 1 gets a set of set-aligned cache lines called lines as input and determines as already described an evictionGroup. The algorithm consists of an outer loop that determines an eviction set at each run. The determination of an eviction set consists of two steps respectively two loops followed by a loop to clean up the lines set. The first step of the algorithm consists of adding lines to an eviction set until the eviction set evicts the current line. In the second step, the algorithm removes any line from the eviction set that is not needed to form a valid eviction set. The cleaning loop at the end removes all lines

Algorithm 2: Prime+Probe method to test whether an eviction set evicts a given cache line. The number of loop iterations used below is system dependent and the given values proved to be a good tradeoff between performance and completeness.

```
1  Input: An eviction set evictionSet and a cache line line
2  Output: evictionSet evicts line return true else return true
3  times ← {}
4  repeat
5      access line
6      repeat
7          foreach member in evictionSet do
8              access member
9      until 20 times
10     timed access to line
11     times ← times + {elapsed time}
12 until 16 times
13 if median of times > predetermined threshold then
14     return true
15 else
16     return false
```

from the lines set that are evicted by the current eviction set. Therefore, they can be ignored in the future. In the last step, the current eviction set is added to the evictionGroup. The algorithm terminates as soon as the set lines is empty.

Extracting a given target line from an evictionGroup results in two different cases. The first case occurs when only the virtual address of the target line is known. Then one has to iterate over all eviction sets of the evictionGroup that match from bit 0 to bit 11 (if the victim uses 4 KiB pages; for larger pages this interval can be extended to speed up the search) with the virtual address and test with algorithm 3 if the target line is evicted. In the second case, the set index of the target line is known. Now the search can be shortened by testing only the eviction sets using algorithm 3 which are set-aligned to the target line. If n is the associativity, at worst n eviction sets have to be tested in this case.

If the physical address or the set index of the target line is already known at the time of determining the eviction set, the performance of the algorithm can be increased again by determining only the required eviction set. See also algorithm 4.

Algorithm 3: Prime+Abort method to test whether an eviction set evicts a given cache line. The number of loop iterations used below is system dependent and the given values proved to be a good tradeoff between performance and completeness.

```
1  Input: An eviction set evictionSet and a cache line line
2  Output: evictionSet evicts line return true else return true
3  aborts ← {}
4  commits ← {}
5  while aborts < 32 and commits < 32 do
6      begin transaction
7      foreach member in evictionSet do
8          access member
9      access line
10     end transaction
11     if transaction committed then
12         increment commits
13     else if transaction aborted with appropriate status code then
14         increment aborts
15 if aborts ≥ 32 then                          // Number based on experience
16     return true
17 else
18     return false
```

Algorithm 4: Determines an eviction set to a target line (set index known)

```
1  Input: The target line called targetLine; a set of cache lines called lines, that is
     set-aligned to the target line
2  Output: An eviction set evictionSet for the target line targetLine
3  evictionSet ← {}
4  while evictionSet do not evict targetLine do
5      line ← random member of lines
6      remove line from lines
7      add line to evictionSet
8  foreach member of evictionSet do
9      remove member from evictionSet
10     if evictionSet do not evict targetLine then
11         add member back to evictionSet
12 return evictionSet
```

3 Combining MemJam and Prime+Abort

We now try to combine the attacks MemJam and Prime+Abort to allow attacks on asymmetric encryptions such as RSA. Our target platform is the Intel Core i7-8650U, a current Intel CPU of the 8th generation. Since Intel is quite discreet regarding the microarchitecture of its processors and only a few details are known about the internals of this new processor, the function of Prime+Abort and MemJam was validated in the first step of the project. Both attacks were successfully implemented for the existing CPU. In order to reduce the measurement noise in our results (especially the MemJam measurements), we used Intel's `pstate` driver [Lin] to fix the CPU frequency of all cores to a constant 500MHz.
In this chapter the procedure of the attack combination as well as its development and occurring problems are explained on the basis of different dummy victims. The idea of developing an attack using different dummy victims is to adapt the victim step by step until it is replaced by a real victim (e.g. a real encryption used in practice), while the attack is optimized for the victim at each iteration. The thesis' project does not contain a functioning attack, but the results support the development of an attack based on this approach in the future.

