SQUIP: Exploiting the Scheduler Queue Contention Side Channel

Stefan Gast[1,2], Jonas Juffinger[1,2], Martin Schwarzl[2], Gururaj Saileshwar[3],
Andreas Kogler[2], Simone Franza[2], Markus Köstl[2], Daniel Gruss[2]
[1] *Lamarr Security Research,* [2] *Graz University of Technology,* [3] *Georgia Institute of Technology*

***Abstract*—Modern superscalar CPUs have multiple execution units that independently execute operations from the instruction stream. Previous work has shown that numerous side channels exist around these out-of-order execution pipelines, particularly for an attacker running on an SMT core.**

In this paper, we present the SQUIP attack, the first side-channel attack on scheduler queues, which are critical for deciding the schedule of instructions to be executed in superscalar CPUs. Scheduler queues have not been explored as a side channel so far, as Intel CPUs only have a single scheduler queue, and contention thereof would be virtually the same as contention of the reorder buffer. However, the Apple M1, AMD Zen 2, and Zen 3 microarchitectures have separate scheduler queues per execution unit. We first reverse-engineer the behavior of the scheduler queues on these CPUs and show that they can be primed and probed. The SQUIP attack observes the occupancy level from within the same hardware core and across SMT threads. We evaluate the performance of the SQUIP attack in a covert channel, exfiltrating 0.89 Mbit/s from a co-located virtual machine at an error rate below 0.8 %, and 2.70 Mbit/s from a co-located process at an error rate below 0.8 %. We then demonstrate the side channel on an mbedTLS RSA signature process in a co-located process and in a co-located virtual machine. Our attack recovers full RSA-4096 keys with only 50 500 traces and less than 5 to 18 bit errors on average. Finally, we discuss mitigations necessary, especially for Zen 2 and Zen 3 systems, to prevent our attacks.

1. Introduction

Modern CPUs are highly optimized for performance and efficiency. One driving factor are out-of-order execution pipelines. CPUs have multiple execution units that operate independently. Instructions are split up into micro-operations (µops) and put into the reorder buffer. This allows executing µops out of order, as soon as their dependencies are met, by scheduling them for the corresponding execution unit.

Previous works have exploited these pipeline optimizations in various ways. Spectre [32] and Meltdown [38] exploit that the CPU runs operations out of order that should not be executed. These operations then leak information while they are executed transiently. Other attacks focus more on side channels in the pipeline rather than its misbehavior. For instance, PortSmash [5] and SMoTherSpectre [18] exploit that an execution unit can only execute one µop simultaneously. Thus, running the same µop simultaneously on two SMT threads will lead to a slight timing variation in one of the two SMT threads until the execution unit is available again. Also, the contention of the reorder buffer has been considered for side-channel attacks [4].

Most of these works have focused on Intel CPUs, where server (Xeon) and client (Core) CPUs have a similar design with a single µop scheduler for all execution units [28]. Consequently, the scheduler and scheduler queues on Intel CPUs are not a high value target for attacks as equivalent information can be obtained by the already known attacks, e.g., on the reorder buffer [4]. In particular, the single-scheduler design inherently does not leak information on the type of instruction as it is shared for all execution units. A fundamentally different design choice is to have per-execution-unit scheduler queues. During our reverse-engineering we discovered that the Apple M1 [30], the AMD Zen 2 [9], and the AMD Zen 3 [10] follow such a per-execution-unit scheduler design. Hence, we ask the following research question in this paper:

Do per-execution-unit schedulers leak more information than a single scheduler? Can scheduler queues be utilized in contention-based attacks to leak secrets such as cryptographic keys from co-located workloads?

In this paper, we present the SQUIP attack, the first side-channel attack on scheduler queues. With SQUIP, we measure the precise degree of **S**cheduler **Q**ueue **U**sage (*i.e.*, occupancy) via **I**nterference **P**robing. We show that this occupancy level measurement works on microarchitectures of different vendors, namely the Apple M1, AMD Zen 2 and Zen 3. The SQUIP measurement method is able to measure the precise occupancy level with a high temporal resolution as it uses only low-latency operations, and observes contention (*i.e.*, exceeding the capacity) via a pipeline stall. We show that these stalls are observable using performance counters or non-serialized timer reads. We show that SQUIP is particularly dangerous when untrusted workloads are co-located on SMT threads, a setup still considered secure on AMD CPUs and not excluded for future Apple CPUs.

We develop and evaluate the SQUIP attack in a series of experiments. We first present the *SQUIP measurement method*, which precisely measures scheduler contention on the Apple M1, the AMD Zen 2 and the AMD Zen 3 microarchitectures. We determine precise thresholds directly related to the corresponding scheduler queue sizes. We evaluate the

performance and robustness of SQUIP in a *covert channel scenario* on SMT-enabled AMD Zen 2 and Zen 3 machines. In a native cross-process scenario, the SQUIP covert channel achieves a bandwidth of 2.70 Mbit/s at an error rate of less than 1 %. Across virtual machines (VMs), we achieve a bandwidth of 0.89 Mbit/s at an error rate of less than 0.8 %.

We then demonstrate that the *SQUIP attack* can exploit the per-execution-unit scheduler queues to leak more information about the instructions executed compared to single-scheduler-queue or reorder-buffer contention. Specifically, we show that we can leak whether specific instructions, grouped by their scheduler, have been executed and even measure the precise degree of contention on this scheduler. On a Zen 3 machine, we demonstrate the significance of this side channel in an attack on an mbedTLS RSA signature service co-located on the other SMT thread of a physical core. In a native cross-process scenario, the SQUIP side channel leaks full RSA-4096 keys with only 50 500 traces and less than 5 (n = 10, $\sigma_{\bar{x}}$ = 1.28) bit errors. Across VMs, we leak full RSA-4096 keys with only 50 500 traces and less than 18 (n = 10, $\sigma_{\bar{x}}$ = 3.26) bit errors.

We discuss how this side channel affects current and future Apple and AMD CPUs, as well as mitigations necessary to prevent our attacks. We propose operating-system-level mitigations to prevent exploitation of the SQUIP side channel across security domains or tenants. We also discuss what CPU designs are unaffected and how future CPU designs can take these design choices into account.

To summarize, we make the following contributions:

1) We present the SQUIP side channel, the first side-channel attack on scheduler queues in CPUs.
2) We evaluate our attack in native and cross-VM covert channels, with bandwidths of 2.70 Mbit/s (native) and 0.89 Mbit/s (cross-VM), at error rates of less than 1 %.
3) We show that the SQUIP side-channel leakage is precise enough to recover full RSA-4096 keys with only 50 500 traces, with on average less than 5 (n = 10, $\sigma_{\bar{x}}$ = 1.28) bit errors across native processes and less than 18 (n = 10, $\sigma_{\bar{x}}$ = 3.26) bit errors across virtual machines.
4) We present potential mitigations against SMT side channels on AMD CPUs.

Outline. Section 2 provides background on pipelines, schedulers, and side channels. Section 3 presents the SQUIP building blocks. Section 4 evaluates the performance in covert channels. Section 5 evaluates SQUIP on mbedTLS RSA. Section 6 discusses limitations and mitigations and Section 7 related work. Section 8 concludes.

Responsible Disclosure. We reported leakage through the SQUIP side channel to AMD on December 15th, 2021. AMD acknowledged our findings and reserved CVE-2021-46778 to this issue. We also reported the issue to Apple and Arm on April 22nd, 2022 and to Intel on May 11th, 2022.

2. Background

In this section, we provide background about the CPU pipeline, execution unit schedulers, and simultaneous multithreading.

2.1. CPU Pipelines

Modern CPUs use a *superscalar* design, where multiple instructions are executed simultaneously to maximize performance. Such CPUs process instructions in a pipeline across several stages [8]: (1) fetch, (2) decode, (3) schedule/execute and (4) retire. The schedule/execute stage can process instructions *out-of-order* to maximize the instruction level parallelism. We briefly describe each of these stages:

Fetch. The CPU fetches the next instruction to be executed from the L1i cache. As branches can make the address of the next instruction to be fetched unknown, the next address is often predicted using the branch prediction unit.

Decode. To enable efficient execution, fetched instructions (macro ops) are decoded into one or multiple simpler micro-ops (µops) and placed into a µop queue. These µops are fed into the backend, where they are scheduled and executed.

Schedule / Execute. The scheduler(s) tracks which µops are ready for execution (have inputs available) and dynamically schedules them (in an out-of-order manner) to available execution units. A CPU core has multiple execution units and can have multiple arithmetic and logic units (ALUs), branch execution units (BRUs), address generation units (AGUs). Figure 1 shows the connection between the schedulers and execution units on Zen 2 and Zen 3. Once a µop has been executed, and its output is written to a register and forwarded to dependent µops, it is removed from the scheduler queue.

Retire. The retire queue hides the out-of-order execution by ensuring executed µops are always retired in order. For instance, a µop is retired only once all prior µops in program order have been executed and retired.

2.2. CPU Scheduler Microarchitecture

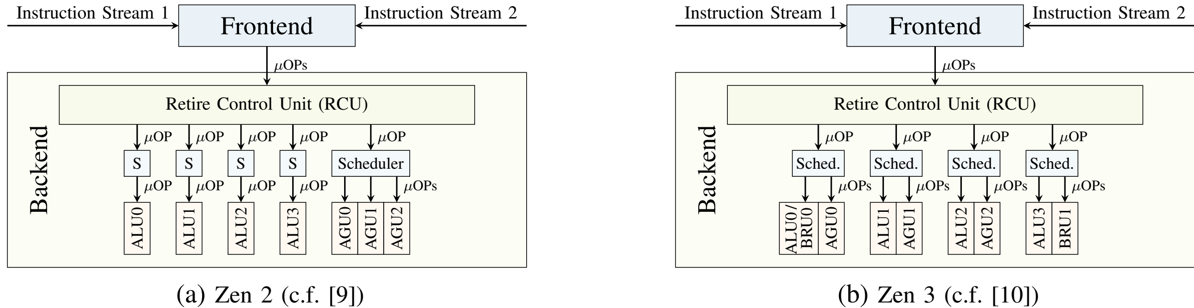
The scheduler design is critical for maximizing the µop throughput. A CPU can have a single scheduler, like Intel CPUs [28] or multiple schedulers, like AMD [9], [10] and Apple CPUs [30]. With an increasing number of execution units, a single monolithic scheduler can become complex and power intensive, necessitating multiple schedulers, one per subset of execution units (or type of µop) [56], [66].

