TeeJam: Sub-Cache-Line Leakages Strike Back

Florian Sieck[1], Zhiyuan Zhang[2*], Sebastian Berndt[1], Chitchanok Chuengsatiansup[2*], Thomas Eisenbarth[1] and Yuval Yarom[3*]

[1] University of Lübeck, Lübeck, Germany
{florian.sieck,s.berndt,thomas.eisenbarth}@uni-luebeck.de
[2] The University of Melbourne, Melbourne, Australia
zhiyuanz5@student.unimelb.edu.au,c.chuengsatiansup@unimelb.edu.au
[3] Ruhr-University Bochum, Bochum, Germany
yuval.yarom@rub.de

Abstract. The microarchitectural behavior of modern CPUs is mostly hidden from developers and users of computer software. Due to a plethora of attacks exploiting microarchitectural behavior, developers of security-critical software must, e.g., ensure their code is constant-time, which is cumbersome and usually results in slower programs. In practice, small leakages which are deemed not exploitable still remain in the codebase. For example, sub-cache-line leakages have previously been investigated in the CacheBleed and MemJam attacks, which are deemed impractical on modern platforms.

In this work, we revisit and carefully analyze the 4k-aliasing effect and discover that the measurable delay introduced by this microarchitectural effect is higher than found by previous work and described by Intel. By combining the rediscovered effect with a high temporal resolution possible when single-stepping an SGX enclave, we construct a very precise, yet widely applicable attack with sub-cache-line leakage resolution. To demonstrate the significance of our findings, we apply the new attack primitive to break a hardened AES T-Table implementation that features constant cache line access patterns. The attack is up to three orders of magnitude more efficient than previous sub-cache-line attacks on AES in SGX. Furthermore, we improve upon the recent work of Sieck et al. which showed partial exploitability of very faint leakages in a utility function loading base64-encoded RSA keys. With reliable sub-cache-line resolution, we build an end-to-end attack exploiting the faint leakage that can recover 4096-bit keys in minutes on a laptop. Finally, we extend the key recovery algorithm to also work for RSA keys following the standard that uses Carmichael's totient function, while previous attacks were restricted to RSA keys using Euler's totient function.

Keywords: Side-Channels · Microarchitectural Attacks · Trusted Execution Environments

1 Introduction

Performance and functionality of modern processors evolve in short intervals and improvements over previous generations are usually significant. To allow for this rapid development, microarchitectural features like caching and speculative execution have been pushed to their limits, often favoring speed over security. As a result, numerous microarchtectural attacks have been found that exploit non-constant-time behavior caused by caches, branch predictors etc. [OST06, AKS07, LYG+15, IAES15]. These attacks were followed by exploits

[*] Work was partially done while the author was affiliated with the University of Adelaide

of out-of-order and speculative execution [KHF$^+$19, LSG$^+$18, vSMÖ$^+$19, CGG$^+$19]. In the meantime, CPU designers have introduced new security features. One prominent example is Trusted Execution Environments (TEEs) such as AMD SEV [KPW16, Adv18] and Intel Software Guard Extensions (SGX) [MAB$^+$13, HLP$^+$13], which can enhance overall system security if properly used. The introduction of TEEs has led to an increased interest in microarchitectural attacks [XCP15, MIE17, BPS17, BMW$^+$18, BMS$^+$20, CCX$^+$19]. While protecting code and data from direct access, TEEs do not protect against microarchitectural leakages. Instead, the operating system (OS) and/or hypervisor are untrusted and thus allow for a much more powerful adversarial model. The malicious OS scenario has enabled the development of various new attack techniques, with improved attack resolution [MBH$^+$20, SBWE21].

The most common strategy to prevent these microarchitectural attacks is by ensuring that protected code is constant-time. That is, the code should contain neither secret-dependent control flows nor data accesses as well as avoiding instructions with data-dependent execution behavior. Implementation of constant-time code, however, is not trivial. As a result, a broad range of tools utilizing various analysis methods has been proposed. A comprehensive overview of constant-time verification tools is provided in a recent study, which showed that correctly using these tools is a challenge to code developers [JFB$^+$22]. One possible pitfall is the level of leakage granularity assumed by these tools. While some of these tools assume cache line resolution for attacks [DFK$^+$13, WWL$^+$17], others such as Microwalk [WMES18, WSPE22] and DATA [WZS$^+$18] keep the resolution configurable, leaving the choice to the developer.

Deciding the right level of granularity for a constant-time assumption can be quite tricky. It is widely accepted that cache line granularity can be observed by sophisticated microarchitectural attacks. Thus, security-critical code like cryptographic implementations must ensure constant-time behavior at the cache line level. The case of more fine granular sub-cache-line leakage is less obvious and contested. The quite powerful and generic CacheBleed attack [YGH17] has sub-cache-line resolution, but it has been fixed in the SkyLake processor generation and is no longer applicable. Other sub-cache-line resolution attacks such as MemJam [MWES19] are more difficult to perform and apply to fewer scenarios. In MemJam, the attacker introduces data-dependent timing behavior in a victim via 4k-aliasing, which is then exploited by measuring the execution time of the victim code. Hence, MemJam has much lower temporal resolution and is much noisier than other modern microarchitectural attacks. As timing variations are statistical and small, the application scenarios are limited. In fact, MemJam only targeted block cipher implementations. As a result, many cryptographic libraries have decided to ignore sub-cache-line leakages, as the cost for constant time at the byte level can be high in terms of performance loss.

In this work, we pose two questions on the limits of the spatial and temporal resolution for microarchitectural attacks on trusted execution environments. As of now, there are many attacks on TEEs with a very high temporal resolution down to instruction level [MIE17, CGYZ22, BPS18]. Most of these attacks use the SGX-Step framework [BPS17] to achieve a high temporal resolution. However, the spatial resolution of these attacks so far is limited by a cache line granularity and combinations of single-stepping and cache attacks on the last-level cache exist [SBWE21]. As explained above, a sub-cache-line spatial resolution by a software-based side-channel attack was achieved by MemJam and CacheBleed [YGH17, MWES19], but these attacks achieve only a very low temporal resolution. In fact, due to the attacker model, timing in MemJam is always observed over complete executions of the victim's program. A natural question is thus whether this tradeoff between temporal and spatial resolution is inherent, i.e., whether there is a bound on a combined temporal and spatial resolution for software-based side-channels when attacking software running in trusted execution environments and whether future attacks will be limited by this bound.

In this work, we show how to achieve a spatial resolution far beyond cache line granularity while keeping a temporal resolution on instruction level.

However, achieving this observation granularity does not necessarily imply that the obtained data can be used to construct attacks, as this might, e.g., be hindered by high noise levels rendering the measurements unreliable. Hence, we are also interested in the question whether the observed fine-granular spatial and temporal information can be used to exploit previously unexploitable implementations.

In summary, the research questions to be answered in this work are:

RQ1 Can an attack be designed which surpasses current bounds on the combination of spatial and temporal resolution for observations on software running in trusted execution environments? Meaning, can a temporal resolution as fine as single-stepping be combined with a sub-cache-line spatial granularity?

RQ2 Can the instruction-wise temporal and sub-cache-line spatial resolution be used to construct new attacks?

To answer RQ1 and RQ2, we revisit and comprehensively analyze the 4k-aliasing effect on modern Intel CPUs. Our careful analysis reveals that, if properly tuned, the delay effect already observed in MemJam can be significantly amplified, making the leakage exploitable with much fewer observations. In fact, the observed leakage is greater than reported in MemJam, and also exceeds the delays described in the Intel Optimization Reference Manual [Int15] due to the 4k-aliasing effect. We note that the 4k-aliasing effect can lead to significant delays is already explicitly mentioned in the Reference manual. Furthermore, by single-stepping through the victim application, the induced leakage no longer needs to be detected over the full execution, as done in MemJam. Instead, our new attack, which we call TeeJam, can exploit data-dependent leakage for a single instruction, thus significantly improving the level of temporal resolution achieved by the attack. TeeJam can observe every victim memory read at a granularity of 4 bytes—achieving sub-cache-line resolution—even if the victim is executed inside the context of Intel SGX.

To showcase the power of TeeJam, we apply this new side-channel to exploit the base64 decoding of RSA private keys in the OpenSSL library. This leakage was previously exploited in `Util::Lookup` [SBWE21], which managed to degrade the security level of the targeted RSA implementation, making recovery of short RSA keys practical. However, the leakage is not sufficient to completely recover keys of larger sizes, such as 2048- or 4096-bit keys. TeeJam can observe the exploited key decoding process with up to 16 times higher resolution. Due to the additional leakage, TeeJam succeeds in reconstructing 4096-bit keys with ease, highlighting the danger of the discovered sub-cache-line leakage and demonstrating for the first time that the 4k-aliasing effect can be used to exploit vulnerable public-key cryptography, not just block ciphers as done by MemJam.

In order to implement a full end-to-end attack for reconstructing RSA private keys generated with OpenSSL [CT23], we extend the Heninger-Shacham key reconstruction algorithm [BPS18] with the ability to reconstruct RSA keys not only with Euler totient as defined in the original publication [RSA78], but also with Carmichael totient as defined in the recent RSA standard [MCK$^+$16], which is used in many recent implementations such as OpenSSL [CT23][rsa_sp800_56b_gen.c] and sometimes required, e.g., by the FIPS standard for digital signatures [KR13].

Moreover, we further extend the generalization of the Heninger-Shacham algorithm from `Util::Lookup` [SBWE21]. The reconstruction algorithm from Heninger-Shacham requires side-channel information to be available as an array of bits. `Util::Lookup` allows the usage of observation partitions instead of an array of bits. We extend the algorithm to support missing observations and unaligned start and end partitions.

Compared to previous work, the generalization of the key reconstruction algorithm presented here necessitates guessing more information in the lower bits of the secret RSA parameters. While the increased uncertainty requires a higher fraction of known bits to be available in the side-channel traces, it enables the reconstruction of keys with Carmichael totient. Our experiments demonstrate that the high-resolution TeeJam attack yields sufficient information for the reconstruction of these keys.

Additionally, we present the recovery of a 128-bit AES key with a last-round known-ciphertext attack using TeeJam's high resolution. The attacked implementation is protected against cache attackers and cannot be exploited with cache-line resolution. We show that the combined high temporal and spatial resolution provided by TeeJam enables the reconstruction of the full key while we reduce the number of required encryption traces by three orders of magnitude compared to previous work MemJam [MWES19].

1.1 Contribution

In short, our contributions are:

- A thorough analysis and improvement of the MemJam 4k-aliasing effect on modern chip architectures and in various scenarios with and without Intel SGX.
- Introduction of the TeeJam attack, which provides sub-cache-line leakage of memory accesses in SGX, with high temporal resolution.
- An efficient end-to-end attack which recovers 4096-bit RSA keys from the base64 decoding process, whose leakage was believed to be practically unexploitable for larger key sizes. Notably, we show for the first time that 4k-aliasing effect can be used to target public-key cryptography as well.
- An extension of the Heninger-Shacham algorithm to support private keys with Carmichael totient and improvements that enable the reconstruction of keys from observation traces with missing information and unaligned partitions.
- Recovery of an AES secret key with an attack on a T-Table based AES encryption with effective protection against cache line level attackers. We reduce the number of required encryption traces by multiple orders of magnitude compared to previous work.

The source code is available on https://github.com/UzL-ITS/teejam and https://github.com/UzL-ITS/rsa-key-recovery.

1.2 Responsible Disclosure

The vulnerability in OpenSSL's base64 decoding was reported by the author's of `Util::Lookup`. We informed the authors of WolfSSL about our findings concerning their cache attack resistant AES T-Table implementation. They acknowledged our findings and added an AES bitsliced implementation.

2 Background

The TeeJam effect is based on microarchitectural details of Intel processors and functions of SGX. This section gives a short overview of the necessary background.

2.1 4K-Aliasing and MemJam

Most Intel processors support out-of-order execution for load and store operations. These operations are tracked by the load and store buffers respectively, while the Memory Order

Buffer (MOB) maintains the order of these operations. According to the Intel memory-ordering model, a load operation can be executed earlier than program order as long as it does not execute earlier than a store to the same physical address.

To avoid waiting for address resolution and allow loads to execute early, the processor performs partial matching using the virtual addresses. Recall that during address translation, bits [11:0] of the virtual address are preserved, i.e., they are not changed by the translation. Thus, to test for potential overlap, the processor first matches bits [11:5] of the virtual addresses and if matched, it compares the offsets for overlap. We use the term *4k-aliasing* to refer to a situation where two addresses are determined to be potentially overlapping based on this test. The Intel Optimization Reference Manual describes the precondition for 4k-aliasing as follows: two addresses are said to be affected by 4k-aliasing when the "... load and store have the same value for bits 5–11 of their addresses and the accessed byte offsets ... have partial or complete overlap." [Int15][Section 15.8].

When the processor detects 4k-aliasing, it delays the load operation until the physical addresses of both operations have been determined. Conversely, when no 4k-aliasing is detected, the load is not delayed and instead can be executed before the store. It is important to note that the conflicting offsets do not have to be actual conflicts, i.e., they do not have to be on the same physical page. The Intel Optimization Manual specifies a five-cycle penalty for 4k-aliasing.

This delay can be used by an active attacker to obtain sub-cache-line leakage as presented in MemJam [MWES19]. The 4k-aliasing leakage, however, is statistical, i.e., several measurements are necessary to reliably observe the delay specified by Intel. The effect is statistical in the sense that both load and stores have to arrive in a short time frame to cause the conflict. Especially, with an attacker trying to induce the delay from a hyper-thread this is not always given. Additionally, the 4k-aliasing effect can be easily disguised by noise and its amplitude depends on the number of conflicting stores in the MOB, which is also not deterministic in a scenario where the conflict is caused by attacking hyper-thread. Consequently, averaging over repeated executions is necessary to obtain reliable results.

2.2 Intel SGX

Intel Software Guard Extensions (SGX) is a Trusted Execution Environment (TEE) targeting at isolating and protecting sensitive workloads in untrusted environments. It consists of several processor instructions and hardware extensions within the processor [HLP+13, MAB+13]. Intel SGX protects the memory by encrypting all data which leaves the processor. The Memory Management Unit (MMU) transparently takes care of encrypting the data in RAM. Furthermore, software is measured and the result is compared against a pre-computed signed measurement before the enclave is started, also allowing for remote attestation [AGJS13, SJBZ18]. To ensure proper isolation, SGX provides specific instructions for entering (`EENTER`), resuming (`EERESUME`), leaving (`EEXIT`) and asynchronously exiting (`AEX`) an enclave. When leaving an enclave, SGX stores the current register file in a secure state save area and restores this state when resuming [CD16]. Furthermore, on exit, SGX flushes the L1 data cache and the TLB [CD16, Int23d].

SGX-Step: SGX-Step [BPS17] is a framework which uses precise APIC timer interrupts to single-step the execution of an SGX enclave. For that, the APIC timer is reconfigured after every "step" before the enclave is resumed, such that the interrupt triggers when the `ERESUME` instruction finishes and the first instruction within the enclave is executed. The interrupt routine will only be executed after the current instruction is committed. If the interrupt arrives slightly early, i.e., within the `ERESUME`, it results in a "zero-step" which can be easily detected by observing page accesses. A more difficult issue are "multi-steps" which execute more than one instruction in the enclave. To avoid this behavior, the timer interrupt can be tuned to rather cause “zero-steps”. Taking timer values before the enclave resumes and directly after the `AEX` in the interrupt handler allows to measure the time of a single-step [BPS18]. This time is dominated by the duration of `ERESUME` and `AEX`, still allows to infer some information about the protected program [BPS18].

Memory Management: The management of memory allocations and mapping of memory pages used within SGX remains in control of the host operating system (OS). While the content of the pages is fully under the control of SGX and can only be decrypted by the owning enclave, the host OS allocates the pages and assigns the virtual to physical page mapping. As such, the OS can control meta information of the page table entries like the page access bit or read / write / execute permissions. All pages, however, are allocated within the Enclave Page Cache (EPC), a processor reserved memory region only accessible from within SGX. To ensure that there are no manipulations of the address translation, SGX keeps track of the page mapping in the Enclave Page Cache Map.