3.1 The general procedure and a first dummy victim

This section covers the general procedure of the attack combination. As before, the description of the attack is based on the general structure of cache attacks, as described in subsection 2.3.1. We introduce a first dummy victim in order to be able to describe the procedure of the attack more clearly and to verify that the two attacks do not in any way interfere unfavorably with each other (e.g. by a processor behavior unknown to us). This victim consists of an array, called sbox, containing 256 1-byte entries and a buffer array containing 640 64-byte entries. The victim accesses the `sbox[200]` and the `buffer[32]` in 640 iterations. Every 64 iterations it accesses `sbox[0]` and `buffer[0]` (See Listing 3.1).
The MemJam part of our attack observes accesses to the first sbox entry, while the Prime+Abort part observes the first entry of the buffer array. We now describe the attack based on this dummy victim. This victim is highly simplified to understand the procedure of the attack. Afterwards, we will adjust the dummy victim slightly so that the

```
for(int x=0;x<640;x++){
        access = buffer[(x%64==0)?0:32];
        access = sbox[(x%64==0)?0:200];
}
```

Listing 3.1: Memory accesses of the first dummy victim

advantage of the attack becomes clearer. Figure 3.1 visualises the chronological sequence of the attack. The attack phases of MemJam+Abort are as described below.

- The pre-attack portion: Before the attacker can start the active portion of the attack, he must satisfy some preliminaries.
  1. MemJam: The attacker has to determine at which address the security-critical data he wants to observe is located.
  2. MemJam: The attacker must find out on which virtual core the victim process is executed and which other virtual core shares the same physical core.
  3. Prime+Abort: The attacker must determine an eviction set.
- The active portion:
  1. Initialization phase:
     - MemJam: The attacker starts a process (same physical core as the victim process) in which he writes to a particular address continuously. This address matches the 12 least significant bits of the target line (`sbox[0]`).
     - Prime+Abort (prime phase): The attacker opens a TSX transaction and accesses enough cache lines from the cache set to be monitored such that this cache set is only filled with his data.
  2. Waiting phase:
     - MemJam: The attacker waits for access to the observed target line. In Figure 3.1, the access times to `sbox[0]` were measured in the victim thread to check if the attacks influence each other and to facilitate the debugging of the attacks. (The next dummy victim will not have to perform any measurements.)
     - Prime+Abort: The attacker waits until the transaction from the prime phase is aborted (with the correct abort criterion). With the aborted transaction, the attacker can now conclude that an address (e.g. `buffer[0]`) in the monitored cache set has been accessed.

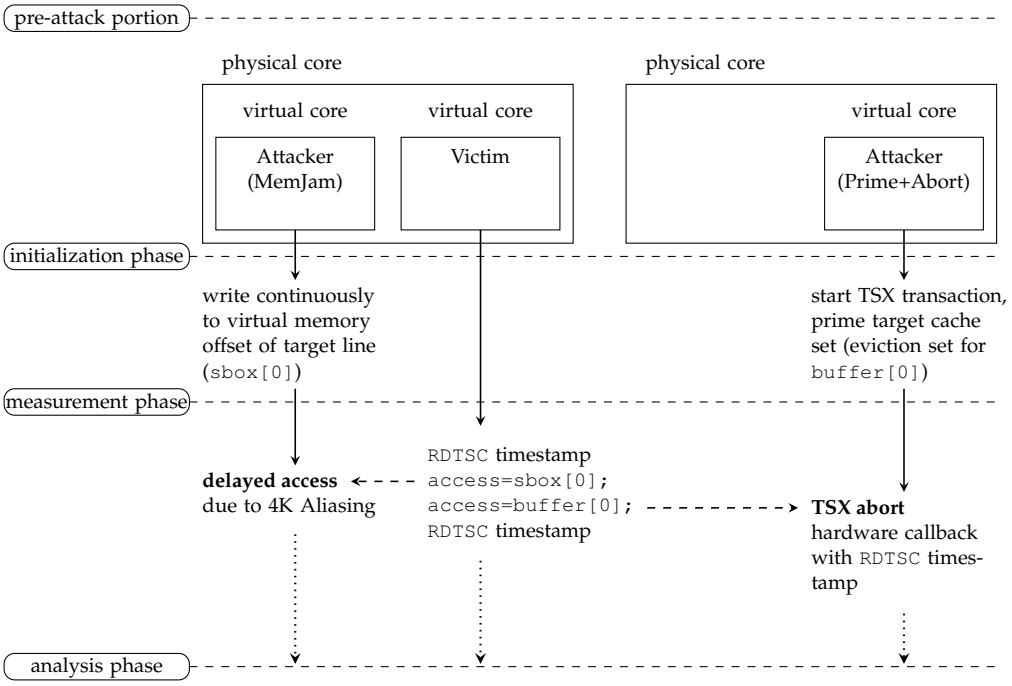

Figure 3.1: The graphic shows the chronological flow of MemJam and Prime+Abort starting with the pre-attack portion followed by the active portion with its three phases: initialisation, measurement and analysis. The memory accesses shown here refer to the first dummy victim.

3. Analysis phase: Since the attack on this dummy victim is less aimed at collecting data than at showing the phases of the attack, there is no real analysis phase. Figure 3.2 shows the measurement results of the attack. The curved course describes the access times of the MemJam attack and the vertical lines the time points of the TSX aborts.

