IAIK

SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Christoph Royer

JavaSQUIP
Implementing a Scheduler-Queue Covert Channel in JavaScript

BACHELOR'S THESIS

Bachelor's degree programme: Computer Science

Advisor: Stefan Gast
Supervisor: Daniel Gruss

Institute of Applied Information Processing and Communications
Graz University of Technology

Graz, October 2024

Abstract

The implementation of out-of-order execution has brought a big performance benefit to current CPUs. With this benefit also come security concerns however, 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 3 and Zen 4 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 thesis, 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. At this speed, we are able to achieve a transmission accuracy of 99.2 % and 99.3 % on Zen 3 and Zen 4 respectively.

Modern CPUs are becoming ever more complex, and with this complexity come more possibilities for side channel attacks. JavaSQUIP makes it clear that the security measures of current browsers have not yet adapted to protect against these kinds of attacks. Our findings suggest that making these attacks impossible may not even be feasible because the drawbacks in performance and features would be too extensive.

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

1 Introduction

Nowadays, CPU manufacturers are in a steady competition against each other to make their products faster and faster. One of the ways this is achieved is *out-of-order execution*, where instructions are not executed in program order, but rather as soon as their respective dependencies are ready and a suitable execution unit is free. 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 physical infrastructure like the schedulers and execution units.

Out-of-order execution and SMT are designed to be transparent to the user, however attacks like PortSmash [ABH+19] and SMoTherSpectre [BSN+19] show that combining the two technologies presents a viable attack surface. By measuring timing variations in out-of-order execution, an attacker can gain insight into sensitive information. This attack, which targets all CPUs with SMT, is called *port contention*. 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.

SQUIP [GJS+22] focuses on CPUs with multiple specialized scheduler queues to implement *scheduler queue contention*. Each scheduler deals with fewer execution units, which is why the attacker can push one of the comparatively small scheduler queues close to its capacity. If the victim also adds to the queue, the pipeline stalls, which results in a significant increase in execution time. This makes SQUIP a fast and reliable covert channel on many modern CPU architectures such as the AMD Zen 3. [GJS+22]

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

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?

Contributions. In this thesis, we contribute the following:

- We present JavaSQUIP, a browser-based covert channel based on the SQUIP [GJS+22] attack.
- We develop a framework for transmitting data between separate instances of a browser at speeds of up to 1000 bit/s.
- We show that JavaSQUIP is reliable and faster than approaches based on CPU architectures with a single scheduler queue.
- We argue that JavaSQUIP relies on essential features of JavaScript, meaning the threat it poses cannot be easily mitigated.

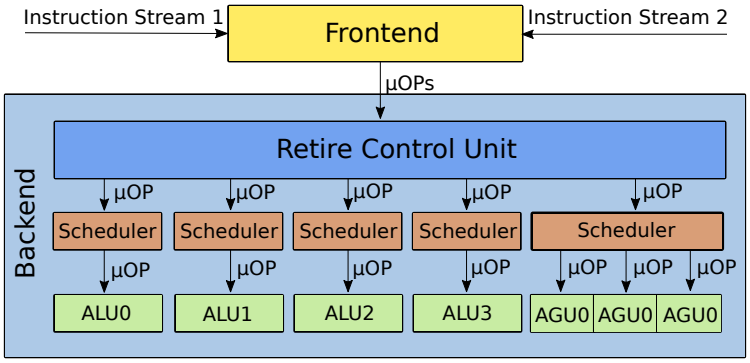
Outline. In Section 2, we provide relevant background information on scheduler queue contention, resulting in several challenges that we will have to overcome when implementing JavaSQUIP. Section 3 shows how we handle these challenges. Section 4 continues by explaining how we optimized JavaSQUIP for speed and reliability. We go into more detail on the mode and results of our evaluation of JavaSQUIP in Section 5. In Section 6, we present comparable covert channels and compare them to JavaSQUIP. We propose ways to prevent JavaSQUIP in Section 7 and discuss whether they are feasible. We conclude with Section 8, outlining potential use cases and discussing the implications of the attack.

2 Background

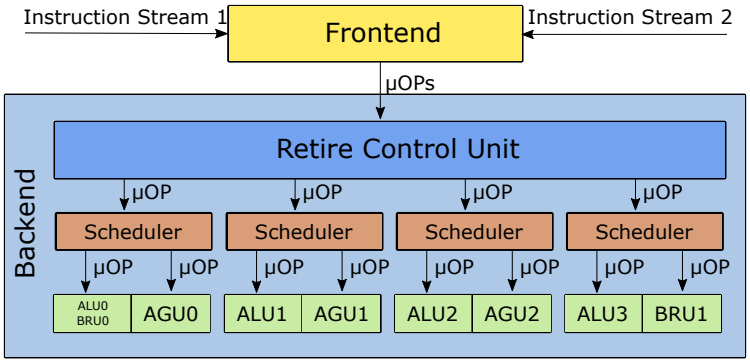
In this chapter, we go into detail on two aspects of CPU scheduling – *Out-Of-Order Execution* and *Simultaneous Multithreading* – and how they have been exploited by prior work. Furthermore, we take a closer look into the browser environment and which features are needed to implement these exploits in JavaScript.

2.1 Out-Of-Order Execution

For performance reasons, modern CPUs use efficiency features at almost every stage of an instruction's execution. One notable feature is *Out-Of-Order Execution*, where instructions are split up into smaller parts. These so-called *µops* can be executed in a different order than the program order, enabling a more efficient utilization of the CPU's resources.



(a) The scheduler layout in AMD Zen 2 (c.f. [AMD20a])



(b) The scheduler layout in AMD Zen 3 (c.f. [AMD20b])

Figure 2.1: Comparison of the scheduler layouts of AMD Zen 2 and 3 architectures

The following paragraphs describe the stages of an instruction in a CISC CPU, where Out-Of-Order Execution is utilized.

Fetch. Before an instruction can be executed, 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* [AMD20a].

Decode. Modern CISC CPUs divide complex instructions into their parts when possible; these parts are called *micro-ops* (µops) [AMD20a]. For example, an addition with memory operands may be split up into three µops: one for loading the operands, one for the addition, and one for storing the result. These µops can be distributed across specialized execution units, enabling more parallelization 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 microarchitecture, there can be one general scheduler queue for all execution units, or multiple. While Intel uses one scheduler queue for all its execution units [Int19], architectures like AMD's Zen 2 [AMD20a], Zen 3 [AMD20b] and Zen 4 [AMD23] use several specialized scheduler queues. When multiple scheduler queues are used, different queues can be specialized for a specific µop or class of µops. Figure 2.1 illustrates the concept. Splitting the scheduler queue reduces complexity and enables more efficient power usage.

Retire. After all the µops of an instruction have completed, the *retire control unit* (RCU) reassembles the results. It commits the finished instructions and schedules dependent instructions – the dissection into µops is transparent to the user.

2.2 Simultaneous Multithreading

As we described in more detail in Chapter 2.1, the cores of a modern CPU are subdivided into multiple specialized execution units. If only one thread would be executed on such a core, most of these execution units would stay unused while an instruction is executed.

With *Simultaneous Multithreading* (SMT), one physical core is split into multiple logical cores – most commonly, two logical cores per physical core are implemented [TEL95]. This means that one core executes two threads simultaneously, distributing the workload across its execution units more effectively. Tests show that this provides an average performance gain of about 25 % [Lar18; Cut20] with relatively little additional architecture needed.

The two threads that are being executed on the same core are called *co-located*. Usually, these threads should be unaffected by each other; no data is exchanged between two co-located threads. However, the sharing of common architecture, particularly the execution units, enables several different side channels. All of them are *timing attacks*, meaning that they gain information by measuring execution times of a piece of code. A selection of these attacks will be explained in the following chapter.

