Port Contention Goes Portable: Port Contention Side Channels in Web Browsers

Thomas Rokicki
Univ Rennes, CNRS, IRISA
Rennes, France

Clémentine Maurice
Univ Lille, CNRS, Inria
Lille, France

Marina Botvinnik
Ben-Gurion University of the Negev
Be'er Sheva, Israël

Yossi Oren
Ben-Gurion University of the Negev
Be'er Sheva, Israël

Abstract

Microarchitectural side-channel attacks can derive secrets from the execution of vulnerable programs. Their implementation in web browsers represents a considerable extension of their attack surface, as a user simply browsing a malicious website, or even a malicious third-party advertisement in a benign cross-origin isolated website, can be a victim.

In this paper, we present the first port contention side channel running entirely in a web browser, despite a highly challenging environment. Our attack can be used to build a cross-browser covert channel with a bit rate of 200 bit/s, one order of magnitude above the state of the art, and has a spatial resolution of 1024 native instructions in a side-channel attack, a performance on-par with Prime+Probe attacks. We provide a framework to evaluate the port contention caused by WebAssembly instructions on Intel processors, allowing to increase the portability of port contention side channels. We conclude from our work that port contention attacks are not only fast, they are also less susceptible to noise than cache attacks, and are immune to countermeasures implemented in browsers as well as most side channel countermeasures, which target the cache in their vast majority.

CCS Concepts

• **Security and privacy** → *Web application security*; *Side-channel analysis and countermeasures.*

Keywords

Side Channel; CPU Port Contention; JavaScript; WebAssembly

ACM Reference Format:
Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. 2022. Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30–June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3488932.3517411

1 Introduction

Microarchitectural features such as SMT, out-of-order execution, caches and branch prediction units are designed with the goal of increasing performance. They can, however, be exploited by attackers to derive secrets from the execution of vulnerable programs, and to enable covert communications between processes. As these microarchitectural attacks gain traction in the security community, their attack surface increases two-fold: 1) more and more components are found vulnerable to side channels, and 2) side-channel attacks, which were originally implemented in native code, are being ported to web browsers, expanding the attacker model and crucially increasing the number of potential victims.

While cache side-channel attacks remain the microarchitectural attacks most studied in the literature [23, 24, 27, 43], port contention attacks have also been shown to be a potential attack vector in a technique introduced in 2018 by Aldaya et al. [3], named PortSmash. This attack on Intel CPUs is based on port contention, where CPU ports act as a bottleneck in the execution pipeline. By sharing ports with the victim, the attacker can exploit timing differences caused by the contention of different instructions. PortSmash has a high temporal resolution and can be used, like its counterparts on the cache, to perform side-channel attacks on cryptographic libraries. While port contention attacks restrict the attacker by requiring that it shares the core it executes on with its victim, they are inherently stealthier than attacks on the memory subsystem. They are also immune to most hardware and system countermeasures which, in their vast majority, target the cache [9, 19, 22, 28, 32, 44].

Web browser-based timing attacks, and in particular microarchitectural attacks, are a real threat to security. Indeed, previous work has shown that it is possible to derandomize ASLR completely from JavaScript [15], to spill secrets via transient execution [18], and to craft covert channels of the same order of magnitude as native code approaches: 320 kbit/s for the nominal approach of Prime+Probe in the browser, 8 kbit/s with a receiver in a virtual machine [27], and 200 bit/s when using Chrome's I/O event loop [40]. However, browser vendors have introduced countermeasures against these attacks, targeting high-resolution timers [30, 33] and introducing resource isolation mechanisms [29]. In practice, this entirely mitigated the event loop side channel, and severely hindered Prime+Probe[1]. Covert channels have been developed after the introduction of these countermeasures, but with significantly lower bit rate. To

[1]Although, to the best of our knowledge, no recent implementation of Prime+Probe has been evaluated.

Table 1: Comparison of covert channels in web browsers.

Covert channel	Bandwidth	Runs with current mitigations	Setup
CPU throttling [31]	0.2 bit/s	●	-
Disk contention [38]	5 bit/s	●	-
RIDL (Evict+Reload) [39]	8 bit/s	●	-
DRAM [33]	11 bit/s	●	-
Hardware interrupts [21]	25 bit/s	●	cross-browser
Event loop [40]	200 bit/s	○	cross-browser
Prime+Probe [27]	320 kbit/s[2]	◐	
Prime+Probe [27]	8 kbit/s[1]	◐	cross-VM
Port contention [our work]	200 bit/s	●	**cross-browser**
Port contention [our work]	80 bit/s	●	**cross-VM**

the best of our knowledge, the highest bit rate demonstrated after the countermeasures is 25 bit/s.

When compared to cache attacks such as Prime+Probe, native port contention attacks offer better speed and spatial accuracy, do not require a complex cache profiling step, are more resistant to noise, and, most significantly, can bypass cache-centric countermeasures. Mounting a port contention attack in a browser setting would therefore deliver a real advantage to attackers. Performing such an attack, however, is far from trivial. The basic step of a Prime+Probe cache attack is sequential access to user-controlled memory. It has been shown that even high-level primitives, such as substring searches, can provide this functionality [35]. Port contention, on the other hand, requires an attacker process which is co-located with the victim on the same processor core and executes assembly language instructions carefully chosen to conflict with the victim's instructions. This is highly challenging in a web browser environment:

C1 : In this setting, the attacker's code is written in a highly-abstracted language which is translated into machine code by a just-in-time compiler;

C2 : The attacker has no control over the physical core selected by the browser to execute the attack code;

C3 : Finally, web-based timers have a lower resolution than native hardware-based timers, increasing the attacker's measurement noise.