AMD Zen 2 has separate schedulers (ALQ0 to ALQ3) for each ALU and a separate scheduler for all AGUs (AGQ) [9]. AMD Zen 3 has separate schedulers for each pair of ALU and AGU/BRU [10]. Apple M1 CPU is also suggested to have distributed schedulers [30].

Each of these schedulers maintains separate queues from where the µops are issued for the corresponding execution units. On Zen 2 and Zen3, scheduler queues have 16 [9] and 24 [10] entries, respectively. If any one of these scheduler queues (for any one type of µop) becomes full, this causes a back-end stall, delaying the execution of subsequent µops from the front-end.

2.3. Simultaneous Multithreading (SMT)

Even good scheduling and a large retire queue cannot keep all execution units utilized all the time from a single instruction stream. With simultaneous multithreading, a



(a) Zen 2 (c.f. [9]) (b) Zen 3 (c.f. [10])

Figure 1: Simplified core block diagrams for AMD Zen 2 and Zen 3.

CPU core is split into multiple logical cores or hardware threads, executing independent instruction streams but sharing resources such as the L1i cache. µops from these threads share the execution units dynamically to enable higher total utilization. The partitioning of different parts of the core is done by competitively sharing, watermarking, or static partitioning. The AMD Zen architectures allow two threads per core. These threads can be from a single program or from different programs, as managed by the OS.

2.4. Prior Microarchitectural Side Channels

Port Contention. If a scheduler is connected to multiple execution units, this connection is established through so-called ports. Each port can forward one µop per cycle to an execution unit. If two programs running on two logical cores of a physical core execute instructions requiring the same port, the instructions of only one program can be issued in any given cycle. Port contention causes delays for instructions of the other program, which can be detected through a timing side-channel. Prior works [5], [18] exploit port contention on Intel CPUs to steal cryptographic secrets by distinguishing timing differences of a few cycles.

Cache Contention. CPU caches such as L1 caches are shared between multiple logical cores of a physical core, while last-level caches (LLCs) are shared by many physical cores. Contention for limited space within a cache *set* can cause accesses by one program (or thread or VM) to evict the address of another, causing an observable timing side channel. The classic *Prime+Probe* attack, exploiting such cache contention, has been shown to leak cryptographic secrets through contention on L1 caches [48], LLC [39], coherence directories [71], and others. Cache contention introduces much higher timing differences (tens to hundreds of cycles) that are more easily observable than port contention. However, cache contention only leaks memory (or cache) accesses of a victim program, whereas port contention observes a larger set of instructions.

Transient-Execution Attacks. Attacks like Spectre [32] and Meltdown [38] exploit the fact that the CPU can transiently execute operations that should not be executed. These operations can be used to leak information during transient execution through any microarchitectural side channel, such as cache side channels or port contention [18].

3. Observing Contention on Scheduler Queues

In this section, we demonstrate how the behavior of scheduler queues introduces interferences across workloads, leveraging simple experiments and reverse engineering. We provide methods to observe scheduler queue contention, first, via performance counters, second, via unserialized timer reads, and third, across sibling threads on the same core. The SQUIP side-channel is based on the contention of a queue (similar to a buffer or cache set) and follows the attack principle of a Prime+Probe-style attack.

3.1. Single-Threaded Scheduler Queue Contention

In our first two experiments, we show that it is possible to deliberately induce and observe scheduler queue contention with multiplications on a single hardware thread.

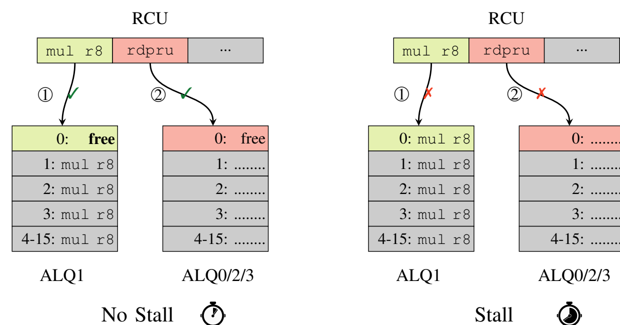
3.1.1. Observation Using Performance Counters. We start by observing contention using performance counters on an AMD Zen 3 Ryzen 7 5800X CPU. Each integer execution unit (ALU) is associated with a separate scheduler queue (see Section 2.2), and, importantly, these ALU execution units can have different capabilities. For example, on AMD Zen, Zen 2, and Zen 3 all four ALUs can perform additions and subtractions. However, multiplications, divisions, and CRC operations can only be executed by one specific ALU. In particular, multiplications can only be executed by ALU1 [8], [9], [10] and, therefore, always use the same scheduler queue.

To control scheduler queue contention, we first reverse-engineer how the scheduler queue can be primed, *i.e.*, how we can fill it to its full capacity. Listing 1 shows the code sequence used in this experiment. Note that the highlighted lines are commented out. We will enable and explain them in Section 3.1.2, yet for this first experiment, we want to ensure that they do not interfere with our measurements.

We use a short loop to drain the backend here (lines 4 to 8), so that any previous outstanding operations can retire before we start filling the queue. Later, for the actual attacks, we replace this loop with a more efficient way to drain the queue. As the priming step, we fill the scheduler queue associated with ALU1, using a dependency chain [65], which consists of:

```
1  # Set ecx = 1 for rdpru (APERF)
2  # movl $1, %ecx
3
4  # 1: Drain the scheduler queue
5  movl $10000, %eax
6  loop:
7  sub $1, %eax
8  jnz loop
9
10 # 2: Delay subsequent multiplications
11 movq $12345678, %r15
12 cvtsi2sd %r15, %xmm0
13 sqrtsd %xmm0, %xmm0
14 sqrtsd %xmm0, %xmm0
15 sqrtsd %xmm0, %xmm0
16 cvtsd2si %xmm0, %r15
17
18 # rdpru
19 # movl %eax, %ebx
20
21 # 3: Fill queue with n multiplications
22 imulq $3, %r15
23 imulq $3, %r15
24 imulq $3, %r15
25 imulq $3, %r15
26 imulq $3, %r15
27 # ... (contention if n > capacity(queue))
28
29 # rdpru
30 # subl %ebx, %eax
```

Listing 1: Causing scheduler queue contention with a dependency chain.



(a) No contention: ALQ1 has (at least) one free entry. The `mul` operation is enqueued (1) and subsequently the `rdpru` operation can also be enqueued (2).

(b) Contention: ALQ1 is already full. Thus, `mul` cannot be enqueued (1) before there is a free entry. As operations are enqueued in-order, `rdpru` (2) is delayed too.

Figure 2: Measuring scheduler queue contention via non-serialized `rdpru` instructions.

1) A block of dependent, long-latency instructions that are not occupying the targeted scheduler and produce an integer result (lines 10 to 16).
2) A block of `imul` instructions, each depending on the result of the previous instruction (lines 21 to 27).

This ensures that no single `imul` can execute before the preceding long-latency block is finished. Thus, the chosen number of multiplications stays in the scheduler queue for a prolonged period of time. By adjusting the number of multiplications, we can cause different occupancy levels of the scheduler queue. If we exceed the capacity of the queue by trying to insert one more (dependent or independent) operation into a full queue, the *pipeline stalls* until there is at least one slot free again.

We want to emphasize that reordering µops is the task of the schedulers (see Section 2.1). They keep track of the dependencies and schedule the µops to the associated execution units, possibly out of program order. In the RCU, µops are still in program order and, thus, also added to the scheduler queues in-order. Therefore, if a µop cannot be enqueued because its designated queue is full, it blocks all subsequent µops from being enqueued as well, stalling the pipeline, see Figure 2. As the RCU cannot reorder µops, it has to wait until the oldest µop can finally be enqueued.

We can maintain this scheduler queue contention situation until the long-latency instructions and, consequently, the first multiplication of the `imul` block finish. This yields a far longer time window to observe contention than other SMT attacks, e.g., port contention [5].

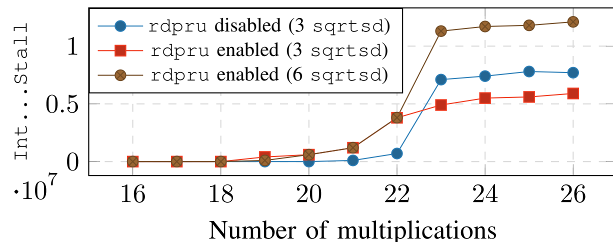


Figure 3: The number of CPU cycles reported by `IntSch1TokenStall` with scheduler queue contention when executing 100 000 iterations of the priming code on Zen 3 with varying lengths of the multiplication block (sibling thread idle).

To verify this, we use the `IntSch1TokenStall` performance counter, introduced with AMD Zen 3 [7], [11]. It counts the number of cycles with scheduler contention [11], *i.e.*, it increments for each cycle where µops could not be dispatched because there were no free entries in the scheduler queue of ALU1. We run 100 000 iterations of Listing 1 with varying numbers of multiplications while monitoring `IntSch1TokenStall`. Between 22 and 23 multiplications, we observe a substantial increase of the performance counter, see the blue curve in Figure 3 (the red and brown curves are explained in Appendix A). This is very close to the 24 ALU1 entries of the Zen 3, indicating that we can intentionally induce scheduler contention. In this aspect, SQUIP is very similar to the Prime+Probe attack on caches, where the eviction set size, depending on the cache activity on a specific cache set, can be smaller than the associativity of that cache. In contrast to Prime+Probe attacks, we observed no cases requiring more than 24 multiplications to fill the ALU1 entries, as the dependency chain makes the scheduler queue follow a first-in-first-out policy (in contrast to complex cache replacement policies).

3.1.2. Observation Using Unserialized Timer Reads. While the performance counter shows that we are indeed observing scheduler queue contention, its use for practical attacks is limited by the fact that access to CPU performance counters requires root privileges on most Linux systems [20]. Furthermore, this counter has only been introduced with the Zen 3 microarchitecture and is unavailable on Zen 2 [7]. Hence, we show that scheduler queue contention can also be exploited by observing timing differences.

For precise timing measurements on `x86-64`, previous attacks, e.g., Flush+Reload [72], read the *Time-Stamp Counter* (`TSC`) using the `rdtsc` instruction. However, unlike on Intel and older AMD CPUs, the `TSC` on the Zen microarchitectures updates only every 20 to 35 cycles [36]. Instead, we use the *Actual Performance Frequency Clock Counter* (`APERF`). Previous work [35] has shown that it has an update interval of 1 cycle and can be read cycle-accurately using the `rdpru` instruction. In contrast to Lipp et al. [35], which required strict serialization of `rdpru`, we, in fact, exploit that it is not serializing. Therefore, `rdpru` can be executed out of order, and the `APERF` counter can be read before all previous instructions have been completed.