2.3 Prior attacks on the CPU pipeline

Many of the stages of an instruction's life cycle are exploitable. In the following, we will list some of the exploits that have already been found.

Speculative Execution. Executing a path before the condition for the branch is known can be exploited, as was shown by Spectre [KHF+19].

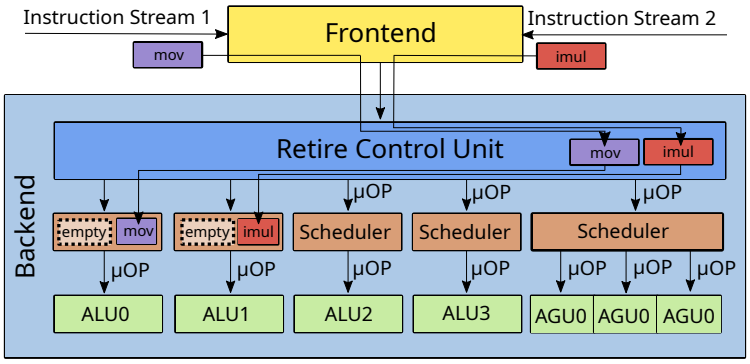
First, the branch prediction unit is primed by repeatedly branching with the condition set to A. In the second step, the branch is called again, but with the condition set to B.

Since the branch prediction unit was trained to expect condition A, the CPU can be tricked into executing an inaccessible path. Under the right circumstances, this can be used to leak data from a co-located process.

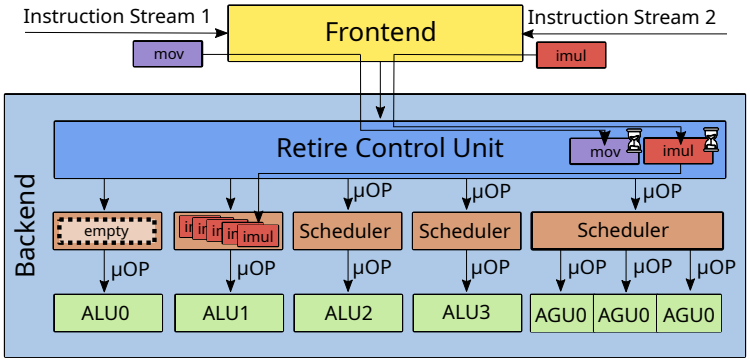
Port Contention. PortSmash [ABH+19] and SMoTherSpectre [BSN+19] exploit port contention. They use the fact that only one of the two threads in an SMT-enabled core can use an execution unit in a single cycle. The attacker can thus find out whether the victim is using a port by measuring the execution time of an instruction on the same port. Delayed execution gives away the fact that the victim is currently using that port. Port contention works independently of the type of scheduler queue used. It affects only one execution unit, and thus creates contention and measures execution time with the same instruction.

Scheduler Queue Contention. SQUIP [GJS+22] on the other hand exploits scheduler queue contention. In scheduler queue contention, the attacker fills up the scheduler queue to the point that it is almost at its capacity limit; any activity of the victim that involves this scheduler queue will overflow the queue, leading to a queue stall. A queue stall occurs when more µops of a particular type are created than what the correlating scheduler queue can hold. Because the CPU cannot drop any µops, it has to stall the loading of new instructions entirely until the scheduler queue is free again. The attacker can measure this easily because it entails a significant increase in execution time – not only for instructions in the affected queue, but on the whole core. Figure 2.2b shows how one instruction stream can stall the other just by filling up the relatively small scheduler queue.

SQUIP is limited in its applicable CPU architectures because it relies on the split-scheduler architecture of AMD Zen 2, Zen 3 and Zen 4 CPUs to distinguish between µops. However, its advantage is a higher transmission rate compared to port contention. Because scheduler queue contention stalls the whole CPU, as compared to port contention stalling just one execution unit, a comparatively bigger increase in execution time is achieved. SQUIP produces stalls with instructions on a different execution unit than the unit where execution time is measured (see Figure 2.2b). The measured execution unit either sees low load (the measurement instructions), or a complete stall (scheduler queue contention), as illustrated in Figure 2.2. This increases the signal-to-noise ratio of the timing measurement, enabling shorter time slices per bit. All of this gives SQUIP [GJS+22] higher potential transmission speeds and reliability than port-contention based side channels. Because of this advantage, JavaSQUIP aims to port this vulnerability to a browser environment.



(a) An AMD Zen 2 processor in normal operation: the two instruction streams do not impede each other



(b) An AMD Zen 2 processor exhibiting scheduler queue contention: a stall on the Scheduler queue for ALU1 stalls both instruction streams

Figure 2.2: Comparison between normal operation and scheduler queue contention in an AMD Zen 2 Processor

2.4 JavaScript in a browser environment

JavaScript [MDN24a] is the main scripting language for running code in a website. It is an interpreted language, which means that the browser reads in code in text form and executes it directly. JavaScript developers benefit from writing high-level platform-independent code, while the interpreter takes care of executing it on the user's machine.

To improve performance, modern browsers use *just-in-time (JIT) compilation* [MDN24b]. The interpreter monitors the program flow during execution. If a function is called very often, it is computationally cheaper to compile it once rather than repeatedly interpreting it. The compilation is done in two stages: baseline compilation and optimized compilation. The baseline compiler creates the binary representation of the function. It also optimizes away unnecessary code, such as unused calculations, while keeping the execution equivalent to the script. If the function is repeatedly called with similar arguments, the optimized compiler creates a second binary code. This second binary is optimized based on assumptions about the arguments – such as their types and properties. The optimized binary is executed after assessing the truth of the assumptions, and is discarded if the assumptions do not hold.

2.5 Browser security measures

Since the advent of many timing side and covert channels [NR18; RMBO22; GMM16; HBBP14], browser developers have implemented many different measures to mitigate the threat [SAO+21; MDN22a; Goo23; SLG18]. 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 and functionality, 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.

2.5.1 Timing functions

There are two contenders for timing measurements in JavaScript: `Date.now()` [MDN22b] and `performance.now()` [MDN22a].

`Date.now()` [MDN22b] returns a synchronized Unix epoch timestamp with millisecond accuracy. The Unix epoch time is essentially the time elapsed since January 1, 1970 and is the same for all processes on a computer. This makes `Date.now()` a cross-instance synchronized timestamp.

The `performance.now()` [MDN22a] function measures the time elapsed since the *time origin* of the current context. This means that `performance.now()` results in different values between browser instances, and even between a main thread and its worker threads. Timing information from `performance.now()` can still be synchronized; The *time origin* of the context is given by the `performance.timeOrigin()` [MDN23a] property. This way, the timestamps from `performance.now()` can be globally synchronized.

The accuracy of `performance.now()` has been restricted heavily by browser developers to counteract the threat 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 [MDN22a; Goo23].

Consequently, both `Date.now()` and `performance.now()` give a synchronized timestamp with 1 ms accuracy, making them viable options for synchronization.

2.5.2 Multithreading

JavaScript does not support low-level control over threading, however the `Web Workers` API [MDN22c] 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 by calling the `Worker.postMessage()` function. Data can be shared across workers using a `SharedArrayBuffer` [MDN22d]. There is no support for selecting the core on which a worker is run, presenting a challenge to co-locate with the sender.