If we analyze this graphical representation of the measurement results, we first notice a large amplitude after the 4th TSX abortion. This noticeable effect was also apparent in further executions of the attack. We dismiss this peak as measurement noise, since it is related to the time measurements in the victim thread and thus no longer occurs in the next dummy victim. It is also noticeable that we get several spikes in the MemJam measurement with each `sbox[0]` access. Especially between the last two TSX aborts the measurement is very noisy. In the case of later attacks we will perform several iterations of the attack and average over these to minimise the noise in the measurement data for further analysis. The Prime+Abort attack provides twelve TSX aborts with the correct abort criterion. Ideally, we would have expected eleven aborts, whereby it is noticeable

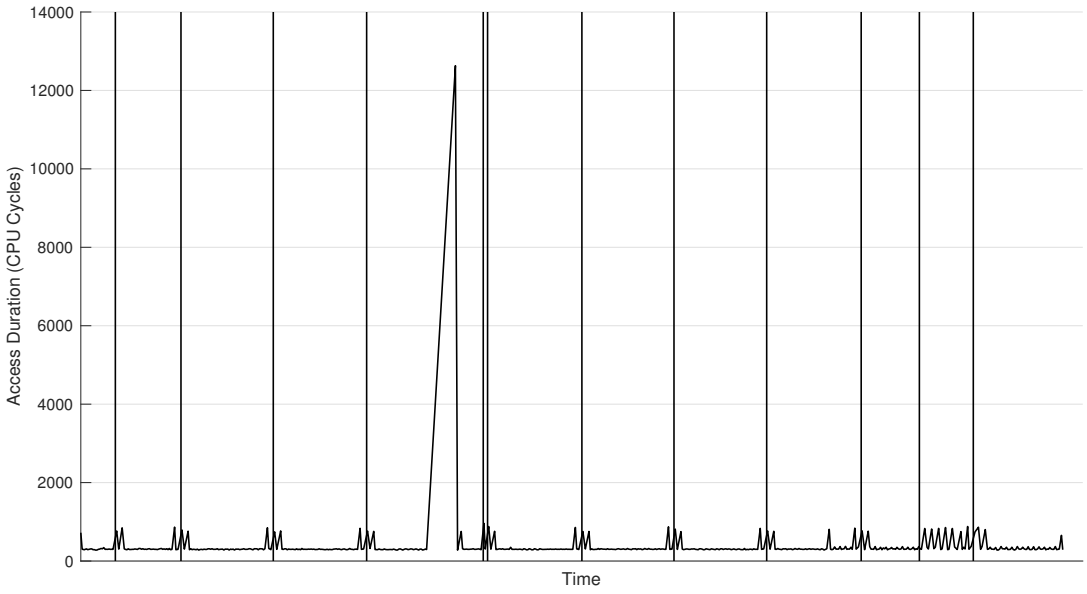

Figure 3.2: The curve shows the MemJam access times and the vertical lines the time points of the TSX aborts.

that the 5th and 6th aborts are very close to each other and can be assigned to the same `buffer[0]` access. In this way, we get a useful measurement for our Prime+Abort attack in the present case. However, further measurements show a further inaccuracy, which did not occur in this measurement. It can also happen that a `buffer[0]` access is not tracked by Prime+Abort. This may be because shortly before the `buffer[0]` access occurs, the TSX transaction has to be terminated and restarted due to a different termination reason and therefore misses the interesting access. We will also compensate for this behavior with several measurement iterations at the next dummy victim.

When this first victim was attacked, accesses to a fixed address in the buffer array were observed, allowing the two attacks to run independently of each other, apart from their start and end times, and requiring no coordination between them. This requirement dramatically simplifies the practical implementation of the attack. When attacking the next victim, we will remove this requirement and analyse the dependencies between the two attacks.

```
for(int x=0;x<640;x++){
        access = buffer[x];
        access = sbox[(x%64==0)?0:200];
        //empty loop for performance degradation
        for(int i=0;i<30;i++){}
}
```

Listing 3.2: Memory accesses of the second dummy victim

3.2 The second dummy victim

We now introduce a second dummy victim, which is slightly modified in comparison to the first dummy victim. Instead of accessing `buffer[0]` every 64 iterations, the victim now accesses `buffer[i]`, where `i` is the loop counter for the 640 iterations of all accesses (See Listing 3.2). The attacker no longer observes continuous accesses to `buffer[0]` but accesses to `buffer[x]`, where `x` is increased by the value of the resolution of the Prime+Abort attack at each TSX abort, with appropriate abort criteria (See Listing 3.3). The sbox accesses remain the same as in the first dummy victim, but the victim no longer performs time measurements. Besides, an empty loop with 30 iterations (number based on experience) is executed after each access to the sbox to reduce the noise of the Prime+Abort attack. Without this loop, the attacker could not start TSX transactions fast enough to watch the buffer accesses. This performance degradation in the dummy victim is not unrealistic, because real targets would also be slower than the dummy victim without this loop.