2.3 Reconstructing RSA Keys from Partial Information

A common scenario in side-channel attacks is that the side-channel only reveals partial information about the sensitive values. The attacker thus needs to reconstruct the complete sensitive value from partial information. For the case of RSA private keys, Heninger and Shacham [HS09] presented an iterative algorithm that reconstructs RSA keys if some of the bits of the key are flipped. The algorithm makes use of the fact that RSA keys are typically stored in a highly redundant format to allow for faster decoding via the Chinese remainder theorem. Using the relations between the variables stored in the secret key, one can set up a set of *conditions* to relate the single bits of these variables to each other. By starting with few candidates, the algorithm expands these candidates using these conditions and prunes all candidates that are not compatible with their observations (e.g., because too many bits are flipped). This algorithm was extended by Henecka et al. and Paterson et al. [HMM10, PPS12] to allow for more bit flips. Recently, Sieck et al. [SBWE21] extended the algorithm to a different type of observation where the only information gained from the side-channel is whether a continuous block of b bits belongs to a certain set. Sieck et al. showed that this information is sufficient to drop the security level of the observed RSA keys by at least one level, i.e., the attack costs are reduced by a factor of about 2^{30}.

2.4 Attacker Model

In this work, we assume a system level attacker with full control over the OS and firmware, i.e. BIOS and UEFI, of the target machine. Thus, the attacker is capable of reading and manipulating the memory page mapping, isolating cores from the OS scheduler, fixing the processor frequency, disabling processor features like Intel Speed-Step and cache prefetching and starting and stopping enclaves at will as well as configuring interrupts and load custom kernel modules. Thereby, controlled channel attacks, such as SGX-Step [BPS17], which allows single-stepping enclaves, are enabled. Additionally, the attacker has read access to the binary of the targeted program. SGX only protects the program at runtime but does not encrypt the binary itself. The attacker does *not* execute any code within the victim’s enclave and cannot modify the enclave binary. In the context of TEEs, as e.g. SGX, these are reasonable assumptions as the goal is to allow users to securely execute software on untrusted machines in, e.g., cloud environments. Similar attacker models are followed by comparable works [BPS18, MBH$^+$20, SBWE21]. Additionally, hyper-threading is enabled.

3 4K-Aliasing Effect Analysis

To answer our first question about natural limits on the temporal and spatial resolution of attacks, we now take a closer look at known attacks with high spatial resolution. Two of the most prominent such attacks are MemJam [MWES19] and CacheBleed [YGH17]. Due to removed cache bank conflicts which only affected older Intel Ivy Bridge and Sandy Bridge CPUs, CacheBleed is not applicable anymore and we thus focus our study on MemJam. Our goal is to understand the underlying vulnerability exploited by MemJam and related attacks in depth and how to use it to obtain maximum leakage. Taking a closer look at these attacks shows that the 4k-aliasing effect caused by false read-after-write (RaW) dependencies was used in MemJam [MWES19] and also in *Microarchitectural Minefields* [SAMJ18]. Both attacks demonstrate that 4k-aliasing delays the load operation issued after a store operation; however, a detailed analysis on the preconditions and effect of 4k-aliasing on the performance penalty is missing.

We investigate the cause and effect of 4k-aliasing in detail. Therefore, we start by verifying Intel's documentation for a conflicting offset, specifically, "... load and store have the same value for bits 5–11 of their addresses and the accessed byte offsets should have partial or complete overlap" [Int15]. Then, we show how the number of store operations executed within a loop on the sibling thread affects the delay caused by 4k-aliasing.

3.1 Measurement Setup

We analyze the effect of 4k-aliasing across hyper-threads with the code listed in Listing 1 and Listing 2. We refer to two sibling logical cores as Thread 0 and Thread 1. Thread 0 measures the execution time of one load operation from a given address, as shown in Listing 1. Thread 1 performs 100 consecutive stores to a given address in an endless loop as shown in Listing 2. For every experiment in this section, we specify the number of repetitions for calculating the average delay. We explore how the operand size, the memory access offset and the number of stores in the loop impact the 4k-aliasing effect. Finally, we present the delay introduced by a 4k-conflict on different processors. Unless otherwise specified, the evaluations in this section are executed on an Intel Core i7-10710U processor with six cores, running Ubuntu 20.04. We configure the processor to run at maximum frequency of 4700 MHz by setting the CPU governor to performance.

Listing 1: Measuring a load operation on Thread 0.

```
mfence
rdtsc
mov r11b, [r10]
mfence
rdtsc
```

Listing 2: Performing 100 store operations to the same address on Thread 1.

```
%rep 100
mov word [%1], cx
%endrep
```

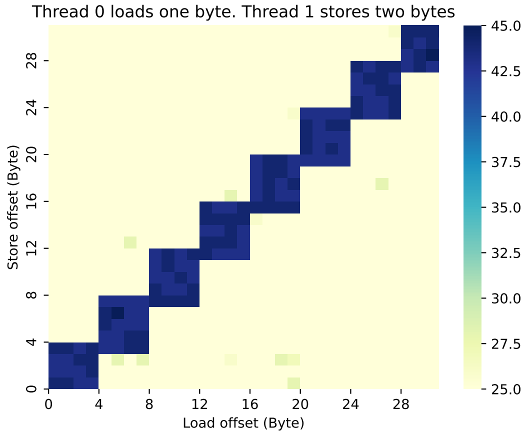


Figure 1: Precondition of 4k-aliasing in terms of address overlaps between conflicting stores and loads. The average time in cycles measured for the memory loads is encoded by color as shown on the right-hand side.

3.2 Bits Overlapping and 4k-Aliasing

In this section, we investigate the overlapping bits of load and store addresses that cause 4k-aliasing. First, we verify 4k-aliasing occurs when bits [11:5] of the load and store addresses are identical, and bits [4:2] are partially or completely overlapping. In the experiment, Thread 0 measures a one-byte load operation from a given address 1,000 times (Listing 1) while Thread 1 infinitely executes 100 two-byte store operations in a loop (Listing 2). We use a two-byte store to show the effect of overlap in bits [4:2]. The load and store addresses are initially pointing to the first byte of two different pages, thus having identical lower 12 bits. We then vary the offset of load and store addresses from byte 0 to byte 31. For each load offset, the average execution time is computed over the 1,000 measurements. The result is shown in Figure 1.

Technically, when storing two bytes to 0x0 on Thread 1 and loading one byte from 0x3 on Thread 0, both operations affect disjunct memory areas. However, as shown in Figure 1, a load operation at offset 0x3 is delayed by a store operation at offset 0x0, which indicates a 4k-aliasing performance penalty. Similarly, a one-byte load at offset 0x0 is delayed by a two-byte store at offset 0x3 although the loaded value is not affected by the stored value. We hypothesize that this effect is caused by the MOB always assuming four-byte aligned load and store operations. For example, a two-byte store to offset 0x3 is treated as if it modifies all data from 0x0 to 0x7 which can also be observed in Figure 1.

Due to the four-byte alignment policy, a load and an earlier store to addresses that share bits [11:5] are considered to be 4k-aliasing if they overlap in bits [4:2]. The results from *Microarchitectural Minefields* [SAMJ18] do not show any alignments above the two bytes stored in their experiment. However, we run our experiments on newer microarchitectures, confirming the leakage features a 4-byte granularity as shown in MemJam.

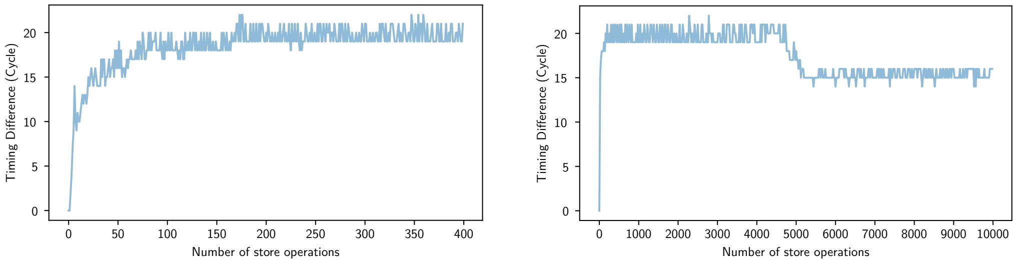
3.3 Number of Conflicting Store Operations

In this section, we demonstrate that 4k-aliasing does not introduce a constant performance penalty, but that the delay is related to the number of conflicting store operations on the sibling logical core. We show that with a proper number of conflicting store operations, 4k-aliasing performance penalty can be increased to 20 cycles under highest processor frequency.

We reuse the code shown in Listing 1 and Listing 2. In this experiment, Thread 1 uninterruptedly executes a certain number of stores to offset 0x0 in an endless loop, while Thread 0 loads an eight-byte value from addresses from the offsets 0x0 and 0x8 alternatingly. The load operation at offset 0x0 will be delayed because of 4k-aliasing. We measure the timing difference between the conflicting and non-conflicting load and change the number of store operations on Thread 1. To reduce measurement noise, we use the average of 500 timed load operations for each address.

First, we investigate the relationship between the performance penalty and the number of stores. Figure 2a shows the increase of the delay when the number of stores is gradually raised from 0 to 400. Around 100 to 200 stores in the loop, the delay is maximal with about 20 cycles difference between the conflicting and non-conflicting load. When further increasing the number of stores in the endless loop, the performance penalty decreases at approximately 4,600 stores as shown in Figure 2b.

As described in Section 2, to re-order a load operation, all older store operations are checked for dependency until a real conflict or 4k-aliasing is detected. In the case that a load operation is 4k-aliasing with a directly preceding store operation, the MOB stops checking the dependency of the load operation with other previous store operations. When the physical addresses of both load and store operations are available and 4k-aliasing turns out to be a false dependency, the MOB starts re-ordering the load operation again and it checks the dependency of the load operation with other store operations. Consequently,



(a)

(b)

Figure 2: Performance penalty of the 4k-aliasing effect in cycles depending on the number of store operations in the loop of Thread 1. Increasing the number of stores in the loop first raises then reduces the penalty. Figure 2b shows the measurements up to 10,000 stores in the loop of Thread 1, while Figure 2a shows a closer investigation of the range from 0 to 400 stores.

Table 1: 4k-aliasing on different processors. Delay is shown for the base frequency (BF) and the maximum frequency (MF). The latter is measured by setting the Intel pstate driver's governor to "performance".

Intel Processor	BF (MHz)	Delay BF (Cycles)	MF (MHz)	Delay MF (Cycles)
Core i5-6260U	1800	36	2900	23
Xeon E-2286M	2400	53	5000	26
Core i5-10210U	1600	70	4200	27
Core i7-10710U	1100	78	4700	18

fewer store operations in the MOB lead to a smaller delay caused by 4k-aliasing. Thus, the increase of the 4k-aliasing effect shown in Figure 2a can be explained by an increase in the number of address dependency checks. When the MOB is entirely filled with store operations, the effect of 4k-aliasing is maximized. We hypothesize that too many store operations in the loop on the sibling thread cause a slow down in filling the MOB with 4k-aliasing store operations as the front-end is busy with fetching and decoding new store operations. Thus, the number of dependency checks decreases.

3.4 MemJam on Different Processors

Finally, we present the delay caused by the 4k-conflict on different processors in Table 1 and show a comparison between the timings for loads with and without a 4k-conflict in Figure 3 on an Intel Core i5-10210U at base frequency.

For the results shown in Table 1 and Figure 3 we measure 500,000 times, each conflicting and non-conflicting load. In Table 1 the delay is shown for the processor's base frequency and the processors maximum single-core frequency. On modern Intel processors, the `rdtsc` instruction measures against a fixed frequency which corresponds to the processor's maximum frequency or the maximum core-clock to bus-clock ratio [Int23e][Vol. 3B, 18.17]. Therefore, a delay measured at lower frequencies appears higher, but in fact remains the same in terms of cycles. Only the temporal measurement resolution is increased compared to the processors running frequency. We show the delay at the maximum frequency as reference and additionally show the base frequency as baseline for a comparison against measurements in Section 4.

Figure 3 shows the distribution of timing measurements for loads with and without

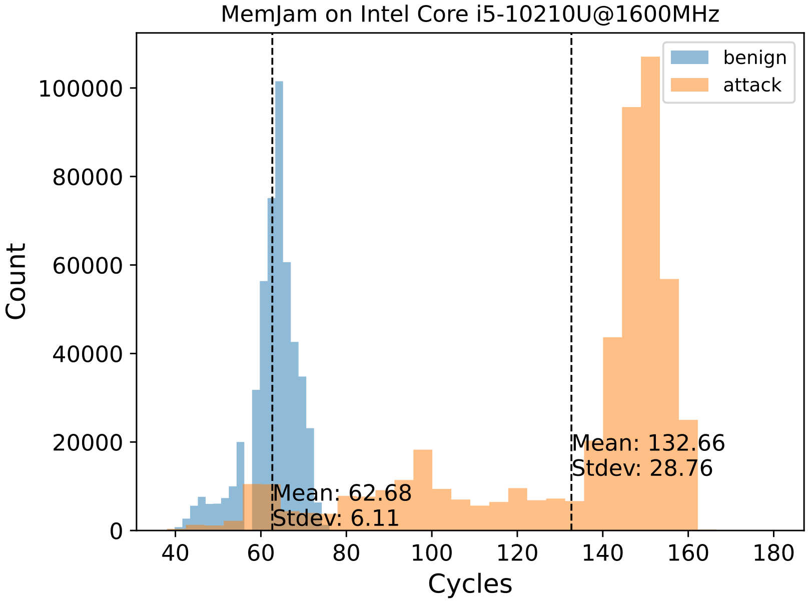


Figure 3: "Conventional" MemJam timing measurements with 12 conflicting bits. The measurement is repeated 500,000 times for conflicting (attacked) and non-conflicting (benign) loads.

4k-conflict at the processor's base frequency of 1600 MHz. At this frequency, the average delay is approximately 70 cycles.

4 TeeJam: Amplifying 4k-aliasing Leakage with Enclave Interruption

To construct an attack that combines the high spatial resolution of 4k-aliasing leakage with the temporal leakage of single-stepping, we now move the 4k-conflict experiments from Section 3 to the SGX single-stepping context. Many workloads today are executed in trusted execution environments allowing for a stronger attacker model. Our findings show that this combination yields a powerful attack, which we call TeeJam. TeeJam inherits the 4-byte intra cache line spatial resolution due to the 4k-aliasing leakage and combines it with the single-instruction temporal resolution of single-stepping, thereby answering our first question positively.

We describe the combination of a 4k-conflict based attack with asynchronous exits from an SGX enclave. As explained in Section 2.1, 4k-aliasing causes the CPU to detect a false read-after-write dependency if a load accesses an aliasing address affected by a preceding store on the same *physical* core, even across hyper-threads. Thus, in a straightforward attack an adversary could determine the addresses of conflicting stores by loading 4k-aliasing addresses. As secret-dependent store locations are extremely rare in cryptographic implementations, the effect direction needs to be reversed in order to build a useful attack. In MemJam the attacker thread performs conflicting stores to affect secret-dependent loads by the victim on the neighboring hyper-threads.

In fact, MemJam provides a sub-cache-line resolution by slowing down loads to specific *offsets* within a specific cache line. All that is left is measuring the caused delay, which is possible but requires millions of observations in a free-running target on the neighboring hyper-threads [MWES19]. MemJam thus measured the execution time of entire encryptions, causing the attack to require millions of observations to recover secret keys from block ciphers. We show that by exploiting the 4k-aliasing leakage in combination with single-stepping the target produces a new attack, achieving both maximal temporal resolution of single-stepping while achieving a 4-byte *intra cache line* resolution, which can thwart implementations that only consider cache-line granularity with ease. We further show that the 4k-aliasing effect is actually amplified by single-stepping into SGX. The delay caused by the false read-after-write dependency is doubled when observed across SGX boundaries, improving the efficiency of the attack. While TeeJam achieves highest *temporal and spatial* resolution for a microarchitectural attack, we will show that the 4k-aliasing effect remains a statistical one, requiring a low number of repetitions. Yet, we show that single-stepping and the SGX-based leakage amplification allow us to succeed with thousands of observations instead of millions of observations that were necessary in the MemJam attack.

