Remote Scheduler Contention Attacks

Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss

Graz University of Technology, Austria

Abstract. In this paper, we investigate unexplored aspects of scheduler contention: We systematically study the leakage of all scheduler queues on AMD Zen 3 and show that all queues leak. We mount the first scheduler contention attacks on Zen 4, with a novel measurement method evoking an out-of-order race condition, more precise than the state of the art. We demonstrate the first inter-keystroke timing attacks based on scheduler contention, with an F_1 score of $\geq$99.5 % and a standard deviation below 4 ms from the ground truth. Our end-to-end JavaScript attack transmits across Firefox instances, bypassing cross-origin policies and site isolation, with 891.9 bit/s (Zen 3) and 940.7 bit/s (Zen 4).

1 Introduction

Modern CPUs execute micro-operations (µops) out-of-order to improve performance. To select which µops to execute next, modern CPUs have one or more schedulers [22,4,5,8]. Gast et al. [16] showed that an attacker-controlled binary, executed natively on the machine, can exploit the integer multiplication scheduler queue on AMD CPUs to leak cryptographic keys.

Side-channel attacks mounted via the browser are considered more dangerous, as a victim only has to visit a malicious website [29]. However, it is unclear if scheduler contention attacks are possible from JavaScript. The shown native code attacks focus on highly repetitive events, such as covert channels and encryptions. This raises the question whether singular low frequency events can also be monitored using scheduler contention. Additionally, the attack surface of other schedulers than the integer multiplication scheduler is unexplored. Furthermore, although AMD acknowledged the security issues [7] reported by Gast et al. [16], it is unclear whether AMD Zen 4 is also affected.

In this paper, we systematically study scheduler-contention side channels in both native and JavaScript contexts, using AMD's Zen 3 and Zen 4 architectures:
RQ1: What distinguishes schedulers, e.g., priming instructions or queue sizes?
RQ2: Which measurement methods are practical on recent processors?
RQ3: Which scheduler queues leak non-repeatable events, e.g., keystrokes?
RQ4: Can a remote attacker exploit scheduler contention using JavaScript?
We systematically address these overarching research questions:

For **RQ1**, we determine generic requirements to prime arbitrary schedulers, resulting in effective priming sequences for all execution units on Zen 3. On Zen

Table 1. Comparison of browser-based covert channels

Covert Channel	Raw Capacity[1]	Error Rate	True Capacity	
Prime+Probe [29][2]	320 kbit/s	-	-	✗
Our work	1 000 bit/s	0.69 %	940.7 bit/s	✓
Port contention [34]	200 bit/s	1 %	184 bit/s	✓
Event loop [45]	200 bit/s	-	-	✗
Hardware interrupts [24]	25 bit/s	-	-	✓
DRAM [38]	11 bit/s	0 %	11 bit/s	✓
RIDL (Evict+Reload) [36]	8 bit/s	-	-	✗
Disk contention [44]	0.5 bit/s	0 %	0.5 bit/s	✓
Memory throttling [35]	0.2 bit/s	0 %	0.2 bit/s	✓

The last column indicates an evaluation with current mitigations.
[1] Sorted by the bandwidth each work reported (cf. [34]).
[2] The work predates current mitigations, including heavy countermeasures against timing attacks. If reimplemented it will likely yield a much lower bandwidth.

4, we found entirely reimplemented instructions, diverging from the behavior on Zen 3 and, based on reverse-engineering that contradicts AMD's documentation.

For **RQ2**, we evaluate bingo race, a timingless, more precise, out-of-order race condition based method to measure scheduler contention. This method yields the correct scheduler 1 capacity on both Zen 3 and Zen 4, in contrast to prior work.

For **RQ3**, we present keystroke-timing attacks on all integer scheduler queues. We leak password keystrokes on a login screen with an F_1 score of $\geq$99.5 %. For the correctly detected keystrokes, the standard deviation from the ground truth is below 4 ms, confirming the high precision of our attack.

Finally, for **RQ4**, we present a 1 kbit/s JavaScript scheduler contention covert channel in Firefox with an error rate of less than 1.5 % (Zen 3) and 0.7 % (Zen 4). This results in a true capacity of 891.9 bit/s (Zen 3) and 940.7 bit/s (Zen 4). Our attack works across websites, with sender and receiver running in different browser windows, bypassing cross-origin policies and site isolation.

In summary, we make the following contributions:

1. We systematically analyze contention on each scheduler on Zen 3 and 4, revealing effective priming sequences for all of them.
2. We present bingo race, a novel timingless and more accurate measurement method based on out-of-order memory reads.
3. We show that contention attacks on each integer scheduler can observe singular events like inter-keystroke timings of a password entry, with F_1 scores $\geq$99.5 % and a standard deviation below 4 ms from the ground-truth timings.
4. We present a scheduler contention covert channel purely in JavaScript running in Firefox 114, with a true capacity of 940.7 bit/s ($n = 10$, $\sigma_{\bar{x}} = 5.15$ bit/s), bypassing cross-origin policies and site isolation.

Responsible Disclosure. We reported our findings to AMD on August 8th, 2023. They acknowledged our findings on August 16th, 2023.

Outline. We present background in Section 2, a systematic scheduler queue analysis in Section 3, and our bingo race measurement in Section 4. We present a keystroke timing attack in Section 5, a JavaScript covert channel in Section 6, and discuss mitigations and related work in Section 7. Section 8 concludes.

2 Background

In this section, we provide background on out-of-order execution, scheduler contention attacks, and timing measurements in JavaScript.

Superscalar CPUs. Superscalar CPUs increase the number of instructions per clock cycle through parallel processing [15,8]. The CPU frontend decodes fetched instructions into micro-ops (µops) and stores them in the retire control unit (RCU). The backend has multiple execution units, such as arithmetic and logic units (ALUs), branch execution units (BRUs) and address generation units (AGUs). Execution units have a scheduler, tracking operand dependencies to determine µops ready for execution. Finally, the out-of-order executed µops retire in instruction stream order, making their results visible in the architectural state.

Execution Unit Schedulers. A CPU can have one scheduler, such as Intel CPUs [20], or multiple schedulers, such as AMD [4,5] and Apple CPUs [22]. AMD Zen 2 CPUs have separate schedulers for each ALU and a dedicated scheduler for all AGUs [4]. Zen 3 [5] and 4 [8] have schedulers for pairs of ALU and AGU, or ALU and BRU. Each scheduler has a queue buffering µops until the execution unit and the input operands are ready. On Zen 3, the integer execution unit schedulers have a capacity of 24 µops [5,16]. Scheduler contention refers to the situation when a µop is about to be enqueued into an already full queue [16].

Simultaneous Multithreading (SMT). Efficiency is maximized if all execution units are in use. This is usually not the case with a single instruction stream. Therefore, many modern CPUs execute multiple instruction streams (2 on AMD Zen CPUs) simultaneously on the same core, sharing the L1 and L2 cache, execution units and schedulers. This sharing enables various side channels [42]. Scheduler contention attacks exploit the shared scheduler queues [16]. Different hardware resources are partitioned in different ways: competitively, each thread can fully use a resource; watermarking, a small fraction is reserved for each thread; static partitioning, each thread can only use their fixed fraction.