Creating a `SharedArrayBuffer` is restricted in two ways: the browser window needs to be SSL-encrypted, and it needs to restrict some Cross-Origin requests. Nevertheless, this is not a problem for an attacker on any site that provides `https` and where all code comes from a single origin. With the help of free, easy-to-use SSL certificates such as certbot [Fou24], almost every website on the internet provides an SSL-encrypted connection. This means that the requirements for a `SharedArrayBuffer` are likely to be met for any public website where a malicious actor would place their code.

2.5.3 Low-level operations in JavaScript

To create contention on one specific scheduler queue, we need a way to create a stream of equal and co-dependent instructions – specifically the `imul` instruction. One way to achieve this in a browser context would be *WebAssembly* [Web24]. For the sake of using as few features of the browser as possible, we implement the needed functionality in JavaScript.

JavaScript handles numbers as floating-point numbers 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 32 bit integers and multiplies them as such, resulting in an `imul` instruction on the CPU.

```
1 function test(a) {
2 a = Math.imul(a, a);
3 a = Math.imul(a, a);
4 a = Math.imul(a, a);
5 a = Math.imul(a, a);
6 a = Math.imul(a, a);
7 return a;
8 }
```

Listing 2.1: A test function for the `imul` instruction

```
1  00001072  49 89 c3                                    mov %rax, %r11
2  00001075  49 c1 eb 2f                                 shr $0x2F, %r11
3  00001079  41 81 fb f1 ff 01 00                        cmp $0x1FFF1, %r11d
4  00001080  0f 85 1a 03 00 00                           jnz 0x00000000000013A0
5  00001086  8b c0                                       mov %eax, %eax
6  00001088  0f af c0                                    imul %eax, %eax
7  0000108B  0f af c0                                    imul %eax, %eax
8  0000108E  0f af c0                                    imul %eax, %eax
9  00001091  0f af c0                                    imul %eax, %eax
10 00001094  0f af c0                                    imul %eax, %eax
11 00001097  b9 ff ff ff ff                              mov $-0x01, %ecx
12 0000109C  48 3b c1                                    cmp %rcx, %rax
13 0000109F  0f 86 01 00 00 00                           jbe 0x00000000000010A6
14 000010A5  cc                                          int3
```

Listing 2.2: The compiled result of the `imul` test function

```
1 function test(a) {
2 a = Math.sqrt(a);
3 a = Math.sqrt(a);
4 a = Math.sqrt(a);
5 a = Math.sqrt(a);
6 a = Math.sqrt(a);
7 return a;
8 }
```

Listing 2.3: A test function for the `sqrt` instruction

```
1 0000107C  41 81 fb f0 ff 01 00                        cmp $0x1FFF0, %r11d
2 00001083  0f 87 1c 03 00 00                           jnbe 0x00000000000013A5
3 00001089  c4 e1 f9 6e c0                              vmovq %rax, %xmm0
4 0000108E  f2 0f 51 c0                                 sqrtsd %xmm0, %xmm0
5 00001092  f2 0f 51 c0                                 sqrtsd %xmm0, %xmm0
6 00001096  f2 0f 51 c0                                 sqrtsd %xmm0, %xmm0
7 0000109A  f2 0f 51 c0                                 sqrtsd %xmm0, %xmm0
8 0000109E  f2 0f 51 c0                                 sqrtsd %xmm0, %xmm0
9 000010A2  c4 e1 f9 7e c1                              vmovq %xmm0, %rcx
```

Listing 2.4: The compiled result of the `sqrt` test function

The functions used for creating specific instructions are run repeatedly during data transmission. This ensures that they are converted into binary code by the JIT compiler (see Section 2.4). We verify which instructions were generated by running code through IonMonkey – the optimizing compiler used in Firefox – using Firefox' js-shell.

A test function in Listing 2.1 contains multiple `Math.imul()` calls. The resulting binary in Listing 2.2 contains the `imul` instruction. Similarly, `Math.sqrt()` [MDN23b] translates to the `sqrt` instruction (see Listing 2.3 and Listing 2.4).

3 Implementing JavaSQUIP

As described in Section 2.5, there are some challenges to overcome when implementing a timing attack in a browser. This chapter explains how we work around these challenges to implement the JavaSQUIP covert channel.

3.1 General architecture

We build a showcase for JavaSQUIP consisting of two web pages: one receiver and one sender. The aim is to build a showcase that, while showing the capabilities of the covert channel, should reflect as realistic a scenario as possible to also prove the feasibility of a side channel attack. The difference between a conventional side channel attack and a covert channel is that a side channel attack involves a (non-cooperating) victim and an attacker, while covert channels have a cooperating pair of sender and receiver.

One example where an attacker would use this sender-receiver pair is data exfiltration. The sender runs on a page where it can obtain confidential information, but cannot transmit it back to the attacker because of cross-origin resource sharing (CORS) [MDN24c] restrictions. It instead sends the data to a receiver on a different website with less strict CORS settings using JavaSQUIP. From here, the received data can be sent back to the attacker.

The sender in the showcase corresponds to the victim in a side channel scenario. Since a victim does not cooperate in a side-channel attack with an uncooperative sender, the sender is kept as simple as possible. It consists of one thread, sending one bit per timeslice (see Section 3.2) via issuing `Math.imul()` instructions or not.

```
self.onmessage = function(event) {
  console.log("Hello from counter!");
  let buffer = event.data;
  let arr = new Uint32Array(buffer);
  let counter = 0|0;
  for(;;)
  {
    counter = ((counter|0)+1)|0;
    arr[0] = counter|0;
  }
}
```

Listing 3.1: The code for the counting thread

The receiver represents the attacker. The assumption is that an attacker can execute arbitrary scripts, for example as an interactive advertisement on a website. Given this, the attacker can get one thread co-located with the victim and exploit the SQUIP side channel to gain information. Details on the implementation are described in the following chapters.

3.2 Getting accurate timing

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

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

For the synchronized clock, we can use `Date.now()` [MDN22b], which has a 1 ms accuracy. This allows us to partition the duration of the transmission into 1 ms *timeslices*. During one timeslice, a single bit can be transmitted by the existence or absence of scheduler queue contention. The alignment of bytes is solved by starting the transmission on a multiple of 8 timeslices since the start of the epoch (see Section 2.5.1). Synchronizing with `Date.now()` [MDN22b] works, but only 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 0.1 ms, we need to provide our own timing.

We solve this with a counting worker, which continually increases one variable in the `SharedArrayBuffer`. As explained in Section 2.5.2, the other receiver workers can then access this counter and use it to measure execution times with adequate accuracy.

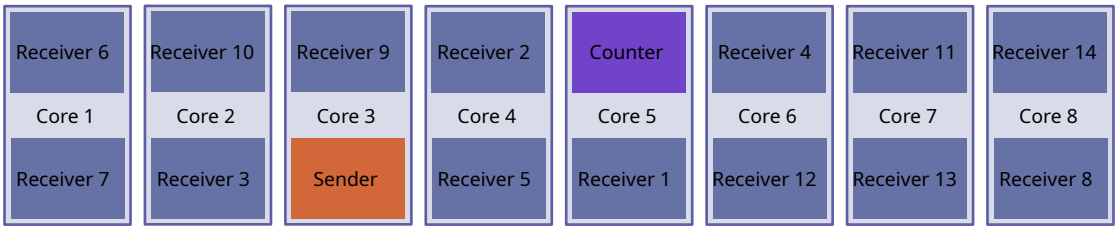


Figure 3.1: One possible co-location configuration. The counting thread is co-located with Receiver 1, the Sender is co-located with Receiver 9.

Listing 3.1 shows the counting thread's code. After initialization, it continually increases the counting variable and stores it to the `SharedArrayBuffer`. The paper on *Fantastic Timers* [SMGM17] explains the concept in more depth.