In what follows, we describe the measurement setup for determining the amplification of the MemJam effect when applied to enclaves and analyze the results on different machines.

4.1 Measurement Setup

We run the basic experiments for evaluating the 4k-aliasing effect on an Intel Xeon E-2286M @2.4 GHz (Coffee Lake) and on an Intel Core i5-10210U @1.6 GHz (Comet Lake). All processors feature Intel SGX. Hyper-threading is enabled and the CPU frequency is fixed to the processor's base frequency. Additionally, to avoid noise in the measurements, we isolate the logical cores used for the measurements from the OS scheduler.

Listing 3: Code for Thread 1: load from #Offset and #Offset + 8.

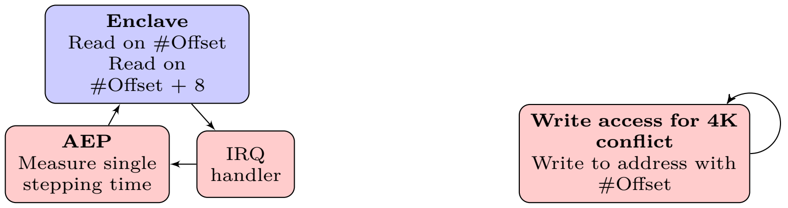
```
void access_for_4k_conflict(
    int pos1, int pos2) {

  for (int i = 0; i < 100; ++i) {
    mem_read(
      (uintptr_t) (mem + pos1),
      (uintptr_t) (mem + pos2));
  }
}

...

[global mem_read]
mem_read:
  %rep 100
  mov rax, qword [rdi]
  mov rax, qword [rsi]
  %endrep

  ret
```

Listing 4: Code for Thread 2: store to #Offset.

```
int main() {
  ...

  while (1) {
    write_conflict(target);
  }
}

...


[global write_conflict]
write_conflict:

  %rep 100
  mov qword [rdi], 100
  %endrep

  ret
```

The experimental setup is depicted in Figure 4. The findings of Moghimi et al. in MemJam [MWES19] show that the 4k-conflict is highest and best measurable with Read-after-Write (RaW) conflicts, meaning load after store. We transfer MemJam RaW conflicts to a TEE scenario. Therefore, load and store operations to pseudo-conflicting offsets are separated onto two threads, as shown in Figure 4a and Figure 4b. Both threads are running on the same physical core but on two different logical cores.

Thread 2, as depicted in Listing 3b, stores continuously and uninterruptedly to a virtual address with page offset `#Offset`. On Thread 1, shown in Listing 3a, alternating loads from the offsets `#Offset` and `#Offset + 8` are implemented and executed within an SGX enclave. Both addresses are located within the same cache line. We repeat the experiment 10,000 times for each offset. A histogram of the measured single-stepping times for each experiment as well as mean and standard deviation computed over 10,000 experiments are shown in Figure 5. As shown in Figure 4a, the enclave is single-stepped



(a) Measurement flowchart for Thread 1. (b) Measurement flowchart for Thread 2.

Figure 4: Setup for measuring the effect of 4k-conflict on SGX enclave exits. Thread 1 and Thread 2 are running on sibling logical cores. The enclave accesses alternatingly an address with page offset #Offset and #Offset+8 while it is single-stepped and the time between `AEX` and `ERESUME` is measured. Thread 2 continuously stores to an address with page offset #Offset.

with SGX-Step [BPS17] and for every step we measure the single-stepping time, i.e., the time between the call to `EERESUME` and the return of `AEX` [BPS18]. The single-steps that belong to the monitored load instructions are filtered by monitoring and manipulating the access bit of the page holding the accessed data.

4.2 Measurement Results

This section presents our measurement results on the Intel Core i5-10210U. Figure 5 shows the results of the TeeJam effect when single-stepping an enclave with memory loads while simultaneously running a hyper-thread which executes memory stores, as described in Section 4.1.

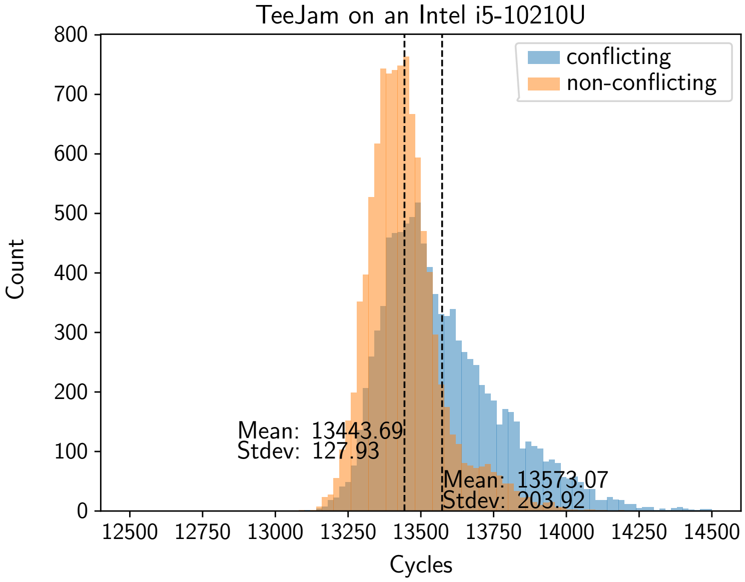


Figure 5: TeeJam measurements with 10,000 measurements of the single-stepping time for each the conflicting and non-conflicting load.

When averaging over 10,000 measurements, we obtain results with a distinguishable difference of about 130 cycles in the mean single-stepping time between single-steps with non-conflicting and conflicting loads. For a comparison, the MemJam effect, as shown in Figure 3, causes a delay of 70 cycles on the same processor at its base frequency. Thus, the delay caused by 4k-aliasing is almost twice as big in the SGX setting. The results of the same experiment on the Xeon E-2286M are shown in the Appendix (Figure 16).

Finally, we repeat the experiment in 4-byte aligned steps over a full memory page to cover all address bits which can be subject to the TeeJam effect. We proceed in 4-byte chunks since we determined in Section 3.2 that this is the maximum spatial resolution achievable with the 4k-aliasing effect. For mapping the influence of the TeeJam effect over a whole page, the experiment from Figure 4 is executed with 90,000 instead of 10,000 measurements on the Intel Core i5-10210. For the evaluation, we compute the difference of the means of the conflicting and non-conflicting loads for each 4-byte chunk. The measurement results are shown in the heatmap in Figure 6 that depicts one cache line per row subdivided into 4-byte chunks. We show larger differences in brighter and smaller differences in darker colors. A black field thus depicts a difference of 0.

Of the 1024 executed offsets, the measurements were successful for 852. For the remaining offsets, we were not able to collect proper measurements due to either (i) excessive zero-stepping or (ii) the measurements resulted in two to three times higher single-stepping times accompanied by a very high variance. Since both effects render the corresponding offsets unusable for side-channel attacks, we assigned them a difference of 0.

A close inspection of the heatmap reveals the existence of some cache lines completely unusable for side-channel measurements, especially in the beginning and end, but also isolated throughout the page. Nevertheless, the majority of the cache lines is suitable for TeeJam.

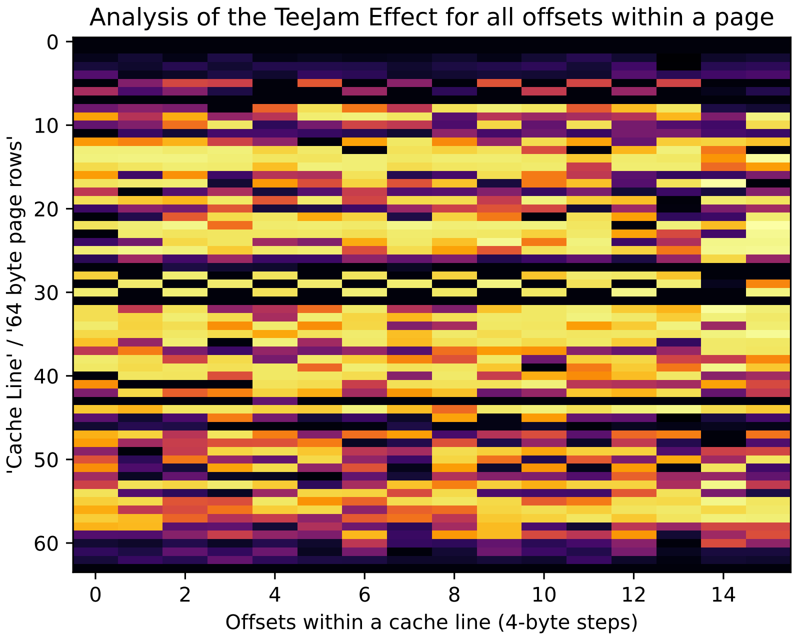


Figure 6: Mapping the TeeJam effect over a full memory page. The experiment is repeated in 4-byte aligned steps. The difference between the mean timing measurement of the conflicting and non-conflicting memory accesses are depicted. Dark colors show small differences, while brighter colors depict large differences. We set "unmeasurable" offsets to 0.

4.3 Evaluation

In this section, we evaluate the results from the previous section in more details to conclude that a high temporal and spatial resolution is possible simultaneously, thus answering RQ1 in the positive. The results from the previous section show an increase in the delay by a factor of approximately two between MemJam and TeeJam. This amplification and the large delay of TeeJam in general allow us to measure 4k-aliasing for memory accesses within enclave execution in a single-stepped fashion, despite the high variance in the single-stepping time. Note, that the original MemJam attack measures the delay introduced by a 4k-conflict over the full execution of a program. With TeeJam, we can measure 4k-conflict on a per-instruction basis and thus drastically increase the timing resolution.

We suspect the reasons for the high variance in the single-stepping time to be attributable to the behavior of the Asynchronous Enclave Exit (AEX) and `EERESUME` instruction as well as the continuously running hyper-thread that executes the conflicting store instructions. Van Bulck et al. [BPS17] found the `ERESUME` instruction to be "relatively deterministic". However, in a more recent work, Constable et al. [CVBC+23] describe the `ERESUME` with "... whose execution time itself varies greatly and can take thousands of CPU cycles" and stating that single-stepping is only possible because of a forced microcode assist associated with the first enclave instruction which might take several hundreds of cycles. Additionally, the relatively complex Asynchronous Enclave Exit, which has to cleanup the enclave state and was subject to changes such as flushing the L1 data cache to counteract attacks such as Foreshadow [BMW+18], introduces additional noise. Finally, measuring the 4k-conflict requires running application on the hyper-thread co-located with the target enclave. Since both threads share microarchitectural components, the core's pipeline additional noise is introduced and the enclave thread is slightly slowed down. In our experiments, this reflects in a higher APIC timer interrupt time for SGX step compared to an execution without hyper-thread and less reliable single-stepping, meaning more frequent zero-stepping with an appropriatley configured APIC timer.

We suspect that the amplification of the MemJam effect in SGX is caused by SGX first waiting for all requests in the store and load buffer to be completed before flushing the L1 data cache and TLB. Due to many pseudo-conflicting stores in the store buffer, this process is delayed causing SGX to wait many cycles before flushing the caches and buffers.

As for the applicability of TeeJam to different offsets, we assume that the difference in the amplitude for most of the working offsets can be explained with measurement noise, i.e. other effects like fetching from last-level cache instead of the second-level cache, which obscures some of the effects. For those offsets, which show excessive zero-stepping or very high single-stepping times and variance, we hypothesize that the 4k-aliasing writes conflict with memory involved in SGX's context switching or the memory translation process during enclave enter and exit [ZMFT22]. A too high variability or the occurrences of such large effects render the timing measurements of these impractical for side-channel attacks. However, the majority of offsets remains fruitful.

RQ1 is answered positively: Single-stepping attacks that achieve a per-instruction temporal resolution can still achieve a sub-cache-line spatial resolution for loads.

4.4 Discussion

In the following we discuss limitations of and potential countermeasures against TeeJam.

Intel SGX and Hyper-Threading: Intel recommends to disable hyper-threading when running secure workloads in Intel SGX [Int23c]. The reasons are attacks like Foreshadow [BMW+18], which can only be fully mitigated by disabling hyper-threading.

Disabling hyper-threading, however, is not desirable in many situations, especially for cloud providers who want to optimize the usage of their resources. Obtaining trusted information about hyper-threading state at runtime is a difficult problem for an enclave [CWC$^+$18, OTK$^+$18] since the `CPUID` instruction is not available within the SGX context. However, the hyper-threading state is verified in some attestation scenarios. For Intel's Enhanced Privacy ID (EPID) attestation scheme, the hyper-threading state is reported as part of the attestation report by returning a code which states that further hardening is required if hyper-threading is enabled [Int23c]. In case of the newer Data Center Attestation Primitives (DCAP) attestation scheme, however, the attestation state is only part of the signed attestation data for multi socket systems [Int23a, Section 3.7]. Attestation, however, is only a one-time check of the system configuration and is not necessarily repeated for every initialization of an enclave after keys were exchanged.

Chen et al. [CWC$^+$18] propose an instrumentation-based technique to detect hyper-threading and AEX side-channel attackers. However, for the hyper-threading countermeasure they require a running trusted hyper-thread on the co-located core. For shared machines offering trusted execution services to customers on all cores, this essentially reduces to disabling hyper-threading as only half of the logical cores remain available. Furthermore, their approach increases overhead to the target program due to frequent checks for asynchronous enclave exits and trusted validation of a running co-located hyper-thread. Moreover, this countermeasure must be preemptively taken by every enclave, especially in the case of vulnerable library code this poses a problem for practical security. Chen et al. also state that their countermeasure can detect an attacker that applies asynchronous enclave exits, however, to mitigate data-flow leakage this requires many AEX checks which will in turn further increase the overhead.

Single-Stepping Countermeasures: Single-stepping SGX enclaves has become a major attack vector. For many years, a plethora of attacks have been enabled by SGX-Step [BPS17]. Thus, it seems only natural that attempts are and were made to reduce the attackers capabilities to mount such attacks. Pridwen [SSL$^+$22] is a tool designed to simplify the application of different attack countermeasures to enclave code. However, it only works with enclaves written in WebAssembly and still requires manual application for each enclave. Among others, Pridwen includes the SGX side-channel countermeasure Varys [OTK$^+$18]. Varys works in very similar manner to the work of Chen et al. It counteracts AEX based attacks by observing the enclaves state save area and hyper-threading based attacks by co-locating a second enclave thread, thus incurring comparable overheads, essentially blocking all logical cores on systems which are used for trusted workloads in shared environments like clouds and requiring manual application to every SGX enclave.

Finally, Intel recently published a revised version of the SGX specification that specifies the AEX-Notify architectural extension [CVBC$^+$23, Int22, Int23b]. This extension enables a hardware assisted single-stepping detection and is implemented in microcode and software. However, a target enclave has to enable this feature and register and implement the response to the detected AEX itself by implementing a trusted handler. Thus, while introducing less overhead due to a push mechanism, AEX-Notify still depends on every enclave handling the countermeasures against single-stepping themselves.

Practical Relevance of TeeJam: While countermeasures against single-stepping and hyper-threading based attacks against SGX exist, most of these incur significant overhead and all of them require the enclave developer to apply these mechanisms to their enclave themselves. Especially the latter is a problem for vulnerabilities in commonly used cryptographic libraries, which have also been used to construct enclaves. The libraries are general-purpose and thus do not contain such countermeasures. Instead, enclave developers would have to include countermeasures when integrating the library for enclave usage, resulting in a practical risk that either the countermeasures are not applied at all, applied incorrectly or are incomplete and can be circumvented by sophisticated attackers. Disabling hyper-threading significantly reduces the available computing power and thus increases costs for infrastructure providers which in turn is not desirable. Moreover, a misconfiguration on the server side or a missing or flawed attestation check could lead to the inadvertent activation of hyper-threading. As such, single-stepping and hyper-threading based attacks remain a practical problem, providing further evidence that countermeasures should be applied by default without requiring the developer's intervention.