The MemJam attacker continues to slow down access to the first sbox entry. Compared to the first attack constellation we increase the resolution of the Prime+Abort attack and it now tracks every 16th access to the buffer. We are also developing a small analysis phase for our second dummy victim. The following measurement results come from 101.000 iterations of the attack, whereby we ignore the first 1.000 iterations to reduce measurement noise on the startup. In the first step of the analysis phase, we normalise the Prime+Abort measurements of all iterations. Therefore, each instance of Prime+Abort starts at time-point 0. Afterwards, we perform a k-means clustering[4] with 40 (buffer accesses/resolution Prime+Abort) cluster centres on the Prime+Abort data. We now look at the distances between these cluster centres to draw conclusions about the intervals at which accesses to `sbox[0]` occurred. If such access occurs between two neighbouring cluster centres, it is slowed down by MemJam and should cause them to lie further apart. A plot of the distances between the cluster centres is depicted in Figure 3.3.

The number of spikes, as well as their distance to each other, corresponds to our expecta-

[4] MathWorks' implementation in MatLab [Mat] with `'MaxIter' 1000` and `'Replicates' 5`

```
while(runPrimeAbort){
        status = _xbegin();
        if(status == _XBEGIN_STARTED){ //transactional path

                //access eviction set for the observed line

                while(1){}
                xend();
        }else{ //non-transactional path

                //save RDTSC timestamp

                //if status = abort reason for conflicted memory access
                if(status==6){
                        observedBufferLine+=resolutionPrimeAbort;
                }

                //save status
                //...
        }
}
```

Listing 3.3: The TSX Transaction procedure in the attacker's Prime+Abort thread

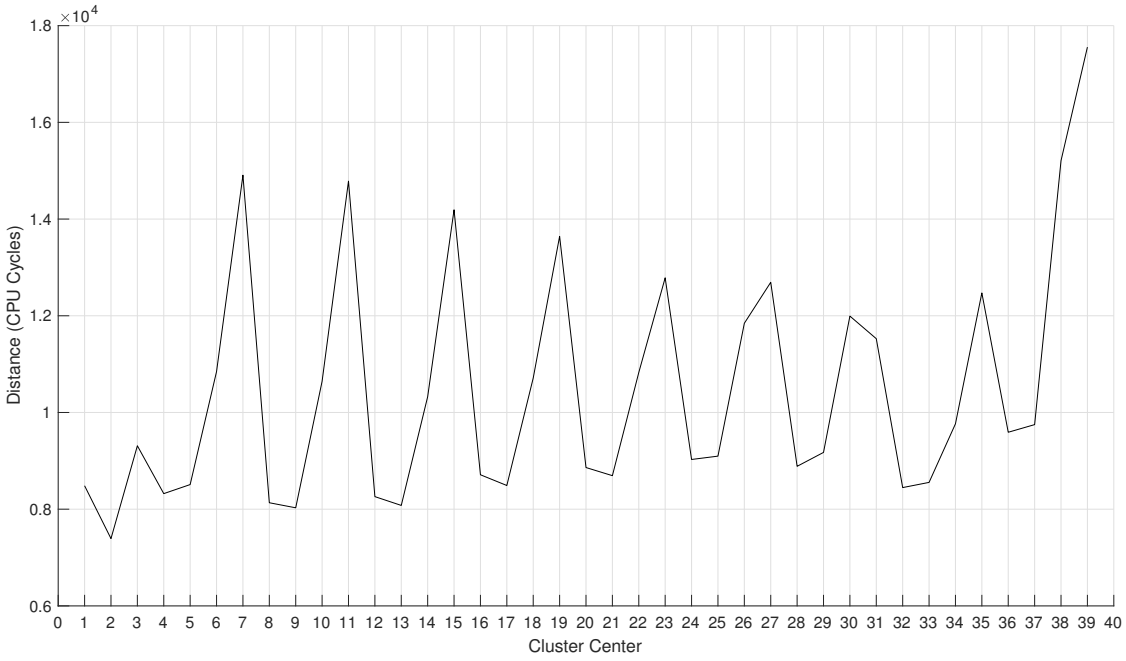

Figure 3.3: Measurements results of the second dummy victim - The curve shows the distances between the cluster centres of the measured TSX aborts (with appropriate termination criteria).

```
//NUMBER_ITERATIONS: Iterations of the Attack
while(iteration<NUMBER_ITERATIONS){
        //notify attacker
        runPrimeAbort=1;
        //empty loop for performance degradation
        for(int i=0;i<10000;i++){}
        for(int i=0;i<64;i++){
                //buffer and sbox Accesses
                access=sbox[hash(buffer+i*64)];
                for(int i=0;i<10000;i++){}
        }
        //notify attacker
        runPrimeAbort=0;
        for(int i=0;i<10000;i++){}
        iteration++;
}
//kill attacker
victimFinished=1;
```