3.3 Co-locating with the sender

JavaScript does not provide the functionality of binding to a particular core. The receiver tries to force co-location by crowding the CPU. The receiver has one thread running on each virtual core, leaving just one virtual core open for the sender's thread to occupy. To do this the receiver creates as many receiving workers as the number of virtual cores in the machine, minus two: one for the counting thread (see Section 3.2), and one for the sending thread. In our experiments, we use an *AMD Ryzen 7 5800X* as representative for Zen 3 and an *AMD Ryzen 7 7700X* as representative for Zen 4. Both are 8-core CPUs, so the experiments are run with $8 \times 2 - 2 = 14$ receiver workers. Since every thread does continuous work, the operating system will most likely assign each thread to a unique virtual core.

To receive the message, one of the worker threads needs to be co-located with the sending thread. Assuming a uniform distribution for the threads, we can calculate the theoretical co-location probability. The only way that the sender is not co-located with a receiver is that the sender is co-located with the counting thread, which is a $\frac{1}{15}$ chance. Thus, the theoretical co-location probability is $\frac{14}{15} \approx 93.3\ \%$. Figure 3.1 illustrates one possible configuration of threads on the 16 virtual cores.

To figure out which receiver is the one co-located with the sender, we need to take advantage of some characteristic of the transmission. With some optimizations of the receiver workers – described closer in Section 4.3 – we are able to get the contention of two co-located receiver workers very low. The counting thread does not cause much contention either. This means that if a '`1`' is sent, the sender will cause the highest scheduler queue contention level – i.e. the longest execution time – among all the physical cores. Detecting the correct receiver worker is now a matter of selecting the receiver with the highest contention; this will correctly identify the co-located receiver if a '`1`' is

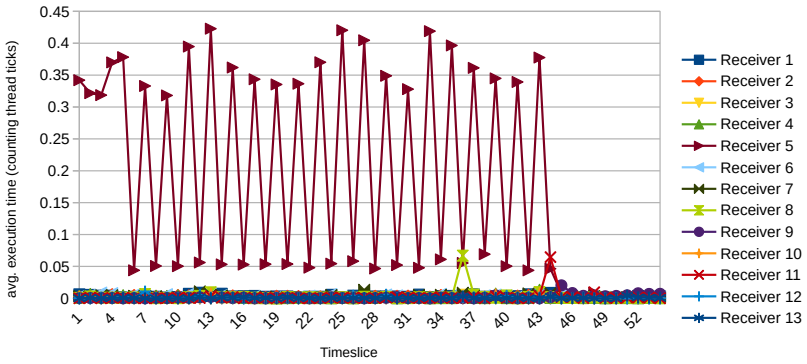


Figure 3.2: Average execution time per timeslice (counting thread ticks) during a test run of sending alternating '0' and '1' bits

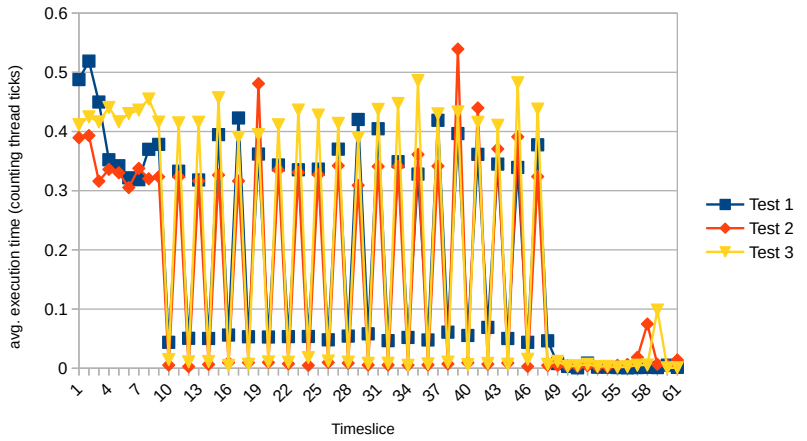


Figure 3.3: Average execution time per timeslice (ticks) of the co-located receiver during three test runs of sending alternating '0' and '1' bits

sent, and a random receiver if a '0' is sent. Since in this case all receivers report low contention, a '0' is still registered regardless of the selected receiver.

Figure 3.2 displays the contention of each receiver for each timeslice. One can easily see that receiver 5 is co-located with the sender; it registers alternating high and low contention, in accordance with the alternating '1' and '0' bits sent by the sender. Meanwhile, the contention measured by the other receivers stays low.

3.4 Targeting a scheduler queue

JavaSQUIP targets AMD Zen 2, Zen 3 and Zen 4; these architectures support multiplication only on ALU1 [AMD20b]. When sending a '1' bit, JavaSQUIP makes multiple

```
1  function send_zero(until) {
2    until = +until;
3
4    while(+Date.now() < +until) {
5      // simply busy waiting
6    }
7  }
8
9  function send_one(until) {
10   until = +until;
11
12   let i = dummy[0]|0;
13   let j = dummy[1]|0;
14   while(+Date.now() < until) {
15     i = (((i|0) + 100069) % 100103) | 0;
16     j = (((j|0) + 997) % 13) | 0;
17
18     // block scheduler queue with 100 imul instructions
19     i = Math.imul(i|0, j|0) | 0;
20     i = Math.imul(i|0, j|0) | 0;
21     // repeats 100 times
22     i = Math.imul(i|0, j|0) | 0;
23     i = Math.imul(i|0, j|0) | 0;
24   }
25
26   // use the result so the calculation is not optimized away
27   dummy[1] ^= i|0;
28 }
```

Listing 3.2: The functions `send_zero()` and `send_one()`

codependent calls to `Math.imul()`, which translate to the `imul` instruction. This fills up the scheduler queue of ALU1, eventually causing a queue stall.

In Listing 3.2 and Listing 3.3, many features of the JavaScript environment are utilized to ensure the correct behavior of the program. The most noticeable is the use of `|0` for many operations with numbers. It forces the interpretation of the given constant or variable as an integer, thus simplifying the operation. For example, `1.4|0 + 4.5|0` will be interpreted as an integer addition and evaluate to `5`.
Similarly, using a variable with a `+` in front – as in line 2 – will force a reinterpretation as a floating-point number. We use `Atomics.load()` [MDN24d] to get the current value of the counting thread. It is an atomic operation, so thread safety and correctness of the value are ensured.

The codependence of the `imul` operations ensures that they have to be executed one after the other, in order. It is ensured simply by using the result of one operation as the input for the next. The input for the `imul` operations is calculated in lines 15 and 16 of Listing 3.2 based on a global variable `dummy`. This prevents the JIT-compiler from pre-computing the result and optimizing away the `imul` calls.