By uncommenting the highlighted lines in Listing 1, we can observe that `rdpru` is executed in parallel. In line 2, we initialize the `ecx` register to 1 to read the `APERF` counter. The first `rdpru` instruction in line 18 runs in parallel to the delay block and starts executing immediately after it has been enqueued, as it does not depend on any long-latency instructions. Therefore, the order of both `rdpru` instructions is preserved, even though they are not serializing.

If the multiplication block does not exceed the scheduler queue capacity, the second `rdpru` instruction in line 29, after the final multiplication, is also executed in parallel to the multiplication block. We confirm this by investigating the difference Δ_t between the first and the second `APERF` value, starting with only 11 multiplications and adding more of them:

- If Δ_t is small, the second `rdpru` was *executed immediately* while the multiplication block was executing in parallel, because `rdpru` is not serializing.
- If Δ_t is large, the second `rdpru` was *delayed until* the pipeline stall caused by *exceeding* the scheduler queue capacity has been resolved by retiring multiplications.

Figure 4 shows the average timing difference Δ_t over 100 000 runs. We can see a strong increase of Δ_t at the step from 22 to 23 multiplications on Zen 3, matching the limit we found in the previous experiment. In contrast, on an AMD Ryzen 7 3700X CPU (Zen 2), we see the same increase at the step from 16 to 17 multiplications. Official AMD documentation [9] states 16 entries as the capacity of the ALU1 scheduler queue, exactly matching our result. This experiment shows that we can indeed observe whether we have exceeded the capacity of the multiplication scheduler queue, with the code in Listing 1. We thus create a timing side-channel signal where the second `rdpru` instruction is either delayed (by a pipeline stall) or not, depending on the occupancy level of the scheduler queue.

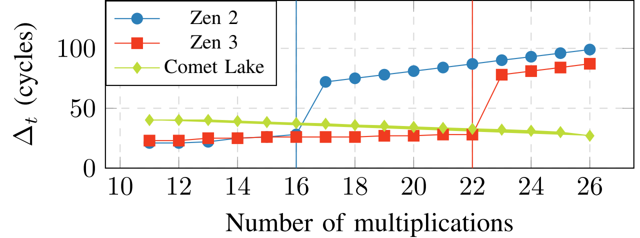


Figure 4: Average timing differences (n = 100000) for different lengths of the multiplication block on Zen 2, Zen 3, Comet Lake (sibling thread idle).

For comparison, we also ran a similar experiment on an Intel Core i5-10210U CPU (Comet Lake). As this CPU does not have the `rdpru` instruction, we use `rdtsc` here. On Intel CPUs, it has a resolution between 1 and 3 cycles [36] and is non-serializing [29], allowing for direct comparisons with the measurements from the AMD CPUs. On Comet Lake, we do not see any increase in Figure 4, as it uses a single large scheduler for all execution units. This highlights that SQUIP is different from port contention [5] and rather follows the semantics of a Prime+Probe, namely on the scheduler queue: While cache contention leads to evictions, scheduler queue contention stalls the backend.

In Appendix A, we present supporting results to rule out any interference from the `rdpru` instructions on the observed capacity. In Appendix B, we perform additional experiments on the contention of different scheduler queues, but in the rest of the paper, we focus on the ALU1 that we use for the RSA key recovery in Section 5. In Appendix C, we show that the same measurement technique can also be applied to the Apple M1, yielding comparable results. However, as the following experiments require SMT, which the M1 does not support, we focus on Zen 2 and Zen 3 for the remainder of this paper.

3.2. Observing Activity of the Sibling Thread

We measure contention of the ALU1 scheduler queue from a sibling thread on the same core, exploiting the sharing of the scheduler queues across SMT threads.

3.2.1. Observing the Queue Watermark on Zen 3. If we execute an empty, endless loop on the sibling thread in parallel to the measurement code on Zen 2, we still observe the same steep increase in the timing difference Δ_t from 16 to 17 multiplications, as in the same-thread measurements, see Figure 5. However, on Zen 3, we observe a steep increase with fewer multiplications, between 18 and 21, showing that there are fewer scheduler queue entries available for one thread if its sibling is busy. This result is in line with AMD's documentation, which states that the schedulers are competitively shared on Zen 2 [9] and watermarked on Zen 3 [10]. This watermark dedicates some entries to each of the hardware threads. Only if the watermark threshold is exceeded, the threads use the remaining, competitively shared entries of the scheduler queue.

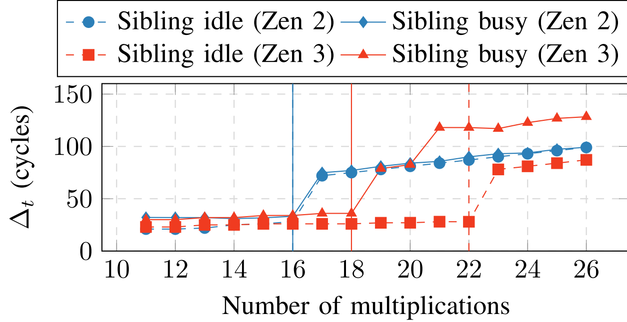
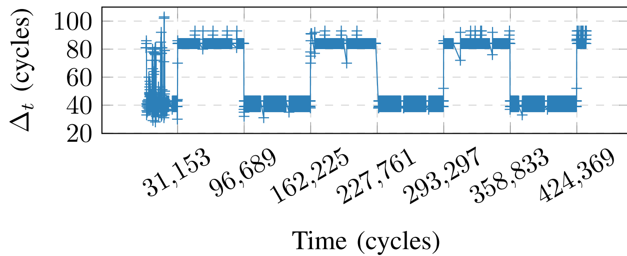
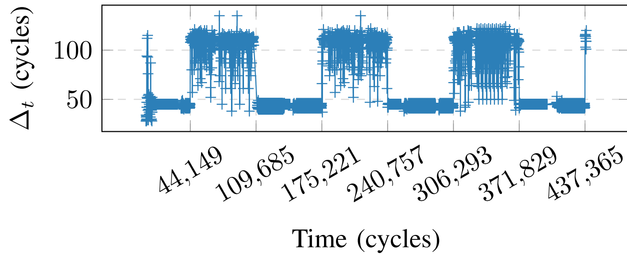


Figure 5: Effect of an empty loop on the sibling thread on the average timing differences on Zen 2 and Zen 3 (n = 100 000).



(a) Zen 2



(b) Zen 3

Figure 6: Observing multiplications on a co-located thread via scheduler contention.

3.2.2. Observing Multiplications of a Co-Located Thread. By measuring the available scheduler queue capacity, a receiver thread can detect multiplications executed by a co-located sender thread on the same core: On Zen 2, the receiver fills the scheduler queue to the overall capacity and measures Δ_t. If Δ_t is high, the sender performed a multiplication and, thus, occupied a scheduler queue entry during the measurement. On Zen 3, the receiver fills the scheduler queue to the watermark limit and measures Δ_t. If Δ_t is high, the sender performed multiple multiplications, exceeding the watermark threshold and causing contention on the competitively shared entries of the queue. In both cases, if the sender performs a sufficiently high number of multiplications, a co-located receiver can detect that.

We verify this with a proof-of-concept sender that generates an alternating sequence of high and low levels, each for 65 536 cycles. For a high level, the sender repeatedly runs a block of 15 dependent multiplications. For a low level, the sender repeatedly runs a block of 15 `nop` instructions. On the receiver side, we repeatedly perform the same measurement as before, using 16 (Zen 2) or 18 (Zen 3) dependent multiplications, respectively. To speed up the measurement, we omit the delay loop at the beginning of the measurement code in the receiver and replace it with a final, independent multiplication after the second `rdpru` instruction. If there was no contention before (caused by the other thread), that last multiplication finally induces it. With this, the CPU always stalls at the end of each iteration until the `sqrtsd` instructions and at least the first multiplication have finished. After the stall, the remaining multiplications finish quickly, within 3 or 4 cycles [9], [10] each. This effectively drains the scheduler queue before starting the next iteration and removes the need for the delay loop from the initial experiment (Listing 1).

For each measurement, we record the current `APERF` time counter value and the timing difference Δ_t. Figure 6a shows the clear alternating levels on Zen 2, demonstrating that a co-located receiver can indeed observe the multiplications of the sender. We also performed this experiment on a Zen 3 machine, using 18 multiplications in the receiver. Here we get more noise for the high level, as shown in Figure 6b, as the watermark mechanism pulls several attempts to transmit the high level down to the low level. However, the low level is unaffected by the watermark mechanism, making the two levels clearly distinguishable.

Finally, we investigated if this effect also applies to the Apple M1. As the Apple M1 does not have SMT, it is currently not possible to perform this experiment on it. However, future Apple CPUs with SMT and split scheduler queues would be affected by SQUIP in the same way.

4. Evaluating SQUIP Covert Channels

In this section, we evaluate the performance of the scheduler queue side-channel, like state-of-the-art [23], [57], [35], in covert channel scenarios across processes (see Section 4.3) and across VMs (see Section 4.4).

4.1. Threat Model

For our SQUIP covert channels, we assume that sender and receiver are co-located on different SMT threads of the same physical core. We assume that the sender and receiver are from different security domains with no legitimate communication channel, in line with previous works on SMT side channels [5], [64], [2], [4], [57], [63], [23], [62], [13]. Moreover, Linux still co-locates different security domains on the same physical core by default [24]. Windows Server, by default, avoids co-locating different virtual machines on the same physical core [43]. We make no assumptions about the CPU frequency of the targeted core.

4.2. Covert Channel Construction

In the following, we describe how the sender transmits a message via scheduler contention, as well as how the

receiver samples, decodes, and reconstructs the message. As the cross-VM scenario requires some parameter fine-tuning, we start with the cross-process scenario and explain the cross-VM changes later in Section 4.4.

4.2.1. Sender and Receiver Logic. In general, we encode a '0' as a low pressure on the scheduler queue of ALU1 and a '1' as a high pressure on that queue. The sender executes loops of different instruction blocks, depending on the value to send: To transmit a '0', the sender executes 190 (Zen 2) / 410 (Zen 3) iterations of 15 `nop` instructions[1] (low pressure on the ALU1 scheduler queue). For a '1', the sender executes 38 (Zen 2) / 35 (Zen 3) iterations of 15 dependent `imul` instructions on Zen 2 and 3 (high pressure on the ALU1 scheduler queue).

The receiver repeatedly samples using the techniques shown in the previous experiment, *i.e.*, it runs a sequence of dependent `imul` instructions with the timing measurements before and after. If Δ_t exceeds a certain threshold, indicating high pressure on the scheduler queue and a pipeline stall, the receiver records a '1', otherwise a '0'. As the receiver is not synchronized with the sender and runs faster, the receiver samples each bit multiple times.