Scheduler Contention Attacks. Gast et al. [16] have shown that contention on scheduler queues measurably delays program execution. A full or almost full queue causes a back-end stall, delaying subsequent µops. Their attack on Zen 2 and 3 exploited that unprivileged `rdpru` timer reads are executed out-of-order, unless the back-end stalls. As the schedulers are shared between SMT threads, it is possible to observe multiplications of a co-located program. They demonstrate a covert channel and an attack on square-and-multiply RSA, both in native code. They also show how to fill (*i.e.*, prime) the scheduler used for divisions. However, they did not investigate all scheduler queues nor JavaScript attacks.

JavaScript. JavaScript is a just-in-time compiled scripting language for the web that operates within a strict sandbox. JavaScript code cannot access high-resolution timers via `rdtsc` or similar instructions. It instead relies on the High Resolution Time API [46]. To address security concerns associated with timing attacks [29,48], all major browsers reduced the frequency to 200 kHz [48,11,10]. However, side channel attacks were still demonstrated [23,38,34,32]. JavaScript supports multithreading through web workers [28], typically using shared array buffers, representing shared memory [25], for communication between threads.

3 Systematic Analysis of Zen 3 and 4 Scheduler Queues

Prior work [16] showed that scheduler queues 0 and 1 can be primed with divisions and multiplications, respectively. To answer **RQ1**, we explore priming instructions for integer execution-unit schedulers 2 and 3, as well as the floating point schedulers. Thus, our analysis fills the gap to a complete coverage of all schedulers on Zen 3 and 4. Zen 4 has a similar scheduler design to Zen 3, also regarding capacities and execution unit connections [5,8]. Thus, we expect similar scheduler queue sizes and usages for Zen 3 and Zen 4.

For each scheduler queue, we need to find a *priming* instruction suitable to fill it. This instruction must be *delayable*, *targeted*, *single-queue*, *non-serializing*, *unprivileged*, and preferably *single-µop*: The instruction has to be *delayable* by a long-latency operation to maintain the desired queue occupancy level. To target a specific scheduler, the instruction must be decoded to a µop that can only be executed on one specific execution unit (*targeted*). Additionally, the instruction must not cause contention on other queues (*single-queue*). Furthermore, the instruction must be *non-serializing*, as observing scheduler contention relies on out-of-order execution. The instruction must be *unprivileged* to allow exploitation from user space. Finally, for precise control over the occupancy level of the target queue, the instruction must not have more than one input-dependent µop for the targeted scheduler (*single-µop*). If decoded into multiple µops, the other µops must either be independent or executable by more than one execution unit.

We compose a set of candidate instructions from a complete x86 instruction table by eliminating instructions that do not fulfill all requirements in two phases:

In the **first** phase, we select a set of candidate instructions that are *delayable*, *non-serializing*, and *unprivileged* from AMD's instruction latency table [5] and uops.info [1]. We initially focus on instructions that are explicitly documented to require the execution unit that is connected to the targeted scheduler. However, for many instructions the used execution unit is undocumented, e.g., `bsf`, `bsr`, and the majority of microcoded instructions. Furthermore, as shown later, there are some discrepancies between the documented execution unit usage and our measurement results, indicating inaccurate or outdated documentation. Therefore, if we do not find a suitable instruction in the first phase, we extend our search to instructions without documentation about the execution units used. We start with instructions that are only decoded into a few µops, as we expect these to allow us to control the occupancy level more precisely (*single-µop*).

In the **second** phase, we check each candidate instruction whether it is *targeted*, *single-µop*, and *single-queue*, by replacing the multiplications in Figure 1 with a varying number of repetitions of the chosen instruction. We monitor which scheduler queues are affected by contention via the performance counters `IntSch[0-3]TokenStall` and `FPSchRsrcStall`. To obtain the exact number of repetitions k required to fill the target queue, we introduce a precise contention measurement approach using an out-of-order read from a bingo variable in Section 4.1. If we observe k to be more than the capacity (24 µops [6]) of a single scheduler queue, we conclude the instruction is not *targeted*. If we observe k to be significantly less than the capacity of a single scheduler queue, we conclude the

instruction is not *single-µop*. If we observe multiple performance counters to be increasing with an increased k, we conclude the instruction is not *single-queue*.

For the sake of brevity, we do not go into detail for integer scheduler queues 0 and 1, as instruction sequences for these were already documented by Gast et al. [16]. In the following, we discuss the results for the remaining scheduler queues.

3.1 Scheduler Queue 2

While AMD's instruction latency table [5] lists several instructions that are executed on ALU2, we found the microcoded `stosb` to be the best candidate.

Instructions moving data from a general purpose to a floating point register (e.g., `movd`, `vmovd` and `cvtsi2sd`) are documented to be executed on ALU2. However, contrary to the documentation, our measurements clearly show that they use ALU0 instead. The instruction latency table also states that bit shift and bit rotation operations, e.g., `rol` or `shl`, can be scheduled on ALU2 and ALU1, violating our *single-queue* requirement, as we also experimentally confirmed.

We tested microcoded instructions and found `stosb`, with its implicit `al` input operand, to cause contention on scheduler 2. We measure [2] that `stosb` is decoded into 3 µops: One µop is always enqueued into scheduler queue 2, but not the other 2 µops. We observe k to be 22 (see Section 4 for why it is not the documented 24), showing that we can precisely prime scheduler queue 2.

3.2 Scheduler Queue 3

The instruction latency table does not report any instructions that exclusively use ALU3. Our search again resulted in a microcoded instruction, `lodsb`, which loads `[rsi]` into the `al` register, then increments `rsi`. The `loadsb` instruction is *delayable*, as it performs a partial register load. When a byte is loaded into the `al` register, *i.e.*, bits 0–7 of the `rax` register, the remaining bits of `rax` retain their previous values, creating a dependency on `rax` [15]. According to nanoBench [2], `lodsb` is decoded into 4 µops. Our measurements show that one µop is always enqueued into scheduler 3 while the other 3 µops are scheduled to other execution units. k is again 22, showing that we can also precisely prime scheduler 3.

We obtain identical results for `lodsw`, `lodsd` and `lodsq`, loading 16, 32 and 64 bit into `ax`, `eax` and `rax`, respectively. This is surprising for the 32 and 64 bit variants, `lodsd` and `lodsq`, as they completely overwrite `rax`. Thus, these instructions likely have a false dependency.

We also evaluated the non-microcoded `bsf` and `bsr` instructions that we found to be executed on ALU3 on Zen 3. Our measurements show that they indeed occupy scheduler 3, however k is only 7. With a scheduler capacity of 24 µops [6], this indicates that each of these instructions generates 3 µops for ALU3, violating our *single-µop* requirement. In contrast, on Zen 4 the two instructions appear to be decoded into only a single µop that can be executed on every of the four integer ALUs, violating the *targeted* requirement.