5 Recovering RSA Private Keys with TeeJam

The TeeJam effect described in Section 4 introduces a novel way to implement an attack achieving high temporal and spatial resolution simultaneously. In this section, we investigate RQ2 and aim to find suitable targets that could not be fully exploited before (due to a lack of spatial or temporal resolution).

In the scenario studied here, we apply TeeJam to the decoding of a base64 encoded RSA private key with OpenSSL and demonstrate how the fine granular resolution allows for reconstructing even 4096-bit private keys. This scenario was considered in `Util::Lookup` but due to a lack of sub-cache-line resolution, it was not possible to reconstruct private keys of length more than 512 bits or 1024 bits without specialized hardware [SBWE21]. Besides significantly extending the range of keys that can be recovered, we also adapt the key reconstruction from `Util::Lookup` to work with unaligned partitions, missing information, and Carmichael's totient function, allowing us to implement a full end-to-end key recovery attack.

We begin with describing the state of the art RSA key recovery from side-channel leakage obtained from the key file's base64 decoding process, continue with the general idea of how to extract information from base64 decoding in Section 5.3 and then describe the actual attack on the decoding process with TeeJam in Section 5.4. Finally, we elaborate on the key reconstruction in Section 5.5 and Section 5.6.

5.1 State of the Art RSA Key Recovery from Base64 Decoding

The recovery of RSA private keys from side-channel information gathered during base64 decoding was previously attempted in `Util::Lookup` [SBWE21]. The attacker in `Util::Lookup` exploits that in most cryptographic libraries base64 decoding is implemented with table lookups, translating from base64 symbols to binary. They run a cache attack, specifically a Prime+Probe attack, on a single-stepped enclave. Meaning, they first create eviction sets for the cache lines they want to observe, start the enclave and then, after every step, probe and prime the cache sets. The attack runs in a single trace, meaning in the optimal case it only requires one repetition. More repetitions will not increase the amount of recovered information. However, the obtained information or in other words the maximum leakage is only one bit per access to the lookup table and thus per translated symbol. Due to the uneven distribution of symbols to the cache lines, the real leakage is even below one bit per access. The private key is stored in a heavily redundant way to speed up the decryption operation. Hence, the private key contains five values that are closely related to each other and the attack obtains one bit for each of these five values. The authors of `Util::Lookup` used this knowledge to employ a combination of the Heninger-Shacham RSA key reconstruction algorithm [HS09] and the lattice algorithm `small_roots` in Sagemath based on Coppersmith's method [Cop97] for reconstructing the complete key from this small leakage.

The Heninger-Shacham reconstruction algorithm expects observations on single bits of the key. The information from the `Util::Lookup` attack, however, delivers information on

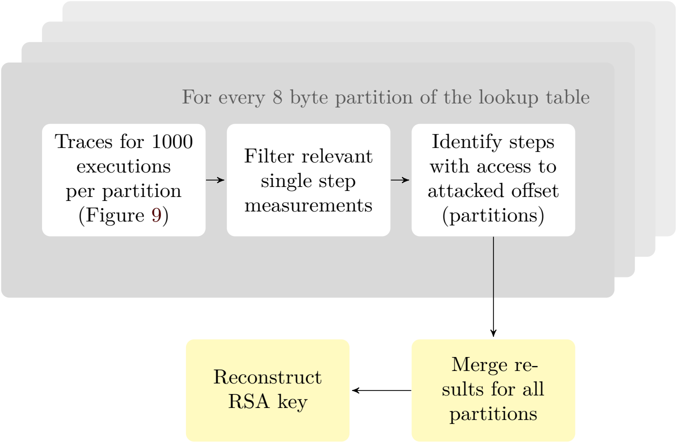

For every 8 byte partition of the lookup table
Traces for 1000 executions per partition (Figure 9)
Filter relevant single step measurements
Identify steps with access to attacked offset (partitions)
Merge results for all partitions
Reconstruct RSA key

Figure 7: Overview of the attack to reconstruct RSA private keys from execution traces recorded during the key's base64 decoding process.

blocks of size 6 bits. Therefore, the authors generalize the algorithm to work with blocks of variable size. Due to the small leakage, so far only keys with a maximum size of 512 to 1024 bits could be reconstructed with the information from the side-channel attack. The reconstruction of a 512-bit key already requires more than 4,000 CPU hours on commodity hardware and realistic key lengths are thus out of reach for `Util::Lookup`.

In the next sections, we will show how TeeJam can be used to significantly increase the amount of leaked information and how this information can then be used to reconstruct even keys with a size of 4096 bits on commodity hardware. Additionally, we will show how RSA keys with Carmichael totient can be reconstructed, which is not possible with the reconstruction algorithm from `Util::Lookup`.

5.2 Applying TeeJam to Table Lookups

Classical cache attacks provide a maximum spatial resolution of cache-line granularity, i.e., of 64 bytes on modern Intel processors. In the case of base64 decoding where the relevant information in most cases only spreads over two cache lines, the attacker is limited to distinguish between only two sets or partitions of symbols which are translated with the observed steps. Using TeeJam, the attacker can launch a statistical attack on the victim enclave by provoking 4k-conflicts from a hyper-thread with a granularity of as little as four bytes, potentially splitting a lookup table of 128 bytes into up to 32 partitions. Since TeeJam is a statistical effect, the decoding has to be observed repeatedly while the attacker provokes conflicts to the same partition. After sufficiently many observations, the same process can be repeated with the next partition. To determine whether the attacked partition was accessed during a table lookup, the attacker computes the average single-stepping time of each observed single-step corresponding to a lookup table access over all observations for which they attacked the same partition. If the average single-stepping time is higher than those of the preceding and following steps, the attacked partition was accessed by the victim.

5.3 Information Retrieval From Base64 Decoding

To reconstruct RSA keys from the information gathered during the decoding of a base64 encoded RSA private key, we proceed in several steps. Figure 7 shows an overview of the complete attack and reconstruction process. First, we collect single-stepping timing traces

```
0x00  ff ff ff ff  ff ff ff ff
0x08  ff e0 f0 ff  ff f1 ff ff # TAB LF CR
0x10  ff ff ff ff  ff ff ff ff
0x18  ff ff ff ff  ff ff ff ff
0x20  e0 ff ff ff  ff ff ff ff # SPACE
0x28  ff ff ff 3e  ff f2 ff 3f # + - /                0
0x30  34 35 36 37  38 39 3a 3b # 0 1 2 3 4 5 6 7      1
0x38  3c 3d ff ff  ff 00 ff ff # 8 9 =                2
0x40  ff 00 01 02  03 04 05 06 # A B C D E F G        3
0x48  07 08 09 0a  0b 0c 0d 0e # H I J K L M N O      4
0x50  0f 10 11 12  13 14 15 16 # P Q R S T U V W      5
0x58  17 18 19 ff  ff ff ff ff # X Y Z                6
0x60  ff 1a 1b 1c  1d 1e 1f 20 # a b c d e f g        7
0x68  21 22 23 24  25 26 27 28 # h i j k l m n o      8
0x70  29 2a 2b 2c  2d 2e 2f 30 # p q r s t u v w      9
0x78  31 32 33 ff  ff ff ff ff # x y z               10
```

Figure 8: OpenSSL's base64 decoding lookup table. The boxes indicate the partitions observable with the TeeJam attack. The partitions are identified by the numbers on their right. The comments list the ASCII representations contained within each partition.

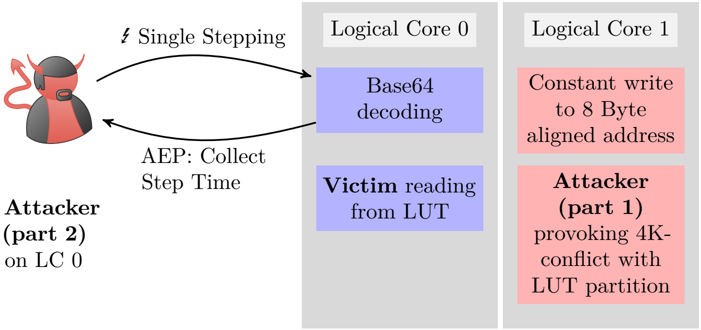
for every partition shown in Figure 8. From these traces, we filter those timings which are related to decoding lookups. The filtered traces for each partition are then analyzed to identify those load operations that accessed the observed partition, meaning the partition attacked with 4k-conflicts. Finally, the results for all partitions are merged and used as input for the RSA key reconstruction algorithm.

The decoding of base64 in many cryptographic and utility libraries is implemented with lookup tables [SBWE21]. There are minor differences in the implementations, but in general the ASCII code of each base64 symbol is used as the index to an array which holds the associated binary values. As each base64 symbol corresponds to $\log_2(64) = 6$ bits of information, the base64 decoding algorithm proceeds by concatenating the information from four table lookups into three bytes.

Exemplarily, the lookup table (LUT) used by OpenSSL [CT23] is shown in Figure 8. It has a size of 128 bytes, potentially holding all ASCII characters, and replaces unneeded bytes with `0xff`. During the translation of a private key Privacy-enhanced Electronic Mail (PEM) file, OpenSSL parses every base64 symbol twice. First, it collects 64 symbols and translates them to verify their validity, then it iterates over the same chunk of 64 bytes again for the actual decoding.

With a cache attack, the maximum information which can be gathered per lookup is one bit, assuming an equal distribution of symbols per cache line. For the OpenSSL LUT the mutual information is approximately $I(B, P) = 0.696$ bit for 64-byte alignment of the LUT or $I(B, P) = 0.974$ bit for 32-byte alignment [SBWE21], where I is the mutual information and the random variables B and P denote the base64 symbol and the partition, respectively. For a cache attack, the partition size is a 64-byte cache line. `Util::Lookup` shows that this information is enough to reconstruct small keys with a size of 512 bytes or 1024 bytes with sufficient computing resources as well as to decrease the security level of larger keys.

Figure 8 shows the partitioning of the OpenSSL LUT we chose for a sub-cache-line attack with TeeJam. As shown in Section 3, TeeJam offers a resolution of up to four bytes. However, we settle, as a tradeoff, for partitions of eight bytes to reduce the amount of necessary observations by 50% and still obtain more than sufficient leakage to reconstruct the decoded keys from the observed side-channel information. Each of the chosen partitions consists of two to eight symbols, depending on the symbol distribution corresponding to the ASCII encoding, as some of the symbols can only occur in certain special positions. More concretely, the symbol '-' does not appear in the base64 standard, but only in some



↯ Single Stepping
AEP: Collect Step Time
Attacker (part 2) on LC 0
Logical Core 0
Base64 decoding
Victim reading from LUT
Logical Core 1
Constant write to 8 Byte aligned address
Attacker (part 1) provoking 4K-conflict with LUT partition

Figure 9: Attack setup for retrieving traces which yield the accessed lookup table (LUT) partitions. Logical core (LC) 0 and 1 are hyper-threads on the same physical core.

variants. Furthermore, the symbol '=' is only ever used for padding and thus cannot appear in arbitrary positions. Hence, we have two partitions in which 2 symbols are possible, five partitions with 8 symbols, two partitions with 7 symbols, and two partitions with 3 symbols. The entropy is maximized by considering the uniform distribution in each partition, yielding mutual information of

$$\begin{aligned} I(B,P) = &\frac{2\cdot 2}{64}\cdot \log_2\left(\frac{64}{2}\right) + \frac{5\cdot 8}{64}\cdot \log_2\left(\frac{64}{8}\right) \\ &+ \frac{2\cdot 7}{64}\cdot \log_2\left(\frac{64}{7}\right) + \frac{2\cdot 3}{64}\cdot \log_2\left(\frac{64}{3}\right) \approx 3.3 \text{ bit} \end{aligned}$$

for each correctly detected lookup table access. This is more than half of the total available information of 6 bits and by far enough to reconstruct the decoded private key from the captured leakage, even with imperfect traces that miss some observations. Choosing smaller partitions would require to collect more traces, thus complicating the attack without any real gain for the attacker.

5.4 Lookup Table Trace Recovery

To extract all the information as described in Section 5.3, we design an attack based on TeeJam. We run the following experiment on an Intel Core i5-10210U. The attack is depicted in Figure 9. The victim, i.e., the decoder of the private RSA key, is running on a fixed logical core in an SGX enclave while *part 1* of the attacker is running on the other logical core (hyper-thread) of the same physical core. *Part 1* of the attacker constantly writes to a pseudo-conflicting 4k-aliasing address of one of the partitions from the victim's LUT. *Part 2* of the attacker runs on the same thread as the enclave and uses SGX-Step [BPS17] to single-step the enclave, measure the single-stepping time and observe the page access bits of the pages holding the LUT and decoding routine. The single-stepping time is defined as described in Section 2.2.

For every partition, as shown in Figure 8, the attacker observes 1,000 decodings while continuously executing stores to the corresponding 4k-aliasing address. They store the recorded traces of single-stepping times and page accesses to the LUT and decoding routines. Afterwards, the page access bits are used to filter only the steps with access to the LUT. Then, only traces with the same length are considered. In all our experiments, traces with the correct length were the clearly dominating share of all traces, thus allowing to identify the correct length easily.

As described in Section 5.3, OpenSSL has a special way of parsing a PEM file twice. Knowing the underlying algorithm, it is simple to select either the first or second pass for

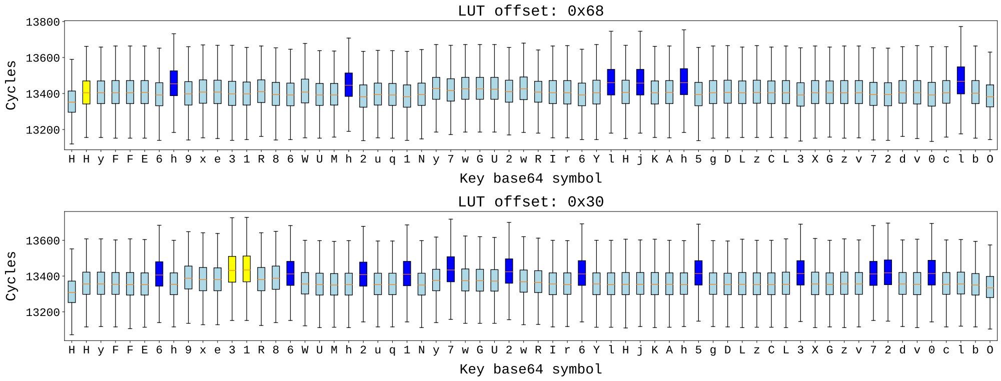


Figure 10: base64 decoding in OpenSSL works by parsing every line of 64 symbols in the PEM file twice: first pass for writing all symbols to a buffer and checking for correctness, second for the actual translation. We depict the first pass for the third line of a 1024-bit key for two different attack offsets into the lookup table (LUT), partition 8 and 1 as defined in Figure 8. Comparing with the symbols in the partitions reveals longer single-stepping latencies when these are decoded. The correctly detected accesses to the partitions are marked in blue, yellow highlights accesses which could not be detected or for which we detected accesses for multiple partitions.

each *64 symbol block* from the trace. We use the first pass, which checks each symbol for validity, as the measurement results are clearer to interpret.

For illustration purposes, Figure 10 shows the measurement results for two selected offsets for the third block of 64 symbols of a 1024-bit key decoded with OpenSSL. The single-stepping times are depicted in a box plot, which shows the median and quantiles for each decoded symbol when superimposing all traces of correct length. Comparing the attacked offset to the corresponding symbols in the offset's partition (Figure 8) reveals a distinguishable higher single-stepping time for nearly all symbols in the attacked partition.

For the automated analysis performed in our end-to-end attack, we use the average A instead of the median for each symbol's single-stepping time. In order to automatically detect attacked LUT accesses, we use a symmetric moving window of size 15 on the traces of average single-stepping times. Let s be the standard deviation, and a be the average over the current window. We detect an access to a partition if $A - a \geq 1.5 \cdot s$ and $A - a \geq 20$ cycles.