Our work tackles these challenges, and asks the following questions: *Can port contention attacks be mounted from within the browser? What are the implications of this new attack vector?*

Contributions. The main contributions are as follows:

- We show that port contention can be ported to web browsers via WebAssembly, despite the strong requirements of this attack and the abstraction of the WebAssembly language. This greatly increases the attack surface that is due to port contention (Section 3).
- We propose an automated framework to find which WebAssembly instructions can cause port contention on a given Intel processor (Section 4).
- We demonstrate a side-channel attack on a synthetic example, to evaluate the resolution of our port contention attack. We show that our attack has a spatial resolution of 1024 instructions with a single trace, equivalent to the best microarchitectural attacks in the browser (Section 5).
- We build a covert channel using port contention. With a sender running unprivileged native code and a receiver inside the browser, we obtain a throughput of 200 bit/s, *i.e.*, one order of magnitude higher than modern covert channels in the browser. Table 1 compares the results of our covert channel with the state of the art. In a virtualized setting where the sender is running inside a virtual machine, we reach a throughput of 80 bit/s. We also build a cross-browser covert channel with an estimated throughput of 200 bit/s. (Section 6).

[2]This work was presented before heavy countermeasures against timing attacks. The covert channel is theoretically still implementable, but with a heavily degraded bandwidth.

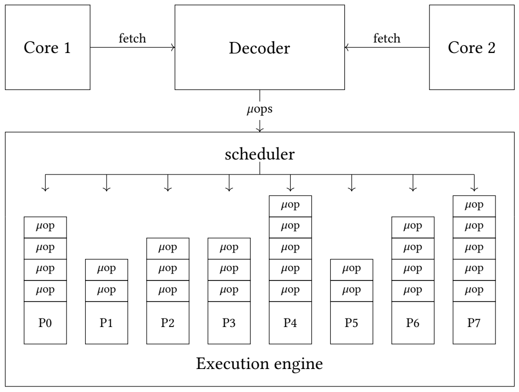


Figure 1: Illustration of the execution pipeline of instructions inside a physical core on an Intel CPU.

2 Background

In this section, we present background information on microarchitectural attacks, and in particular port contention side-channel attacks, JavaScript, WebAssembly, and literature on microarchitectural attacks in the browser.

2.1 Microarchitecture and Port contention

Hyper-Threading and CPU ports. Modern Intel CPUs have an implementation of simultaneous multithreading (SMT) commercially referred to as Hyper-Threading Technology. It aims at allowing more parallelization with the same microarchitectural components. At an abstract level, the CPU splits each of its physical cores into two logical cores, running their own processes. The logical cores are independent at the OS level, acting as different physical cores. At the microarchitectural level, however, they share common hardware, such as L1 and L2 caches, or execution engines.

To optimize out-of-order execution, modern CPUs decompose native instructions into smaller, atomic operations, called micro-operations, or μops. Figure 1 illustrates how the physical core decoder fetches the instructions and decomposes them into μops. The μops are then distributed to the execution engines by the scheduler, through multiple *CPU execution ports*. Each port leads to several execution units that will process the μops. Then, when all μops of an instruction are executed, the instruction is completed and committed to the microarchitecture. The distribution of μops to ports is deterministic, with each execution unit being specialized to process certain types of instructions. For instance, arithmetic μops are distributed to port 0, 1, 5 or 6 (P0156). The port usage of instructions have been documented by Abel and Reineke [2]. The ports are shared by all processes running on the same physical core. This means that threads running on different logical cores, but on the same physical core, output μops to the same CPU ports.

Port contention side-channel attacks. Sharing microarchitectural components between processes can leak information through timing attacks. By timing the execution time of specific operations, attackers can infer the state of the microarchitecture, possibly granting them access to secret information. Aldaya et al. [3] introduced a timing attack based on port contention named PortSmash. As a CPU port can handle a single μop per cycle, it can act as a bottleneck in the flow of operations. Thus, by repeatedly calling and timing instructions with a specific port usage, a spy process can monitor μops from other threads on the same physical core. For instance, an attacker can repeatedly call the `crc32` instruction, which is decomposed into a single P1 μop. This will create a bottleneck on P1. Next, by measuring the execution time of the instruction, the attacker knows if instructions from other processes co-located on the same physical core are distributed on the same port. More specifically, if the attacker's instruction has a longer execution time than usual, this means that another process has issued one or more μops to P1. Aldaya et al. exploited this vulnerability to mount an end-to-end attack on OpenSSL's TLS implementation and recover private keys. Their side channel offer a spatial resolution, *i.e.*, the smallest event they can distinguish, of a single instruction.

Bhattacharyya et al. [5] leveraged port contention as a side channel in their speculative execution attack SMoTherSpectre, with a spatial resolution of a single victim instruction. They also presented a methodology to find vulnerable gadgets. Gras et al. [14] introduced ABSynthe, an automated framework to identify on-core contention-based side channels. Their blackbox model does not focus on specific microarchitectural components, e.g., CPU ports, but on the interaction between different instructions.

Other microarchitectural side-channel attacks. The cache is a small, fast memory. It is used to dynamically store copies of frequently used memory to reduce access latency. Modern Intel CPUs often have three levels of cache of different sizes. The L1 cache is the smallest and fastest, while the L3 cache, also known as last-level cache, or LLC, is bigger and slower. Both L1 and L2 are private to each core, whereas the LLC is shared by all physical cores. Modern caches are set associative, meaning a cache line is stored in a fixed set determined by its address, virtual or physical. It can be stored in any of the ways of a cache set, based on the replacement policy of this level. Modern Intel LLCs often have several ways, ranging from 12 to 20. When the CPU needs to access a specific address, it first queries the cache. If the address is stored in the cache, the data will be directly served from the cache, resulting in a short access time (a cache hit). If not, the CPU will access the data from the DRAM, resulting in a slower access time (a cache miss).