Listing 3.4: New synchronization mechanism of the second dummy victim

```
int hash(uint8_t thisBuffer[]){
        int sum=0;
        //iterate over one buffer index with 64 bytes
        for(int i=0;i<64;i++){
                sum+=*thisBuffer;
                thisBuffer++;
        }
        return sum%256; //sbox has 256 entries
}
```

Listing 3.5: Hash-function for the memory accesses - The for loop enables a simple and effective performance degradation attack.

tions regarding the sbox accesses. However, we expect MemJam to only generate a delay of about twenty cycles, which means that the spikes in the graphic are too large and not related to the MemJam attack. A simple additional test with an increased number of sbox accesses confirms the assumption because the measurements again show ten peaks.

In order to ensure that these peaks are not caused by TLB misses due to an unfavourable alignment of the buffer or insufficient synchronisation of the attack, adjustments of the dummy victim are necessary. These include a shortening of the buffer from 640*64 bytes to 64*64 bytes (= 4096 bytes = size of standard 4 KiB pages), as well as several empty loops to slow down the attack and variables for synchronisation (see Listing 3.4 and 3.6). Also, the buffer and sbox accesses are now generated using a simple hash function (see listing 3.5) in order to allow performance degradation attacks later.

Besides, the already mentioned changes we increase the resolution of the Prime+Abort

```
while(!victimFinished){
        observedBufferLine=0;
        //number of aborts with appropriate abort criterion
        int rightAborts=0;
        //wait for victim to start round
        while(!runPrimeAbort && !victimFinished){}
        while(rightAborts<NUMBER_OBSERVED_CACHELINES &&
        numberOfAborts[iteration]<NUMBER_MEASURED_ABORTS &&
        observedBufferLine < NUMBER_OBSERVED_CACHELINES &&
        runPrimeAbort && !victimFinished){
                status = _xbegin();
                if(status == _XBEGIN_STARTED){

                        //access eviction set for the observed line

                        // Wait for abort
                        while(runPrimeAbort && !victimFinished){}
                        xend();
                }else{

                        //save RDTSC timestamp

                }
                if(status==6){
                        observedBufferLine+=resolutionPrimeAbort;
                        rightAborts++;
                }

                //save status
                //...
        }
        //wait for victim to notify end of round
        while(runPrimeAbort && !victimFinished){}
}
```

Listing 3.6: New Synchronisation mechanisms in the attacker's Prime+Abort thread

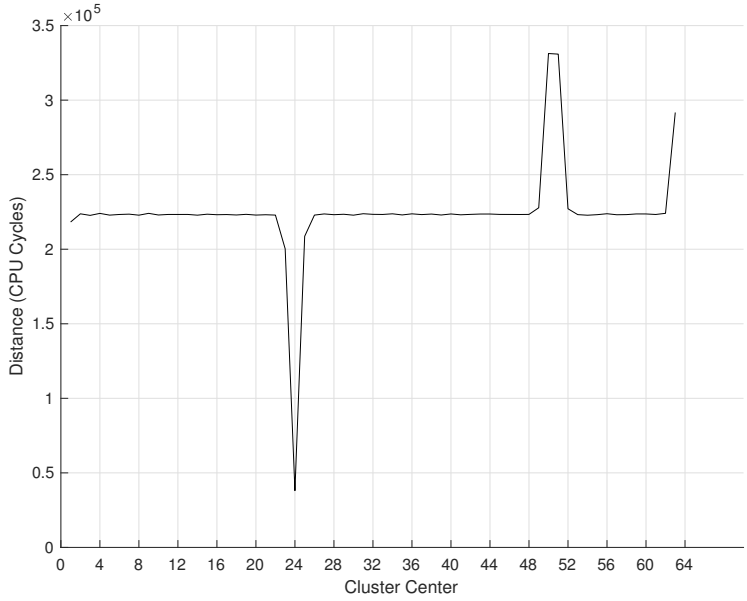

Figure 3.4: Measurement results of the updated second dummy victim. The curve shows the distances between the cluster centres of the measured TSX aborts (with appropriate termination criteria).

attack to the maximum so that every buffer access is now tracked. Before the attack goes to the analysis phase, we filter our recordings so that we only look at iterations in which Prime+Abort tracks exactly the 64 expected aborts with the appropriate abort criterion (Between the aborts with the corresponding abort criterion, an arbitrary number of aborts with a different abort reason can occur).