4.2.2. Channel Encoding. With an ideal transmission, each bit would be received as a block of consecutive samples with the same value. For noise resilience, the receiver considers blocks shorter than len_{min} = 4 samples as errors and corrects them to the same value as the previous block.

To detect the start of the message and to determine how many samples correspond to a '0' and a '1' in the receiver, the sender transmits a preamble consisting of the bytes `\xff\x00\xaa\xaa` before we send the actual payload. The receiver observes the first byte of the preamble as a long sequence of '1' samples. After having seen at least 25 consecutive '1' samples, the receiver assumes to have received the start of the preamble. The second, zero-byte of the preamble only separates the first from the third byte and is not used otherwise.

The `\xaa\xaa` bytes in the preamble form an alternating bit pattern, which the receiver uses to calculate the average number of samples (*i.e.*, iterations) representing a single '0' bit (len_0) and a single '1' bit (len_1). Note that we cannot include the last '0' when calculating len_0, as the payload might start with a '0'. Instead, for the last bit of the pattern, we just skip ahead len_0 samples, and consider the subsequent sample to be the start of the payload.

After this, the receiver reconstructs the payload by iterating over the remaining samples. First, we infer each block of consecutive samples of the same value (after performing the len_{min} correction). Based on whether the samples are consecutive '0's or '1's, we then divide the number of consecutive samples by len_0 or len_1 respectively, to get how many '0' or '1' payload bits they correspond to.

To mark the end of the message, the sender transmits a trailer of n '1' bits. After the message transmission, the receiver only rarely observes contentions. Assuming the receiver knows the approximate length of the transmission, it can find the exact end of the payload by searching backward through the recorded samples for the last n consecutive '1's. Note that this block has to be long enough to be distinguishable from occasional contentions caused by other activity on the system. We observed that we can reliably detect trailers with a length of 15 '1' samples (approximately n = 3, *i.e.*, three '1' bits).

4.3. Cross-Process Evaluation

We first evaluate the performance of the covert channel in a cross-process setting on an AMD Ryzen 7 3700X (Zen 2) running Ubuntu 20.04 with Linux kernel v5.15.6 and on an AMD Ryzen 7 5800X (Zen 3) running Ubuntu 20.04 with Linux kernel v5.16.13. We transmit 10 000 random messages, each with a size of 32 768 bit. Receiver and sender run in separate processes and are co-located on different SMT threads on the same core.

In the cross-process setting, we use the same parameters for the receiver as described in Section 3.2, priming the scheduler queue with 16 (Zen 2) and 18 (Zen 3) `imul` instructions that are delayed by 3 `sqrtsd` instructions. For each iteration of the receiver, we drain the scheduler queue by issuing a final `imul` instruction after the measurement. We choose a Δ_t threshold of 65 cycles on both machines, which clearly distinguishes the *contention* and *no contention* scenarios in Figure 4 and Figure 5.

On Zen 2, we achieve an average raw capacity of 2.19 Mbit/s (n = 10000, $\sigma_{\bar{x}}$ = 0.002295) with an average bit-error rate of 0.71 % (n = 10000, $\sigma_{\bar{x}}$ = 0.0274).

On Zen 3, we achieve an average raw capacity of 2.70 Mbit/s (n = 10000, $\sigma_{\bar{x}}$ = 0.000008) and an average bit-error rate of 0.62 % (n = 10000, $\sigma_{\bar{x}}$ = 0.0159).

4.4. Cross-VM Evaluation

To demonstrate that SQUIP also works across virtual machine boundaries, we evaluated the performance of the covert channel in a cross-VM setup, with sender and receiver running in separate VMs. For our proof of concept, each virtual machine has one virtual CPU (vCPU), statically assigned to one SMT thread of a shared physical core. We start using the same Zen 2 and Zen 3 hardware as in Section 4.3. Finally, we evaluate the covert channel on an AMD EPYC 7443 machine, with and without SEV enabled on both VMs, showing that SEV does not prevent leakage across VMs that allow SMT in their SEV policy (cf. [14]).

4.4.1. Adaptions for the Cross-VM Setup. One challenge with this cross-VM setting is that KVM does not support `rdpru` inside virtual machines, and neither does Xen 4.16 [68]. Both hypervisors trigger an invalid instruction exception if a virtual machine tries to execute `rdpru`. Future versions may support `rdpru`, as unofficial patches [17] already provide such support. However, to show that SQUIP works across virtual machines on unmodified hypervisors,

1. We require more `nop` iterations in Zen3 to establish a sufficiently large bit-period, as we observe `nops` execute faster on Zen3.

TABLE 1: Parameters and results of the covert-channel evaluation.

Setup		**Receiver parameters**					**Sender parameters**		**Average results** (n = 10000)	
Scenario	CPU	Timing method	Number of `sqrtsd`	Number of `imul` (prime/drain)	Δ_t Threshold	‘1’ samples Preamble / Trailer	Iterations for ‘0’	Iterations for ‘1’	Raw transmission rate	Error rate
Cross-Process	Ryzen 7 3700X (Zen 2)	`rdpru`	3	16 / 1	65	25 / 15	190	38	2.195 Mbit/s ($\sigma_{\bar{x}}$ = 0.002295)	0.71 % ($\sigma_{\bar{x}}$ = 0.0274)
	Ryzen 7 5800X (Zen 3)	`rdpru`	3	18 / 1	65	25 / 15	410	35	2.700 Mbit/s ($\sigma_{\bar{x}}$ = 0.000008)	0.62 % ($\sigma_{\bar{x}}$ = 0.0159)
Cross-VM	Ryzen 7 3700X (Zen 2)	`rdtsc`	12	16 / 1	130	50 / 15	230	90	0.873 Mbit/s ($\sigma_{\bar{x}}$ = 0.000031)	3.18 % ($\sigma_{\bar{x}}$ = 0.0738)
	Ryzen 7 5800X (Zen 3)	`rdtsc`	12	15 / 27	150	50 / 55	1320	113	0.892 Mbit/s ($\sigma_{\bar{x}}$ = 0.000039)	0.75 % ($\sigma_{\bar{x}}$ = 0.0297)
	EPYC 7443 (Zen 3)	`rdtsc`	12	15 / 27	150	50 / 55	1250	113	0.874 Mbit/s ($\sigma_{\bar{x}}$ = 0.000052)	0.96 % ($\sigma_{\bar{x}}$ = 0.0398)
Cross-VM (SEV)	EPYC 7443 (Zen 3)	`rdtsc`	12	15 / 27	150	50 / 55	1250	113	0.873 Mbit/s ($\sigma_{\bar{x}}$ = 0.000071)	1.47 % ($\sigma_{\bar{x}}$ = 0.0639)

we let the receiver use the non-serializing [6] `rdtsc` instruction for timer reads instead.

As described in Section 3.1.2, `rdtsc` has a lower update rate and is thus less precise. However, we can compensate for a less precise timer by holding scheduler contention for a longer period of time, by further delaying the dependent multiplications with a longer chain of `sqrtsd` instructions. This approach is similar to the extension of transient execution windows [69], [51] in Spectre attacks [32].

Furthermore, while the `APERF` counter frequency is tied to the current frequency of the core, the `TSC` frequency is constant and does not change with the CPU frequency [3]. In consequence, we observe that CPU frequency changes affect the Δ_t threshold and lead to transmission errors: With a higher frequency, the CPU performs more operations per `TSC` cycle, hence we require a lower threshold than with a lower CPU frequency. If the threshold is too high, we do not detect scheduler contention and we only receive ‘0’s. On the other hand, if the threshold is too low, the observed times are above the threshold even without contention, resulting in only ‘1’s. However, with 12 (Zen 2 and 3) `sqrtsd` instructions, we can amplify the timing difference between the ‘0’ and ‘1’ cases enough, so that a threshold of 130 (Zen 2) / 150 (Zen 3) works with both, the minimum and the maximum CPU frequency. Additionally, before starting the actual transmission, the sender and the receiver execute a delay loop, stabilizing the CPU frequency beforehand.

An interesting observation on Zen 3 was that, using `rdtsc` instead of `rdpru` in the receiver together with 18 dependent multiplications to measure ALU1 contention, similar to the cross-process setup in Section 4.3, resulted in receiving all ‘1’s erroneously due to spurious contentions. This was the case even if both the sender and the receiver ran natively on the host. We suspect the reason for this is that `rdtsc` on Zen 3 is a complex microcode-assisted [10] instruction that probably requires ALU1 for execution, and thus shares the scheduler queue with the multiplications, leaving fewer slots for the multiplications. Note that we have not seen such an effect on Zen 2.

To work around this issue on Zen 3, we use only 15 dependent multiplications between the `rdtsc` instructions of the receiver. Consequently, to drain the scheduler queue after each measurement, we now have to issue more multiplications after the second `rdtsc` instruction. In our case, we use a chain of 27 multiplications, which depend on a single load of an immediate value into a register. With this, we no longer observe spurious stalls on Zen 3.

Another challenge we face in this setting is increased measurement noise with one vCPU per VM, as now one hardware thread has to handle the housekeeping tasks of both the hypervisor and the guest operating system. This can cause the receiver and the sender to be interrupted more frequently, resulting in more errors like missed samples (if the receiver is interrupted) or inserted zeroes (if the sender is interrupted) [41]. Some of these errors are overcome with techniques described in Section 4.2, using multiple samples for each bit and correcting blocks shorter than len_{min}. This makes the channel more robust against short interruptions, but it cannot correct longer interruptions.

We observed that we have to increase the number of iterations in the sender for each ‘0’ and ‘1’ of the message (see Table 1), to achieve a reliable transmission. In the first test runs, we furthermore observed that the preamble was sometimes detected too early and that the trailer was sometimes detected too late, both due to occasional bursts of contentions. To address this problem, we increase the number of consecutive ‘1’ samples the receiver has to observe to detect the preamble and the trailer.

4.4.2. Results. Using the parameters from Table 1, we achieve an average raw capacity of 0.87 Mbit/s (n = 10000, $\sigma_{\bar{x}}$ = 0.000031) with an average bit-error rate of 3.18 % (n = 10000, $\sigma_{\bar{x}}$ = 0.0738) on the Ryzen 7 3700X (Zen 2).

On the Ryzen 7 5800X (Zen 3), we achieve an average raw capacity of 0.89 Mbit/s (n = 10000, $\sigma_{\bar{x}}$ = 0.000039) and an average bit-error rate of 0.75 % (n = 10000, $\sigma_{\bar{x}}$ = 0.0297).