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

  let dummy = +start_value;

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

  dummy = +Math.sqrt(+dummy);
  dummy = +Math.sqrt(+dummy);
  // repeats 10 times
  dummy = +Math.sqrt(+dummy);
  dummy = +Math.sqrt(+dummy);

  dummy = dummy|0;

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

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

  return {
    dummy: dummy|0,
    time: ((end_time|0) != (start_time|0))|0
  };
}
```

Listing 3.3: The squip() function

Listing 3.2 shows two functions of the sending process. The functions are for sending a zero or a one respectively, in a given time frame. The parameter `until` gives the end of the timeslice to send a bit. The function `send_one()` creates contention with `imul` instructions, while `send_zero()` only waits until the timeslice is over.

3.5 Detecting scheduler queue contention

The receiving process needs to detect whether the sender has sent a '`1`' or a '`0`', which is equivalent to detecting scheduler queue contention. This is implemented by issuing several `imul` instructions, and measuring the delay with the help of a separate counting thread.

Listing 3.3 shows how the receiver issues 10 `sqrt` and 20 `imul` instructions, retreiving the counter value before and after (see lines `7` and `23`). If the retire control unit is not blocked by an overflowing scheduler queue, the instructions can be reordered so both reads of the counting value are right after one another. This results in the measured time delay being zero, which means no scheduler queue contention took place. If the two values are not equal, we can deduce that the program flow was held up by scheduler queue contention: The program flow was not reordered because the scheduler was stalled, and the counting thread increased the counter between the execution of line `7` and line `23`.

The combination of `sqrt` and `imul` instructions is selected because they work on different scheduler queues. If the scheduler queue for ALU1 overflows because of the additional `imul` instructions, the whole instruction queue stalls – which prevents the reordering of lines `7` and `23`. In this case, the `sqrt` instructions are also stalled, further increasing the execution time between lines `7` and `23`. In the case that no stall occurs, the additional `sqrt` instructions do not hinder the program flow, since they target a different scheduler queue.

In line `27`, we do not check the actual difference between `start_time` and `end_time`, but rather only return a '`1`' if the two are different, and a '`0`' otherwise. Since we can assume that a difference in `start_time` and `end_time` is an indicator of scheduler queue contention, this gives a good measurement; a '`0`' is returned when there is no contention, and longer execution times do not throw off the average, since at maximum a `1` can be returned. Returning the actual difference of the counting thread values, i.e. returning the actual execution time, mostly gives equal results. Some measurements are, however, thrown off by longer delays in execution, which increases the average execution time unproportionally.

The `squip` function is run 100 times per timeslice, summing up the return values of each iteration. At the end of the timeslice, this sum is divided by 100, resulting in the relative likelihood of a queue stall between time measurements, or *stall likelihood*. If the stall likelihood is above the threshold (see Section 4.1), a '`1`' is registered, otherwise a '`0`'.

Figure 3.3 shows that scheduler queue contention usually affects around $30 - 50$ % of the tries in a timeslice. Since this metric is closely related to the execution time of the measurement, the two terms are used interchangeably in the rest of this thesis.

4 Optimization

JavaSQUIP is intended to show the potential of the SQUIP exploit as an attack in a browser setting. Because of this, we aim to make it as fast and stable as possible. The transmission speed is limited by the environment, as the time slices cannot be smaller than 1 ms. The remaining settings and features are adjusted so the transmission is as accurate as possible at the speed of 1000 bits/s.

4.1 Delay Threshold

The first parameter we optimize is the delay threshold. It determines whether the summed up delay of a timeslice is registered as a '0' or a '1'. If the threshold is too high or too low, results are skewed towards registering a '0' or a '1' respectively.

To tune this parameter, the resulting bitstream is checked against the actual sent stream. All of the bits that are wrongly registered as '1' or wrongly '0' are tallied up separately. Then the delay threshold is adjusted until faulty '1' bits and faulty '0' bits are approximately equally frequent, meaning that no skew toward high or low bits is present and the delay threshold is set optimally.

Figure 4.1 shows the distribution of execution times per timeslice with and without scheduler queue contention. Timeslices without scheduler queue contention show very short execution times, except for a small number of outliers. Execution times for timeslices with scheduler queue contention vary more, though they are clearly separable from those without contention. The empirical optimum on our test setup (see Section 5.1) is **0.21**. It gives the best margin for '0' and '1' transmissions.

4.2 Number of instructions for a queue stall

The sender blocks a targeted scheduler queue by sending a large number of instructions to it. The number of instructions is another tunable parameter: it needs to be big enough to cause a queue stall and a spike in execution time, yet it needs to be as small as possible to minimize the chance of a stall bleeding into the following timeslice.

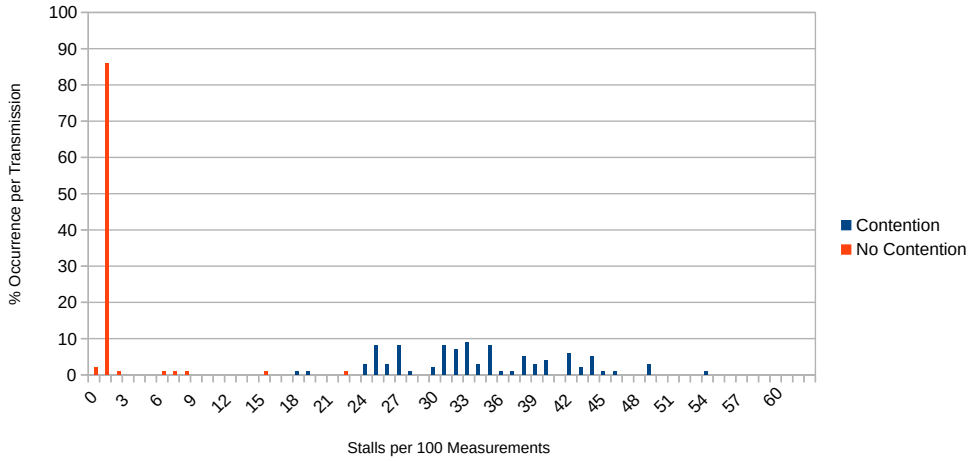


Figure 4.1: Distribution of stall likelihood per timeslice with and without scheduler queue contention by the sender

To tune the number of `imul` instructions, a bleed-over test is constructed. The test consists of a string of alternating '`1`' and '`0`' bits, so contrasts between different bits could be easily observed. The amount of `imul` instructions is tuned by successively reducing the number and running a test. The point where no spike in execution time is registered any more is set as the lower limit for the amount of `imul` instructions. Figure 3.2 shows the final tuned system; the spikes for a '`1`' are pronounced, while the execution time in a '`0`' timeslice is only slightly higher than for the remaining receivers. For our test setup, **200** co-dependent `imul` instructions are the optimum. Small changes to the number of instructions do not create meaningful differences in performance. Figure 3.3 illustrates that this system is quite robust across different tests.

4.3 Minimizing contention by the receivers

The receiver detects a queue stall by measuring whether two operations were reordered to be right after one another, or whether the scheduler queue was stalled between the two operations by contention. To detect a stall, the execution time for `imul` instruction calls is measured. See Section 3.5 for a detailed explanation of the stall detection.

To improve the signal-to-noise ratio, the receivers should create as little contention as possible themselves. This is helpful in the transmission, as it reduces the contention for two co-located receivers, maximizing the contrast to the receiver that is co-located with the sender (see Section 3.3). Three optimizations are employed in the receiver worker: adjusting the *number of stall instructions*, using *buffer instructions*, and a *delay loop*.

