JavaSQIP

An application of a scheduler queue-based covert channel to JavaScript

Christoph Royer

Introduction to Scientific Working 2022/23

Supervisor: Stefan Gast

Institute of Applied Information Processing and Communications
Graz University of Technology

Abstract

The implementation of out-of-order execution has brought a big performance benefit to current CPUs. But with this benefit come also security concerns, since attackers can exploit the timing variations which are bound to occur in an out-of-order pipeline. Attacks which target only one execution unit through a separate scheduler queue – as seen in the Apple M1 and AMD Zen 2 and Zen 3 microarchitectures – have proven even more powerful than port contention on single-scheduler systems. As with any attack, porting this side channel to the web would greatly increase its reach and number of victims.

In this paper, we present the JavaSQUIP attack, which is a port of SQUIP [GJS+22] to JavaScript. We look into the security measures used in common browsers to prevent timing attacks, and show how we worked around them successfully. Our covert channel can provide communication across separate browser instances at a speed of 1000 bits/s, which is faster than any current covert channel of this type.

JavaSQUIP makes it clear that current browsers have not yet been able to adapt to be secure in light of the ever increasing complexity of modern CPUs. Our findings suggest that making these attacks impossible may not even be feasible because the drawbacks in performance and features would be too large.

Keywords: Covert channel · Scheduler queue contention · JavaScript · AMD Zen

1 Introduction

Nowadays, CPU manufacturers are in a lasting battle against each other to make their product faster and faster. One of the ways this is achieved is *out-of-order execution*, where instructions are not executed in the order given by the binary, but rather as soon as their respective dependencies are ready. Although this toes the line of breaking the hardware-software contract, the CPU ensures that out-of-order execution remains transparent to the user. To better balance usage of its execution units, modern CPUs often also include *Simultaneous Multithreading* (SMT). Here, one physical core is split up into two logical cores, which share some physical architecture. SMT is also designed to be transparent to the user, but problems arise from the combination of out-of-order execution and SMT.

By measuring timing variations in out-of-order execution, an attacker can gain insight into sensitive information. One of the ways attacks achieve this is through scheduler queue. If the attacker is co-located with the victim on the same core, they can measure delays when issuing instructions to a shared execution unit. In this way, the attacker can gain knowledge on the victim's execution based on whether instructions are delayed by the scheduler or not. SMoTherSpectre [BSN+19] and PortSmash [ABH+19] already use this principle successfully.

While SMoTherSpectre and PortSmash rely on an architecture where there is only one scheduler for all ports, SQUIP [GJS+22] focuses on CPUs with multiple specialized scheduler queues. Because each scheduler deals with fewer execution units, pipeline stalls are easier to achieve and allow more accurate measurements. This exploitation of *scheduler queue contention* makes SQUIP a fast and reliable covert channel on many modern CPU architectures such as the Apple M1, and the AMD Zen 2 and Zen 3.

Since implementations of port contention-based side channels in a browser setting already exist [RMBO22], we ask the following research question:

Can per-execution-unit scheduler queue contention be exploited in a browser setting? Are there significant benefits compared to single scheduler port contention-based approaches?

In this paper, we present JavaSQUIP, a browser-based covert channel using the SQUIP attack. We show that it is reliable and faster than approaches based on CPU architectures with a single scheduler queue. We develop a framework for transmitting data between separate instances of a browser at speeds of up to 1000 bit/s.

Outline. In Section 2, we explain the relevant knowledge around pipelines, schedulers and scheduler queue contention. We also observe several challenges that come with trying to implement a timing attack in a browser. Section 3 shows how we dealt with these challenges. Section 4 concludes by comparing JavaSQUIP's performance to similar attacks and outlining potential use cases.

2 Background

In this chapter, we go into detail on CPU scheduling and how it has been exploited in the past. Afterwards, we take a closer look into the browser features needed to implement these exploits in JavaScript.

2.1 CPU Scheduling

For performance reasons, modern CPUs use efficiency features at almost every level of the lifecycle of an instruction. In the following, we will list these features for each of the stages along with the way they are exploitable.