In addition to the Ryzen 7 3700X (Zen 2) and Ryzen 7 5800X (Zen 3) machines, we also evaluated the covert channel on an EPYC 7443 machine supporting AMD SEV. With AMD SEV, the memory content of each virtual machine is encrypted with one separate key per guest, isolating virtual machines from each other and the host [12]. We achieve a raw capacity of 0.87 Mbit/s (n = 10000, $\sigma_{\bar{x}}$ = 0.000071) at a bit-error rate of 1.47 % (n = 10000, $\sigma_{\bar{x}}$ = 0.0639) with SEV enabled for both VMs. This shows that the isolation provided by SEV does not prevent leakage via SQUIP. For comparison, we repeated this experiment with SEV and memory encryption both disabled, yielding the same capacity at a slightly lower error rate of 0.96 % (n = 10000, $\sigma_{\bar{x}}$ = 0.0398).

Table 1 summarizes the results and parameters used for each machine and in each setting. Table 2 in Appendix D compares the bit rate in the cross-VM covert-channel setting for SQUIP with state-of-the-art cross-VM covert channels. We highlight that our covert channel is among the fastest of them, as it does not require complex eviction strategies or memory accesses (only requires low latency ALU instruc-

tions) and produces a low-noise signal. One limitation of our channel is that it only works across SMT threads and not cross-core, unlike some prior covert channels.

5. Side-Channel Attacks on Scheduler Queues

In this section, we show the significance of the scheduler queue side channel in a practical attack on the RSA signature process of mbedTLS [15] (version 3.0.0) on an AMD Ryzen 7 5800X CPU (Zen 3). We recover a full RSA private key with only 50 500 traces. We demonstrate our attack across processes and across KVM virtual machines.

5.1. Threat Model

The unprivileged attacker's goal is to steal the RSA secret key from the victim process. Following the threat models of state-of-the-art SMT attacks [5], [53], [18], [63], [57], we assume that the attacker and victim are co-located on the same physical core but run on different SMT threads. The victim performs a signature process that the attacker can trigger via a legitimate interface (e.g., a signing service) an arbitrary number of times. We assume that the attacker can make side-channel observations throughout the square-and-multiply algorithm (e.g., the start and end of square-and-multiply are detected using timing or a side channel).

5.2. Environment

Various libraries implement the RSA algorithm like OpenSSL [45], IntelIPP [27] and mbedTLS [15]. We attack the mbedTLS implementation following previous work [59]. mbedTLS implements the RSA algorithm via a sliding window Montgomery *modulo exponentiation* algorithm [42]. The sliding-window exponentiation combines multiple key bits to optimize performance. The default window length in mbedTLS increases with the key length. However, Liu et al. [39] show that attacks on window length '1' can be extended to an arbitrary length. Therefore, we set the window length to '1' while attacking RSA-4096. Note that the mbedTLS implementation does not use distinct functions for *square* and *multiply* but reuses the multiplication function. For both the cross-process and the cross-VM attack, we again make no assumptions about the CPU frequency.

Cross-Process Setup. For the *cross-process* attack setup, we only assume that the attacker and the victim run co-located, without any further restrictions. This is possible on Linux by default [24] if the application has not set `PR_SCHED_CORE` via `prctl`. We scanned the source code of OpenSSL 3.0.4, OpenSSH 9.0, QEMU 7.0.0, and Firefox 99.0, and did not find these calls in any of them, making co-location a realistic scenario for the cross-process setting.

Cross-VM Setup. For the *cross-VM* attack, the attacker and the victim process run in separate virtual machines, but instead of a single vCPU per VM as in Section 4.4, we now run the virtual machine with two vCPUs. In the attacker VM, the attacker assigns one vCPU for the attacker process, and the other for housekeeping tasks of the hypervisor and the guest operating system. We assume that in the victim VM, the victim process is pinned to one vCPU, which helps avoiding unnecessary movements of tasks between hardware cores and, thus, cold caches, *i.e.*, it is plausible to find this configuration in practice. We make no assumptions about the other vCPU of the victim. We assume and focus on the setup where the attacker has achieved co-location [26] with the victim such that the RSA computation runs on the same physical core (on a sibling SMT thread) as the attacker's SQUIP attack. To reduce interference from other tasks and timer interrupts, we enable the full task-isolation mode [55] in the guest and the host. This avoids interference from operating system or hypervisor tasks. Note that this is not a requirement for the attack, as filtering techniques (as described by Yarom et al. [73]) can also be used for timing measurements degraded by timer interrupts. Previous work has shown that achieving co-location in the cloud is possible [26]. While larger cloud providers will avoid co-location of different tenants on the same core, with the associated performance cost (see Section 6.2), smaller cloud providers may not have the margins to pay this performance cost and, therefore, may not use the co-scheduling approach. Furthermore, also on private servers or personal computers, where co-scheduling is not enabled by default, virtual machines are used for security (isolation) in practice.

5.3. SQUIP Attack on RSA

In our attack, we target the `mul` instructions executed during the multiplication function of the square-and-multiply algorithm. In particular, for a '0' in the exponent, the `multiply` function is only executed once (square), whereas, for a '1', it is called twice (square and multiply). This results in more multiplications for a '1' in the exponent and, therefore, a higher chance to observe scheduler queue contention. The second additional invocation of the `multiply` function in the '1' case has a different parameter, also changing the microarchitectural behavior of that second invocation. It is important to note that the `multiply` function will effectively perform more than just a single integer multiplication. Consequently, the occupancy level in the ALU1 scheduler queue varies over time, depending on the value of the secret bit. If the attacker can observe these differences, they can directly infer the exponent, *i.e.*, the private key.

Using the SQUIP side channel, we create a contention profile, observing the probabilities for scheduler contention over the execution time of the square-and-multiply algorithm. Such a contention profile is formed by multiple traces: For each trace, we continuously monitor the scheduler queue of ALU1 using the SQUIP measurement code, while the victim is executing the square-and-multiply algorithm. Again, if the observed Δ_t exceeds a certain threshold, we record that sample as a '1', otherwise as a '0'.

To compensate for run-time variations (e.g., due to varying clock frequencies), we stretch each trace to a fixed length, uniformly filling the resulting gaps with '0's. We

do not use any interpolation between the samples as we do not want to lose precision in the timing information of the observed ‘1’s. Finally, we compute the contention profile by adding up the stretched traces, sample by sample.

Given enough traces, even slight differences in the contention probabilities between the two cases become visible in the contention profile. We observed stable results with 5000 traces and 50 additional warmup rounds.

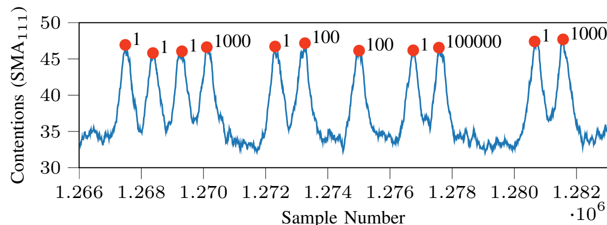
On Zen 3, the scheduler queue of ALU1 has a watermark limit of 18 (see Section 3.2), implying that the attacker cannot fill the queue completely when priming the scheduler queue. In consequence, the attacker can only observe a stall if the victim has at least $23 - 18 = 5$ operations enqueued at the same time. We observed that the probability for that is higher during the ‘multiply’ step than during the ‘square’ step in the square-and-multiply implementation of mbedTLS. Consequently, the ‘multiply’ steps show up as peaks in our contention profiles.

5.3.1. Trace Recording. For each key, we generate 10 contention profiles. For each profile, we collect 5050 traces. Thus, in total, we record 50 500 traces. The first 50 traces of each recording serve as a warmup phase and are discarded, *i.e.*, only the remaining 5000 traces are actually used per contention profile. Out of the 10 contention profiles, we use the best one for key extraction, as explained subsequently in Section 5.4.

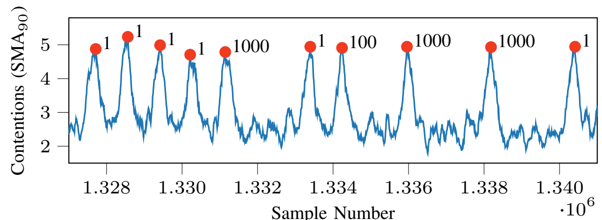
For each trace, the attacker repeatedly executes the SQUIP measurement code while the victim is performing the RSA signature, recording the samples as a trace, from the start to the end of a victim’s execution of the square-and-multiply algorithm. As with the covert channels shown before, if the observed Δ_t exceeds a threshold of 65 cycles (cross-process) or 200 cycles (cross-VM), the attacker stores that sample as a ‘1’. Otherwise, it is stored as a ‘0’. After recording each trace, we stretch it to a length of 2 700 000 samples and add it to the contention profile.

Our first proof-of-concept attack runs in the cross-process setting, in which the attacker uses the same parameters as the receiver in the covert channel, *i.e.*, 18 dependent multiplications, delayed by 3 `sqrtsd` instructions, again with one final multiplication after the second `rdpru` instruction to drain the queue. To increase the chances of observing contention for a ‘1’ but not for a ‘0’, we perform dummy memory operations between each sampling period. This gives the CPU time to drain the scheduler queue from the last ‘1’ so that we are at the same occupancy level again for the next bit. In the cross-VM setting, we observed more noise, *i.e.*, higher occupancy levels on the scheduler queue, such that 14 dependent multiplications were already sufficient, albeit delayed by 12 `sqrtsd` instructions to extend the contention window. We also observed that we can significantly reduce the timing jitter if we replace the final multiplications after the second `rdpru` with an `lfence`.

5.3.2. Trace Postprocessing. We extract the key from the contention profile in a postprocessing step. To reduce the impact of outliers, we compute a simple moving average (SMA), resulting in the signals shown in Figure 7. We observe a clear difference in the number of contentions caused by the `multiply(x,x)` and the `multiply(x,a)`, which is only executed for a ‘1’. The `multiply(x,a)` causes contentions with a slightly higher probability, resulting in the peaks. With these peaks, we can clearly see the ‘1’s in the exponent and also distinguish multiple succeeding ‘1’s. To extract the ‘0’s of the key, we measure the distance between the peaks. From the minimal distance between two peaks in the signal, we can infer the duration of processing a ‘1’, which is double that of a ‘0’. If two peaks are further apart than this minimal time, the number of ‘0’s in between is computed from the distance.



(a) Cross-Process



(b) Cross-VM. The absolute height of the peaks, which corresponds to the number of scheduler contentions, is approximately 10 % of those in the cross-process scenario.