3.3 Observing Scheduler Contention on the FPU

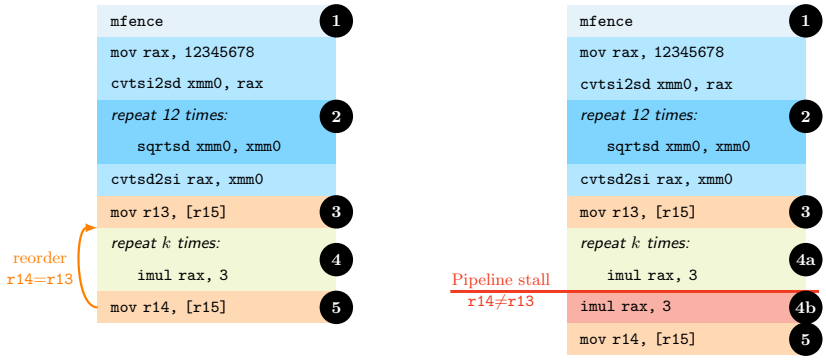
In addition to the integer execution unit schedulers, the floating point unit (FPU) schedulers of Zen 3 and 4 can also be primed. The FPU has 2 schedulers with 32 entries each [5,8]. Each scheduler is connected to 3 execution units. In contrast to the integer execution unit, the FPU has an additional 64 µops non-scheduling queue, buffering µops between the RCU and the FPU schedulers [5]. Hence, we expect to observe stalls when exceeding $64 + 2 \cdot 32 = 128$ repetitions of a priming instruction that can be enqueued into both schedulers.

When priming the FPU schedulers, we can no longer use `sqrtsd` for the delaying dependency chain, as it is executed on the FPU and influences the measurement. We move it to the integer unit, by chaining 18 `div` instructions and converting the final result to floating point, using `cvtsi2sd`. According to AMD [5], `cvtsi2sd` is decoded into 2 µops with one of them utilizing the FPU, reducing the observable capacity by one. We verify that we can prime both FPU scheduler queues and the non-scheduling queue, using `vaddsd` as the priming instruction, which targets both FPU schedulers. With this, we observe back-end stalls with $k > 127$, showing that we can fill the queues exactly to their capacity.

To target a single FPU scheduler, we search for an instruction that can only be executed by a single execution unit. According to AMD [5], almost all operations can be handled by 2 or 4 execution units, which are evenly distributed across the schedulers. The only documented exceptions are `divsd` and `sqrtsd`. Hence, we expect to observe a lower k but again measure a capacity of $k = 127$. This indicates that these operations are executed on multiple execution units, connected to both schedulers, contrary to the documentation. A possible explanation is that both schedulers have a uniform set of FPU execution units. Therefore, each FPU scheduler can handle all FPU µops, making priming a single scheduler impossible. As demonstrated, it is however possible to prime and probe both schedulers together.

4 The Accuracy of the Measurement

With this section we address **RQ2**. Our initial experiments show that the two measurements methods, performance counters and non-serialized hardware timer reads, described by Gast et al. [16] yield imprecise results. Their measurements with both methods show a scheduler capacity of 22 instead of the documented 24 µops. Our hypothesis is that the complex `rdpru` and `rdtsc` instructions use multiple scheduler entries for themselves, influencing the measurement. Additionally, our JavaScript covert channel in Section 6 requires a measurement method that only uses instructions emitted by the JavaScript JIT compiler. We therefore develop a novel method, using a race condition between a read from a bingo variable and the targeted scheduler queue. With this method we measure the exact scheduler 1 queue size of 24 entries on Zen 3 and Zen 4.



(a) Scheduler capacity k not exceeded: The second bingo read ❺ is reordered and retrieves the same bingo number as the first read ❸.

(b) Scheduler capacity k exceeded: Contention causes a pipeline stall, hence the second read ❺ is not reordered and retrieves a different bingo number than the first read ❸.

Fig. 1. Measuring scheduler contention with a bingo race. After draining the pipeline ❶, we fill the scheduler queue with repetitions of the priming instruction ❹, delayed by a high-latency input operand dependency chain ❷. The bingo variable at `[r15]` is constantly updated by the bingo thread on another core. If the pipeline stalls due to scheduler contention, `r14` will contain a different value than `r13`.

4.1 Measurements using an Out-of-Order Bingo Race Condition

Gast et al. [16] use timer reads to detect if the CPU executes instructions out of order. We show that this is also possible with a bingo variable that is modified by a thread running in parallel and that this yields a more precise result.[1]

In Figure 1, we replace the hardware timer reads (*i.e.*, `rdpru`) with loads from a bingo variable at a specific memory address (in `r15`). The bingo thread constantly updates this variable with new bingo numbers. We have no requirements on the bingo numbers except for them to be frequently updated, without frequent value repetitions, e.g., a (pseudo-random) number sequence. In contrast to the timer reads [16], the `mov` instruction to read the bingo variable is decoded into only a single µop [5], minimizing its influence on the measurement while still subjected to out-of-order execution. The frequent updates keep the bingo variable in the L1 cache, preventing delays from memory accesses.

The second bingo access is only executed out-of-order when there is no contention on the targeted scheduler queue. Thus, contention is detected based on whether the second read of the bingo variable is executed out-of-order or not. If it is executed out-of-order, the second read of the bingo variable will retrieve the

[1] Our approach is named after the game Bingo, where numbers are openly announced, and matching numbers have a special meaning. Likewise, in our case, matching bingo numbers have a special meaning, namely that the scheduler queue was not full.

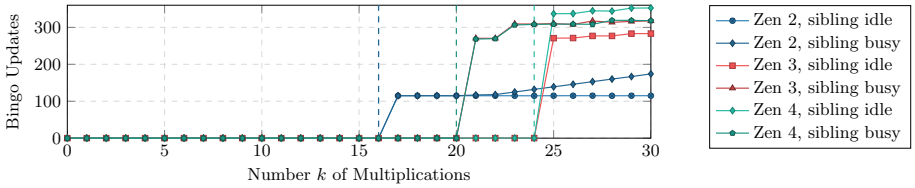


Fig. 2. Average load delay for different lengths k of the multiplication block for Zen 2, 3 and 4; with sibling thread being busy or idle. ($n = 100\,000$).

exact same bingo number, otherwise, it will retrieve a new, different bingo number. In other words, when the queue is *full* and the bingo variable read is about to be enqueued, the pipeline stalls and subsequent µops are not reordered, *i.e.*, they do not reach their respective scheduler queues and thus the second bingo variable read does not succeed out of order. With all measurement methods, there can be spurious stalls, which we investigate further in Appendix B.

4.2 Evaluation

We evaluate our bingo race measurement on an AMD Ryzen 7 3700X CPU (Zen 2), an AMD Zen 3 Ryzen 7 5800X CPU (Zen 3), and an AMD Ryzen 7 7700X CPU (Zen 4) with a varying number of multiplications k. For each k, we repeat the measurement 100 000 times, and track the average number of updates of the bingo number. We avoid interference between the bingo thread and the measurement thread by pinning each of them to separate cores.

Figure 2 shows the increased precision of our method. On Zen 2, we measure a queue capacity of 16 µops for scheduler 1, exactly matching the documentation [4] and previous work [16]. On Zen 3 and Zen 4, we measure a capacity of 24 µops, exactly matching the capacity published by AMD [6] and two µops more than reported by Gast et al. [16]. We further analyze this difference in Appendix B. Running an empty loop on the other hardware thread of the same core yields a reduced capacity of 20 µops on Zen 3 and Zen 4, whereas there is no reduction on Zen 2. This reveals the watermark mechanism on Zen 3 and Zen 4, in line with previous work [16]. Our results for more instructions are shown in Table 3.