Fetch. Before an instruction can be used, it needs to be loaded from cache or memory. As branches can make loading unpredictable and loading the wrong instruction is expensive, a *branch prediction unit* tries to guess which path is more likely to be loaded. Taking this one step further, the predicted path may also be executed before the condition for the branch has been computed. This is called *speculative execution*, and it can be exploited, as was shown in the Spectre [KHF+19] attack.

Decode. Modern CISC CPUs divide normal instructions into smaller, RISC-like instructions; these are called *micro-ops* (µops). These µops can be distributed across specialized execution units, enabling more parallelisation and load balancing, and thus an increase in performance.

Schedule/Execute. Once created, the µops are queued to be executed as soon as their dependencies are met – sometimes out of the order of their arrival. Depending on the CPU micoarchitecture, there can be one general scheduler queue for all execution units, or multiple. On CPUs that support simultaneous multithreading (SMT), an attacker may be co-located with the victim on the same core. This opens up the vulnerability of a timing attack based on scheduler queue contention; an attacker can find out whether the victim is executing a particular µop based on timing differences in the scheduler queue. To achieve this, the attacker fills up the scheduler queue to the point that it is almost full; any activity of the victim that involves this scheduler queue will overflow the queue, leading to a queue stall. The attacker can measure this easily because it entails a significant spike in execution time in the affected queue. SMoTherSpectre [BSN+19] and PortSmash [ABH+19] implement this attack in an environment with a single scheduler queue. SQUIP [GJS+22] adapts the exploit to architectures with multiple scheduler queues. JavaSQUIP aims to port this vulnerability to a browser environment.

Retire. After all the µops of a macro-operation have completed, the *retire control unit* (RCU) reassembles the results. These are then passed on to the frontend – the dissection into µops is transparent to the user.

2.2 Browser security measures

Since the advent of many timing side- and covert channels [NR18; RMBO22; GMM16], browser developers have devised many measures to mitigate the threat [SAO+21]. This has always meant striking a balance between security and efficiency. Because of this, many potential attack surfaces cannot be eliminated without unreasonable losses in performance, though some measures do slow down side- and covert channels significantly.

In the following we expand on security measures regarding multithreading, accurate timing in browsers, and low-level operations. All three of these are needed to mount the SQUIP attack.

2.2.1 Multithreading

Though JavaScript does not support low-level control over threading, the `Web Workers` API [MDN22a] allows the user to create workers running in a separate thread. An instantiated worker runs a predefined script. The main thread can interface with the worker via calls to the `Worker.postMessage()` function and via a `SharedArrayBuffer` [MDN22b]. Creating a `SharedArrayBuffer` was restricted to SSL-secured browser windows, which is not a problem for this attack. There is no support for selecting a core on which a worker is run on, presenting a challenge to co-locate with the sender.

2.2.2 Timing functions

There are two contenders for timing measurements in JavaScript: `Date.now()` [MDN22c] and `performance.now()` [MDN22d].

`Date.now()` [MDN22c] function, gives a synchronized Unix epoch timestamp with millisecond accuracy. This is synchronized across all contexts and browser instances.

`performance.now()` measures the time elapsed since the *time origin* of the current context. This means that `performance.now()` gives different values between browser instances, and even between a main thread and its worker threads. `performance.now()`'s accuracy has been heavily restricted by browsers to counteract the thread of timing attacks. For example, in Firefox it has an accuracy of 1 ms, and in Chrome it measures with an accuracy of 0.1 ms.

2.2.3 Low-level operations in JavaScript

JavaScript handles numbers as floats by default. This means that a simple invocation of a multiplication like `a * b;` in JavaScript will not result in an `imul` instruction on the CPU. Rather, `Math.imul(a, b)` [MDN22e] gives the user a lower-level command: It interprets the parameters as 32bit integers and multiplies them as such, resulting in an `imul` instruction on the CPU.

3 Implementing JavaSQUIP

As described in Section 2.2, there are some challenges to overcome when implementing a timing attack in a browser. This chapter explains how we worked around these challenges to implement the JavaSQUIP covert channel.