Figure 7: The peaks show a higher number of contentions because of the ‘1’ multiply. The time difference between two peaks is directly related to the number of ‘0’s between two ‘1’s. The peak labels denote the recovered key bits. The plot is from the center of the contention profile. The peaks become sharper and higher on both sides.

We automate the postprocessing in Python with the help of the SciPy library. To find the peaks corresponding to the ‘1’s, we use SciPy’s `find_peaks` function. Due to the normalization of each trace’s time domain and runtime differences, the traces have near-perfect alignment at the start and end. However, they degrade in the middle, influencing the sharpness and height of the contentions peaks. The `find_peaks` function requires well-chosen parameters to return a valuable and reliable result, and these parameters change significantly throughout the trace because of the changing peak heights. We split the trace into five parts and chose different `find_peaks` parameters for every part.

Our key extraction process is fine-tunable via additional parameters, including the window size of the moving average and factors to refine the extraction of ‘0’s from the

duration between peaks. All parameters are shown in Table 3 in Appendix F.

This parameterization allows us in a last step to automatically optimize the extraction result with an evolutionary optimization algorithm. We split a data set of different measured RSA keys into a training and validation set to optimize our parameters and an evaluation set for our final evaluation. During the optimization, we first randomize the parameters and then optimize multiple sets of parameters with SciPy's `optimize` function. The fittest group of parameters is selected by the lowest edit distance of the extracted key, mutated multiple times, and each set is optimized again. The fitness converged after approximately ten generations.

5.4. Evaluation

We evaluate the attack across processes and across VMs as described in the following.

Optimizing the Postprocessing Parameters with Training Keys. First, we create a set of 10 training keys using `openssl genrsa`. For each of these 10 training keys, we record 10 contention profiles, using 5000 traces and 50 warmup rounds for each profile. Out of the resulting contention profiles, we choose the best profile for each key. The best profile is defined by the quality and quantity of the peaks we find with `find_peaks`. A higher number of peaks is better if it is in the range of the expected number of peaks, around 2100. As bad contention profiles can have a very high number of peaks, mainly consisting of noise, these profiles are excluded. The quality is defined by the "prominence" *i.e.*, the height of the peaks in relation to their surrounding baseline. The weighting of the number of peaks and prominence for the final contention profile quality was defined empirically and is different for the cross-process and cross-VM profiles.

With the 10 best contention profiles, we build a training set of 6 profiles and a validation set of 4 profiles for the evolutionary optimization algorithm.

Extracting Evaluation Keys. After optimization, we create a fresh set of 10 new evaluation keys, again using `openssl genrsa`. For each evaluation key, we again record 10 contention profiles, again using 5000 traces and 50 warmup rounds for each profile. We choose the best contention profile for each key. With the chosen contention profile, we then apply our postprocessing to extract the key. Finally, we compute the edit distance between the actual ground truth key and the extracted key.

Results for the Cross-Process Scenario. In the cross-process scenario, we can extract the evaluation keys with an average edit distance of 4.9 bit (n = 10, $\sigma_{\bar{x}}$ = 1.28). This corresponds to an error rate of less than 0.12 %. Our best result is one key without any errors and three keys with an edit distance of only 2 bit. Our worst result is one key with an edit distance of 15 bit.

Recording a single contention profile takes 244.41 s (n = 100, $\sigma_{\bar{x}}$ = 0.1911) on average. Consequently, recording the 10 contention profiles we require for a key takes about 41 min. Given the strong signal and the low error rate, we consider it possible to reduce the required time even more by reducing the number of traces used for the contention profiles or by recording fewer profiles to select from, at the cost of a possibly higher error rate.

Results for the Cross-VM Scenario. In the cross-VM scenario, we can extract the evaluation keys with an average edit distance of 17.8 bit (n = 10, $\sigma_{\bar{x}}$ = 3.26), corresponding to an error rate of less than 0.5 %. Our best results are two keys with an edit distance of 8 bit. Our worst results are one key with an edit distance of 28 bit and one with 43 bit.

Recording a single contention profile takes 230.12 s (n = 100, $\sigma_{\bar{x}}$ = 0.1238) on average in the cross-VM setting. Consequently, recording the 10 contention profiles we require for a key takes about 38 min.

6. Discussion and Limitations

As shown, using the SQUIP side channel, an unprivileged attacker can extract sensitive information from a co-located victim within less than 45 min, achieving very low error rates. In this section, we discuss the limitations of our attack and possible hardware and software countermeasures.

To summarize, the SQUIP attack exploits 1) that the ALUs are connected to different schedulers, 2) that the ALUs have different capabilities, 3) that co-located processes compete for free slots in the scheduler queues and 4) that the control flow of the RSA implementation is secret-dependent. Without any of these four prerequisites, the demonstrated attack no longer works, so that possible countermeasures can target all of them.

6.1. Hardware Countermeasures

Future CPU designs can avoid being vulnerable to the SQUIP attack by 1) using a single scheduler design, 2) making the schedulers symmetric, or 3) isolating hardware threads more strictly in the scheduler queues.

Single Scheduler Design. Single scheduler designs such as Intel's are not affected by SQUIP, as the single scheduler does not differentiate between different µops. However, there may be costs in terms of complexity and power for unified schedulers, as the scheduling algorithm input size (queue sizes and the number of functional units) grows [66], necessitating distributed scheduler designs in the future.

Symmetric Scheduler Design. Another approach might be a design with multiple schedulers, where there is no difference in the capabilities of the ALU(s) connected to each of them. With such a symmetric design, it is no longer possible to specifically target one scheduler queue, as operations are now distributed over all available queues. Consequently, an attacker can no longer deduce that a specific group of operations is performed, and the attack no longer works.

Stricter Partitioning Mechanisms. As the scheduler queues are only shared between threads of the same core and not between cores, an attacker can only observe the scheduler queue usage of a co-located victim. Our attack does not work on Apple's M1, as it does not support SMT.

However, if future Apple CPUs support SMT and keep the split scheduler design, they might also be affected.

Future CPU designs might consider more strict partitioning mechanisms [62], also for the scheduler queues. For example, with static, spatial partitioning and two hardware threads, a single thread can only occupy at most half of the capacity of a resource [62]. With a scheduler queue size of, e.g., 16 slots, each thread can always use at most 8 of them. The capacity available to one thread is, therefore, completely unaffected by the scheduler queue usage of the other thread. Consequently, an attacker can no longer monitor the other thread, and our attack will not work, either. Adaptive partitioning is another, more sophisticated mechanism that, however, does not entirely prevent leakage [62].

6.2. Software Mitigations

In this subsection, we discuss three mitigation possibilities. First, security-critical software can protect itself by ensuring secret-independent execution flow with constant-time implementations. Second, as a more hypothetical option, we also briefly discuss exploiting the watermark mechanism for protection. Third, operating systems can protect against the attack by either disabling SMT or by using co-scheduling.

Constant-Time Algorithms. With our attack, we exploit that mbedTLS performs more multiplications for a '1' in the key than for a '0'. Secret-dependent control-flow like this, where different instructions are executed depending on a secret value, has been attacked using various other side channels in previous works [48], [72], [39], [5], [37]. Most of these works [48], [72], [39], [5] recommend using constant-time cryptography algorithms, such as square-and-multiply-always, to mitigate the attacks. These algorithms ensure that the execution flow is independent from secret input, so that always the same instructions are executed. With this, we can no longer distinguish a '1' from a '0' and our attack no longer works. However, attacks on non-cryptographic software, which cannot be implemented in constant time (e.g., user input), might be possible too, so this mitigation might be incomplete.

Exploiting the Watermark Mechanism on Zen 3. Security-critical software could leverage the watermark mechanism on Zen 3 (see Section 3.2.1) by ensuring that its usage of the critical scheduler queue remains too low for an attacker to observe. For example, inserting some dummy instructions into a code path containing many multiplications could space out the multiplications and keep their queue usage below the watermark limit. However, this mitigation would be very specific to the Zen 3 microarchitecture, as the scheduler queues are not watermarked on Zen 2, and future CPUs might have different watermark limits. Furthermore, unlike constant-time algorithms, this would not help against other side channels. Because of these limitations, we did not further investigate this approach.

Disabling SMT. Various works [2], [5], [64], [25], [73], [44] have shown attacks on SMT before. Furthermore, Taram et al. [62] have analyzed contention-based covert channels on SMT CPUs, albeit without considering the scheduler queues as a possible attack target. Consequently, some operating systems warn about SMT or even disable it by default [31] on Intel CPUs. As most academic offensive research has focused on Intel CPUs, the severity of the threat has thus far been less clear on AMD CPUs – so SMT is still enabled there by default, because it accelerates certain workloads significantly: In a 2005 study, when enabling SMT, Ruan et al. [54] observed a speedup of 1 % to 15 % on a dual-CPU Xeon system running the Specweb99 [60] web server benchmark. In 2018, Phoronix [33] published a direct comparison for various workloads on an Intel Coffee Lake CPU, showing SMT speedups of up to 30 % for certain workloads like 3D rendering or code compilation. In 2020, Cutress [19] performed such a comparison specifically for an AMD Zen 3 CPU, showing SMT speedups of up to 34 % for data compression and up to 26 % for 3D rendering. However, disabling SMT comes at the cost of reducing performance and it can reduce measurement noise in other attacks [50]. Furthermore, while we focus on exploiting scheduler queue contention in cross-SMT scenarios, single-threaded attacks exploiting scheduler queue contention may also be possible.

Co-Scheduling. A more fine-grained approach would use co-scheduling [47] (or core-scheduling [34]), also on AMD CPUs. With co-scheduling, processes from different security domains are prevented from running co-located on the same core, effectively defeating our attack. During the development of the core-scheduling feature for the Linux kernel, Faggioli [21] performed an extensive performance evaluation with several standard benchmarks. His evaluation included an over-committed scenario, where two VMs with 8 VCPUs were run on a quad-core host with two SMT threads per core. One of these VMs generated CPU, memory and I/O stress, while he ran the benchmark in the other VM. With the STREAM benchmark, he observed a performance drop between 26 % and 32 % when disabling SMT, which is in line with the results cited in the section before. When enabling core-scheduling, so that only VCPUs of the same VM can share the same core simultaneously, he only observed a performance drop between 12 % and 20 %. In contrast, with sysbench, he observed a performance loss of 8 % to 53 % when disabling SMT, while core-scheduling resulted in performance losses between 19 % and 91 %. These results indicate that the performance loss from disabling SMT or enabling co-scheduling is highly workload specific, thus we unfortunately cannot give general guidance for all workloads. Also, like disabling SMT, co-scheduling cannot prevent potential single-threaded attacks.