5 Detecting User Behavior via Scheduler Contention

In this section, we demonstrate detection of singular non-repeatable events via scheduler contention, addressing **RQ3**. We describe an attack to recover inter-keystroke timings from the X server and evaluate it on all integer execution unit schedulers, also extending our insights towards **RQ1**. Our evaluation shows successful keystroke spying, with an F_1 score $\geq$99.5 %. We observe that the average deviation from the ground truth for correctly detected inter keystroke-timings is below 4 ms, indicating the high level of precision in our attack.

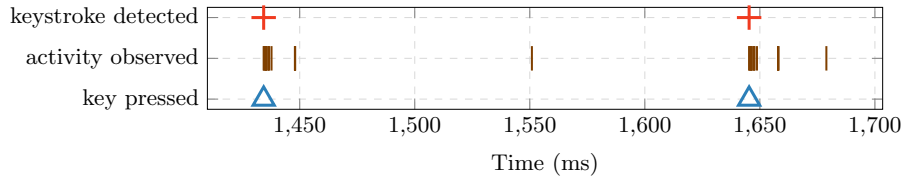


Fig. 3. Keystroke (△) timings observed via scheduler 3 by a co-located observer due to the watermark mechanism (|||). Multiple samples with reduced scheduler capacity are clustered and filtered, resulting in a clear recovered keystroke signal (+).

5.1 Threat Model and Experimental Setup

The attacker wants to recover inter-keystroke timings from a user entering their password into Gnome Display Manager (gdm3, the default login manager on Ubuntu) on a multi-user machine. We assume that the attacker can execute native code on the target machine and has achieved co-location to the X server, *i.e.*, attack code and X server run on the same core but on different hardware threads. We evaluate our attack on an AMD Ryzen 7 5800X CPU (Zen 3) running Ubuntu 20.04.6 LTS with X server 1.20.13 and gdm3 3.36.3.

5.2 Inter-Keystroke Timing Attack

Our attack exploits the watermark mechanism of the scheduler queues to detect bursts of high activity of the co-located X server. The attack consists of an observation phase and a postprocessing phase. First, the observer process repeatedly samples any of the scheduler queues to infer if the X server is active, recording timestamps each time it detects activity. In the postprocessing phase, we group the timestamps to obtain the start time of each activity burst.

Observation phase. The observer process is co-located to the X server, which provides graphical services and handles user input. User input, such as pressing a key, results in an activity burst. Conversely, without user input, the X server remains predominantly idle, causing only low activity. Since the scheduler queues are shared, an activity burst of the X server can be detected by the observer.

The observer continuously checks whether it can exceed the watermark limit on the targeted scheduler queue without causing a back-end stall. If the available capacity is *above* the watermark limit, the X server is currently *idle*. If the available capacity is *below* the watermark limit, this indicates that the X server is *busy* performing tasks like updating the display, handling mouse movements, or, notably, processing *keystrokes*. In these instances, the observer records a timestamp, resulting in a chronological list when the X server was active.

Postprocessing. A single keystroke triggers X server activity over several sampling intervals, causing the observer to record multiple timestamps, as illustrated in Figure 3. The postprocessing phase filters these timestamps in two steps: In the first step, we group them into clusters. We consider two subsequent timestamps to be part of the same cluster if their difference is below or equal a threshold t_{idle}.

Table 2. Inter-keystroke Timing Attack Evaluation

	Scheduler 0	Scheduler 1	Scheduler 2	Scheduler 3
Recorded Keystrokes	100	100	100	100
False negatives	0	0	1	0
False positives	1	0	0	0
F_1 score	0.995	1.000	0.995	1.000
Standard deviation (ms)	2.555 ($n = 99$)	3.244 ($n = 100$)	2.307 ($n = 99$)	0.901 ($n = 100$)

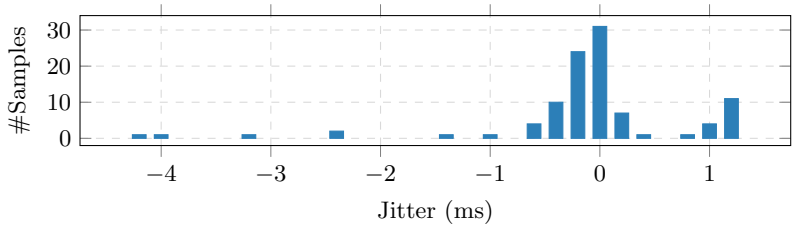


Fig. 4. The jitter of the individual recorded keystrokes (from the aligned ground truth), observed via scheduler 3. The low number of outliers and high concentration around 0 ms shows that inter-keystroke timings are extracted with very high accuracy.

In the second step, we filter out clusters with less than $n_{\min}$ samples, removing noise caused by X server activity unrelated to user input. The first timestamp in each remaining cluster is the assumed point in time of a keystroke.

Figure 3 exemplifies the postprocessing phase, where the observed activity clusters (|||) in the timer intervals 1434-1448 and 1645-1659 are interpreted as keystrokes (+) at timestamps 1434 and 1645, respectively. Conversely, the observed activities at timestamps 1551 and 1679 are filtered out as noise.

5.3 Evaluation

We evaluate our attack on all four integer execution unit schedulers as follows:

Using a programmable USB keyboard emulator, we inject 100 keystrokes into the gdm3 password prompt. Between each pair of keystrokes, we insert a delay between 150 ms and 300 ms, simulating the typing speed of a skilled typist [31]. The keyboard emulator records the actual keystroke timings as the ground truth. We post-process ($t_{\text{idle}} = 0.1$ ms, $n_{\min} = 10$) the recorded timestamps, resulting in a trace of keystroke times. The values for t_{idle} and $n_{\min}$ were found empirically to achieve good filtering for traces recorded on all integer scheduler queues.

For inter-keystroke timing attacks, the exact timing between individual keystrokes is the crucial information. Due to other system activity, the measured inter-keystroke timings can vary slightly, making the attack, and possibly recovered keys, less precise. These slight inaccuracies are called jitter and should be as low as possible. We compute the jitter of our measurement as follows:

We first align the trace with the ground truth by computing the mean squared error between the ground truth and the signal shifted by Δt and minimizing it. We then check for false negatives and false positives against the ground truth

and count them, giving us the numbers in Table 2. The very low numbers of 1 or less are possible due to the filtering with t_{idle} and $n_{\min}$. To compute the jitter, we remove false negatives and positives from the signal. For each of the remaining true positive keystroke timings, we compute the distance to the ground truth. Figure 4 shows the distribution of the jitter of 100 recorded keystrokes. The small number of outliers and high concentration around 0 ms indicate a very low jitter. The high accuracy of our attack is summarized in Table 2. For all scheduler queues, we obtain F_1 scores close to 1, showing reliable keystroke detection. The standard deviation is below 1 ms for scheduler queue 3 and maximum 3.244 ms for scheduler queue 1, indicating very precise timings.

As we attack the login screen, before the user session is started, the only sources of X server activity are password keystrokes and mouse movements. Mouse movements become long bursts in the signal, easily filtered out. While typing, users typically have both hands on the keyboard without using the mouse.

6 JavaScript-based Scheduler Contention Covert Channel

In this section, we show that scheduler contention can exploited from a website, addressing **RQ4**. We design and evaluate a covert channel using JavaScript in Firefox 114, similar to other state-of-the-art works [34,36,38,45,29]. We achieve a raw transmission rate of 1 kbit/s with an error rate of less than 1.8 % in a cross-browser-window setting, bypassing cross-origin policies and site isolation.