3.1 Co-locating with the sender

For co-locating the receiver with the sender, the receiver creates as many workers as the number of cores on the machine's CPU. Now, every one of these workers will create a lot of entries in a particular execution unit's queue. This makes it very likely that each of the workers will be moved to a separate physical core to reduce port contention. Since every worker is in a separate core, one of them is bound to be co-located on the same core with the sending thread.

3.2 Getting accurate timing

JavaSQUIP needs two kinds of timing in order to work properly:

- a cross-browser syncronized clock to determine the start of bit transmissions: on the order of $f_{transmission}^{-1} \approx 1\text{ms}$
- a fine-grained measurement to measure port contention: on the order of 100 CPU cycles $\approx$ 25ns

This method only works for transmissions of up to 1000 bits/s, since there is no more precise timestamp in JavaScript which is synchronized across browser instances.

The receiver also needs a way to measure the small variations in timing whenever a scheduler queue gets blocked. Since no timing function in JavaScript gives a better accuracy than 0.1ms, we need to provide our own timing.

We can instantiate a counting worker, which continually increases one variable in the `SharedArrayBuffer`. The other receiver workers can then access this counter and use it to measure execution times with adequate accuracy.

3.3 Targeting a scheduler queue

JavaSQUIP targets AMD Zen, Zen2, and Zen3; these architectures support multiplication only on ALU1 [AMD20]. By sending multiple codependent `imul` instructions, we can fill ALU1's scheduler queue. JavaSQUIP utilizes the `Math.imul()` function, which does exactly this.

4 Conclusion

In this paper, we ported the SQUIP [GJS+22] attack to a JavaScript context, thus creating the JavaSQUIP covert channel. JavaSQUIP has achieved a transmission rate of 1000 bits/s, proving an advantage in performance over attacking single-scheduler based architectures [RMBO22]. This shows that architectures with multiple scheduler queues present a bigger attack surface than their single-scheduler counterparts.

JavaSQUIP breaks the sandboxing model of a web browser context. This means that one browser instance can communicate with another without using the network, thus being virtually undetectable for an unknowing victim. This could potentially be used by malicious code within a website – no matter its origin – to exfiltrate data to another browser instance on the same computer.

Though security updates in modern browsers slow down and hinder timing attacks, microarchitectural attacks have not yet been eliminated. JavaSQUIP mostly uses JavaScript features that have already been restricted to improve security as much as feasible; mitigating microarchitectural attacks in a browser setting proves to be increasingly hard as CPUs grow more and more complex.

Bibliography

[ABH+19] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit". In: *S&P*. 2019.

[AMD20] AMD. *Software Optimization Guide for AMD EPYC 7003 Processors*. Nov. 2020.

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. "SMoTherSpectre: exploiting speculative execution through port contention". In: *CCS*. 2019.

[GJS+22] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. "SQUIP: Exploiting the Scheduler Queue Contention Side Channel". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2022, pp. 468–484.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer. js: A remote software-induced fault attack in javascript". In: *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer. 2016, pp. 300–321.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *Security and Privacy – SP 2019*. IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[MDN22a] MDN. *Web Workers API - Web APIs | MDN*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API (visited on 12/14/2022).

[MDN22b] MDN. *SharedArrayBuffer - JavaScript | MDN*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer (visited on 12/14/2022).

[MDN22c] MDN. *Date.now() - JavaScript | MDN*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now (visited on 12/14/2022).

[MDN22d] MDN. *performance.now() - Web APIs | MDN*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now (visited on 12/15/2022).

[MDN22e] MDN. *Math.imul() - JavaScript | MDN*. 2022. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/imul` (visited on 12/15/2022).

[NR18] Lucas Noack and Tobias Reichert. “Exploiting speculative execution (spectre) via javascript”. In: *Advanced Microkernel Operating Systems* (2018), p. 11.

[RMBO22] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. “Port Contention Goes Portable: Port Contention Side Channels in Web Browsers”. In: *AsiaCCS*. 2022.

[SAO+21] Anatoly Shusterman, Ayush Agarwal, Sioli O’Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. “{Prime+ Probe} 1,{JavaScript} 0: Overcoming Browser-based {Side-Channel} Defenses”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2863–2880.