7. Related Work

Resource conflicts on SMT CPUs have already been exploited in previous side-channel attacks. In this section, we summarize and discuss such related work.

7.1. Prime+Probe

Prime+Probe exploits the limited capacity of CPU caches [46], [48], [39]. The attacker first *primes* the cache

by filling it with its own data. After the victim process has been executed for a short time, the attacker *probes* the cache by timing memory loads, to observe which of the previously primed cache sets have been evicted by the victim accessing congruent memory addresses. While the first Prime+Probe attacks were single-threaded [46] or limited to co-located processes [48], they were later extended to work across cores, by attacking the shared LLC [39]. Other Prime+Probe-style attacks target coherence directories [71], branch prediction caches [1] or translation lookaside buffers [25] instead of the memory caches.

Our SQUIP attack follows a scheme similar to Prime+Probe, as we also *prime* a capacity-limited resource (*i.e.*, a scheduler queue) and use timing differences to *probe* for resource usage conflicts. As we target the scheduler queues, which are exclusive to the physical core, we cannot extend our attack to work across cores.

7.2. SMT Attacks

The co-location of two threads on the same core with SMT can be exploited in different ways.

Port Contention. Aldaya et al. [5] (and prior works [64], [2]) observe the contention of ALU ports instead of the scheduler queue. They run their PortSmash attack on an Intel CPU that has a single-scheduler design but also ALUs with different capabilities. In their side-channel attack, they profile the port usage of a victim on a target port. To detect port usage of the victim, they choose an instruction executed on the same port simultaneously and measure the time over a block of that instruction. With this approach, they observe that the throughput for that instruction type is halved for the attacker when the victim also runs similar instructions simultaneously, as that port is now effectively multiplexed between the attacker and the victim process. Using this observation, they attack an OpenSSL P3-384 ECDSA signature process.

In contrast to port contention, which only delays attacker operations of the same type as the victim operation, SQUIP delays the execution of *all* subsequent operations regardless of the type. These pipeline stalls are much easier to observe. Moreover, our attacker can surgically control the length of the delay for the multiplications, using a dependency chain of `sqrtsd` – this makes the delay more easily observable even when the timing measurements are less precise (e.g., `rdtsc` from within a VM). Lastly, with a single measurement, we only observe whether scheduler queue contention caused a pipeline stall or not. So a single measurement is sufficient to distinguish between only two latency levels (much like a cache hit and miss), unlike PortSmash, which requires inference of a decrease in port bandwidth that requires many measurements to ascertain reliably.

Bhattacharyya et al. [18] demonstrate that port contention can be exploited in Spectre [32] gadgets. Compared to the original Spectre attack that leaked information via a cache side channel, they leaked information via a port contention side channel. Similarly, our scheduler contention side channel, SQUIP, can also be used as a channel to leak information in Spectre gadgets.

Attacks on Other SMT Shared Resources. Aimoniotis et al. [4] show that reorder-buffer contention can leak information, yet they do not demonstrate a full attack. Yarom et al. [73] induce L1 cache bank conflicts to recover a 4096 bit RSA key. We demonstrate an orthogonal vulnerability with a split scheduler queue design in multiple generations of AMD CPUs and Apple CPUs, and demonstrate that it can leak RSA keys on an AMD Zen 3 CPU.

Generalized Approaches. Fogh [23] introduced Covert Shotgun, an automated SMT covert channel finder. In the sender, Covert Shotgun runs a loop of a chosen "signal instruction". In the receiver, it measures the execution time of a chosen "receiver instruction". Covert Shotgun repeats this process with different instruction combinations. Note that for the SQUIP side channel, we have to prime the scheduler queue. As Covert Shotgun does not do any specific preparation before running the receiver instruction, it cannot find our SQUIP side channel automatically.

Taram et al. [62] systematically analyzed SMT-enabled CPUs for contention side-channels and evaluated multiple hardware mitigation strategies. However, they based their analysis on a single-scheduler design, so they have not considered split scheduler queues and different capabilities of the associated ALUs as a possible security risk.

8. Conclusion

In this paper, we introduced the SQUIP attack. SQUIP is the first side-channel attack on scheduler queues, which are separate per execution unit on Apple M1, AMD Zen 2, and Zen 3 microarchitectures. We reverse-engineered the behavior of the scheduler queues on these CPUs and showed how they can be primed and probed. We evaluate the SQUIP attack across SMT threads in different scenarios on AMD Zen 2 and Zen 3 CPUs. First, a covert channel to measure the bandwidth of the side channel. We were able to transmit 0.89 Mbit/s across virtual machines at an error rate below 0.8 %, and 2.70 Mbit/s across processes at an error rate below 0.8 %. In our full side-channel attack on an mbedTLS RSA signature process, we can recover the full RSA-4096 key with only 50 500 traces and less than 5 to 18 bit errors on average across processes and virtual machines. Our work highlights that pipelines with multiple scheduler queues have to be reevaluated for security and future CPUs need mitigations to prevent our attack.

Acknowledgments

We thank the reviewers and our shepherd for their valuable feedback. We also thank Moritz Lipp, Claudio Canella for their valuable input and Jonathan Montineri for help with initial experiments. Part of the funding was provided by generous gifts from Amazon and Red Hat. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] O. Acıiçmez, c. K. Koç, and J.-p. Seifert, "On the Power of Simple Branch Prediction Analysis," in *AsiaCCS*, 2007.

[2] O. Acıiçmez and J.-P. Seifert, "Cheap Hardware Parallelism Implies Cheap Security," in *FDTC*, 2007.

[3] *Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh*, 3rd ed., Advanced Micro Devices Inc., 7 2018.

[4] P. Aimoniotis, C. Sakalis, M. Själander, and S. Kaxiras, "Reorder Buffer Contention: A Forward Speculative Interference Attack for Speculation Invariant Instructions," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 162–165, 2021.

[5] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port Contention for Fun and Profit," in *S&P*, 2019.

[6] AMD, "AMD64 Architecture Programmer's Manual," 2017.

[7] ——, "Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processors," 2017.

[8] ——, "Software Optimization Guide for AMD EPYC 7001 Processors," June 2017.

[9] ——, "Software Optimization Guide for AMD EPYC 7002 Processors," March 2020.

[10] ——, "Software Optimization Guide for AMD EPYC 7003 Processors," November 2020.

[11] ——, "Processor Programming Reference (PPR) for AMD Family 19h Model 21h, Revision B0 Processors," 2021.

[12] ——, "AMD Secure Encrypted Virtualization (SEV)," 2022. [Online]. Available: https://developer.amd.com/sev/

[13] ——, "PortSmash Mitigations," 2022. [Online]. Available: https://www.amd.com/en/support/kb/faq/pa-210

[14] ——, "SEV Secure Nested Paging Firmware ABI Specification," January 2022.

[15] ARM, "mbed TLS," 2020. [Online]. Available: https:///tls.mbed.org

[16] ——, "Arm Architecture Reference Manual for A-profile architecture," Feb 2022.

[17] J. Beulich and A. Cooper, "[v3,7/8] x86emul: support RDPRU," Sep 2019. [Online]. Available: https://patchwork.kernel.org/project/xen-devel/patch/1fc41c75-7e6d-5a34-c500-8f769e4374bb@suse.com/

[18] A. Bhattacharyya, A. Sandulescu, M. Neugschwandt ner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[19] I. Cutress, "Investigating Performance of Multi-Threading on Zen 3 and AMD Ryzen 5000," 2020. [Online]. Available: https://www.anandtech.com/show/16261/investigating-performance-of-multithreading-on-zen-3-and-amd-ryzen-5000/2

[20] J. Edge, "Disallowing perf_event_open()," 2016. [Online]. Available: https://lwn.net/Articles/696216/

[21] D. Faggioli, "Re: [RFC PATCH v3 00/16] Core scheduling v3," 2019. [Online]. Available: https://lore.kernel.org/lkml/277737d6034b3da072d3b0b808d2fa6e110038b0.camel@suse.com/

[22] A. Fog, "The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers," 2021. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

[23] A. Fogh, "Covert Shotgun: automatically finding SMT covert channels," 2016. [Online]. Available: https://cyber.wtf/2016/09/27/covert-shotgun/

[24] Github, "Systemd TODOs," 2022. [Online]. Available: https://github.com/systemd/systemd/blob/aaec2216602ce3a26b7bca30eaf28e525ef5e762/TODO#L1272

[25] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium*, 2018.

[26] M. S. Inci, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "Co-location detection on the cloud," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2016, pp. 19–34.

[27] Intel, "Developer Reference for Intel Integrated Performance Primitives Cryptography," 2019. [Online]. Available: https://software.intel.com/en-us/ipp-crypto-reference

[28] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2019.

[29] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z," 2019.

[30] D. Johnson, "And a little more... big changes to the LDQ/STQ sizes, and new 'coalescing retire queue' theory and sizes," Mar 2021. [Online]. Available: https://twitter.com/dougallj/status/1373973478731255812

[31] M. Kettenis, "CVS: cvs.openbsd.org: src," 2018. [Online]. Available: https://www.mail-archive.com/source-changes@openbsd.org/msg99141.html

[32] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[33] M. Larabel, "Intel Hyper Threading Performance With A Core i7 On Ubuntu 18.04 LTS," June 2018. [Online]. Available: https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4

[34] Linux Kernel Documentation, "Core Scheduling," 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html

[35] M. Lipp, D. Gruss, and M. Schwarz, "AMD Prefetch Attacks through Power and Time," in *USENIX Security Symposium*, 2022.

[36] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors," in *AsiaCCS*, 2020.

[37] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *S&P*, 2021.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[39] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[40] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-Cores Cache Covert Channel," in *DIMVA*, 2015.

[41] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.

[42] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, Oct 1996. [Online]. Available: https://cacr.uwaterloo.ca/hac/

[43] Microsoft, "Managing Hyper-V hypervisor scheduler types," 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types

[44] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack against Constant-Time Crypto Implementations in SGX," in *CT-RSA*, 2018.

[45] OpenSSL, "OpenSSL: The Open Source toolkit for SSL/TLS," 2019. [Online]. Available: http://www.openssl.org

[46] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[47] J. K. Ousterhout *et al.*, "Scheduling Techniques for Concurrent Systems," in *ICDCS*, 1982.

[48] C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.

[49] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[50] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend," in *USENIX Security Symposium*, 2021.

[51] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks," in *USENIX Security Symposium*, 2021.

[52] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *CCS*, 2009.

[53] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port Contention Goes Portable: Port Contention Side Channels in Web Browsers," in *AsiaCCS*, 2022.