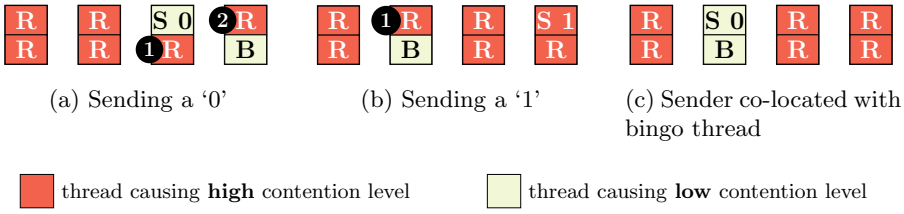
6.1 Threat Model and Experimental Setup

We use a Ryzen 7 5800X (Zen 3) and a 7700X (Zen 4), 8 cores and 16 SMT threads each, with a default Ubuntu 20.04 and Firefox 114. Like other browser-based covert channels [34,36,38,45,29], we assume receiver and sender have no other communication channel, e.g., due to cross-origin policies and site isolation. We make no assumptions on CPU frequency or co-location of sender and receiver.

6.2 Scheduler Contention in JavaScript

Compared to microarchitectural attacks from native code, browser-based attacks have to overcome additional challenges. In particular, to induce and observe scheduler contention across multiple browser instances, these are:

C1 Find a scheduler that can be targeted from JavaScript, with instructions that the JIT compiler produces.
C2 Craft code that generates a tight and non-bloated sequence of priming instructions for the targeted scheduler, *i.e.*, surrounding code must not interfere with other schedulers or prevent the contention of the targeted scheduler.
C3 Measure contention without access to hardware timers.
C4 Achieve co-location between sender and receiver.



(a) Sending a '0' (b) Sending a '1' (c) Sender co-located with bingo thread

Fig. 5. Spawning multiple receiver threads to achieve co-location. Each core has two hardware threads: one occupied by the bingo thread (B) the other by the sender (S). Other hardware threads run receiver threads (R), measuring scheduler contention. The transmitted bit is recovered from the number of threads that have observed a low contention level (a, b), unless the sender is co-located with the bingo thread (c).

We solve each challenge and then combine them to our end-to-end covert channel.

C1. Up to this point, priming a scheduler required fine-tuned assembly code with carefully chosen instructions. JavaScript not providing direct control over the machine code makes priming more difficult, hence the targeted scheduler must be carefully chosen: Scheduler 0 is heavily used by several instructions, increasing the noise and reducing the transmission rate. Scheduler 1 is best primed with `imul`, which is easy to emit from JavaScript via `Math.imul`. While we found instructions to prime scheduler 2 and 3 in Section 3, we have not found these specialized string operations in code emitted by the JIT compiler. The FPU schedulers can only be targeted at once, with a high number of operations, and are also heavily used, as all JavaScript numbers are floating point numbers by default [27]. This leaves scheduler 1 as the most plausible choice, solving **C1**.

C2. We use asm.js (e.g., `|0` or the unary `+`) to tell the JIT compiler to use integer data types [26,19]. These type hints enable the compiler to omit type checks and to directly emit machine instructions suitable for the hinted data type. We prime scheduler 1 with a chain of 20 dependent calls to `Math.imul`, delayed by a chain of 12 calls to `Math.sqrt` (see Appendix D, Listing 1.1). To prevent the compiler from over-optimizing, we pseudo-randomize the initial value for the square roots and one operand of the multiplications. Disassembling the JIT-compiled code shows that our instruction sequence is compiled into a chain of 12 `sqrtsd`, followed by 20 `imul` (see Listing 1.2 in Appendix D).

C3. Gast et al. [16] measured scheduler contention via non-serialized hardware timer reads, which are not available in JavaScript. However, our bingo race approach (Section 4.1) eliminates that requirement, solving **C3**. Like other state-of-the-art microarchitectural attacks from JavaScript [34,38,24,17], this requires a Web Worker with a `SharedArrayBuffer`. The Web Worker constantly updates the bingo variable within the shared memory. The receiver reads the bingo variable using `Atomics.load`. Each read from the bingo variable is JIT-compiled into a single `mov r8d, [rsi+rbp*4]` instruction. This instruction immediately follows the last multiplication, allowing it to be executed out-of-order when the scheduler is not full.

C4. To observe scheduler contention across processes, threads must be co-located on a physical core. As JavaScript cannot control this, prior work [34] ran multiple sender instances. If the number of tasks matches the number of hardware threads, this results in a one-to-one assignment of tasks to individual hardware threads, with inert core affinities. We instead run multiple receiver Web Workers, leaving one for the sender and one for the bingo thread.

Consequently, the majority of the receiver workers is co-located to another receiver, as shown in Figure 5. These receivers observe a high level of scheduler contention, as both receivers continuously prime and probe the shared scheduler queue. Typically, one of the receivers is co-located with the bingo thread and one of them is co-located with the sender. A receiver co-located with the bingo thread observes a low contention level, as the bingo thread does not perform integer multiplications. A receiver co-located with the sender observes either a low or a high contention level, depending on the bit that is transmitted.

However, it is possible that the OS co-locates the sender with the bingo thread, see Figure 5c. In this case, each receiver is co-located to another receiver and, therefore, observes high contention levels. This case is trivial to detect and the attacker can just restart the communication until the threads are distributed in a different way. Hence, this approach solves the last challenge, **C4**.

6.3 Time-sliced Bit Transmission

Our covert channel continuously transmits the same bit over a time slice, with a shared clock for synchronization. We use the coarse-grained `Date.now` for start and end of each time slice but not for any measurements, as `Date.now` has side effects (possibly triggering a serializing system call) that prohibit use during out-of-order execution. The minimum time slice is limited by the resolution of `Date.now`, which is 1 ms on Firefox 114, *i.e.*, the raw transmission rate is 1 kbit/s. To transmit a '0', the sender repeatedly checks `Date.now` to wait for the end of the time slice. To transmit a '1', the sender additionally executes a chain of dependent `Math.imul` calls to cause scheduler contention.

Each receiver continuously measures contention on scheduler 1. Every millisecond, one of the receivers collects and evaluates the results from all receivers. As stated previously, most receivers measure a constantly high contention level, one measures no contention and one the actual signal from the sender, which is the second-lowest average contention level. If this receiver's contention level exceeds a predefined threshold, we consider the retrieved bit a '1' (see Figure 5b); otherwise, we consider the retrieved bit a '0' (see Figure 5a).

6.4 Evaluation

Our covert channel is around 5 times faster than other unmitigated state-of-the-art covert channels in the browser (cf. Table 1). In contrast to many of these works, our covert channel does not rely on high-overhead operations, such as intra-browser message passing [45], cache evictions [29,38,36] or disk accesses [44]. Additionally, the back-end stalls caused by scheduler contention

enable our receiver to easily distinguish contention and non-contention cases, even from within a sandboxed JavaScript environment.

We evaluate our covert channel by transmitting 15 random messages, each 5 000 B in size, each divided into 10 packets. For AMD Zen 3, we detected 17 lost packets, out of 150, due to the sender being co-located with the bingo thread. We achieved a bit error rate of 1.46 % ($n = 15$, $\sigma_{\bar{x}} = 0.14\,\%$). With a raw bandwidth of 1 kbit/s, this results in a true capacity of 891.9 bit/s ($n = 15$, $\sigma_{\bar{x}} = 8.75$ bit/s).