Such timing differences can be exploited by an attacker to mount side-channel attacks or covert channels. Yarom and Falkner presented Flush+Reload [43], a cache attack that exploits shared memory to infer whether the victim accessed a certain cache line. The attacker evicts said line by using the native instruction `clflush` and then, after a certain period, times the access to the address. If the access time is short, this means the value has been loaded into the cache between the flush and the reload, meaning the victim has accessed said cache line. Flush+Reload has a spatial resolution of a single cache line, *i.e.*, 64 bytes. It, however, requires access to native instructions, as well as shared memory. Liu et al. [23] implemented Prime+Probe, a cache attack that does not require shared memory or access to native instructions. Instead of sharing a cache line with the victim and flushing it, the attacker uses an eviction set, *i.e.*, a group of addresses indexed on the same cache set, to evict all previous lines in this cache set. This attack has a slightly reduced spatial resolution compared to Flush+Reload, consisting of one cache set. The size of a cache set varies between processors, but it usually ranges from 12 to 20 cache lines, *i.e.*, from 768 to 1280 bytes.

2.2 JavaScript and WebAssembly

JavaScript. JavaScript is a high-level object-oriented interpreted scripting language that follows the ECMAscript standard [10]. It is a major part of the World Wide Web as it is in charge of most client-side computing in almost all websites. A user visiting a website downloads and executes various scripts. As a consequence, it is meant to run on the client's hardware, and needs to be system-independent. For security reasons, JavaScript is executed in a sandbox, restricting access to local files, native instructions, and memory addresses.

JavaScript code is interpreted and executed in the browser by the JavaScript engine [13, 26]. The just-in-time (JIT) compilation approach taken by these engines means that the same code can be executed differently based on the engine, browser, or even the OS and microarchitecture.

WebAssembly. WebAssembly [42] (or wasm) is an open-sourced binary instruction format designed to be deployed on the web, for clients or servers. Its main feature is to allow compilation from various languages and executing them at native speed. On the client side, WebAssembly is designed to run inside of the JavaScript sandbox, hence ensuring the same security restrictions. WebAssembly is currently supported by major web engines, including V8 (found on Google Chrome and Microsoft Edge), WebKit (found in Apple Safari) and SpiderMonkey (found in Mozilla Firefox).

WebAssembly functions as a low-level, assembly-like, program. It is built around a stack-based virtual machine. It supports two main formats: binary, which is directly interpretable by the engine, and the text format, human-readable format, allowing to read and modify compiled WebAssembly code. WebAssembly's specification is still under development, and it currently has around 100 specified instructions, with various operands.

2.3 Timing attacks and microarchitectural attacks in the browser

JavaScript timers. With the development of microarchitectural attacks, in particular Spectre, browser vendors introduced several countermeasures in order to provide more isolation to the JavaScript sandbox. In particular, Reis et al. [29] introduced a new browser architecture based on site isolation, where each site runs in a different process. This prevents an attacker to access the memory space of other sites in the same browser. COOP and COEP [7, 8] extended site isolation. They are a set of header between the top level domain and all loaded resources. When enabled, the site is considered cross-origin isolated, ensuring a unique process for the context group and safe external resources.

To prevent the threat of timing attacks, most browser vendors have removed access to high-resolution timers. The highest resolution timer available in recent browsers, `performance.now()`, has a resolution of 5 µs with jitter in Chrome 94 and 20 µs in Firefox. This is highly insufficient to mount microarchitectural attacks, as we need to measure timing differences in the order of 10 ns. However, auxiliary timers, able to recover a high resolution in the sandbox, were described by Schwarz et al. [33].

The most powerful of these auxiliary timers is based on `SharedArrayBuffer`, an array shared between the main thread and a subthread (Web Worker in JavaScript). The main thread initializes a Web Worker and shares the array with it. Then, the Web Worker constantly increments a variable in the array. As this operation has a low and constant execution time, it can be used as a unit of time by the main thread. The main thread can then read the shared value to get a timestamp. This timer grants a resolution ranging from 10-100 ns on recent browsers [30]. In the past, `SharedArrayBuffer` has been disabled by default to prevent timing attack threats. However, they are available by default when the web page is cross-origin isolated in Chrome 94 and Firefox 90 [7, 8]. Unless stated otherwise, all timing measurements in the paper use `SharedArrayBuffer`-based clocks, thus the time unit is an increment.

JavaScript timing attacks. The fact that microarchitectural attacks can be mounted from JavaScript brings major changes to their threat model. On the one hand, it allows running code on the victim's hardware on a very large scale. For instance, an attacker can buy an advertisement on a popular website and will be able to run its scripts on all visitors of said website [11]. On the other hand, the sand-boxed execution brings many major restrictions to the implementations of such attacks. The lack of native instructions or memory addresses, for instance, removes the possibility to implement some classes of attacks, such as attacks based on Flush+Reload [43].

However, in 2015, Oren et al. [27] implemented the first entirely web-based cache attack. Many different types of web-based microarchitectural attacks were since demonstrated, exploiting other components or features, including the DRAM [16], ASLR [15], and even speculative execution [18].

Covert channels in browsers. Covert channel in the browsers break the fundamental principle of the JavaScript sandbox isolation. In particular, previous work has studied covert channels based on hardware timing attacks. Oren et al. [27] presented a covert channel based on Prime+Probe with a bandwidth estimated at 320 kbit/s. However, this number was estimated before the introduction of countermeasures against microarchitectural attacks in this browser. To the best of our knowledge, there has been no work on Prime+Probe subsequently to these countermeasures. The closest covert channel is the one used to extract data in the transient execution attack RIDL [39], with a bandwidth of 8 bit/s using Evict+Reload. Rushanan et al. [31] used CPU-throttling to build a covert channel with a bitrate of 0.2 bit/s. Schwarz et al. [33] implemented a DRAM-based covert channel in the browser. They reached a bitrate of 11 bit/s when using `SharedArrayBuffer`-based clocks. Lipp et al. [21] presented a cross-browser channel using network interruptions, reaching a bandwidth of 25 bit/s. Van Goethem and Joosen [38] exploited disk or memory contention to send bits every 200 ms, thus granting a maximal raw bandwidth of 5 bit/s.