```
let k = 0|0;
while(k < 1000|0) {
  dummy  ^= k|0;
  k += 1;
}
```

Listing 4.1: The delay loop after a measurement

Number of stall instructions. To get a more stable result, several successive `imul` instructions are used. On the other hand, too many instructions would create contention themselves, so fewer instructions would be better. A detection test is set up to determine the minimum amount of instructions to be able to detect contention by the sender.

The detection test is constructed similarly to the bleed-over test: alternating '`1`' and '`0`' bits are received, and the delays measured. The number of `imul` instructions in the receiver worker is reduced until no signal is retrievable, resulting in a lower bound for the number of `imul` instructions in the receiver worker. For Zen 4, the optimum number of `imul` instructions is **20**.

Buffer instructions. In the measurement function, **10** `sqrt` instructions serve as a buffer to delay the `imul` instructions: they delay the execution, while not creating contention on ALU1. This means that the effect of a stalled queue is more pronounced, because the `sqrt` instructions will also be stalled. On the other hand, they have little effect on an unstalled queue, since they operate on a different scheduler. The implementation can be seen in Section 3.5.

Delay loop. A delay loop is added after each measurement to clearly separate measurements. To this effect, we fill the scheduler queue with other operations for a short time between measurements. Since no normal delay function gives a better accuracy than 1 ms, a busy-waiting loop is used. Listing 4.1 shows the delay loop that is repeated **1000** times after each measurement.

Figure 3.2 demonstrates these measures working: Receiver 5 – which is co-located with the sender – receives a clean signal of alternating '`1`' and '`0`' bits. Meanwhile, all the remaining receivers – which are co-located either with each other or with the counting thread – show very little contention.

5 Evaluating JavaSQUIP

To evaluate and measure different parameters and settings, we set up a test scenario for JavaSQUIP. It consists of two web pages communicating with each other solely over the JavaSQUIP covert channel.

5.1 Test setup

To evaluate JavaSQUIP on both the Zen 3 and Zen 4 architecture, we test on an *AMD Ryzen 7 5800X* as representative for Zen 3 and an *AMD Ryzen 7 7700X* as representative for Zen 4.

The test setup consists of two webpages: a sender and a receiver. The sender simulates the target, and the receiver simulates a page that is controlled by the attacker. The attacker has little or no control over the target – we keep the sender page minimal as a consequence. The sender has only one working thread. The working thread is sending either a '`0`' (no action) or a '`1`' (contention with codependent `Math.imul()` calls) for each timeslice. With more control over the receiver, the attacker can ensure co-location from this side. For our tests on the *AMD Ryzen 7 5800X* (Zen 3) and *AMD Ryzen 7 7700X* (Zen 4), we need 14 receiving workers (see Chapter 3.3).

To distinguish a message from random noise on the CPU, the sender sends a preamble before the actual data. The preamble is a sequence of the numbers from 100 through 109. If the receiver sees two ascending bytes between 100 and 109, it is highly likely that this is part of the preamble – a message was detected. For easier evaluation of the transmission accuracy, the message is also stored on the receiver's side, so that the number of "faulty '`1`'" and "faulty '`0`'" bits can be automatically calculated. In our test setup, messages of 500 random bytes are sent at a transmission speed of 1000 bits/s. This results in a transmission time of $\frac{500 \times 8}{1000} = 4$ seconds.

5.2 Speed

The sender and receiver need to know the desired transmission speed beforehand to match up their timeslices. This means that the transmission speed is a fixed value and needs no further calculation. Due to the reduced resolution of the `Date.now()` timer used for the time slices (see Section 3.2), our best achievable speed is 1000 bits/s.

There is, however, still some overhead for the preamble (see Section 5.1). It is a fixed size of 10 bytes regardless of the message, so the relative overhead tends to 0 % as the message size increases.

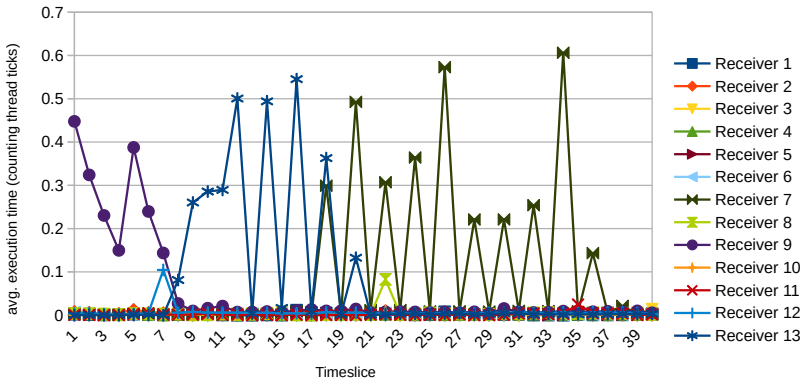


Figure 5.1: Execution time per timeslice during a test run of sending alternating '0' and '1' bits with `stress -cpu 1`

5.3 Reliability

There are two aspects that affect the reliability of the covert channel: First, the sender being co-located with a receiver, and second, the resilience of the covert channel against noise.

Co-location. To measure the actual co-location probability, we use the preamble system as described in Section 5.1. Threads are unlikely to be relocated during the 4 seconds of a transmission, and the preamble is likely to be recognized if co-location is given (see noise resilience further down). We can conclude that the rate of recognized preambles is a good measurement for the actual co-location probability.

In our tests, we find an average co-location probability of 88.7 % in a Zen 3 architecture and 96 % in Zen 4. This is consistent with the theoretical co-location probability of 93.3 % calculated in Section 3.3.

When tested under stress with `stress -cpu 1`, we achieve an average co-location probability of 53.7 %. Most likely, the sender is co-located to a receiver more often than that, but the corresponding receiver is not able to detect the preamble due to the noise. When run simultaneously with `stress -cpu 2`, JavaSQUIP is not able to receive the preamble. As a result, co-location is never detected and transmission of data is not possible.

Noise. For measuring noise resistance, we only take those transmissions into account where the preamble was recognized. This results in an average transmission accuracy of 99.2 % on Zen 3, and 99.3 % on Zen 4. With `stress -cpu 1`, we observed an accuracy of 74 %.

Figure 5.1 shows the effect of outside CPU stress on the measurement. Although the `stress` command creates more noise in the measurement, the overall contention levels go down (compare with Figure 3.2). This is because `stress` does not target a specific scheduler queue, but instead takes up CPU time with a varied set of instructions. The result is more relocation of threads (the sending thread is co-located with receiver 13 at first, later with receiver 7), and less contention.

Effective transmission rate. We define the *effective transmission rate* as the product of a channel's transmission rate and its channel capacity according to Shannon's noisy coding channel theorem [Mac03]. For example, a transmission at 2000 bits/s with an error rate of 10 % has an effective transmission rate of 1062.01 bits/s:

$$2000 \text{ bits/s} * \left(1 - \left(0.1 \; log_2 \frac{1}{0.1} + (1 - 0.1) \; log_2 \frac{1}{1 - 0.1}\right)\right) = 1062.01 \text{ bits/s}$$

The effective transmission rate may differ from the usable bandwidth of the covert channel, depending on the transmission protocol and error correction. Nevertheless, it gives a good estimate to compare usable bandwidth between different covert channels.

For JavaSQUIP, we take into account both the co-location probability and the transmission accuracy. By multiplying these values, we calculate the *cumulative error rate* – the probability that a given bit is flipped during transmission. The cumulative error rate is $1 - 88.7\ \% * 99.2\ \% = 0.1201$ for Zen 3 and $1 - 93.3\ \% * 99.3\ \% = 0.0745$ for Zen 4.

We calculate an effective transmission rate of 470.36 bits/s on Zen 3:

$$1000 \text{ bits/s} * \left(1 - \left(0.1201 \; log_2 \frac{1}{0.1201} + (1 - 0.1201) \; log_2 \frac{1}{1 - 0.1201}\right)\right) = 470.363 \text{ bits/s}$$

The effective transmission rate on Zen 4 is 617.63 bits/s:

$$1000 \text{ bits/s} * \left(1 - \left(0.0745 \; log_2 \frac{1}{0.0745} + (1 - 0.0745) \; log_2 \frac{1}{1 - 0.0745}\right)\right) = 617.634 \text{ bits/s}$$

6 Comparison to other browser covert channels

We have shown that JavaSQUIP is a fast and robust cross-instance covert channel for modern browsers. To further illustrate the power of JavaSQUIP, we will now compare its performance to similar modern browser-based covert channels. For a fair comparison, they need to exhibit data transfer between browser instances, with all the current security mitigations in place (see Section 2.5).

The selected covert channels are the following:

- *Event Loops* [VK17]
- *Hardware Interrupts* [LGS+17]
- *Port Contention* [RMBO22]