For AMD Zen 4, we transmit 10 random messages, 5 000 B in size, divided into 10 packets. Out of 100 packets, 4 packets were lost due to the sender being co-located with the bingo thread. The average bit error rate is 0.69 % ($n = 10$, $\sigma_{\bar{x}} = 0.08\,\%$), with a true capacity of 940.7 bit/s ($n = 10$, $\sigma_{\bar{x}} = 5.15$ bit/s). Based on our findings, we conclude that JavaScript-based scheduler contention side-channel attacks are practical. Additionally, we highlight that the attack surface created by these attacks is larger than previously anticipated.

7 Discussion

Our results on Zen 4, non-repeatable events, and from JavaScript, show that scheduler contention side channels are more relevant than previously known. Prior work focused on native attacks and non-constant-time RSA code [16]. Consequently, the mitigations they discuss focus primarily on this scenario and do not mitigate the attacks we presented. First, generic constant-time code for, e.g., user input, would incur extreme runtime and latency overheads, rendering systems completely unusable. Second, using a watermark to implement strong non-interference is not trivial, especially with an attacker controlling the instruction stream (via JavaScript). Third, disabling SMT has prohibitive performance overheads [12]. Fourth, co-scheduling and core-scheduling have significant performance overheads [14] in a range of 8 % to 91 %, rendering these software-level mitigations unpractical against our attacks. Using a single scheduler queue, like Intel CPUs, or a symmetric scheduler and execution unit design is also insufficient: The FPU execution units have a symmetric scheduler design and we still were able to fill the entire scheduler queue including the non-scheduling queue; Intel CPUs, in turn, are susceptible to port contention attacks [3,9,34].

Related Work. Oren et al. [29] demonstrated the first microarchitectural side-channel attack in the browser, in the form of a website-fingerprinting attack and a covert channel with a reported raw transmission rate of 320 kbit/s, using Prime+Probe. As the high-resolution timer they used is no longer available on current browsers, several works demonstrated attacks without a high-resolution timer [38,40,39]. Lipp et al. [24] demonstrated interrupt-timing attacks in JavaScript, showcasing website and user fingerprinting, inter-keystroke timing attacks, and a covert channel. Schwarz et al. [38] presented a JavaScript-based covert channel using DRAM row access timing differences. Rokicki et al. [34] constructed a covert channel in WebAssembly based on port contention. They achieved a true capacity of 184 bit/s, making it currently the fastest browser-based covert channel on up-to-date browsers. With a true capacity of 940.7 bit/s,

our covert channel is more than 5 times faster. Other works exploit speculative data loads [36], disk contention [44] or memory throttling [35] from JavaScript, or focus on power, frequency, and temperature side channels [13,43]. In addition to these attacks, targeting specific hardware aspects, side-channel attacks can also target the browser itself [45,47].

Several works demonstrated timing attacks on user input, specifically inter-keystrokes, potentially recovering or reducing the entropy of passwords. The seminal work by Song et al. [41] inferred keystroke timings from SSH connections. Ristenpart et al. [33] presented a cross-VM cache timing attack on keystrokes. Gruss et al. [18] presented a semi-automated Flush+Reload attack on keystrokes. Other works exploited DRAM row accesses [30], interrupts [24], browser event loops [45] or the CPU frequency [13] to infer inter-keystroke timings.

Keyboard input typically triggers interrupts. Thus, interrupts have been exploited in several works, using `rdtsc` differences [37], Prime+Probe [37], and `procfs` files [37,49,21]. We extend the state-of-the-art by demonstrating keystroke timing attacks via another microarchitectural element that would remain exploitable even when the other side channels are closed.

8 Conclusion

In this paper, we extended the state-of-the-art on scheduler queue side channels in four directions, covering (1) all scheduler queues on (2) both Zen 3 and the new Zen 4 microarchitecture, (3) singular non-repeatable keystroke events, and (4) a pure JavaScript-based attack running in Firefox. We showed that all scheduler queues can be exploited and introduced a new measurement method, based on a timingless out-of-order race condition. We demonstrated keystroke timing attacks on all scheduler queues, with F_1 scores close to 1 and a standard deviation from the ground truth below 1 ms for scheduler queue 3 and at most 3.244 ms for scheduler queue 1. As native code execution is not a requirement for scheduler queue side channels, the research for efficient and effective mitigations becomes a more urgent issue. Our end-to-end JavaScript attack underlines this with covert data transmission across browser windows, bypassing cross-origin policies and site isolation, and achieving true capacities of 940.7 bit/s on Zen 4.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback and Martin Schwarzl for help with initial experiments. This research is supported in part by the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85), and the Austrian Research Promotion Agency (FFG project SEIZE 888087). Additional funding was provided by generous gifts from Red Hat, Google, Intel and AWS. Any opinions, findings, and 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. Abel, A., Reineke, J.: uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: ASPLOS (2019)
2. Abel, A., Reineke, J.: nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In: ISPASS (2020)
3. Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port Contention for Fun and Profit. In: S&P (2018)
4. AMD: Software Optimization Guide for AMD EPYC 7002 Processors (3 2020)
5. AMD: Software Optimization Guide for AMD EPYC 7003 Processors (11 2020)
6. AMD: Where Gaming Begins: AMD Ryzen 5000 Series (11 2020), https://www.slideshare.net/AMD/amd-where-gaming-begins-239086719
7. AMD: Execution Unit Scheduler Contention Side-Channel Vulnerability on AMD Processors (2022), https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1039.html
8. AMD: Software Optimization Guide for the AMD Zen4 Microarchitecture (1 2023)
9. Bhattacharyya, A., Sandulescu, A., Neugschwandt ner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: SMoTherSpectre: exploiting speculative execution through port contention. In: CCS (2019)
10. Christensen, A.: Reduce resolution of performance.now. (2015), https://bugs.webkit.org/show_bug.cgi?id=146531
11. Chromium: window.performance.now does not support sub-millisecond precision on Windows (2015), https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110
12. Cutress, I.: Investigating Performance of Multi-Threading on Zen 3 and AMD Ryzen 5000 (2020), https://www.anandtech.com/show/16261/investigating-performance-of-multithreading-on-zen-3-and-amd-ryzen-5000/2
13. Dipta, D.R., Gulmezoglu, B.: DF-SCA: Dynamic Frequency Side Channel Attacks are Practical. arXiv:2206.13660 (2022)
14. Faggioli, D.: Re: [RFC PATCH v3 00/16] Core scheduling v3 (2019), https://lore.kernel.org/lkml/277737d6034b3da072d3b0b808d2fa6e110038b0.camel@suse.com/
15. Fog, A.: The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers (2021), https://www.agner.org/optimize/microarchitecture.pdf
16. Gast, S., Juffinger, J., Schwarzl, M., Saileshwar, G., Kogler, A., Franza, S., Köstl, M., Gruss, D.: SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In: S&P (2023)
17. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS (2017)
18. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security (2015)
19. Herman, D., Wagner, L., Zakai, A.: asm.js: Working Draft – 18 August 2014 (2014), http://asmjs.org/spec/latest/
20. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual (2019)
21. Jana, S., Shmatikov, V.: Memento: Learning Secrets from Process Footprints. In: S&P (2012)
22. Johnson, D.: And a little more... big changes to the LDQ/STQ sizes, and new 'coalescing retire queue' theory and sizes (3 2021), https://twitter.com/dougallj/status/1373973478731255812