As to be expected from noisy side-channel measurements, not every access can be reliably determined. Additionally, a drift over time can be observed in the overall trace. This prevents detecting attacked offsets with a simple threshold and requires the classification within a moving window. While most of the LUT accesses in the example in Figure 10 can be clearly determined, few like the yellow marked symbols in the lower box plot cannot clearly be distinguished from their surrounding measurement values and are false negatives. We choose conservative parameters for the selection algorithm, meaning rather rejecting the detection of a memory access than risking false positives. Even with this approach that results in missing true positives, enough information is readily available due to the high sub-cache-line resolution. With a proper amount of repetitions and selection threshold, we did not encounter any false positives during our experiments.

Due to the good results, it was not necessary to determine whether one of the unusable offsets, as described in Section 4.2, had to be excluded from the measurements. However, we left out those offsets of the LUT that do not contain information used for the base64 decoding.

Figure 11 shows the merged classification results of all partitions for a full 4096-

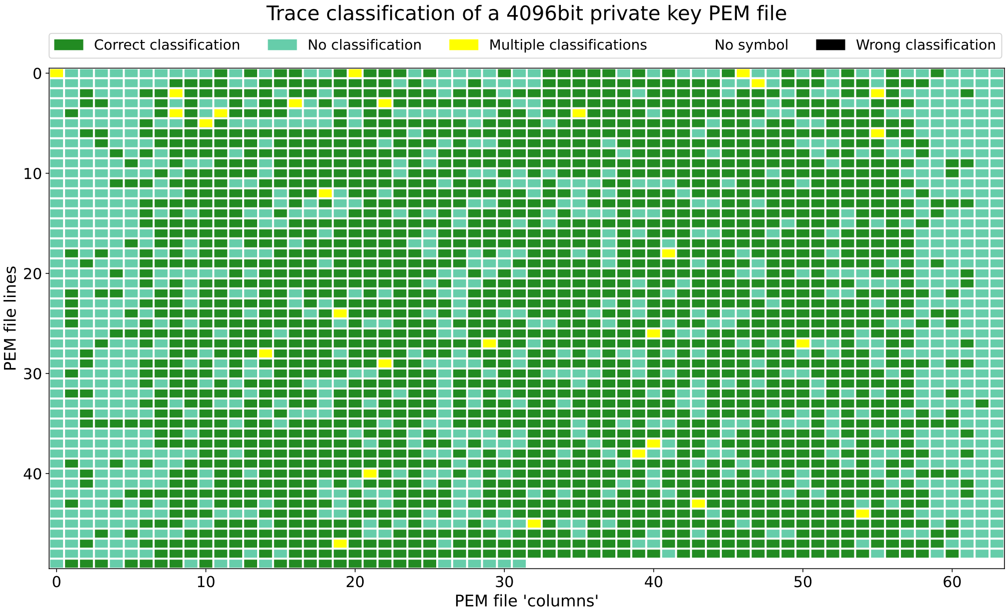


Figure 11: Result of the trace recording with 1,000 repetitions for each attacked offset during the attack on a 4096-bit key with Euler totient function. Classification was chosen to be "conservative" as described in Section 5.4 to avoid undetectable errors. Missing and invalid observations: 1117

bit private key. It shows whether a step or rather LUT access was classified with the correct partition. It also shows whether we did not detect any LUT access for any of the partitions ("no classification"), whether multiple partitions were detected, in which case we interpret the measurement as missing as well, or whether a false positive access was detected ("wrong classification"). Performing 1,000 repetitions for each attacked offset and choosing conservative parameters, our experiment contains no false positive and about 65% of the bits are correctly detected. We leave further optimization of the detection algorithm's parameters to future work, as the current results provide sufficient information for reconstructing the key.

RQ2 is answered positively: Attacks with high temporal and high spatial resolution allow to attack implementations that are explicitly hardened against side-channel attacks.

5.5 Key Recovery

While the previous discussion already shows the applicability of TeeJam when loading and decoding the private key, it does not constitute a complete end-to-end attack. Similarly, `Util::Lookup` also omitted several steps needed for a complete end-to-end attack [SBWE21]. In order to present a complete end-to-end attack, we now fill the remaining gaps of the attack, including dealing with missing partition classifications, trace alignment, and handling keys using the Carmichael totient instead of Euler's totient.

The obtained trace as described in Section 5.4 is a base64 representation of the key's binary blob that is composed of the parameters $(N, e, p, q, d, d_p, d_q, {q_p}^{-1})$ in Abstract Syntax Notation One (ASN.1) encoding. We modify and generalize the algorithm from `Util::Lookup` that is based on the algorithm by Heninger and Shacham [HS09]. In `Util::Lookup`, an *idealized* and aligned trace is generated for evaluation of the reconstruction algorithm. In contrast, we implement a full end-to-end attack; however, this requires to handle missing classifications and to identify the correct base64 symbols corresponding to the private key parameters which we discuss at the end of Section 5.5.

The Key Recovery Algorithm of `Util::Lookup`: The main idea of the key recovery algorithm is to reconstruct the different bits of the secret key $sk^\star = (p^\star, q^\star, d^\star, d_p^\star, d_q^\star, q_p^{\star\,-1})$ iteratively by building up a set of *candidates*. Each candidate corresponds to a potential RSA secret key compatible with our observations. In the first step, we find all possible values k, k_p, and k_q such that

$$\begin{aligned} e \cdot d &= k(N - p - q + 1) + 1 \\ e \cdot d_p &= k_p(p - 1) + 1 \\ e \cdot d_q &= k_q(q - 1) + 1. \end{aligned}$$

Due to our observations, the number of such possible triples, denoted by obs, is typically very low, i.e., usually two. Then, we initialize the set of candidates with a few candidates (described later) where a few bits are already set. The *depth* of a candidate $\widetilde{sk}$ corresponds to the number of bits already reconstructed. Next, we apply the expand operation on each candidate $\widetilde{sk}$ to obtain two candidates $\widetilde{sk}_1$ and $\widetilde{sk}_2$. Whenever possible, we compare the set of current candidates to our observations via the check operations and discard candidates not matching the observations obs. A short description following [SBWE21] of the complete algorithm is presented in Figure 12.

```
Input: Observation obs, target depth $D$
find valid triples $(k, k_p, k_q)$
for each possible triple $(k, k_p, k_q)$:
  initialize empty stack $S$
  add initial candidates $\widetilde{sk}(k, k_p, k_q)$ to $S$
  while $S$ is not empty:
    let $\widetilde{sk} = S$.pop()
    let $\widetilde{sk}_1, \widetilde{sk}_2 = \mathsf{expand}(\widetilde{sk})$
      for $\beta \in \{1, 2\}$:
        if depth of $\widetilde{sk}_\beta \geq D$: output $\widetilde{sk}_\beta$
        if depth of $\widetilde{sk}_\beta = j \cdot b$ and $\mathsf{check}(\mathsf{obs}, \widetilde{sk}_\beta)$:
          $S$.push($\widetilde{sk}_\beta$)
        else if depth of $\widetilde{sk}_\beta \neq j \cdot b$ : $S$.push($\widetilde{sk}_\beta$)
```

Figure 12: Concise description of our adapted key-reconstruction algorithm

Adapting the Algorithm: The original reconstruction algorithm assumes private keys with the Euler totient function $\varphi(N) = (p - 1) \cdot (q - 1)$ for the derivation of the equations used in the check operation. However, OpenSSL generates private keys with Carmichael totient function as defined by the standard [MCK+16]. The Carmichael totient function uses $\lambda(N) = \operatorname{lcm}((p-1),(q-1)) = \frac{\varphi(N)}{\gcd((p-1),(q-1))}$ instead of $\varphi(N)$, which complicates the equations used in the expand operation. Since p and q are prime, it holds $\gcd((p-1),(q-1)) \geq 2$. Consequently, $\lambda(N) < \varphi(N)$ for all N. However, if the Euler totient function is used and $d < \lambda(N)$ holds for the private exponent d, then the keys generated with OpenSSL are compatible with Euler's totient function. However, this is not guaranteed and we thus need to adapt the key recovery algorithm. We analyze the state of the art from `Util::Lookup` and adapt the algorithm to handle both types of keys in Section 5.6 by adapting the expand operation.

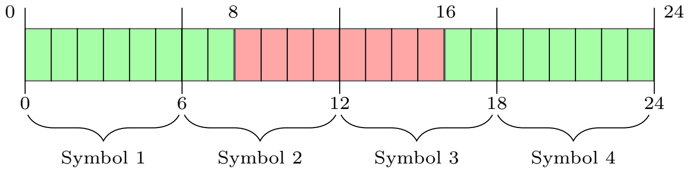


Figure 13: ASN.1 DER encoded key bytes with their representation in base64 encoded PEM files. For determining the overlap of bits from one byte into the next base64 symbol it is only necessary to analyze a partition of 24 bits (as the least common multiple of 6 and 8 is 24).

Missing Classifications: Dealing with missing partition classifications is a rather simple endeavor due to the amount of observed information. Our key recovery algorithm is based on the algorithm by Heninger and Shacham [HS09]. The algorithm from `Util::Lookup` generalizes the available information to be provided in partitions of variable size (here: six bits) instead of single bits. To compensate for the missing partition information, we omit the corresponding checks against the missing information from the side-channel observation and thus continue with a "non-pruned" set of initial candidates.

Trace Alignment: ASN.1 or for cryptographic purposes more specifically Distinguished Encoding Rules (DER) encoded private keys follow a strict structure [ITU23a, ITU23b]. For the sake of simplicity, it suffices to say that the keys start with some meta information of fixed and known length and that every parameter or variable is prefixed with information about its type and length. However, the length of the parameters themselves can vary within a rather bounded range. As (N, e) are public key parameters, we can derive the maximum length of $(p, q, d, d_p, d_q, {q_p}^{-1})$ from them. Nevertheless, the parameters might also be shorter than their maximum possible length by a few bits. For example, d_p is clearly in the range $\{0, \ldots, p-1\}$ and thus can be represented by $\log_2(p)$ bits, but about half of the values in this range only need at most $\log_2(p) - 1$ bits, a quarter only need $\log_2(p) - 2$ bits and so on. Hence, it is unlikely (but not impossible) that the length of any parameter is eight bits shorter than expected. Since the parameters in ASN.1 are byte aligned, a variation of up to eight bits introduces only two possible lengths in bytes for each parameter. Still, per parameter, two lengths in bytes are possible because ASN.1 inserts a zero byte in the most significant position if a parameter has its most significant bit set. For the key reconstruction algorithm, we proceed by assuming parameter lengths in previously explained boundaries. With a maximal parameter length variation of eight bits, there are at most $8^5 = 32,768$ variations. But as we know the file size of the complete key, we can use this knowledge to reduce the number of variations significantly, especially if many parameters have full length and thus a leading zero byte. Since this is only a rather small factor and reconstruction with the available information from the fine grained TeeJam attack is very fast (see Section 5.6), we assume the parameters' length to be known. Having the correct (or assumed) parameter length, one can determine the corresponding elements from the partition trace created as explained in Section 5.4.

However, a base64 symbol or trace element itself does not align with the bytes from the DER representation as depicted in Figure 13. To use the correct information for the key reconstruction, we cannot simply use the first trace element that overlaps with a parameter's start and end byte. Instead, we create four new partition tables with partitions which contain only the information about the lower two and four bits as well as the upper two and four bits from the original partitions. Additionally, we adapt the algorithm to accept these "sub-partitions" as observations for checking the least and most significant bits of the candidates. Since usually not all parameters are aligned in the same way, we compare the single parameters against their observation at different points during the reconstruction.

5.6 Reconstructing Carmichael Keys

In the following, we describe how to adapt the key recovery algorithm by Heninger and Shacham [HS09] to also recover RSA keys where the private exponent d is determined by the congruence $e \cdot d \equiv 1 \pmod{\lambda(N)}$ where $\lambda(N) = \mathrm{lcm}(p-1, q-1)$. We denote the secret key by $sk^\star = (p^\star, q^\star, d^\star, d_p^\star, d_q^\star, q_p^{\star-1})$, but will ignore the variable q_p^{-1} from now on (as described in [SBWE21], the variable q_p^{-1} behaves quite differently from the other variables).

To perform the expand operation, we make use of four equations that can be derived from the structure of the secret key for the three integers k, k_p, and k_q, namely

$$\begin{aligned} N &= p \cdot q, \\ e \cdot d &= k \cdot \lambda(N) + 1, \\ e \cdot d_p &= k_p \cdot (p-1) + 1, \\ e \cdot d_q &= k_q \cdot (q-1) + 1. \end{aligned}$$

Heninger and Shacham only consider RSA keys with regard to Euler's totient function $\varphi(N)$ instead of the Carmichael function $\lambda(N)$ [HS09], As the algorithm rewrites the above equations into polynomials, this allows them to write $\varphi(N) = (p-1) \cdot (q-1) = N - p - q + 1$. For the Carmichael function, the situation is more complicated, as $\lambda(N)$ cannot be expressed simply as a sum of N and its prime factors. However, as $\lambda(N)$ always divides $\varphi(N)$, we know that $\lambda(N) = \varphi(N)/r$ holds for some integer r. Furthermore, with high probability, this integer r is quite small [MvOV96, Note 8.5]. To accommodate for this, we introduce another integer γ and multiply both sides of the relation $e \cdot d = k \cdot \lambda(N) + 1$ with it to obtain (with a redefinition of k) the relation $\gamma \cdot e \cdot d = k \cdot (N - p - q + 1) + \gamma$. Note that this relation holds as long as γ is divided by r. More formally, Diaconis and Erdős showed that the expected size of the greatest common divisor of two random x bit numbers is $6/\pi^2 \cdot \log(x) + O(\log(x)/\sqrt{x})$ [DE04]. Hence, even for primes consisting of 2048 bits, we only need to test at most 2.000 values for γ with sufficiently high probability. In the following, we suppose that our choice of γ is correct. In our implementation, we simply choose γ as a small factorial, i.e., $\gamma \in \{2!, 3!, 4!\}$ to capture the most likely values of r.

By slightly adapting the ideas of Heninger and Shacham, the values of k, k_p, and k_q can be found efficiently. It is easy to see that $k \leq \gamma \cdot e$ and for each candidate k, we can find a value $\hat{d}(k) = \lfloor (k' \cdot (N+1) + \gamma)/(\gamma \cdot e) \rfloor$ that agrees with d on the upper half of the most significant bits. Due to our observations, this allows us to determine k uniquely. From k, we can set up a quadratic polynomial where the roots are exactly k_p and k_q. See Appendix C.1 for a more thorough discussion.

Finding the initial candidates: After we have determined k, k_p, and k_q, we can now describe the first few candidates for the secret key. For an integer x, let $\tau(x)$ be the largest integer such that $2^{\tau(x)}$ divides x. For reasons shown later, we need to initialize the first $\tau(\gamma)$ bits of p and q, the first $\tau(k)$ bits of d, the first $\tau(k_p) + \tau(\gamma)$ bits of d_p, and the first $\tau(k_q) + \tau(\gamma)$ bits of d_q. As described in [HS09], we can deduce the corresponding bits for d, the $\tau(k_p)$ LSBs of d_p, and the $\tau(k_q)$ LSBs of d_q, but only know the least significant bit of p and q. We thus need to enumerate the remaining $\tau(\gamma) - 1$ bits of p, the remaining $\tau(\gamma) - 1$ bits of q, the remaining $\tau(\gamma)$ bits of d_p, and the remaining $\tau(\gamma)$ bits of d_q via brute-force. Hence, we initialize our list of candidates with $2^{4\tau(\gamma)-2}$ candidates for each valid triple (k, k_p, k_q).