Software covert channels have also been implemented in the browser. For instance, Vila and Köpf exploited Chrome's event loop, shared between tabs, to create a covert channel with a raw capacity of 200 bit/s for a same-browser channel and 5 bit/s in a cross browser setting. However, this vulnerability has been mitigated with the introduction of site isolation [29], as different tabs or processes do not share an event loop anymore.

3 Web-Assembly-Based Port Contention

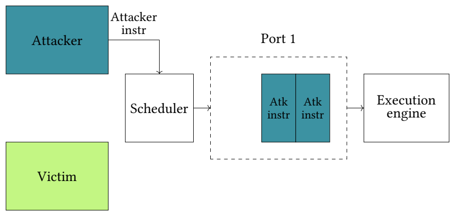
We introduce, to the best of our knowledge, the first implementation of port contention inside a browser. We can create and measure port contention from the JavaScript sandbox, on both Mozilla Firefox and Google Chrome. We found instructions that create contention on both P1 and P5, allowing diverse potential victims.

Experimental setup and threat model. Unless stated otherwise, we run all experiments on an Intel i5-8365U CPU with a maximal frequency of 1.60 GHz running Ubuntu 20.10, with Mozilla Firefox 90 and Google Chrome 95 desktop version, both using WebAssembly 1.1[3]. As Safari and Edge support WebAssembly, the attack can theoretically be carried on these browsers, but they remain outside of the scope of this paper. The threat model is similar to a user visiting a malicious website with his browser. The browser scripts run in a cross-origin isolated browser [7, 8], granting more context isolation and allowing access to `SharedArrayBuffer` and higher resolution timers.

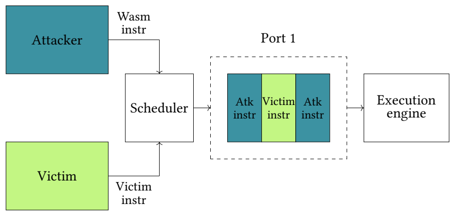
Description. Figure 2 illustrates the principle of our web-based port contention attack. The attacker is situated inside of the browser sandbox, in the blue process. During the attack, he repeats specific instructions that cause contention on a specific port. Section 4 explains how we find these instructions on different systems. For instance, on our processor, the WebAssembly `ctz` (Count Trailing Zeros) instruction creates contention on P1. Similarly, instructions that truncate floats to integers, e.g., `trunc_f32_u`, create contention on P5. The attacker then times the execution of these instructions. If no other processes use the same port at the same time, these instructions will all be executed in a row, resulting in a fast execution time, as exemplified in Figure 2(a). However, if another process emits μops on the same port, these μops will be queued with the attacker-generated μops, resulting in a slower execution time for the attacker, as illustrated in Figure 2(b). By measuring these differences in timings, the attacker process can monitor the port usage on a specific port, and thus monitor other processes.

Challenges. We face three challenges when implementing port contention in the browser. First, as browser-based scripts run in a controlled sandbox, we have no access to native instructions,

[3]We used the latest version available in November 2021. This version did not support vectorial types and SIMD instructions.



(a) Victim has not used port 1: all attacker instructions are executed in a row.



(b) Victim emitted one μop on port 1: attacker instruction will be delayed.

Figure 2: Illustration of web port contention.

and must instead use higher-level language constructs (**C1**). Furthermore, as browser-based scripts are meant to be portable, the instructions are translated to different assembly language instructions by the browser's engine on different systems. This means that the same script generates different native instructions depending on the CPU architecture, each with a different port usage, varying from vendors and generations. The code is also highly optimized by the engines, and execution can vary even on the same system, based on the variables or structure of the code. To gain more control over the port usage of our attacks, we mounted our attack with WebAssembly. This grants us access to smaller, more atomic instructions. However, these instructions are still executed through the browser's JIT engine, and their translation to machine language can vary from a system to another. For instance, the WebAssembly instruction `ctz` is translated into the native Intel instruction `TZCNT` on our system, as we describe in more detail in Section 4. The `TZCNT` instruction, in turn, is implemented using a single μop which is executed on P1 [1]. Thus, repeatedly executing the WebAssembly instruction `ctz` can cause contention on P1. The Intel instruction `TZCNT` is only available, however, on CPUs starting from the Broadwell generation. Thus, the WebAssembly `ctz` instruction may generate contention on another port in older CPU generations. Directly compiling native code using x86 assembly instructions to create contention is not possible. Since WebAssembly is designed as a portable language, the compilers cannot compile instructions that are directly architecture-dependent, as they could not run on non-Intel CPUs.

Secondly, the high level of abstraction provided by the browser means that an attacker can neither know nor control on which core the attack is executed (**C2**). Furthermore, the operating system's scheduler dynamically moves processes between cores to optimize computing and save energy. We address this challenge by performing our attack on multiple cores simultaneously by using Web Workers, JavaScript multi-threading implementation, which

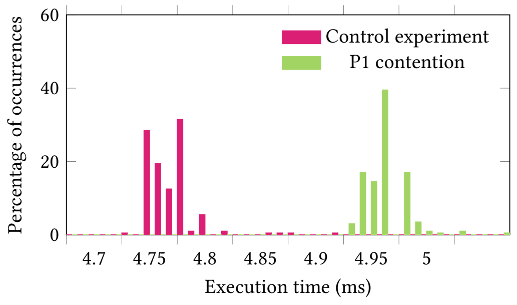


Figure 3: Port 1 contention experiment on `i64.ctz` for 1 000 000 instructions.

creates a sub-thread running in a different process. This lets the attacker create as many attacker processes as physical cores, and as they all have a high workload, they are spread on different physical cores. Then, one of the attacker processes runs on the same core as the victim process, able to monitor it.

Finally, our attack requires high-resolution timers to monitor processes at the μop level (**C3**). Native implementations of port contention attacks all use the cycle-accurate `rdtsc` instruction. As explained in Section 2, browser vendors have restricted access to such timers inside of the sandbox to prevent timing attacks. In our attack, unless stated otherwise, we use `SharedArrayBuffer`-based timers, which offer a resolution and measurement time in the order of 20 ns [30, 33].