23. Kohlbrenner, D., Shacham, H.: Trusted Browsers for Uncertain Times. In: USENIX Security (2016)
24. Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C.m.t.n., Mangard, S.: Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS (2017)
25. Mozilla: SharedArrayBuffer (2012), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer
26. Mozilla: asm.js - Game development — MDN (2 2023), https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js
27. Mozilla: Number - JavaScript — MDN (6 2023), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number
28. Mozilla: Using Web Workers - Web APIs — MDN (3 2023), https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
29. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS (2015)
30. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security (2016)
31. Pinet, S., Ziegler, J.C., Alario, F.X.: Typing Is Writing: Linguistic Properties Modulate Typing Execution. Psychon Bull Rev **23**(6), 1898–1906 (4 2016)
32. Purnal, A., Bognar, M., Piessens, F., Verbauwhede, I.: ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In: AsiaCCS (2023)
33. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS (2009)
34. Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y.: Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In: AsiaCCS (2022)
35. Rushanan, M., Russel, D., Rubin, A.D.: MalloryWorker: Stealthy Computation and Covert Channels Using Web Workers. In: International Workshop on Security and Trust Management (2016)
36. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue In-flight Data Load. In: S&P (2019)
37. Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., Mangard, S.: KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS (2018)
38. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC (2017)
39. Shusterman, A., Agarwal, A., O'Connell, S., Genkin, D., Oren, Y., Yarom, Y.: Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In: USENIX Security (2021)
40. Shusterman, A., Kang, L., Haskal, Y., Meltser, Y., Mittal, P., Oren, Y., Yarom, Y.: Robust Website Fingerprinting Through The Cache Occupancy Channel. In: USENIX Security (2019)
41. Song, D.X., Wagner, D., Tian, X.: Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security (2001)
42. Szefer, J.: Survey of microarchitectural side and covert channels, attacks, and defenses. Journal of Hardware and Systems Security **3**(3), 219–234 (2019)
43. Taneja, H., Kim, J., Xu, J.J., van Schaik, S., Genkin, D., Yarom, Y.: Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and ARM SoCs. arXiv:2305.12784 (2023)

44. Van Goethem, T., Joosen, W.: One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In: WOOT (2017)
45. Vila, P., Köpf, B.: Loophole: Timing Attacks on Shared Event Loops in Chrome. In: USENIX Security (2017)
46. W3C: High Resolution Time Level 2 (2016), `https://www.w3.org/TR/hr-time/`
47. Weinberg, Z., Chen, E.Y., Jayaraman, P.R., Jackson, C.: I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In: S&P (2011)
48. Zbarsky, B.: Reduce resolution of performance.now. (2015), `https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab`
49. Zhang, K., Wang, X.: Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security (2009)

A Validation of the Bingo Race Approach

To verify that we indeed observe scheduler contention with our out-of-order bingo variable race condition, we replace the `imul` priming instruction in block ❹ of Figure 1 with `add rax, 3`, which can be executed by all the 4 ALUs of the Zen 2 and the Zen 3 microarchitectures [4,5]. On Zen 2, we measure a capacity of 64 µops, matching the documented total capacity of all the 4 schedulers that are connected to an ALU. On Zen 3, we measure a capacity of 90 µops, which is 6 µops less than the documented total capacity ($4 \times 24 = 96$ µops [6]) of all the 4 integer execution unit schedulers on that machine, which is again 2 µops more than measured by previous work [16]. We discuss the reason for the lower observed capacity in Appendix B.

One drawback of this method is that the memory access ordering rules of the ISA prevent reordering of the second bingo variable read in some cases. For a simple example, if we ensure `rdi` contains a valid but uncached address and replace the priming instruction with `imul rax, [rdi]`, then the second bingo variable read will not retire until `[rdi]` has been read for the multiplication. We verify this by inserting a `clflush [rdi]` instruction before block ❶ in Figure 1 and replacing the multiplication in block ❹ with `imul rax, [rdi]`. With this, we only observe the same bingo number twice if $k = 0$, whereas for $k = 1$ we immediately see 198.395 (Zen 2) and 443.839 (Zen 3) bingo number updates on average ($n = 100\,000$). This shows that the uncached memory access indeed delays the subsequent second bingo variable read, entirely overshadowing the delay caused by scheduler contention. In a control experiment using `rdpru` instead of a bingo thread, we observe no such effect. Also, if we keep the bingo thread and the memory operand, but remove the `clflush [rdi]` instruction, we observe the same capacity limits as with the original experiment, because the memory operand is cached and accessing it does not cause such a large delay.

For a second example, we ensure `rdi + rax` contains a valid, cached memory address after block ❸ in Figure 1. We now initialize `r10` with a fixed value and replace the priming instruction in block ❹ with `imul r10, [rdi+rax]`, so that the address of its memory operand depends on the long-latency calculation of `rax`. We again observe that the second bingo variable read is delayed for $k > 0$,

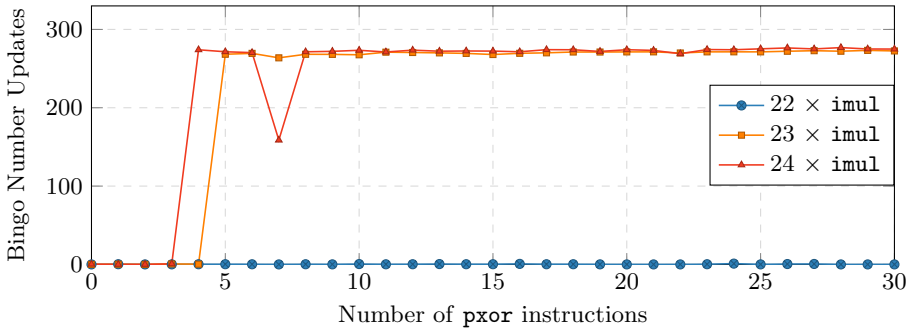


Fig. 6. Spurious back-end stalls on Zen 3. Even though the scheduler queue capacity is not exceeded, the CPU stalls a few instructions after the queue was primed with at least 23 multiplications.

as (per memory access order rules) now `[rdi + rax]` has to be read before the bingo variable, which in turn can only be done after the result for `rax` has been computed.

B Spurious Scheduler Stalls

Unlike earlier [16] scheduler contention measurement methods, which used performance counters or non-serialized hardware timer reads via the `rdpru` instruction, our bingo method result in the exact capacity of scheduler 1 on Zen 3. In this section, we show that the Zen 3 backend spuriously stalls after a few instructions, if the scheduler queue is primed with 23 or 24 µops, explaining the 2 µops difference between the actual capacity and the capacity observed with performance counters or `rdpru`.

First, we run the code in Figure 1a again with varying numbers of multiplications k, while monitoring the `IntSch1TokenStall` performance counter. Like previous work [16], we observe a steep increase in stalled cycles with $k \geq 23$ multiplications, whereas the results from the bingo thread show that, with $k \leq 24$ the backend does not stall before reaching the second bingo variable read. This indicates that the backend might stall *after* the second bingo variable read, even if the scheduler queue capacity is not exceeded.