Figure 3.4 shows the measured data of the new setup. 101.000 iterations of the attack have been performed, and again the first 1000 iterations have been ignored. Of the 100.000 iterations considered, 61.120 matched the filter rules and had precisely 64 aborts with the desired abort reason. The first thing to notice when looking at the measurement results in the graph is the negative peak. This indicates that TLB misses do not cause the measured deflections. A miss would slow down the victim and not accelerate it. We also have the buffer array aligned with the 4 KiB pages so that we would expect a peak due to a TLB miss at the beginning or end of our measurement. This is also not the case and supports the thesis that these are not TLB misses.

Figure 3.5 shows a plot of the cumulative distribution (CDF) over the timepoints of the TSX aborts. For each of the 64 buffer entries, the graph shows a curve describing the sum of the aborts over time for each buffer index. Contrary to what is expected, the curves do not have a sharp increase, but a stepped increase, with these steps "stacked on top" of each

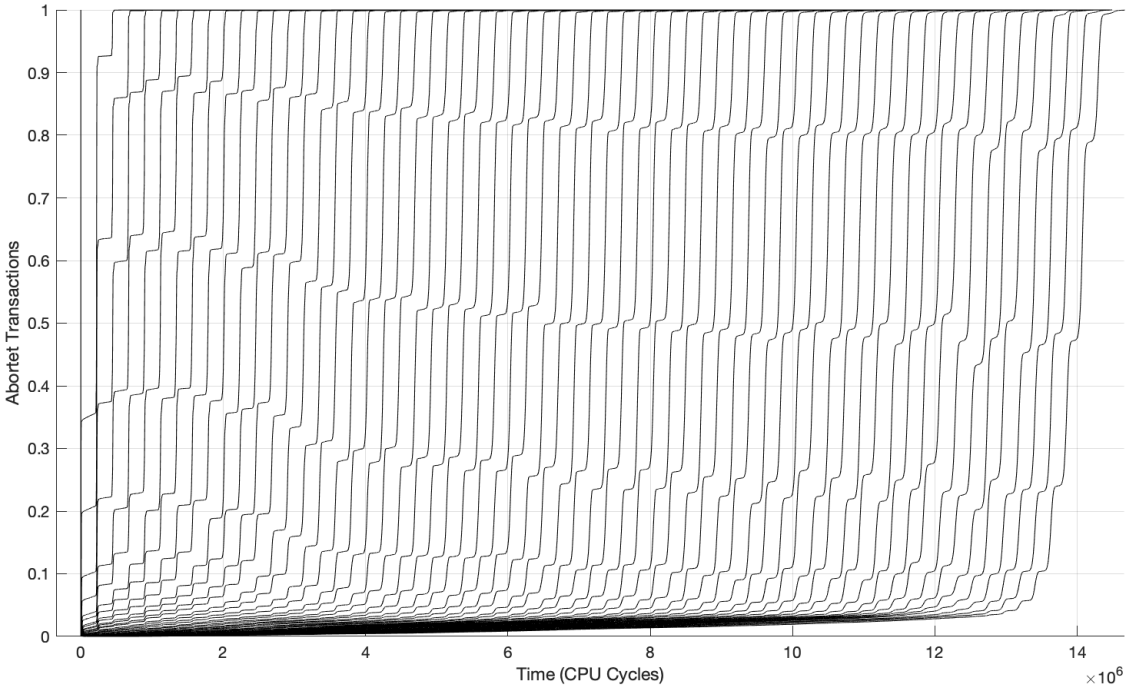

Figure 3.5: The plot shows the cumulative distribution over the timepoints of the TSX aborts. Contrary to what was expected, the curves have a stair-like ascent. We would have expected an abrupt steep rise, as in the curve to the far left, and that they would all be shifted at about the same distance from each other.

other. It would have been expected that all curves would have a course like the first curve and be shifted at the same distance from each other. It can be assumed that the shape and displacement of the individual curves lead to artefacts in k-means clustering, which are manifested as the peaks in Figure 3.3 and Figure 3.4. The reason for this unexpected course of the CDF remains as an open problem (see section 4.2).

To support later analyses, two further interesting measurements will be presented here. In the second development stage of the second dummy victim, a filter is used that only considers measurement iterations with exactly 64 aborts with the appropriate abort criterion. In addition to the expected aborts, a high number of aborts with a different abort reason also occur in these iterations. The distribution of the reasons for the aborts can be seen in Figure 3.6.