Proof-of-concept. Figure 3 shows a proof-of-concept illustrating the contention on P1 caused by the WebAssembly `i64.ctz` instruction.

In this experiment, we time the execution of 1 000 000 WebAssembly `i64.ctz` instructions using the low-resolution JavaScript function `performance.now`. We run the experiment on Firefox 90, where this timer offers a resolution of 20 µs without jitter. In parallel with the Firefox code, we also run a sender program written in native code and pinned to the same processor. In the P1 contention experiment, the native sender runs the Intel instruction `crc32` in a loop. This assembly language instruction is known to cause contention on P1. In the control experiment, the native sender runs a simple loop designed not to cause port contention. We run this program, instead of simply not executing the sender at all, to ensure that the difference stems from port contention, and not from other sources. As the figure shows, the timings measured during the P1 contention experiment are on average 5% higher than the control experiment, allowing the browser to efficiently distinguish between the two distributions. We observe similar results on Chrome 95.

In the following sections, we describe how to convert this proof-of-concept into practical attacks. In particular we obtain a higher spatial resolution and evaluate 100 WebAssembly instructions (**C1**), we ensure the attacker does not have to pin processes (**C2**), and we use a higher resolution timer (**C3**).

4 PC-detector

The translation of WebAssembly instructions into μops is variable on different systems: it can depend on the microarchitecture, instruction extension sets or JavaScript engine. In this context, it can be hard to find WebAssembly instructions that reliably cause port contention. In this section, we propose PC-detector, a Selenium-based framework to dynamically detect and characterize the port usage of WebAssembly instructions. Using the methodology described in Section 3, PC-detector automatically tests multiple WebAssembly instructions and checks if they cause contention on P1 or P5.

4.1 Description

Framework. Our framework is composed of two components. The first component is a native C script that either runs an empty loop, creates contention on P1, or creates contention on P5. The second component is a Selenium-controlled browser which runs automatically generated WebAssembly code. For each WebAssembly instruction *instr*, we create a binary file with 1 000 000 calls. This file is then executed in the browser, and its runtime is measured using `performance.now()`. We run three experiments:

(1) Repeatedly executing and timing the WebAssembly file, used as a control.
(2) Creating contention on P1 with native code and timing the WebAssembly file.
(3) Creating contention on P5 with native code and timing the WebAssembly file.

By evaluating the timing distributions of these three experiments, we can evaluate the port usage of *instr*. If the three distributions are mixed, *instr* is not affected by the port contention (thus it cannot cause it). If the P1 timings (respectively P5) are, on average, higher to both the control and P5 (respectively P1), this means *instr* can detect, and cause, contention on P1 (respectively P5).

We evaluate all standardized single and double operand operations [41], including arithmetic operations and memory operations. Due to the stack machine structure of WebAssembly, each experiment includes a `load` operation to add values to the stack between each operation. We discovered that due to JIT optimizations, it is not possible to load many values on the stack before running double operand operations in a row, as the compiler reorders the instructions to alternate between loads and the tested operation. Therefore, we could not run all double operand operations one after the other. We evaluate single instructions when instructions have an output the same type as their input, and pairs of complementary instructions in the other case (e.g., convert a 32 bit integer into a 64 bit float). We do not evaluate control flow operations, e.g., loops or jumps.

Metrics. We propose two main metrics to automatically evaluate if a WebAssembly instruction can create contention on P1 or P5. The first one is based on the error rate between timings from the P1 and P5 experiments. For this metric, we compare P1 to P5 instead of P1 to control, as the control experiment does not run calculation on the native side. This means that the timing differences could originate from other sources than port contention, e.g., variation in frequency or contention on another shared hardware component.

Table 2: WebAssembly instructions causing port contention. For clarity, we group together the 32- and 64- bits versions of instructions under one line marked i32/i64.

Instruction	P1 contention	P5 contention	Cohen's d
`i32/i64.ctz`	●	○	1.2
`i32/i64.clz`	●	○	1
`i32/i64.popcnt`	●	○	1
`i32/i64.div_s`	●	○	10
`i32/i64.div_u`	●	○	10
`i32/i64.rem_u`	●	○	34
`i32/i64.rem_s`	●	○	5
`f32.convert_i32_s and i32.trunc_f64_s`	○	●	1
`f32.convert_i32_s and i32.trunc_f32_s`	○	●	2
`f32.convert_i64_s and i64.trunc_f32_s`	○	●	8
`f32.convert_i32_u and i32.trunc_f32_u`	○	●	2
`f32.demote_f64 and f64.promote_f32`	○	●	3
`i32.wrap_i64 and i64.extend_i32_u`	●	○	16
`i32.wrap_i64 and i64.extend_i32_s`	●	○	11

P1 and P5 have two timing distributions, and one distribution (X_{low}) has lower timings than the other distribution (X_{high}) when there is contention. Given a temporal threshold τ, we define the error rate as the proportion of values of $X_{low} > \tau$ and values of $X_{high} < \tau$ over all experiments. We define the error rate for a given threshold as

$$er_{\tau} = \frac{|X_{low} > \tau| + |X_{high} < \tau|}{|X_{low}| + |X_{high}|}.$$

Then, by computing er_{τ} for $[min(X_{low}) < \tau < max(X_{high})]$, we can retrieve the lowest error rate possible, giving us the probability for a program to blindly distinguish between port contention and standard usage from experiment timings. By inverting X_{low} and X_{high} and computing the best error rate, we can see if an instruction creates contention on P1, P5 or none. In PC-detector, we infer that if $er < 5\%$, an instruction creates contention.

The error rate calculation lets us identify whether an instruction creates contention. It does not, however, illustrate the efficiency of this contention, *i.e.*, how separated both distributions are or how spread they are. This parameter is important in our attacks, as the more distance between the distributions, the easier it is to distinguish between contention and standard usage. In order to measure the distance between P1 and P5, we compute the effect size, also known as Cohen's d. In our case, Cohen's d between P1 and P5 is defined as

$$d = \frac{|mean(P1) - mean(P5)|}{\sqrt{(stdev(P1) + stdev(P2))/2}},$$

with stdev() the standard deviation of the distribution. A high Cohen's d means that distributions are highly separated and concentrated, and that we can more easily distinguish contention from standard usage.