[54] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey, "Evaluating the Impact of Simultaneous Multithreading on Network Servers Using Real Hardware," in *SIGMETRICS*, 2005.

[55] M. Rybczyńska, "A full task-isolation mode for the kernel," Apr 2020. [Online]. Available: https://lwn.net/Articles/816298/

[56] P. G. Sassone, J. Rupley, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix Scheduler Reloaded," in *ISCA*, 2007.

[57] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[58] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.

[59] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: abusing Intel SGX to conceal cache attacks," *Cybersecurity*, vol. 3, no. 1, p. 2, 2020.

[60] Standard Performance Evaluation Corporation, "SPECWeb99," 2008. [Online]. Available: https://www.spec.org/web99/

[61] D. Sullivan, O. Arias, T. Meade, and Y. Jin, "Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds," in *NDSS*, 2018.

[62] M. Taram, X. Ren, A. Venkat, and D. Tullsen, "SecSMT: Securing SMT processors against Contention-Based covert channels," in *USENIX Security Symposium*, Aug 2022.

[63] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, 2019.

[64] Z. Wang and R. B. Lee, "Covert and Side Channels due to Processor Architecture," in *ACSAC*, 2006.

[65] H. Wong, "Measuring Reorder Buffer Capacity," 2013. [Online]. Available: http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/

[66] H. Wong, V. Betz, and J. Rose, "High-Performance Instruction Scheduling Circuits for Superscalar Out-of-Order Soft Processors," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 1, 2018.

[67] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud," in *USENIX Security Symposium*, 2012.

[68] Xen Project, "Xen Project 4.16.0 Archives," Dec 2021. [Online]. Available: https://xenproject.org/downloads/xen-project-archives/xen-project-4-16-series/xen-project-4-16-0/

[69] Y. Xiao, Y. Zhang, and R. Teodorescu, "SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities," in *NDSS*, 2020.

[70] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *CCSW*, 2011.

[71] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *S&P*, 2019.

[72] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[73] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *JCEN*, 2017.

Appendix A. Investigating Potential Interference from `rdpru` Instructions

To rule out any interference from the `rdpru` instructions on the observed scheduler queue capacity, we repeated the performance counter measurement from Section 3.1.1 with the highlighted lines in Listing 1 enabled, shown by the red curve in Figure 3. We see a shallower increase in stalled cycles between 20 and 22 multiplications because another instruction is sometimes occupying the ALU1 scheduler. As the `mov` instruction (line 19) is fully handled by a register rename [9], [10], [22], it is not enqueued into the scheduler queue. Thus, we conclude that we see `rdpru` occupying the ALU1 scheduler. The largest increase is between 21 and 22 and not between 20 and 21 multiplications, indicating that `rdpru` does not strictly require ALU1 for execution but can also use other ALUs.

To substantiate this finding, we increased the latency of the delayed multiplications by increasing the number of `sqrtsd` instructions in lines 13 to 16 from 3 to 6, extending the contention window using the same approach as described in Section 4.4. With this, the brown curve in Figure 3 again shows the steep increase between 22 and 23 multiplications, while, between 16 and 22 multiplications, the curve looks identical to the red curve with only 3 `sqrtsd` instructions, including the minor increase between 20 and 22 multiplications. The signal from the multiplications now clearly surmounts the increase caused by the `rdpru` instruction, on which the additional `sqrtsd` instructions have no effect. This is because `rdpru` executes immediately before the multiplications, since, in contrast to them, it does not have any long-latency dependencies. Consequently, even if the first `rdpru` runs on ALU1, this does not affect the measured capacity, as its scheduler queue entry is freed when `rdpru` finishes, making it available for another multiplication.

Appendix B. Contention on Different Scheduler Queues

Using the approach from Section 3.1.2, other scheduler queues can be targeted as well: On Zen 3, we replaced the

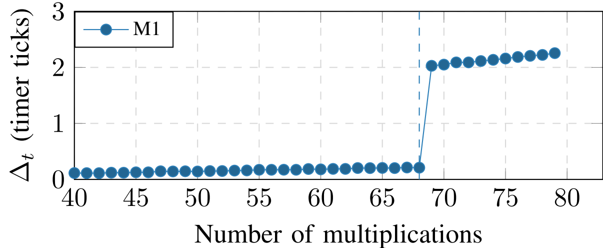


Figure 8: Average timing differences (n = 100000) on the Apple M1 for different lengths of the multiplication block.

dependent multiplications with dependent divisions. Zen 3 exclusively executes divisions on ALU0 [10], so we target a single scheduler queue different from the queue associated with ALU1. With this, we observed a limit of 22 dependent divisions, showing that the scheduler queues for ALU0 and ALU1 have the same length. We also targeted the scheduler queues of ALU0 and ALU1 on Zen 3 in parallel, using an alternating sequence of `mul`, `div`, `mul`, `div`, etc. We observed contention as soon as we added one more `mul` or `div` to a chain of 44 instructions (22 `mul`, 22 `div`), exceeding the capacity of one of the two queues. For comparison, Zen 3 executes `add` instructions on all of its four ALUs. In an additional experiment with a chain of dependent additions, we observed a total scheduler queue capacity of 88 entries (22 per queue) for all the four ALUs.

To summarize, this shows that we are hitting a capacity limit that is dependent on the capabilities of the ALUs (unlike ROB contention). Using dependency chains and unprivileged timing measurements, we can induce and observe back-end stalls caused by contention on scheduler queues. By measuring contention on a single scheduler queue, we can observe execution of instructions that specifically require the associated ALU.

Appendix C. Characterizing Timing Difference on Apple M1

To demonstrate that our reverse-engineering approach can be generically applied to any architecture, given a non-serializing instruction to read a sufficiently precise timestamp, we applied it to Apple's M1 CPU that implements the ARM AArch64 instruction set. We ported the code from Listing 1 to ARM AArch64 using the `CNTVCT_EL0` register as a time source. The reference manual explicitly states that reads from this register are executed out of order [16]. One challenge on the Apple M1 is the low frequency of the `CNTVCT_EL0` timer, which on our system is only 24 MHz. To compensate for this, we increase the number of `fsqrt` instructions to 20 to extend the time until a back-end stall, caused by contention, is resolved. With this, we see a steep increase in the average timing difference Δ_t over 100 000 experiments when we increase the number of multiplications from 68 to 69 (see Appendix C). This is in line with previous reverse-engineering [30] of the Apple M1 high-performance cores (called "Firestorm"). The Firestorm cores have two multiplication-handling ALUs, each with a separate 28-entry scheduler [30]. Additionally, in contrast to Zen 2 and Zen 3, the Apple M1 has additional dispatch buffers before the scheduler queues. In our concrete case, a 12-entry dispatch buffer is placed in front of the multiplication ALU scheduler queues [30]. Hence, we observe a back-end stall when we exceed the total capacity of that dispatch buffer (12 entries) and of the two multiplication ALU schedulers (each 28, *i.e.*, 56 entries), *i.e.*, when we exceed 68 multiplications in our measurement code. This shows that scheduler queue contention is also observable on the M1, using the same approach.

Appendix D. Comparison of the SQUIP Covert Channel with State-Of-The-Art Cross-VM Covert Channels

The following table compares the bit rate of our SQUIP covert channel in the cross-VM setting with other state-of-the-art cross-VM covert channels (adapted from Schwarz et al. [57]):

TABLE 2: Comparison of state-of-the-art cross-VM covert channels, sorted by the true channel capacity (CC - works cross-core) each work reported.

Covert channel	Raw Capacity	Error Rate	True Capacity	CC
Sullivan et al. [61]	1.49 Mbit/s	8.7 %	854.7 kbit/s	yes
SQUIP (this work)	892.2 kbit/s	0.75 %	835.3 kbit/s	no
Van Schaik et al. [63]	608 kbit/s	0 %	608 kbit/s	no
Liu et al. [39]	600 kbit/s	1 %	551.5 kbit/s	yes
Maurice et al. [40]	751.2 bit/s	5.7 %	514.3 kbit/s	yes
Maurice et al. [41]	378.72 kbit/s	0 %	378.72 kbit/s	yes
Pessl et al. [49]	411 kbit/s	4.11 %	309.4 kbit/s	yes
Schwarz et al. [57]	26.8 kbit/s	0 %	4.3 kbit/s	no
Wu et al. [67]	746.8 bit/s	0.09 %	739 bit/s	yes
Xu et al. [70]	215 bit/s	5.12 %	152 bit/s	no
Schwarz et al. [58]	11 bit/s	0 %	11 bit/s	yes
Ristenpart et al. [52]	0.2 bit/s	0 %	0.2 bit/s	no

Appendix E. Attacked Code Sequence in mbedTLS

The following listing shows the attacked code sequence in the modular exponentiation implementation of mbedTLS:

```
1  int mbedtls_mpi_exp_mod(/*...*/)
2  {
3    //...
4    while( 1 )
5    {
6      if( bufsize == 0 )
7      {
8        if( nblimbs == 0 )
9          break;
10       nblimbs--;
11       bufsize = 64;
12     }
13
14     bufsize--;
15     ei = (E->p[nblimbs] >> bufsize) & 1;
```

```
16
17      if( ei == 0 && state == 0 )
18        continue;       //skip leading 0s
19      if( ei == 0 && state == 1 )
20      {
21        //only square!
22        mpi_montmul( X, X, N, mm, &T );
23        continue;
24      }
25
26      state = 2;
27      nbits++;
28      wbits |= ei;
29
30      //square...
31      mpi_montmul( X, X, N, mm, &T );
32
33      mpi_select( &WW, W, 2, wbits );
34      //...and multiply
35      mpi_montmul( X, &WW, N, mm, &T );  //!!!
36
37      state--;
38      nbits = 0;
39      wbits = 0;
40    }
41    //...
42  }
```

In line 35, an additional call to `mpi_montmul` (`multiply`) is performed for each '1' in the exponent, resulting in clearly distinguishable peaks in our contention profiles.

Appendix F. Key Extraction Parameters

TABLE 3: The parameters used for the optimization of the key extraction. Parameters in the trace scope, apply to all sections of the trace. The parameters in the section scope are different for every section.

Scope	Parameter
Trace	Window size N of the smoothing average
	Window size N_m of a second larger average
	Shift of the larger average
	Factor multiplied with larger average before subtracting it from the smoothed average [0-1]
	Factor multiplied with the distance between two peaks, before computing the number of 0s in between
Section	Length of the section
	Correction factor for the length of a 1
	Correction factor for the length of a 0
	`find_peaks` height argument
	`find_peaks` prominence argument
	correction factor for the `find_peaks` distance argument