Expanding a candidate: Now, we only need to describe the expand operation. To do so, we denote the i-th least significant bit of x by $x[i]$. The least significant bit of x is thus $x[0]$. In the following, we will focus only on a single candidate $\widetilde{sk} = (p', q', d', d'_p, d'_q)$ that we want to expand. Suppose that variable p' has length $i + \tau(\gamma)$, variable q' has length $i+\tau(\gamma)$, variable d has length $i+\tau(\gamma)+\tau(k)$, variable d_p has length $i+\tau(k_p)+\tau(\gamma)$, and variable d_q has length $i + \tau(k_q) + \tau(\gamma)$. The goal is to construct all possibilities to extend each variable by a single bit, i.e., by the bits $p[i+\tau(\gamma)]$, $q[i+\tau(\gamma)]$, $d[i+\tau(\gamma)+\tau(k)]$, $d_p[i + \tau(k_p) + \tau(\gamma)]$, and $d_q[i + \tau(k_q) + \tau(\gamma)]$. Using Hensel's Lemma, we can derive the following check equalities that need to be fulfilled for the `expand` operation:

$$p[i + \tau(\gamma)] + q[i + \tau(\gamma)] \equiv (N - p'q')\,[i + \tau(\gamma)] \pmod 2 \tag{1}$$

$$(p[i{+}\tau(\gamma)]{+}q[i{+}\tau(\gamma)]{+}d[i{+}\tau(k)]) \equiv (k(N{+}1){+}\gamma{-}k(p'{+}q'){-}\gamma e d')\,[i{+}\tau(\gamma){+}\tau(k)] \pmod 2 \tag{2}$$

$$p[i{+}\tau(\gamma)]{+}d_p[i{+}\tau(k_p){+}\tau(\gamma)] \equiv \left(k_p(p'{-}1){+}1{-}e\cdot d'_p\right)[i{+}\tau(k_p){+}\tau(\gamma)] \pmod 2 \tag{3}$$

$$q[i{+}\tau(\gamma)]{+}d_q[i{+}\tau(k_q){+}\tau(\gamma)] \equiv \left(k_q(q'{-}1){+}1{-}e\cdot d'_q\right)[i{+}\tau(k_q){+}\tau(\gamma)] \pmod 2. \tag{4}$$

For a detailed derivation of these equalities, we refer to Appendix C.2.

5.7 Experimental Evaluation

Complexity: It can easily be seen from the analysis in `Util::Lookup` that our key recovery algorithm will run in polynomial time, as the amount of partial information derived from our attack is sufficiently high. Clearly, for each candidate generated in the run of the algorithm, the operations run in polynomial time. We thus only need to bound the total number of candidates and, as there is only a single correct key, this approximates the number of incorrect candidates. More formally, we make use of the following theorem implied by Theorem 1 and Theorem 2 in [SBWE21] to bound the number of incorrect candidates generated by single initial candidate. Here, b denotes the block length of the observations (which is equal to 6 in our application), $H_2(\mathrm{pr})$ denotes the entropy of the observations (which is equal to 3.3 bits in our application), and $\langle N\rangle$ denotes the key length.

Theorem 1. *For each initial candidate, the expected number of incorrect candidates produced by the algorithm is* $2^b \cdot \sum_{i=0}^{\lceil \langle N\rangle/b\rceil}(2^{b-5\cdot H_2(\mathrm{pr})})^i$.

As $(2^{b-5\cdot H_2(\mathrm{pr})}) \leq 1$ due to the large number of information gained by our attack (we have $H_2(\mathrm{pr}) \geq 3$), we expect at most $2^b \cdot (\langle N\rangle/b + 2)$ incorrect candidates per initial candidate. As we start with $2^{4\tau(\gamma)-2}$ initial candidates, the total number of expected incorrect candidates is at most $2^{b+4\tau(\gamma)-2} \cdot (\langle N\rangle/b+2)$. For example, in the case of 4096-bit RSA keys, we only generate about 43,840 incorrect candidates per initial candidate. For $\gamma = 4! = 24$, we have $\tau(\gamma) = 3$, as 24 is divisible by $2^3 = 8$ and the number of initial candidates is thus $2^{4\tau(24)-2} = 2^{10}$ for each valid triple (k, k_p, k_q). Hence, we generate about $2^{16} \cdot 43{,}840 = 2{,}873{,}098{,}240$ incorrect candidates for each such triple.

Reconstruction from Experimentally Collected Data: In Section 5.5 we describe how we use TeeJam to obtain the "partition trace" of a 4096-bit RSA private key shown in Figure 11. We use our reconstruction algorithm to successfully reconstruct the key in 13 seconds with $\gamma = 1$ and 124 seconds with $\gamma = 2$ on an AMD Ryzen 7950X with 16 cores.

Evaluation of the Extended Reconstruction Algorithm: To further demonstrate the practical feasibility of the generalization to keys using Carmichael's totient function, we generated ten such keys each with $\log_2(N) = 4096$ and artificially generate traces that exactly correspond to traces created by our side-channel analysis. Since the trace from the side-channel measurements include measurement noise of about 35%, we also randomly remove this portion from the artificial traces. The corresponding values for $\gcd(p-1, q-1)$

were 2, 6, and 8. We then used our algorithm for $\gamma \in \{1, 2, 6, 24\}$, i.e., some keys were only recovered for $\gamma = 24$. The main difference between the two types of keys is the number of starting candidates produced due to the additional brute-force step. The maximal numbers of such initial candidates were 90 ($\gamma = 1$), 2,352 ($\gamma = 2$), 3,920 ($\gamma = 6$), and 4,428,288 ($\gamma = 24$). Even in the slowest configuration, our algorithm needed at most 57 minutes total computation time to reconstruct the complete key. In more detail, the maximal total running time for $\gamma = 1$ was 18 seconds, for $\gamma = 2$ was 54 seconds, for $\gamma = 6$ was less than 2 minutes and finally, for $\gamma = 24$, the maximal time was less than 57 minutes. All of these computations were performed on a Intel(R) Xeon(R) Gold 6438Y+ dual socket machine with in total 128 logical cores on 2 CPUs, each consisting of 32 physical cores. Due to the very high parallelity of the algorithm (see [SBWE21]), the complete computation for the case with 4,428,288 initial candidates ($\gamma = 24$) takes less than seven minutes on an off-the-shelf server using 128 threads.

6 AES

AES is the most widely used symmetric cipher, included in almost all modern crypto libraries. Over time, different implementation variants have emerged, from plain software implementations using S-Boxes or T-Tables to hardware assisted variants supported by vector extensions, to full hardware implementations like AES-NI. WolfSSL [wol23a] provides multiple implementations for AES. While supporting AES-NI, WolfSSL does not yet support AES-NI for SGX compilation [wol23b].

Recent analysis revealed that the WolfSSL AES T-Table implementation is not constant-time [WPS+23]. This issue was fixed in WolfSSL version 5.6.2 [wol23d] by always accessing every cache line of a T-Table when one of its entries is looked up. All cache lines are accessed at the same offset corresponding to the entry which is looked up. The correct value is arithmetically selected after being read into a buffer. Hence, the approach as a constant access pattern at cache line resolution, preventing classic cache attacks.

In this section, we shortly present the current implementation and countermeasure of WolfSSL's AES T-Table implementation and show how to use the TeeJam effect to slow down memory access to specific offsets in each T-Table. To demonstrate how the leakage introduced by the TeeJam effect can be exploited, we present a known-ciphertext attack targeting the offset leakage of the last round that successfully recovers the entire AES key. The resulting attack demonstrates that TeeJam can be used to overcome implementations that only consider cache line leakage. By exploiting fine-grain single-instruction leakage rather than the aggregate execution time of the full AES encryption, we manage to reduce the number of observations needed for full key recovery by three orders of magnitude when compared to MemJam [MWES19].

6.1 WolfSSL AES T-Table Implementation

First, we shortly describe WolfSSL's AES T-Table implementation [wol23c] which does not feature exploitable leakage at cache line resolution, effectively thwarting classic cache attacks such as Flush+Reload or Prime+Probe. As presented in Listing 5, in the last AES round, the T-Table is accessed using the function `GetTable_Multi`. In preceding rounds, both `GetTable_Multi` and the almost identical function `XorTable_Multi` are used. Both functions behave the same, except for the final assignment, which is replaced with a `xor and assign` in the latter function. We will use both function names synonymously. `GetTable_Multi` retrieves all entries from a T-Table required for a round and is called four times per round for AES128. Within `GetTable_Multi` for every T-Table entry, a for-loop accesses the same offset of every cache line of that T-Table and selects the correct entry in constant time. As every T-Table has a size of 1024 bytes and a cache line on x86 systems is of size 64 bytes, the mitigation requires 16 accesses per table lookup. An attacker with cache line resolution will not be able to differentiate between these accesses, leaving them with no information to be obtained.

6.2 Applying TeeJam to AES Encryption

With the sub-cache-line resolution provided by TeeJam, we demonstrate that the cache attack mitigation applied by WolfSSL AES can be broken:

First, we determine the offset of the T-Tables in the enclave binary. Then we use the results shown in Figure 6 to determine those indices or respectively page offsets which are suitable for an attack. As there are four T-Tables with 1024 bytes each, we search for four offsets, one per table, which show good delays when applying the TeeJam attack. For simplicity, we decide to attack the same index in each table, meaning we filter the results for 4-tuple of offsets which are always 1024 bytes apart. We select one of the tuples with the best average delay induced by TeeJam and configure the attacker code to change the attacked address, based on the progress of the victim enclave. Therefore, the attacker synchronizes with the target enclave by employing a page access side-channel and by using their general knowledge of the algorithm the victim is executing.

The attacker single-steps the enclave and removes the page access bits of the pages holding the T-Table and T-Table lookup routines after every step and waits until these are accessed to recognize the beginning of the AES encryption. Next, it is a simple matter of counting the steps with access to the T-Table and T-Table lookup routines and updating the attacked address in the second attacker thread after each $16 \cdot 4 = 64$ accesses as this is the number of accesses performed in the `GetTable_Multi` function. The attacker constantly writes four bytes to the selected address, matching the size of the T-Table entries, which are also four byte words. If a 4k-conflict occurs, it means that the victim enclave performed a lookup of one of the 16 T-Table entries at the corresponding cache line offset. Even though the cache line is *not* known (just the offset within), the attacker obtains the same information a cache attack would obtain from an unprotected T-Table implementation, since there are 16 possible offsets which can be distinguished with TeeJam.

In the meantime the first attacker thread that single-steps the enclave measures the TeeJam effect as before. The correctness of a trace can be verified by the total number of T-Table accesses determined by the enclave's page accesses. For AES128, these are 2,560 with 10 rounds and 256 lookups per round.

We run the experiment on an Intel Core i5-10210U at base frequency (1.6 GHz) on Ubuntu 22.04 with the WolfSSL master branch from June 08, 2023 that already contains the AES cache line attack mitigation that was published with version 5.6.2 on June 21, 2023.

6.3 AES Last-Round Known-Ciphertext Attack

To obtain the secret key we run a last-round known-ciphertext attack. We observe the encryption of up to 100,000 distinct blocks, where each block has a size of 16 bytes, with the attack described above and store the publicly accessible ciphertext. We then use the recorded data in an offline step to retrieve the last round's round key with a Difference-of-Means based distinguisher [AIES14, GIA+15]:

For each of the 16 last-round table lookups, we iterate over all 256 key byte guesses `g` and calculate the expected result `res = g xor c` of the table lookup based on the recorded ciphertext `c` and key byte guess `g`. As T-Table lookups are bijective, we calculate the index `idx` corresponding to `res` and compare it to all indices which would result in a 4k-conflict with the attacked address (remember that for each looked up index the T-Table is accessed 16 times and the attacker does not know which access is selected). If the key

Listing 5: WolfSSL cache line granular cache line attacker resistant implementation [wol23c] (slightly simplified and reformatted). The listing shows the primitive that executes all four access to one T-Table per AES round (`GetTable_Multi`) and its usage in the last round.

```
#define WC_CACHE_LINE_BITS    4
#define WC_CACHE_LINE_MASK_HI 0xf0
#define WC_CACHE_LINE_MASK_LO 0x0f
#define WC_CACHE_LINE_ADD     0x10

static void GetTable_Multi(const word32* t, word32* t0, byte o0,
    word32* t1, byte o1, word32* t2, byte o2, word32* t3, byte o3)
{
    word32 e0 = 0; word32 e1 = 0; word32 e2 = 0; word32 e3 = 0;

    byte hi0 = o0 & WC_CACHE_LINE_MASK_HI;
    byte lo0 = o0 & WC_CACHE_LINE_MASK_LO;
    ...
    byte hi3 = o3 & WC_CACHE_LINE_MASK_HI;
    byte lo3 = o3 & WC_CACHE_LINE_MASK_LO;

    for (int i = 0; i < 256; i += (1 << WC_CACHE_LINE_BITS)) {
        e0 |= t[lo0 + i] & ((word32)0 - (((word32)hi0 - 0x01) >> 31));
        hi0 -= WC_CACHE_LINE_ADD;

        // Accesses for e1 and e2
        ...

        e3 |= t[lo3 + i] & ((word32)0 - (((word32)hi3 - 0x01) >> 31));
        hi3 -= WC_CACHE_LINE_ADD;
    }

    *t0 = e0; *t1 = e1; *t2 = e2; *t3 = e3;
}

...

word32 u0, u1, u2, u3;

s0 = rk[0]; s1 = rk[1]; s2 = rk[2]; s3 = rk[3];

GetTable_Multi(Te[2], &u0, GETBYTE(t0, 3), &u1, GETBYTE(t1, 3),
                        &u2, GETBYTE(t2, 3), &u3, GETBYTE(t3, 3));
s0 ^= u0 & 0xff000000; s1 ^= u1 & 0xff000000;
s2 ^= u2 & 0xff000000; s3 ^= u3 & 0xff000000;

// Last round access for Te[3] and Te[0]
...

GetTable_Multi(Te[1], &u0, GETBYTE(t3, 0), &u1, GETBYTE(t0, 0),
                        &u2, GETBYTE(t1, 0), &u3, GETBYTE(t2, 0));
s0 ^= u0 & 0x000000ff; s1 ^= u1 & 0x000000ff;
s2 ^= u2 & 0x000000ff; s3 ^= u3 & 0x000000ff;
```

```
for t in t_table_accesses:
  for g in key_byte_guesses:

    st = select_single_step(t, attacked_address)

    for o in observations:
      c = get_cipher_text_byte(t, o)
      res = $g \oplus c$
      idx = reverse_t_table_lookup(t, res)

      st_time = get_single_step_time(o, st)
      if idx in attacked_indexes:
        set_attacked.append(st_time)
      else:
        set_benign.append(st_time)

      differences_in_means.append( average(set_attacked) - average(set_benign))

select_correct_key_guess(differences_in_means)
```

Figure 14: Pseudocode illustrating the recovery of the last round AES key bytes with a Difference-of-Means method on the observed single-stepping times.

guess `g` is correct and the expected index `idx` is in the list of T-Table indices which would result in a 4k-conflict, we should observe a delay in the access time of the single-step corresponding to the T-Table access that conflicts with the attacked address. Since the attacker naturally knows the attacked address and the victim always accesses the T-Table and the T-Table's cache lines in the same sequence, we can easily determine the correct single-steps that have to be observed.

For every T-Table access and key byte guess, the attacker iterates over all observations of encrypted blocks, selects the correct single-step and creates a hypothesis whether the lookup is delayed by the 4k-conflict caused by the attacker. This hypothesis is used to sort the single-stepping time of the selected single-step in one of two sets.

Assuming a uniform distribution, the ratio between the set representing the stepping-times of attacked lookups and the set with benign stepping times is 1 : 16. Finally, the attacker calculates the difference between the average single-stepping times of both sets for every key byte guess and T-Table lookup. If the hypothesis was correct, the difference of the means diverges from 0 and grows while a wrong hypothesis results in a difference of means approach 0 with a growing number of observations. In other words, the correctly recovered round key byte can be identified by a higher difference in the average single-stepping time compared to all other key guesses for the same table lookup. A simplified version of the algorithm is shown in Figure 14.

Figure 15 shows the recovery of four last-round key bytes, one byte for each T-Table. For the full result, please refer to Figure 17 in the Appendix. The experiment run on an Intel Core i5-10210U and the attacked offsets are {0x738, 0xB38, 0xF38, 0x338} corresponding to the index 142 into each T-Table. For more than half of the lookups, the correct key byte guess can already be separated from the remaining guesses after the observation of 10,000 to 20,000 encryptions, which is in line with previous cache attacks on AES [IAES15, MIE17], enabling a direct key reconstruction. The remaining key bytes require 40,000 to 60,000 observations due to either a weaker TeeJam effect or more noise.