4.2 Results

We have tested 100 different instructions, including numerical, memory, bit-wise, and type conversion operations.

Table 2 lists which instructions cause contention on the i5-8365U. The results are identical between Chrome and Firefox, although the distance varies because of the different browser architectures. In total, we found 21 instructions causing contention. As most instructions have 32- and 64-bit variants, some instructions are doubled. Generally, we observe that 64-bit variants have a greater Cohen's d than their 32-bit counterparts. Similarly, the unsigned variants of integer operations often grant better results than the signed variants.

P1 contention seems to be caused by arithmetic instructions, whereas conversion/truncation operations create contention on P5. This result is coherent with the specialization of ports and execution units. `i64.rem_u` shows the highest effect size of all detected instructions.

To demonstrate the portability of port contention and PC-detector, we have ran the same benchmark on different Intel CPUs. In total, we have tested 4 recent CPUs: i5-8365U, i7-8650, i7-10510 and i7-10610. The instructions creating contention remain constant, but Cohen's d can vary based on the CPU frequency. This is logical, as all tested cores have the same instruction set extensions, meaning that the WebAssembly instructions are translated to the same native instructions.

5 Side-channel Attack on Artificial Applications

In this section, we present an artificial gadget, illustrating the side-channel threat of web-based port contention. We built a synthetic and generic example showing how a program, which execution depends on secret information, is vulnerable to WebAssembly port contention. Indeed, if a program has branches depending on secret bits, an attacker can use a side-channel attack to infer the secret. The victim process is an unprivileged native process. The attacker is a JavaScript and WebAssembly script, running inside of the browser's sandbox. The attacker has no access to addresses, native instructions, and no control or knowledge of physical or logical cores.

In our implementation, an attacker, running code inside the browser's sandbox, monitors the victim's execution with a spatial resolution of 1024 native instructions, *i.e.*, 3072 bytes. This spatial resolution is of the same order of magnitude as other microarchitectural attacks in the browser, e.g., Prime+Probe, which has a resolution of a cache set (typically 12 to 20 cache lines), *i.e.*, 1280 bytes on our system.

5.1 Description

The victim is a native unprivileged program, running different code sections based on the bits of secret information. As port usage differs between branches, an attacker monitoring port contention could infer parts of the secret. Listing 1 illustrates our gadget, implemented in native assembly code. Depending on a secret bit, the code will execute either instructions creating contention on P1 or P5. To detect from within the browser which path is taken by the victim, we time the execution of nb_{instr} WebAssembly `rem_u` instructions, which creates contention on P1 (Section 4). If the execution time is high, then we know that the native code also creates contention on P1, whereas if it is standard, we know that the native code does not create contention. By repeating this process, we detect the branch that was executed by the native script, and hence the value of the secret bit.

After resolving **C1** with PC-detector and **C3** with `SharedArrayBuffers`, we still face the inability to pin the attack code to the same physical core as the victim (**C2**). Most schedulers try to balance the workload between physical cores. By creating a number of listening Web Workers equal to the number of physical cores, we maximize our chances that one of them listens on the victim's physical core, thus circumventing **C2**. Information about the system's core count is available through the `navigator.hardwareConcurrency` JavaScript API [25], available by default on both Chrome and Firefox.

Listing 1: Side channel artificial example. Depending on the key bit passed in parameter, the code will have different port usage.

```
TEST %rdi, %rdi
JE .p1
JMPQ .p5

.p1
POPCNT %r8,%r8
POPCNT %r8,%r8
...
POPCNT %r8,%r8
POPCNT %r8,%r8

.p5
VPBROADCASTD %xmm0, %ymm0
VPBROADCASTD %xmm0, %ymm0
....
VPBROADCASTD %xmm0, %ymm0
VPBROADCASTD %xmm0, %ymm0
```

5.2 Results

An important metric for our evaluation is the spatial resolution, *i.e.*, the smallest number of instructions we can detect in a branch. To detect contention, we measure the execution time of nb_{instr} WebAssembly `rem_u` instructions. This parameter is important: a high number of instructions lowers our spatial resolution, but a lower number yields noisier time measurements. Furthermore, for values of nb_{instr} ranging from 1 to 10, the execution time of the instruction is slower than the read access to the shared array and other overhead introduced by JavaScript. This means that contention is measured at only specific times in the measurement. To reduce the measurement time of `SharedArrayBuffer`, we access the array directly, without using concurrent access libraries. This grants a better resolution to the timer but creates more noise and outliers. On our system, we were able to create a web listener running in the same physical core as the victim in 95% of our experiments. We infer that the remaining errors stem from the scheduler moving our process to different cores because of other threads creating noise.

On our system, we found $nb_{instr} = 10$ to be the best compromise between noise and resolution. To reduce the noise, we process the data with a median sliding window with a width of 10 measurements. Figure 4 illustrates the resulting values when the victim runs the code with the secret 1101001, for a single trace of the victim. The high values represents the execution of the victim branch creating contention on P1 *i.e.*, a bit set to 1. The width of a peak or a pit is proportional to the number of bits inside the sequence.

Our implementation is able to detect the executed branch with a resolution of 1024 native instructions on both Google Chrome and Mozilla Firefox. To obtain this result, we first implement Listing 1

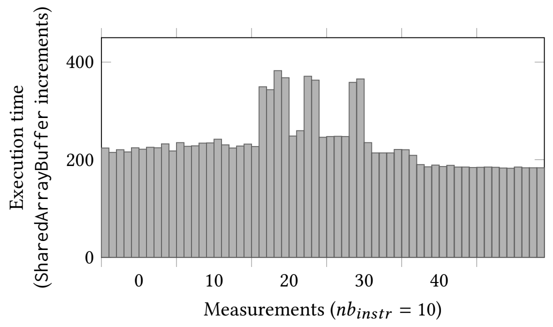


Figure 4: Single-trace execution with secret information 1101001, on Chrome 95.

with a very high number of POPCNT and VPBROADCASTD instructions, that we progressively lower. The resolution is the lowest number of instructions where we can clearly retrieve the secret bits without error on a single victim trace.