The most significant portion, around 53% of the aborts, is due to abort Reason 9, which we cause using the Prime+Abort attack. With about 44% of the aborts follows abort reason 1, i.e. the overflow of an internal buffer to track the transaction state. Interrupts (reason 4)

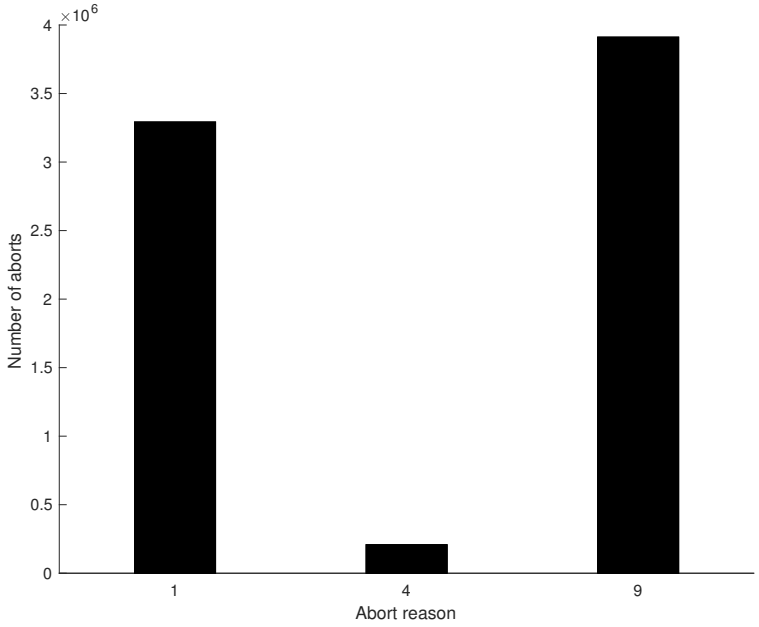

Figure 3.6: Distribution of the aborts to the their various reasons (1: Internal buffer for tracking the transaction state overflows, 4: Interrupts, 9: Caused by Prime+Abort).

cause about 3% of the aborts. Besides, nine aborts occurred due to combinations of reason 9 and 1, which we ignore here. To what extent the aborts due to reason 1 and 4 correlate with the unexpected form of the CDF results in Figure 3.5 remains as an open problem. To explore this correlation, the unexpected aborts should be reduced (see section 4.2).

The sbox accesses in the second development stage of the second dummy victim are done using a hash function to allow a performance degradation attack (to be able to study the influence of such an attack on the measured values, if it becomes necessary by attacking a real Victim, which is not slowed down by empty loops). First attacks on the dummy victim 2, following the example of Allan et al. [ABF+16], enabled a performance degradation by a factor of 22 (using `clflush`; a prime approach would be less effective, since one needs to reload the entire eviction set each time). The hash function implements a loop which is called repeatedly. Allan et al. speak of a hot code section, which the processor regularly executes and therefore keeps in the cache. If this section of code is now flushed out of the cache, the processor has to load it repeatedly from memory, which massively slows down the execution of the code. A significant advantage of this approach over many other performance degradation attacks is that not all program parts that depend on the microarchitecture of the CPU are slowed down, but only the attacked code segment. The performance degradation by a factor of 22 described above was realised by three

attacker threads, which continuously flush the code segment, in which the loop of the hash function is located, from the cache. Together with the victim thread, the Prime+Abort thread, the MemJam thread and the main thread (starts Prime+Abort and victim), seven threads run in parallel on eight logical cores, leaving one logical core for the operating system.

4 Conclusion

4.1 Summary

In the project of the present bachelor thesis, we dealt with cache-based side channel attacks, a particular subgroup of the large family of side channel attacks. In detail, we have studied the possibility of combining the attacks MemJam and Prime+Abort to develop the new attack MemJam+Abort, which (in theory) allows attacks on asymmetric cryptosystems with intra-cacheline granularity. We developed the new attack step by step on a current Intel CPU of the 8th generation. In the first step, we verified the executability of the attacks MemJam and Prime+Abort on this current CPU. In the same step, we determined all essential parameters of the cache architecture of our victim CPU and developed our implementation of the algorithm for determining eviction sets.

In the next step, we implemented the combination of attacks and developed dummy victims to record measurement series with MemJam+Abort. The first of the two dummy victims was used to debug the attack and verify the possibility of combining the attacks to prevent them from interfering with each other. Compared to the first dummy victim, we have slightly adjusted the second dummy victim to create a more realistic scenario in which the Prime+Abort component observes consecutive addresses as the attack proceeds. An analysis phase has also extended the active portion of the attack. The main idea of this first analysis phase is that clustering of the TSX aborts should result in a distribution of the cluster centres that allows conclusions about the sbox accesses. It has been shown that the measurements of the dummy victim 2 are too noisy for the analysis phase to generate a distribution that reveals the sbox accesses in this way.