We further investigate this with variations of Figure 1a, in which we prime the scheduler queue with fixed numbers of multiplications and insert a varying number of `pxor xmm1, xmm1` instructions between the multiplications and the second bingo variable read. As the `pxor` instruction is handled by the FPU [5], it does not use the integer execution unit schedulers, however it adds a short delay between enqueuing the multiplications and the second bingo variable read. Figure 6 shows the effect of this: With 22 multiplications, followed by 0 to 30 `pxor` instructions, the second bingo variable read is still executed immediately without stalling the backend. However, with 23 multiplications, we observe that

the second bingo variable read is delayed when we insert 5 or more `pxor` instructions. With 24 multiplications, we observe the same delay when we insert 4 or more `pxor` instructions. According to official AMD documentation [5], the frontend can dispatch up to 6 µops to the FPU per cycle. We conclude that, with 23 or 24 multiplications, there is a short time window of about one cycle, after which the Zen 3 backend spuriously stalls.

With nanoBench [2] we measure that a single `rdpru` instruction (with `ecx = 1`) requires 17 CPU cycles on our machine. Apparently, with 23 or 24 multiplications, the backend spuriously stalls before `rdpru` has finished reading the `APERF` counter, explaining the 2 µops difference between earlier and our measurement methods. We also run a similar experiment on our Zen 2 machine, where we fill the scheduler queue to its capacity limit of 16 µops and insert up to 100 `pxor` instructions. On that machine, we do not observe these spurious stalls, showing that these are specific to the Zen 3 microarchitecture.

C Scheduler Queue Usage of Various Instructions on Zen 3 and 4

Table 3 shows the scheduler queue usage for various instructions on both the Zen 3 and the Zen 4 microarchitectures.

Table 3. Scheduler Queue Usage of Various Instructions on Zen 3 and 4[1]

Instruction	**Measurements**		**Documentation**	**Comment**
	Capacity	Scheduler	ALU / Scheduler	
`idiv r10`	23	Integer 0	ALU0 / Integer 0	rax=1, rdx=0, r10=1
`movd xmm1, eax`	23	Integer 0	ALU2 / Integer 2	
`vmovd xmm1, rax`	23	Integer 0	ALU2 / Integer 2	
`cvtsi2sd xmm1, rax`	23_3 22_4	Integer 0	ALU2+FPU2/3 / Integer 2+FP0/1	
`imul rax, 3`	24	Integer 1	ALU1 / Scheduler 1	
`stosb`	22	Integer 2	undocumented (ucode)	
`lodsb`	22	Integer 3	undocumented (ucode)	*delayable* due to partial register write
`lodsw`	22	Integer 3	undocumented (ucode)	*delayable* due to partial register write
`lodsd`	22	Integer 3	undocumented (ucode)	*delayable* due to false input dependency
`lodsq`	22	Integer 3	undocumented (ucode)	*delayable* due to false input dependency
`bsf rbx, rax` (Zen 3)	7	Integer 3	undocumented	not *single-µop*
`bsf rbx, rax` (Zen 4)	89	Integer 0/1/2/3	ALU0/1/2/3 / Integer 0/1/2/3	not *targeted*
`bsr rbx, rax` (Zen 3)	7	Integer 3	undocumented	not *single-µop*
`bsr rbx, rax` (Zen 4)	89	Integer 0/1/2/3	ALU0/1/2/3 / Integer 0/1/2/3	not *targeted*
`rol rax, 3`	46	Integer 1/2	ALU1/2 / Integer 1/2	not *targeted*
`shr rax, 3`	46	Integer 1/2	ALU1/2 / Integer 1/2	not *targeted*
`add rax, 3`	91_3 89_4	Integer 0/1/2/3	ALU0/1/2/3 / Integer 0/1/2/3	not *targeted*
`vaddsd xmm0, xmm0, xmm0`	127_3 124_4	FP0/1	FPU0/1/2/3 / FP0/1	not *targeted*
`divsd xmm0, xmm0`	127_3 124_4	FP0/1	FPU1 / FP1	not *targeted*
`sqrtsd xmm0, xmm0`	127_3 124_4	FP0/1	FPU1 / FP1	not *targeted*
`xor rax, rax`	-	none	none (zeroing idiom)	not *targeted*
`mov rbx, rax`	-	none	none (rename only)	not *targeted*

[1] We see small differences in the capacity measurements for most instructions, the Zen version is denoted as a subscript in these cases. The reason for that is not a different scheduler queue size but smaller differences in the microarchitecture, like different scheduling of other instructions executed for the measurement. We only observe a large difference between Zen 3 and Zen 4 with the `bsf` and `bsr` instruction. While consisting of multiple µops executed only on ALU3 in Zen 3, they are single µop instructions that can be executed on all integer ALUs in Zen 4.

D Source Code Excerpts for the JavaScript-based Covert Channel

The following listings show the attack code for our JavaScript-based Covert Channel and a disassembly of the JIT-compiled code generated by Firefox:

```
// Shared array buffer
var timer = new Uint32Array(...);

function squip(start_value, factor) {
  start_value = +start_value;
  factor = factor|0;

  let dummy = +start_value;
  let dummy2 = start_value|0;

  let bingo1 = Atomics.load(timer, 0)|0;

  dummy = +Math.sqrt(+dummy); // repeat 12 times

  dummy = dummy|0;

  dummy = Math.imul(dummy|0, factor|0)|0; // repeat 20 times

  let bingo2 = Atomics.load(timer, 0)|0;

  return {
    dummy: dummy|0,
    time: ((bingo1|0) != (bingo2|0))|0
  };
}
```

Listing 1.1. The JavaScript-based covert channel attack code.

```
mov ebx, [rsi+rbp*4]

; ...

xorpd xmm1, xmm1
cvtsi2sd xmm1, eax
.rept 12
sqrtsd xmm1, xmm1
.endr
cvttsd2si rdi, xmm1

cmp rdi, 1
jo 570h
mov edi, edi
mov r11, 0fffe000000000000h
xor r11, [r9 + 58h]
```

```
17 mov r9, r11
18 shr r11, 2fh
19 jnz 69dh
20 mov r8, 22aa1cc9c660h
21 cmp r9, r8
22 jnz 6a7h
23
24 .rept 20
25 imul edi, ecx
26 .endr
27 mov r8d, [rsi+rbp*4]
```

Listing 1.2. Disassembly of the JIT-compiled code generated by Firefox.

Lines 25 to 27 show the `imul` instructions for priming Scheduler 1, which depend on the `sqrtsd` instructions and the subsequent integer conversion in lines 7 to 10. The bingo variable is read in lines 1 and 27, with the second read directly following the last `imul` instruction.

Firefox inserts some additional code (lines 12 to 22) between the integer conversion and the `imul` instructions, which does not hinder our measurements: Lines 12, 14, 15, 17, 18, 20 and 21 are low-latency operations that only occupy the scheduler queues for a very short time. Line 16 has a memory operand, however this operand is cache-hot, therefore it also has a low latency and does not prevent the second bingo variable read from being executed earlier in the non-contended case (see Appendix A). Lines 13, 19 and 22 are jumps that are never taken, hence they are speculated over, after a few warm-up iterations.