MemJam [MWES19] requires 40 to 50 million observations to recover 14 out of 16 bytes in their SGX experiment which always observes the full execution of one encryption.

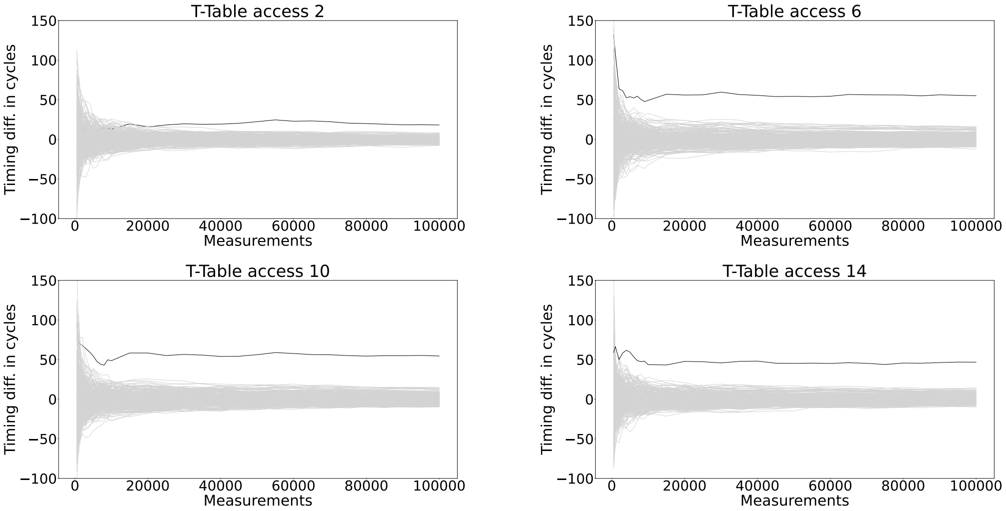


Figure 15: AES128 key recovery results. The last round of AES128 encryption consists of 16 T-Table lookups, 4 from each table. In this graphic the recovery of one key byte for each T-Table is depicted (the access count starts from 0). The attacked offsets were {0x738, 0xB38, 0xF38, 0x338} corresponding to the index 142 into each T-Table. The experiment was executed on an Intel Core i5-10210U@1.6 GHz.

Still about 20 million observations are required to recover half of the key bytes. In their non-SGX experiment they require 2 million observations to recover 15 out 16 key bytes and 200,000 to recover half of the key bytes. Thus with TeeJam we require a factor 1,000 fewer observations in the SGX case due to the high temporal resolution. When comparing our attack against a protected enclave with the attack on an unprotected victim in MemJam, we still require a factor of 10 to 100 fewer repetitions.

6.4 Countermeasures

To mitigate the vulnerable T-Table implementation in software without setting in place any specific hardware requirements, it is necessary to write true constant-time code. A simple but slow variant would access every entry of a T-Table for every lookup and select the correct entry in constant-time. This, however, results in 256 memory loads per lookup. Another option are bitsliced implementations [BP10, KS09] as, e.g., offered in BearSSL [Bea23a, Bea23b] and BoringSSL [Goo23]. These implementations use circuits to define the S-Box computations instead of table lookups. Their performance can be improved by encrypting up to four blocks in parallel [Bea23a].

7 Related Work

Several prior works have also analyzed microarchitectural effects similar to the 4k-aliasing effect we have used in TeeJam.

In CacheBleed [YGH17] cache bank conflicts are exploited on a Sandy Bridge processor to achieve a sub-cache-line resolution and successfully attack RSA decryption. Cache bank conflicts are, however, no longer exploitable on modern microarchitectures.

The authors of MemJam [MWES19] develop a technique to exploit 4k-aliasing in the load after store scenario and obtain a sub-cache-line leakage. In MemJam [MWES19] the authors measure a 10-cycle penalty on load operations delayed by 4k-aliasing stores on the sibling thread. The processor frequency is not specified and the experiment is only executed on one processor with Kaby Lake architecture. We evaluate the 4k-aliasing on multiple CPUs. The delays presented in Section 3 are higher, indicating a stronger MemJam effect; however these differences might be caused by the different microarchitectures and mobile CPUs used in this work. While the authors of MemJam also apply 4k-aliasing to extract keys from symmetric cryptosystems in SGX, they measure the full execution trace and need at least tens of thousands of observations. TeeJam instead shows how to amplify the observed leakage and measures the delay of individual loads, gaining a much higher temporal resolution than MemJam.

In *Microarchitectural Minefields* [SAMJ18] 4k-aliasing is used to build a simultaneous multithreading 4k-aliasing covert channel where a sender fills or flushes the store buffer. The channel is used to achieve multi tenancy detection in the cloud. The work exploits the 4k-aliasing effect on read-after-write with addresses sharing all 12 LSBs and show a statistical 5-cycle delay on the measurement of a 4k-aliasing reading. By increasing the number of loads, they obtain delays up to 15 to 17 cycles. These delays, however, are taken in single threaded measurements on the same hyper-thread and with older microarchitectures and are thus difficult to compare to the results from Section 3. We investigate the preconditions for 4k-aliasing in more detail, discover higher delays and apply TeeJam to trusted execution environments, which allows us to develop a precise, high resolution attack.

Binoculars [ZMFT22] relies on 4k-aliasing to create a false dependency on addresses loaded during a page walk. The sub-cache-line leakage in Binoculars focuses on a victim executing stores, which is the opposite scenario to our work and rarely occurs as a secret-dependent leakage. While Binoculars potentially causes delays of up to 20,000 cycles, their work has a low time resolution as they measure the slow down of page walks to infer the victim's activity. The page walk, however, is inherently slow.

SPOILER [IMB$^+$19] exploits address aliasing as well. However, they focus on speculative load hazards with 1MB (20 bits) aliasing to gain information about the virtual to physical address mapping. The effect is not exploitable for a direct inference attack.

Ragab et al. [RBBG21] shortly study 4k-aliasing in the context of the memory disambiguation on a single thread. The work shows that incorrect memory ordering results in a machine clear and the load buffer re-issues the impacted loads. In our work, we do not observe a machine clear when the load is 4k-aliasing with an older store as the load operation is delayed until the 4k-conflict is resolved. Thus no machine clear is necessary to correct a false state.

Finally, there are other works [SBWE21, MLSS20] which also attack RSA key encodings. However, while `Util::Lookup` [SBWE21] also attacks the key decoding of RSA private keys, they cannot reconstruct keys of 2048- or 4096-bit length. Medusa [MLSS20] focuses on the transient domain and attacking the `rep mov` instruction.

8 Conclusion

In this work, we studied the 4k-aliasing effect and its potential to implement powerful side-channel attacks against TEEs when combined with single-steppinig primitives like SGX-Step. To show the significance of our findings, we focused on Intel SGX and showed that we are able to obtain side-channel leakage with a per instruction, sub-cache-line resolution. The high resolution and information content of the leakage caused by TeeJam enabled us to improve the attack presented in `Util::Lookup` to construct an end-to-end attack that is able to reconstruct RSA keys of at least 4096 bits. To implement the end-to-end attack, we extended the key recovery algorithm of `Util::Lookup` to also handle unaligned partitions and RSA keys with Carmichael's totient function. Moreover, we show that TeeJam can also be used to break symmetric cryptographic implementations protected against classic cache attacks by recovering an AES key from the induced single-stepping delay with a Difference-of-Means distinguisher.

Our results emphasize that the assumption of an attacker model with cache line attack resolution is not sufficient to consider a software secure. Through combination of high temporal and sub-cache-line spatial resolution even tiny leakages, which could previously only be exploited to a limited extend, are fully exploitable. Thus, cryptographic libraries to be used in SGX must ensure to use truly constant-time implementations.

Acknowledgments

We thank the anonymous reviewers for their useful feedback. This work has been supported by the ARC Discovery Project number DP210102670, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and additionally under grants 439797619 and 456967092, as well as by the BMBF through projects ENCOPIA and AnoMed.

References

[Adv18] Advanced Micro Devices. Secure encrypted virtualization API Version 0.16. http://support.amd.com/TechDocs/55766_SEV-KM%20API_Specification.pdf, 2018.

[AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. White Paper, August 2013.

[AIES14] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In Masayuki Abe, editor, *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2007.

[BDF98] Dan Boneh, Glenn Durfee, and Yair Frankel. An attack on RSA given a small fraction of the private key bits. In *ASIACRYPT*, volume 1514 of *Lecture Notes in Computer Science*, pages 25–34. Springer, 1998.

[Bea23a] BearSSL. Bearssl's aes bitsliced documentation. https://bearssl.org/constanttime.html#aes, Accessed: September 2023.

[Bea23b] BearSSL. Bearssl's aes bitsliced implementation. https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/aes_ct.c, Accessed: September 2023.

[BMS+20] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 54–72. IEEE, 2020.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 991–1008. USENIX Association, 2018.

[BP10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2010.

[BPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6. ACM, 2017.

[BPS18] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 178–195. ACM, 2018.

[CCX+19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 142–157. IEEE, 2019.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.

[CGG+19] 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 Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 769–784. ACM, 2019.

[CGYZ22] Chitchanok Chuengsatiansup, Daniel Genkin, Yuval Yarom, and Zhiyuan Zhang. Side-channeling the Kalyna key expansion. In *CT-RSA*, volume 13161 of *Lecture Notes in Computer Science*, pages 272–296. Springer, 2022.

[Cop97] Don Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptol.*, 10(4):233–260, 1997.

[CT23] OpenSSL: Cryptography and SSL/TLS Toolkit. https://github.com/openssl/openssl, Accessed: 2023.

[CVBC+23] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-notify: Thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves. In *USENIX 2023*, page (in print). USENIX Association, 2023.

[CWC⁺18] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 178–194. IEEE Computer Society, 2018.

[DE04] Persi Diaconis and Paul Erdős. On the distribution of the greatest common divisor. *Lecture Notes-Monograph Series*, 45:56–61, 2004.

[DFK⁺13] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 431–446. USENIX Association, 2013.

[GIA⁺15] Berk Gülmezoglu, Mehmet Sinan Inci, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic Flush+Reload attack on AES. In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, volume 9064 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2015.

[Goo23] Google. Boringssl's aes bitsliced implementation. `https://boringssl.googlesource.com/boringssl/+/master/crypto/fipsmodule/aes/aes_nohw.c`, Accessed: September 2023.

[HLP⁺13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ISCA*, page 11. ACM, 2013.

[HMM10] Wilko Henecka, Alexander May, and Alexander Meurer. Correcting errors in RSA private keys. In *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 351–369. Springer, 2010.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[IAES15] Gorka Irazoqui Apecechea, 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 *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 591–604. IEEE Computer Society, 2015.

[IMB⁺19] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: speculative load hazards boost Rowhammer and cache attacks. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 621–637. USENIX Association, 2019.

[Int15] Intel. Intel 64 and IA-32 architectures optimization reference manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf, Accessed: 2022-11-15.

[Int22] Intel. Intel architecture instruction set extensions and future features - programming reference. https://cdrdv2-public.intel.com/671368/architecture-instruction-set-extensions-programming-reference.pdf, December 2022.

[Int23a] Intel. Intel software guard extensions (Intel SGX) data center attestation primitives: ECDSA quote library API. https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/Intel_SGX_ECDSA_QuoteLibReference_DCAP_API.pdf, March 2023.

[Int23b] Intel. Asynchronous enclave exit notify and the EDECCSSA user leaf function - white paper. https://cdrdv2-public.intel.com/736463/aex-notify-white-paper-public.pdf, Accessed: 2023.

[Int23c] Intel. Intel SGX attestation technical details. https://www.intel.com/content/www/us/en/security-center/technical-details/sgx-attestation-technical-details.html, Accessed: 2023.

[Int23d] Intel. L1 terminal fault / CVE-2018-3615, CVE-2018-3620, CVE-2018-3646 / INTEL-SA-00161. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1-terminal-fault.html, Accessed: 2023.

[Int23e] Intel. Intel 64 and IA-32 architectures software developer's manual. https://cdrdv2.intel.com/v1/dl/getContent/671200, June 2023.

[ITU23a] ITU. X.680: ASN.1 specification. https://www.itu.int/rec/T-REC-X.680/en, Accessed: 2023.

[ITU23b] ITU. X.690:ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER). https://www.itu.int/rec/T-REC-X.690, Accessed: 2023.

[JFB+22] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "They're not that hard to mitigate"': What cryptographic library developers think about timing attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 632–649. IEEE, 2022.

[KHF+19] 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 *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019.

[KKY11] Jun Kogure, Noboru Kunihiro, and Hirosuke Yamamoto. Generalized security analysis of the random key bits leakage attack. In *WISA*, volume 7115 of *Lecture Notes in Computer Science*, pages 13–27. Springer, 2011.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. Technical report, Advanced Micro Devices, 2016.

[KR13] Cameron F Kerry and Charles Romine. FIPS PUB 186-4 federal information processing standards publication digital signature standard (dss), 2013-07-19 2013.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[LSG$^+$18] 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 William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018.

[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, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 605–622. IEEE Computer Society, 2015.

[MAB$^+$13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ISCA*, page 10. ACM, 2013.

[MBH$^+$20] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled instruction-level attacks on enclaves. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 469–486. USENIX Association, 2020.

[MCK$^+$16] K. Moriarty, EMC Corporation, B. Kaliski, Verisign, J. Jonsson, Subset AB, A. Rusch, and RSA. PKCS #1: RSA cryptography specifications version 2.2. https://www.rfc-editor.org/rfc/rfc8017, November 2016.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 69–90. Springer, 2017.

[MLSS20] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1427–1444. USENIX Association, 2020.

[MvOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[MWES19] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations. *Int. J. Parallel Program.*, 47(4):538–570, 2019.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[OTK+18] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In Haryadi S. Gunawi and Benjamin C. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 227–240. USENIX Association, 2018.

[PPS12] Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 386–403. Springer, 2012.

[RBBG21] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1451–1468. USENIX Association, 2021.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[SAMJ18] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural minefields: 4K-aliasing covert channel and multi-tenant detection in IaaS clouds. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society, 2018.

[SBWE21] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util::Lookup: Exploiting key decoding in cryptographic libraries. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2456–2473. ACM, 2021.

[SJBZ18] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for Intel SGX with Intel data center attestation primitives, 2018.

[SSL+22] Fan Sang, Ming-Wei Shih, Sangho Lee, Xiaokuan Zhang, Michael Steiner, Mona Vij, and Taesoo Kim. PRIDWEN: universally hardening SGX programs via load-time synthesis. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 455–472. USENIX Association, 2022.

[vSMÖ+19] 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 *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 88–105. IEEE, 2019.

[WMES18] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 161–173. ACM, 2018.

[wol23a] wolfSSL: Embedded TLS Library. https://github.com/wolfSSL/wolfssl, Accessed: 2023.

[wol23b] wolfSSL: Embedded TLS Library. Static library: Building libwolfssl.sgx.static.lib.a for use with SGX enclaves. https://github.com/wolfSSL/wolfssl/tree/master/IDE/LINUX-SGX, Accessed: 2023.

[wol23c] wolfSSL: Embedded TLS Library. Wolfssl AES source code. https://github.com/wolfSSL/wolfssl/blob/master/wolfcrypt/src/aes.c, Accessed: 2023.

[wol23d] wolfSSL: Embedded TLS Library. wolfssl release 5.6.2 (jun 09, 2023). https://github.com/wolfSSL/wolfssl/releases/tag/v5.6.2-stable, Accessed: 2023.

[WPS⁺23] Jan Wichelmann, Christopher Peredy, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. MAMBO-V: dynamic side-channel leakage analysis on RISC-V. In Daniel Gruss, Federico Maggi, Mathias Fischer, and Michele Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12-14, 2023, Proceedings*, volume 13959 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2023.

[WSPE22] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. Microwalk-CI: Practical side-channel analysis for JavaScript applications. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2915–2929. ACM, 2022.

[WWL⁺17] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 235–252. USENIX Association, 2017.

[WZS⁺18] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - differential address trace analysis: Finding address-based side-channels in binaries. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 603–620. USENIX Association, 2018.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015.