As browser-based timing covert channels, they share a few traits. All of them need a way to create contention and a way to measure it. The common solution is with a sender that either idles or performs an action, while the receiver continually monitors the execution time of a loop that is sensitive to this action.

6.1 Event Loops

The *Loophole* Timing Attack [VK17] uses shared event loops to get timing information. An event loop is a piece of code that continually checks for an event condition and starts registered event handlers if the event has occurred. An example would be an event loop that checks for a *Space* key press, which starts an event handler to start a video on the website.

The version of the covert channel attack we are interested in relies on two pages sharing the same *renderer process*. The renderer process takes care of the parsing, execution, and rendering of the JavaScript sandbox. Consequently, pages in the same renderer process also share resources more closely. This is used to get access to shared event loops with more accurate timing information.

The following ways of sharing a renderer process have been found:

- embedding in an Iframe
- creation of the new page by the renderer, e.g. through a link-click, or scripted redirect
- reuse of renderers by Chrome for resource optimization

Having one of these is possible, though it does limit the applicability of this covert channel. The authors of the paper also presented a version of the covert channel that works without a shared renderer process, at the cost of a significant decrease in transmission rate.

Vila et al. [VK17] proposed a covert channel where the receiver continually checks the delay of an event handler that is triggered by a shared event loop. With a shared renderer process, they achieved a transmission speed of **200 bits/s**. Their solution without a shared renderer process ran at a transmission speed of only **5 bits/s**. Vila et al. did not discuss the error rate of transmissions in their paper.

6.2 Hardware Interrupts

In their paper on *Practical Keystroke Timing Attacks in Sandboxed JavaScript* [LGS+17], Lipp et al. explored a side channel based on hardware interrupts. To get information about a hardware interrupt being triggered, a script continually requests and logs the time increase of `performance.now()` [MDN22a]. If this script is preempted by a hardware interrupt (e.g. for a keystroke), the delta to the last measured timestamp is significantly higher. This allows the attacker to infer that a hardware interrupt was triggered.

To convert this side channel into a covert channel, the researchers needed a hardware interrupt that the sending web page could trigger. They found that issuing an `XMLHttpRequest` to an invalid site would cause an I/O interrupt. The sender would now issue these requests to send a '`1`' and remain idle to send a '`0`'. The receiver measures the bits the same way as described above – by continually looking up the value of `performance.now()`.

This hardware interrupt-based covert channel has no notable limitations: It works in a cross-browser setting on a wide range of devices and processors. With time slices of 40 ms length, the resulting transmission speed is **25 bits/s**. Lipp et al. [LGS+17] did not share the error rate of their covert channel.

6.3 Port Contention

The principle of port contention is the most similar to JavaSQUIP's scheduler queue contention. The receiver runs a loop of issuing an instruction on a particular port and measures its execution time. Whenever the sender issues an instruction on the same port, the receiver's instructions on that port take longer, thus delaying the measurement loop.

In *Port Contention Goes Portable* [RMBO22], Rokicki et al. showcased a feasible application of port contention in a browser-based covert channel. As opposed to the 1-sender multi-receiver approach of JavaSQUIP (see Section 3.1 and Section 3.3), their solution features one receiver and multiple senders. The measurement loop of the receiver was implemented in *WebAssembly* to directly issue a particular instruction. Since the single receiver is sandboxed in JavaScript, it can neither bind to a physical core, nor know which core it is on. Consequently, the sender creates as many threads as there are cores on the machine. The sender was implemented as a native program running unprivileged code. This gave the researchers the opportunity to bind each thread to a different core, ensuring that one of them would be co-located with the receiver.

Rokicki et al. [RMBO22] discussed the possibility of altering their covert channel to be fully within two browser instances. They proposed a similar solution to the problem of co-location as the one we have implemented for JavaSQUIP. On this basis, they believably argued that this fully web-based attack would have the same transaction speed as the version with a native sender at **200 bits/s**.

Covert channel	raw capacity	true capacity
Event Loops (shared renderer)	200 bits/s	*n.a.*
Event Loops	5 bits/s	*n.a.*
Hardware Interrupts	25 bits/s	*n.a.*
Port Contention	200 bits/s	131.03 bits/s
JavaSQUIP(Zen 3)	1000 bits/s	470.36 bits/s
JavaSQUIP(Zen 4)	1000 bits/s	617.63 bits/s

Table 6.1: A comparison of transmission speeds between browser-based covert channels.

They reported a packet loss rate of 5.5 % and an error rate of 1 %, resulting in a cumulative error rate of $1 - (1 - 5.5\ \%) * (1 - 1\ \%) = 0.06445$. The resulting effective transmission rate is **131.03 bits/s**:

$$200\ \text{bits/s} * \left(1 - \left(0.0645\ log_2 \frac{1}{0.0645} + (1 - 0.0645)\ log_2 \frac{1}{1 - 0.0645}\right)\right) = 131.028\ \text{bits/s}$$

6.4 Feasibility of JavaSQUIP

Affected CPUs. JavaSQUIP exploits a vulnerability in CPUs with split-scheduler design. It additionally requires SMT to co-locate the sender with the receiver. This means that JavaSQUIP only works on clients running on a CPU with these features present and enabled. This limits the attack surface, though the number of affected computers is still significant. Gast et al. [GJS+22] found that AMD's Zen 2, Zen 3 and Zen 4 architectures are affected, along with the Apple M1. As of Q3 of 2024, AMD holds a market share of around 33 % in CPUs [Als24]. Most of these CPUs run under the Zen 2, Zen 3 and Zen 4 architecture, meaning that large portion of current computers is susceptible to this attack.

Website prerequisites. Some of the above-mentioned covert channels need a particular environment in the browser to mount the attack. The *Loophole* covert channel needs a shared renderer process. While the port contention approach can theoretically work without a native sender, the receiver still requires *WebAssembly* support to work.

JavaSQUIP on the other hand has none of these limitations. The covert channel can be established on any two browser instances where normal JavaScript can be run. Other than `Date.now()` [MDN22b] and `SharedArrayBuffer` [MDN22d], no higher-level APIs are needed. As an example, even two malicious adverts on different browser instances could conceivably communicate with each other with JavaSQUIP.

Speed. JavaSQUIP transmits data at a speed that is currently unmatched by any covert channel in a browser with up-to-date security settings. Its theoretical transmission capacity is **1000 bits/s**. We observe an effective transmission speed of **470.36 bits/s** and **617.63 bits/s** for Zen 3 and Zen 4 respectively. This is enough bandwidth to support a live transmission of all user inputs, or to transmit a short audio recording within minutes. JavaSQUIP's effective transmission speed surpasses the effective speed of all other covert channels mentioned in this chapter at least by a factor of 3.

7 Discussion of Countermeasures

The preceding chapters have shown that JavaSQUIP is a powerful covert channel. We want to discuss several ways to impede or prevent the usage of this vulnerability for nefarious purposes. First, we will discuss possibilites to alter the three features needed to implement JavaSQUIP: *timing functions*, *multithreading*, and *low-level operations*. We finish by discussing more ways of *preventing co-location* of the sender and receiver.

7.1 Timing functions

JavaSQUIP uses two timers: `Date.now()` [MDN22b] and a counting thread (see Section 3.2). The counting thread is based on multithreading, which will be discussed in the next section.

The resolution of `Date.now()` directly affects the available transmission speed. The usefulness of JavaSQUIP can be easily decreased by decreasing the time resolution. On the other hand, going below 1 ms accuracy would make the feature too weak. Many legitimate programs need this accuracy in timing measurements, for example for performance assessments of a website.

Even if the resolution of `Date.now()` was reduced, the covert channel would still be able to synchronize clocks between sender and receiver. Thus, there is no feasible way to counteract the JavaSQUIP attack by decreasing timer accuracy.