For debugging purposes and obtaining more accurate measurements, the dummy victim was extended in the next step to include synchronisation mechanisms and a vulnerable (for performance degradation) hash function. The analysis of this further adapted dummy victim showed that the attack provides an unexpected temporal distribution of TSX aborts that may be responsible for clustering artefacts. The analysis of the distribution of the aborts over their causes also showed us that besides the aborts generated by Prime+Abort, many aborts with a different cause also noise the measurements.

4.2 Discussion and open problems

In the time available for the bachelor thesis it was not possible to answer the question of the realizability of the Abort+MemJam attack completely. However, it was possible to lay a foundation for the further development and analysis of the attack. The thesis closes with some open problems, whereby in particular the unexpected temporal characteristic of the TSX aborts (see Figure 3.5) complicates a further analysis of the attack data. Probably, this problem can be solved if the high number of aborts that are not caused by Prime+Abort can be reduced. This may be achieved by limiting interrupts to CPU cores, that are not affected by the Prime+Abort attack, or by preventing the overflow of the internal buffer to track the transaction state of TSX through making the code within the TSX transaction (i.e., the eviction set accesses) as minimal and efficient as possible.
If we can solve the remaining open problems and enable deductions about the sbox accesses through the temporal distribution of the TSX aborts, we get a time resolution on our MemJam data. With the help of this time resolution, it becomes possible to attack asymmetric encryptions like RSA as well as symmetrical encryptions (like they were already possible with the conventional MemJam attack).

References

[ABF+16] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 422–435. ACM, 2016. 31

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005. 1

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997. 1

[Deb] Debian Wiki contributors. Hugepages. `https://wiki.debian.org/Hugepages`. [Online; accessed 6-September-2018]. 9

[Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, Sep 1970. 9

[DHKL15] Dave Dice, Tim Harris, Alex Kogan, and Yossi Lev. The influence of malloc placement on tsx hardware transactional memory. *arXiv preprint arXiv:1504.04640*, 2015. 13

[DKPT17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel tsx. In *26th USENIX Security Symposium (USENIX Security 17),(Vancouver, BC)*, pages 51–67, 2017. 3, 10, 13, 14, 17, 18

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505. IEEE, 2011. 11

[GMWM16] 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. 12

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912, 2015. 12

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *International cryptology conference*, pages 444–461. Springer, 2014. 2

[gui18] Intel® 64 and ia-32 architectures software developer's manual. 2018. 3, 13, 15

[HEM93] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, page 289–300. IEEE Comput. Soc. Press, 1993. 13

[Hen17] John L. Hennessy. *Computer architecture: a quantitative approach*. Elsevier, 6th edition edition, 2017. 10

[Hu92] W.-M. Hu. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, page 52–61. IEEE Comput. Soc. Press, 1992. 2

[IES15a] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 591–604. IEEE, 2015. 12

[IES15b] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design (DSD)*, pages 629–636. IEEE, 2015. 7, 17

[IGI+16] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016. 7

[KAGPJ16] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, page 1–6. ACM Press, 2016. 7, 12

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018. 2

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999. 1, 2

[Koc96] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996. 1

[lib] libhugetlbfs contributors. libhugetlbfs. `https://github.com/libhugetlbfs/libhugetlbfs`. [Online; accessed 6-September-2018]. 17

[Lin] Linux Kernel Organization. Intel p-state driver. `https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt`. [Online; accessed 29-September-2018]. 21

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018. 2

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, page 605–622. IEEE, May 2015. 7, 10, 12, 17

[Mat] MathWorks. k-means clustering. `https://de.mathworks.com/help/stats/kmeans.html`. [Online; accessed 29-September-2018]. 25

[MLSN+15] 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 Workshop on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015. 7

[MWES18] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, Nov 2018. 15, 16

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006. 12

[Per05] Colin Percival. Cache missing for fun and profit, 2005. 12

[PH14] David A. Patterson and John L. Hennessy. *Computer organization and design: the hardware/software interface*. The Morgan Kaufmann series in computer architecture and design. Elsevier/Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, fifth edition edition, 2014. 5, 6, 7, 8, 10, 15

[Pub01] NIST FIPS Pub. 197: Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441):0311, 2001. 3

[Sco13] Michael L. Scott. *Shared-memory synchronization*. Synthesis lectures on computer architecture. Morgan & Claypool, 2013. 13

[Sha49] Claude E Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949. 3

[Til11] *Encyclopedia of cryptography and security*. Springer reference. Springer, 2nd ed edition, 2011. 1

[Wik] WikiChip contributors. Core i7-8650u - intel. `https://en.wikichip.org/wiki/intel/core_i7/i7-8650u`. [Online; accessed 2-September-2018]. 5, 6

[YF14] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014. 11

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017. 15

[YGL+15] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015. 7

[Yuv] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. `https://cs.adelaide.edu.au/~yval/Mastik/`. [Online; accessed 8-September-2018]. 17

[ZF05] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005. 1, 2