This experimental limit of 1024 instructions is mainly due to the lack of access to high resolution timers. Note that we observe two peaks per secret bit with a single trace. We have found that a higher resolution of 512 instructions could introduce errors with a single-trace attack. One solution to increase the resolution would be to revert to multiple-trace attacks. Moreover, by using a custom browser implementing `performance.rdtsc()`, based on the native cycle accurate timer, we observed that our implementation has a resolution of 256 instructions, *i.e.*, a better spatial resolution than Prime+Probe. This means that our experimental limit could be lowered with better auxiliary timers or noise filters, which could offer a more fine-grained attack vector than existing microarchitectural side channels in the browser.

6 Covert channel

In this section, we present a port contention-based covert channel with a throughput of 200 bit/s for a 1% error rate. This covert channel is composed of a sender running unprivileged native code, and a receiver running completely inside the browser (similarly as Schwarz et al. [33]). We also show that our covert channel runs with a sender located inside a VM, and can even be used in a cross-browser fashion (similarly as Lipp et al. [21]).

The sender runs unprivileged C code on the victim's hardware. The sender can therefore freely use most native instructions, and has access to cycle-accurate timers. It can also pin itself to a certain physical or logical core. The receiver, on the other hand, runs fully inside a cross-origin isolated web page. As it runs inside the browser's sandbox, the receiver has no access to native instructions. Port contention must be created and measured by using WebAssembly (**C1**). Moreover, the web script must share a physical core with the sender, but cannot control or know on which physical core it is running (**C2**). Finally, the receiver does not have access to high resolution timers (**C3**). Instead, we use SharedArrayBuffers to get the best resolution available.

6.1 Description

We implemented a half-duplex asynchronous channel based on port contention, between a native sender and a web-based receiver. In addition to data, the sender and receiver exchange control messages to handle acknowledgments and synchronization. Both parties must therefore be able to send and receive bits. Our side channel can be decomposed into two layers. The lower layer, sending and receiving bits, is equivalent to the physical layer of the TCP/IP model. This layer uses CPU ports as its transmitting channel, and must be able to distinguish between 0 and 1 bits. The upper layer is equivalent to the data-link layer. This layer handles the synchronization between the parties, as well as error management.

Physical layer. The two parties send 1-bits by creating contention on P1 for a fixed duration (t_{bit}), and send 0-bits by idling for t_{bit}. t_{bit} is an important factor, as a high duration lowers the channel's bandwidth but allows the receiver to tolerate more noise when attempting to distinguish bits. In our covert channel implementation, we have fixed t_{bit} = 1 ms. To create contention, the sender and receiver repeatedly call an instruction, respectively the native Intel instruction `crc32` and the WebAssembly instruction `rem_u`. To receive a bit, the sender or receiver repeatedly call these instructions while timing them. A high execution time means the emitting party is sending a 1, while a standard time means a 0. As both instructions are handled by the CPU port P1, both the sender and receiver cannot emit at the same time, making our channel a half-duplex channel. Besides their high resolution, another advantage of a SharedArrayBuffer-based timer is that it is based on a Web Worker, and therefore runs on a different core. This lowers potential noise on the covert channel.

We also need to ensure both the sender and receiver are running on the same core (**C2**). As the browser cannot control which core it is running on, the sender creates as many sub-senders as physical cores. The sender runs native unprivileged native code, so it knows the number of physical and logical cores, and can pin each of its sub-threads to a specific core. This ensures that at least one sender thread is running on the same physical core as the receiver.

Although SharedArrayBuffers offer a high resolution, they can introduce errors at the physical layer level. In particular, concurrent accesses between the thread incrementing a value and the main thread reading the timestamp can cause insertion or deletion errors. We have determined two error-prone scenarios at the physical level. In the first scenario, the main thread reads the shared value too frequently. This prevents the clock thread from incrementing the value, and as a result the measured time is much lower than the real time value. The other scenario stems from particularly high measurement outliers when contention is created. We assume it also comes from concurrent accesses. As this access is longer than usual, it means that we can get less measurements during t_{bit}, thus creating bit deletion errors on higher layers.

Protocol and frame format. To ensure synchronization and correct potential errors, we implemented a simple protocol above our physical layer, similarly as Maurice et al. [24]. Figure 5 illustrates a typical exchange, as well as packet loss management. It is based on a simple request-to-send scheme: the receiver sends a request frame (described in Figure 6(a)), containing a 4-bit sequence number. Upon reception, the sender sends a data frame (described in Figure 6(b)),

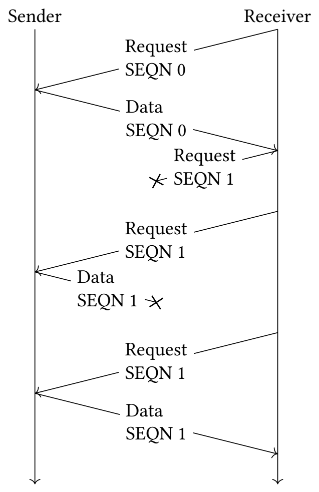
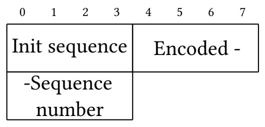
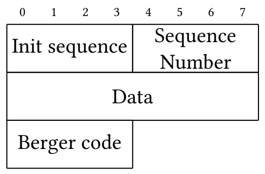


Figure 5: Illustration of the protocol's synchronization in case of lost or incorrect packet.



(a) Request frame.



(b) Data frame.