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptogr. Eng.*, 7(2):99–112, 2017.

[ZMFT22] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Binoculars: Contention-based side-channel attacks exploiting the page walker. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 699–716. USENIX Association, 2022.

A Experiments on the Intel Xeon E-2286M (Coffee Lake)

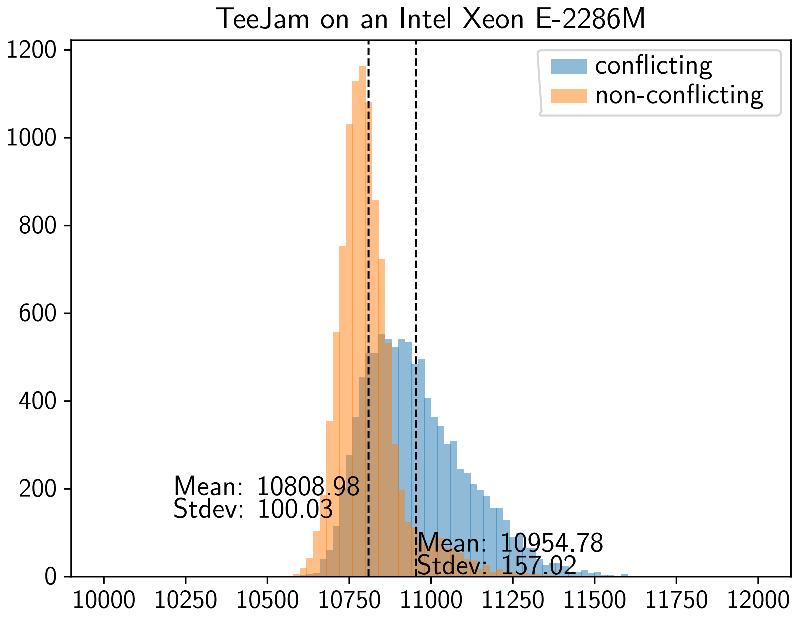


Figure 16: TeeJam measurements with 10,000 measurements of the single-stepping time for each the conflicting and non-conflicting loads.

B AES Last Round Key Recovery

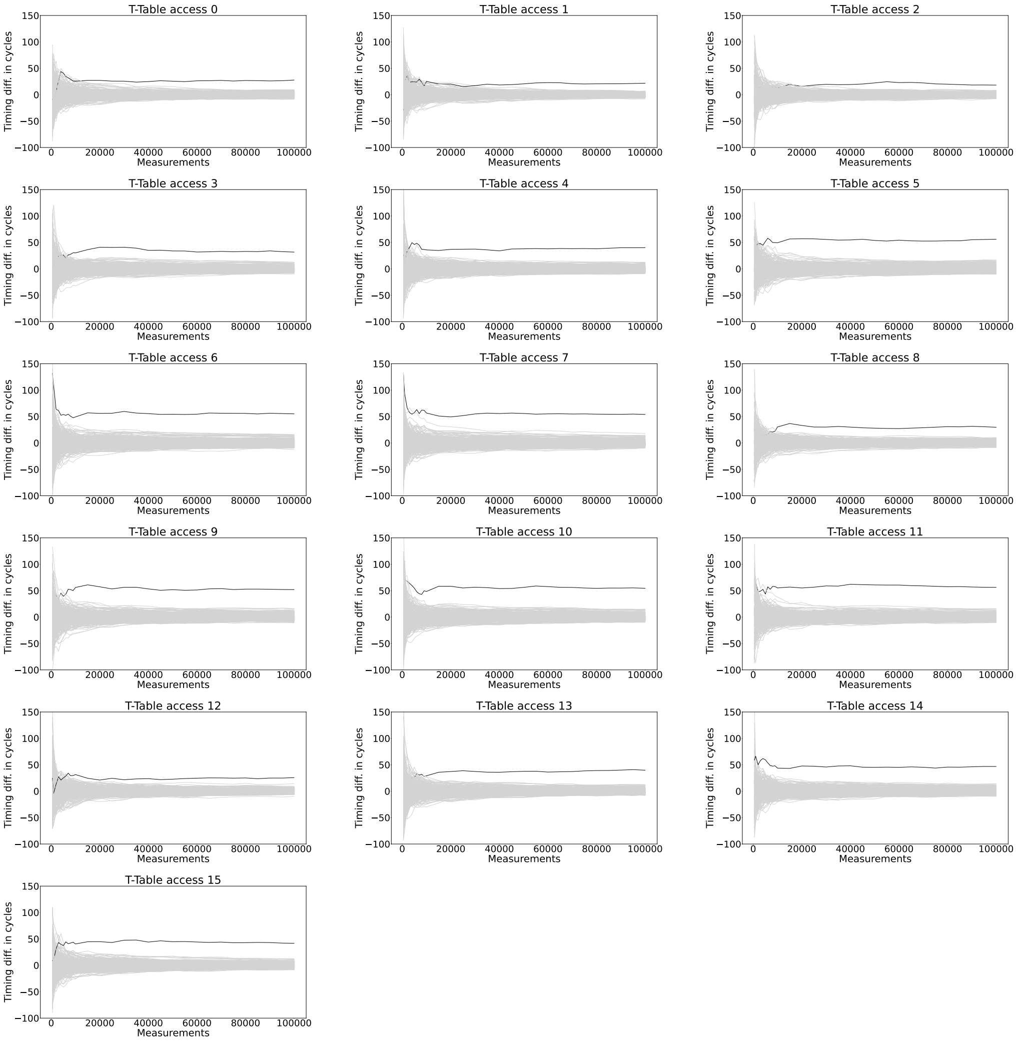


Figure 17: The last round of AES128 encryption consists of 16 T-Table lookups, four from each table. This graphic shows the recovery results for all last-round key bytes from Section 6.3. The attacked offsets were {0x738, 0xB38, 0xF38, 0x338} corresponding to the index 142 into each T-Table. The experiment was executed on an Intel Core i5-10210U@1.6 GHz.

C Details About the Key Recovery Algorithm

C.1 Initial Candidates

In the following, we describe how to find the first initial candidates for the secret key. In order to do so, we will first determine the integers k, k_p, and k_q.

Finding $\boldsymbol{k}$: To find the integer k used in the relation $\gamma \cdot e \cdot d = k \cdot (N - p - q + 1) + \gamma$, we follow the approach of [HS09], first described by Boneh, Durfee and Frankel [BDF98] to show that k has small size: First, it is easy to see that $d < \varphi(N)$. If $k > \gamma \cdot e$, this means that $k \cdot \varphi(N) + \gamma > k \cdot \varphi(N) > \gamma \cdot e \cdot d$, which is a contradiction to the relation. Hence, we have $k \leq \gamma \cdot e$. We can thus enumerate all possibilities of k, as $e = 2^{16} + 1$ is by far the most used choice for the public exponent.

Now, we need to determine whether our choice of k is correct. To do so, we define

$$\hat{d}(k') = \left\lfloor \frac{k' \cdot (N+1) + \gamma}{\gamma \cdot e} \right\rfloor .$$

For the correct choice of k, we have

$$\begin{aligned}
\hat{d}(k) - d &\leq \frac{k \cdot (N+1) + \gamma}{\gamma \cdot e} - \frac{k \cdot \varphi(N) + \gamma}{\gamma \cdot e} \\
&= \frac{k \cdot (N + 1 - \varphi(N))}{\gamma \cdot e} \\
&= \frac{k \cdot (p+q)}{\gamma \cdot e} \\
&\leq p + q,
\end{aligned}$$

where the last inequality follows from the fact that $k \leq \gamma \cdot e$. Hence, $\hat{d}(k)$ and d agree on half of their most significant bits, if $p + q \leq 3\sqrt{N}$. Comparing $\hat{d}(k)$ to our observation of d thus allows us determine k, as only one choice of k will agree with our observation of d on this many bits.

Finding $\boldsymbol{k_p}$ and $\boldsymbol{k_q}$: Knowledge of k allows us also to deduce k_p and k_q. We reduce the relations modulo e and obtain

$$\begin{aligned}
N &\equiv p \cdot q \pmod{e} && (5)\\
0 &\equiv (k \cdot \varphi(N) + \gamma) \pmod{e} && (6)\\
0 &\equiv (k_p \cdot (p-1) + 1) \pmod{e} && (7)\\
0 &\equiv (k_q \cdot (q-1) + 1) \pmod{e} && (8)
\end{aligned}$$

Using the relations $0 \equiv (k_p \cdot (p-1) + 1) \pmod{e}$, we know $(p-1) \equiv -1/k_p \pmod{e}$ and $(q-1) \equiv -1/k_q \pmod{e}$ and can conclude that

$$0 \equiv (k \cdot \varphi(N) + \gamma) \equiv (k \cdot (p-1)(q-1) + \gamma) \equiv (k \cdot (-1/k_p)(-1/k_q) + \gamma) \pmod{e}.$$

This is equivalent to $-\gamma \cdot k_p \cdot k_q \equiv k \pmod{e}$. Replacing k with this term thus gives

$$\begin{aligned}
0 &\equiv (k \cdot \varphi(N) + \gamma) \\
&\equiv (k \cdot (N-1) - k \cdot (p - 1 + q - 1) + \gamma) \\
&\equiv (k \cdot (N-1) + (\gamma \cdot k_p \cdot k_q)(-1/k_p - 1/k_q) + \gamma) \\
&\equiv (k \cdot (N-1) - \gamma(k_p + k_q) + \gamma) \pmod{e}.
\end{aligned}$$

Rewriting $k_q = -k/(\gamma k_p)$ gives

$$0 \equiv (k \cdot (N-1) - \gamma k_p + (k/k_p) + \gamma) \pmod{e}$$

and multiplication with k_p and rewriting thus results in

$$(\gamma k_p^2 - k_p(k(N-1)+\gamma) - k) \equiv 0 \bmod e.$$

If e is prime, this quadratic congruence has two efficiently computable solutions, namely k_p and k_q. We thus just need to try these two possibilities for the values of k_p and k_q.

C.2 Expanding the Key

We consider the following variant of Hensel's Lemma described in [HS09] and [KKY11]:

Lemma 1. *Let $f(x_1, x_2, \ldots, x_n) \in \mathbb{Z}[x_1, x_2, \ldots, x_n]$ be a multivariate polynomial with integer coefficients and π be a positive integer. Let $\mathbf{r} = (r_1, \ldots, r_n)$ be such that $f(\mathbf{r}) \equiv 0 \pmod{\pi^i}$ for some i. Then $f(\mathbf{r}+\mathbf{b}) \equiv 0 \pmod{\pi^{i+1}}$ for $\mathbf{b} = (b_1\pi^i, b_2\pi^2, \ldots, b_n\pi^i)$ with $0 \le b_j \le \pi - 1$ if*

$$f(\mathbf{r}) + \sum_{j=1}^{n} b_j \pi^i f_{x_j}(\mathbf{r}) \equiv 0 \pmod{\pi^{i+1}}.$$

Here, f_{x_j} is the partial derivative of f with respect to x_j.

Using this lemma, Heninger and Shacham were able to derive the following conditions for keys using Euler's totient function that need to be fulfilled:

$$p[i] + q[i] \equiv (N - p'q')\,[i] \pmod{2} \tag{9}$$

$$\begin{aligned} & p[i] + q[i] + d[i+\tau(k)] \equiv \\ & \quad (k \cdot (N+1) + 1 - k \cdot (p'+q') - e \cdot d')\,[i+\tau(k)] \pmod{2} \end{aligned} \tag{10}$$

$$\begin{aligned} & p[i] + d_p[i+\tau(k_p)] \equiv \\ & \quad \left(k_p(p'-1) + 1 - e \cdot d'_p\right)[i+\tau(k_p)] \pmod{2} \end{aligned} \tag{11}$$

$$\begin{aligned} & q[i] + d_q[i+\tau(k_q)] \equiv \\ & \quad \left(k_q(q'-1) + 1 - e \cdot d'_q\right)[i+\tau(k_q)] \pmod{2}. \end{aligned} \tag{12}$$

In order to adapt to the Carmichael function, we introduced the variable γ and thus also need to adapt these equations. Equations (9), (11), and (12) can be used without much modification (besides a small shift in the indices of $p[i]$, $q[i]$, and $d[i+\tau(k)]$). But Equation (10) needs to be derived from scratch by making use of Lemma 1. We thus need to re-derive the corresponding condition for the relation $\gamma \cdot e \cdot d = k\varphi(N) + \gamma$. We define a polynomial $f(p,q,d) = k \cdot (N+1) + \gamma - k \cdot (p+q) - \gamma \cdot e \cdot d$. Now, due to our assumptions, we have $f(p', q', d') \bmod 2^{i+\tau(\gamma)+\tau(k)} = 0$. We now want to extend these suffixes by some bits b_1, b_2, and b_3 such that

$$f\left(p' + b_1 2^{i+\tau(\gamma)}, q' + b_2 2^{i+\tau(\gamma)}, d' + b_3 2^{i+\tau(k)}\right) \equiv 0 \pmod{2^{i+\tau(\gamma)+\tau(k)+1}}.$$

Computing the partial derivatives gives $f_p(p,q,d) = f_q(p,q,d) = -k$ and $f_d(p,q,d) = -\gamma \cdot e$ and Lemma 1 thus implies the following condition

$$f(p', q', d') - (b_1 + b_2) \cdot k \cdot 2^{i+\tau(\gamma)} - b_3 \cdot \gamma \cdot e \cdot 2^{i+\tau(k)} \equiv 0 \pmod{2^{i+\tau(\gamma)+\tau(k)+1}}.$$

Now, we know that $k = 2^{\tau(k)} \cdot k'$ for some odd integer k' and $\gamma = 2^{\tau(\gamma)} \cdot \gamma'$ for some odd integer γ'. Hence, the condition can be written as

$$f(p', q', d') - (b_1 + b_2)k'2^{i+\tau(\gamma)+\tau(k)} - b_3\gamma' e2^{i+\tau(\gamma)+\tau(k)} \equiv 0 \pmod{2^{i+\tau(\gamma)+\tau(k)+1}}.$$

As k', γ', and e are odd integers, reducing everything modulo $2^{i+\tau(\gamma)+\tau(k)}$ gives the condition

$$f(p', q', d')[i + \tau(\gamma) + \tau(k)] - b_1 - b_2 - b_3 \equiv 0 \pmod 2$$

or, in simplified form

$$\begin{aligned}&p[i + \tau(\gamma)] + q[i + \tau(\gamma)] + d[i + \tau(k)]\\ &\quad\equiv (k(N+1) + \gamma - k(p' + q') - \gamma ed')\,[i + \tau(\gamma) + \tau(k)] \pmod 2.\end{aligned}$$

Summing up the above discussion, we can extend our candidate $\widetilde{sk}$ by the bits $p[i+\tau(\gamma)]$, $q[i + \tau(\gamma)]$, $d[i + \tau(\gamma) + \tau(k)]$, $d_p[i + \tau(k_p) + \tau(\gamma)]$, and $d_q[i + \tau(k_q) + \tau(\gamma)]$ if the following conditions hold:

$$\begin{aligned}&p[i + \tau(\gamma)] + q[i + \tau(\gamma)] \equiv (N - p'q')\,[i + \tau(\gamma)] \pmod 2 && (13)\\ &(p[i + \tau(\gamma)] + q[i + \tau(\gamma)] + d[i + \tau(k)])\\ &\quad\equiv (k(N+1) + \gamma - k(p' + q') - \gamma ed')\,[i + \tau(\gamma) + \tau(k)] \pmod 2 && (14)\\ &p[i + \tau(\gamma)] + d_p[i + \tau(k_p) + \tau(\gamma)]\\ &\quad\equiv \left(k_p(p' - 1) + 1 - e \cdot d'_p\right)[i + \tau(k_p) + \tau(\gamma)] \pmod 2 && (15)\\ &q[i + \tau(\gamma)] + d_q[i + \tau(k_q) + \tau(\gamma)]\\ &\quad\equiv \left(k_q(q' - 1) + 1 - e \cdot d'_q\right)[i + \tau(k_q) + \tau(\gamma)] \pmod 2. && (16)\end{aligned}$$