7.2 Multithreading

Removing the support for `Web Workers` [MDN22c] or a `SharedArrayBuffer` [MDN22d] would break JavaSQUIP. Obviously, disabling these features outright is not a satisfactory

solution. It would force every website to run all code in the main thread, drastically reducing responsiveness for websites with bigger computational needs.

One possibility would be that browsers provide a way to disable access to these resources on a *per-site* or a *per-script* basis.

With *per-site* restriction of `Web Workers`, a user could prevent an untrusted website from implementing JavaSQUIP. It is not reasonable to ask the user's permission to this widely used API, so only a blacklist by the user would be feasible. Most users would likely not use this feature, making it generally ineffective.

A *per-script* permission system could give a benign website the option to load scripts from an untrusted source (e.g. third-party advertisements), running them with constrained permissions. This way, the host website could still make use of the multithreading features while preventing a JavaSQUIP attack by third-party scripts. Because there is no ad-hoc way for the browser to tell a trusted script from an untrusted one, website developers would have to explicitly give the information to the browser not to trust a script. Again, this mitigation would most likely lose its effect due to not being used.

7.3 Low-level operations in JavaScript

JavaSQUIP targets ALU1 with `imul` instructions using `Math.imul` [MDN22e]. Browser providers could disable this API call without affecting too many websites, but there are simple workarounds for this.

WebAssembly [Web24] is currently supported in all major browser engines. It promises to provide web developers with faster execution times by circumventing JavaScript's interpreter and just-in-time compiler. Pre-compiled WebAssembly binaries can be executed directly in the browser.

JavaSQUIP could easily be implemented in WebAssembly, using its fine-grained control over instructions to target a scheduler queue. WebAssembly is a rising technology – more and more websites are implementing it [MWJR19]. It is unlikely that browser developers are going to disable support for it. Consequently, there is currently no way to prevent a website from targeting a scheduler queue.

7.4 Preventing co-location

Because JavaSQUIP depends on co-location of the sender and one of the receivers, preventing co-location would present a way to make the covert channel impossible. The simplest way would be to disable simultaneous multithreading, though few users will accept this because of the performance loss of about 25 % [Lar18; Cut20]).

Process-based separation of physical CPU cores also works to mitigate the vulnerability. This approach still makes use of SMT, but prevents threads of different processes to be co-located. This way, multi-threaded programs (and in extension websites) can still benefit from the technology, while SMT-based side channels are fundamentally prevented. SMT-COP [TP19] expands on this idea. This would have to be implemented at the scheduler level, requiring an operating system-level update. Similar to the security update to mitigate Spectre [KHF+19], this update would have a negative impact on the computer's overall performance. Implementing SMT-COP results in a performance loss of 4 % to 10 %, depending on the workload.

8 Conclusion

In this thesis, 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, showing an advantage in performance over all other current JavaScript-based covert channels [VK17; LGS+17], in particular over attacking single-scheduler based architectures [RMBO22].

Additionally, we observed a co-location rate of 88.7 %/96 % with a transmission accuracy of 99.2 %/99.3 % (Zen 3/Zen 4). 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. Its speed and accuracy enable an attacker to transmit large amounts of data, such as images or short audio clips, within the lifetime of a malicious web page.

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. We have shown that there are few options for counteracting JavaSQUIP and similar attacks. All of the solutions proposed in this paper have requirements that make them unlikely to be implemented. Thus, we can expect JavaSQUIP – along with many other browser-based covert channels – to remain viable for the near future.

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.

[Als24] Thomas Alsop. *Intel/AMD x86 computer CPU market share 2024 | Statista*. 2024. URL: `https://www.statista.com/statistics/735904/worldwide-x86-intel-amd-market-share/` (visited on 09/23/2024).

[AMD20a] AMD. *Software Optimization Guide for AMD EPYC 7002 Processors*. Mar. 2020.

[AMD20b] AMD. *Software Optimization Guide for AMD EPYC 7003 Processors*. Nov. 2020.

[AMD23] AMD. *Software Optimization Guide for the AMD Zen4 Microarchitecture*. Jan. 2023.

[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.

[Cut20] Ian Cutress. *Investigating Performance of Multi-Threading on Zen 3 and AMD Ryzen 5000*. 2020. URL: `https://www.anandtech.com/show/16261/investigating-performance-of-multithreading-on-zen-3-and-amd-ryzen-5000/2`.

[Fou24] Electronic Frontier Foundation. *Certbot*. 2024. URL: `https://certbot.eff.org/` (visited on 06/07/2024).

[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.

[Goo23] Google. *Align performance API timer resolution to cross-origin isolated capability*. 2023. URL: `https://chromestatus.com/feature/6497206758539264` (visited on 01/30/2023).

[HBBP14] Grant Ho, Dan Boneh, Lucas Ballard, and Niels Provos. "Tick Tock: Building Browser Red Pills from Timing Side Channels". In: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, Aug. 2014. URL: https://www.usenix.org/conference/woot14/workshop-program/presentation/ho.

[Int19] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

[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.

[Lar18] Michael Larabel. *Intel Hyper Threading Performance With A Core i7 On Ubuntu 18.04 LTS*. June 2018. URL: https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4.

[LGS+17] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. "Practical keystroke timing attacks in sandboxed javascript". In: *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*. Springer. 2017, pp. 191–209.

[Mac03] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[MDN22a] MDN. *performance.now() - Web APIs | MDN*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now (visited on 12/15/2022).

[MDN22b] 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).

[MDN22c] 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).

[MDN22d] MDN. *SharedArrayBuffer - JavaScript | MDN*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer (visited on 12/14/2022).

[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).

[MDN23a] MDN. *Performance: timeOrigin property - Web APIs | MDN*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/timeOrigin (visited on 11/03/2023).

[MDN23b] MDN. *Math.imul() - JavaScript | MDN*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sqrt (visited on 12/28/2023).

[MDN24a] MDN. *JavaScript | MDN*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 10/15/2024).

[MDN24b] MDN. *A crash course in just-in-time (JIT) compilers*. 2024. URL: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/ (visited on 10/15/2024).

[MDN24c] MDN. *Cross-Origin Resource Sharing (CORS) | MDN*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (visited on 10/16/2024).

[MDN24d] MDN. *Atomics.load() - JavaScript | MDN*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics/load (visited on 09/19/2024).

[MWJR19] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer. 2019, pp. 23–42.

[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.

[SLG18] Michael Schwarz, Moritz Lipp, and Daniel Gruss. "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks." In: *NDSS*. Vol. 18. 2018, p. 12.

[SMGM17] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC*. 2017.

[TEL95] Dean M Tullsen, Susan J Eggers, and Henry M Levy. "Simultaneous multithreading: Maximizing on-chip parallelism". In: *Proceedings of the 22nd annual international symposium on Computer architecture*. 1995, pp. 392–403.

[TP19] Daniel Townley and Dmitry Ponomarev. “Smt-cop: Defeating side-channel attacks on execution units in smt processors”. In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2019, pp. 43–54.

[VK17] Pepe Vila and Boris Köpf. “Loophole: Timing attacks on shared event loops in chrome”. In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 849–864.

[Web24] WebAssembly. *WebAssembly*. 2024. URL: `https://webassembly.org/` (visited on 09/27/2024).