Figure 6: Format of the request and data frames.

containing the sequence number as well as the associated data (1 byte). If the data frame is received correctly, the receiver requests the next sequence number. Both frames start with a 4-bit preamble consisting of an initial sequence which is always set to 1010. This initial sequences serves as calibration for the receiver.

To handle possible insertion or deletion errors, we added an error detection code. More specifically, the sequence number is encoded with (8,4) Hamming code [17] in request frames, and the last 4 bits of the data frame contain a Berger code [4], counting the number of zeros in the data and sequence number fields. As the type of errors we face are mainly bit insertion or deletion, we do not use the error correcting properties of Hamming code, and instead use it as an error-detection code.

Our protocol encodes 8 bits of payload into a 31-bit message, including the preamble, sequence numbers and error detecting code. This means that, with t_{bit} =1 ms, we can reach a maximal raw throughput of 1 kbit/s, *i.e.*, a theoretical maximum of data bit rate of 260 bit/s.

Our protocol also manages packet loss and desynchronization. This is handled by the sequence number and the request-to-send scheme. As illustrated in Figure 5, after sending a request, the receiver waits for a fixed timeout value. If it has not received an answer at the end of this time period, it simply re-sends the request. This lets the covert channel recover from packet loss from the sender to the receiver, and from the receiver to the sender.

Receiving frames. The sender and receiver do not share a common clock. Hence, the party receiving bits does not know in advance the demarcations between successive bits, nor when the frame starts. It is processing execution time of instructions as a real-time stream of information, not in post-processing. In order to automatically detect the start of the frame, as well as the actual bits, both sender and receiver run DenStream [6], a density-based data-stream clustering algorithm. It dynamically creates clusters of data, based on the execution time and their time of arrival. The listening party then detects the start of the frame when it detects 4 consequent small clusters with variation in execution time, corresponding to the initial sequence of 1010. The initial sequence is used to calibrate two major values: the temporal threshold between 0-bits and 1-bits, as well as the average number of instructions in a single bit. The average number of points lets the algorithm detect the number of bits in a sequence. As DenStream computation can be slow when we reach a high bit rate, we only use it to detect the preamble. For the rest of the frame, we use a simple stream-based threshold detection: timings above the calibration threshold are identified as 1-bits, and others as 0-bits.

To infer the actual number of bits in such a sequence, we use the average number of instructions calibrated from the initialization sequence. Then, by dividing the number of instructions in our same bit sequence, we can infer how many bits it contains. This step is prone to insertion or deletion errors.

When the stream algorithm has detected a number of bits corresponding to the frame size, it stops listening. If the frame is invalid because of insertion and deletion errors, we try to reinterpret it with slightly modified calibration values. Indeed, variation in frequency can cause slight changes on the number of measurements in a bit, e.g., a frequency raise means we measure more instructions in a bit, thus potential insertion errors.

6.2 Evaluation

We evaluated our covert channel in two different scenarios. The first scenario is the baseline implementation, where both the native sender and web-based receiver run in a standard OS. In the second scenario, the native sender now runs in a virtual machine running on the victim's physical hardware, while the browser runs in the standard OS. This scenario is common, as malware analysis is often conducted in sandboxed environments such as VMs. We also evaluate the impact of noise on our covert channel.

Native sender. This threat model represents the most common scenario, where both the browser and the native sender run as unprivileged processes in the OS. We evaluated our covert channel by transmitting 10 kB of data from the native sender to the web-based receiver. To compute the error rate, we compare the original and received bit sequences bit-by-bit.

Table 3: Evaluation of the port-contention covert channel in different conditions.

Experimental setup	Bit rate	Packet Loss rate	Error rate
Noiseless	200 bit/s	5.5%	1%
`stress -c 2`	170 bit/s	8%	3%
`stress -m 2`	120 bit/s	15%	3%
`stress -c/-m 3`	25 bit/s	80%	5%
`stress -c/-m 8`	<1 bit/s	99%	5%

Table 3 illustrates the bit rate and error rate of our channel in different noise conditions. The transmission takes, on average, slightly less than 7 min. During the transmission, on average 600 frames arrive incorrectly or are lost from the sender to the receiver, over a total of 10 600 frames. This represents a total frame loss rate of 5.5%. Most of the incorrect frames were the result of insertion or deletion errors. The lost frame rate from the receiver to the sender is negligible. We achieve a bit rate of 200 bit/s. This is 80% of the maximal bandwidth possible when using t_{bit} =1 ms. The difference between the bit rate upper bound and our implementation stems from the loss of frames, which requires the sender to wait for some time before requesting the data again, as well as from the short computation time required to handle the protocol.

In this setup, our covert channel presents a better bit rate than previous web-based covert channels [21, 31, 33, 38, 40]. The only covert channel with a better resolution is Prime+Probe by Oren et al. [27]. However, recent countermeasures had a substantial negative impact on the bit rate. To the best of our knowledge, no other Prime+Probe covert channel has been implemented since that allows us to compare between the two approaches. The closest cache covert channel is the one presented by van Schaik et al. in RIDL [39], with a bit rate of 8 bit/s.

We now evaluate our covert channel in the presence of noise. Noise can impact both the bit transmission through port contention, and the `SharedArrayBuffer` clock. Indeed, we observe that when stressing the physical core used by the `SharedArrayBuffer` clock, the number of ticks we measure in each time period decreases, in turn decreasing our resolution. However, our covert channel shows strong resilience to sources of noises with a low thread count. That is because port contention depends on the physical core. As our sender and receiver already use a major part of the core computing capacities, the OS scheduler tends to move other noisy processes to different physical cores, thus lowering their impact on our covert channel. For instance, when running `stress` with square root (`-s`) or malloc (`-m`) on two threads, the bit rate remains on the same order of magnitude. The loss of performance stems from a higher rate of lost frames due to clock outliers. Our channel also shows better resilience to sources of noise with a low thread count than cache covert channels, as the LLC is shared between cores [24]. However, if a noisy thread runs on a physical core shared either by the clock or the receiver, the performance significantly drops, as illustrated in the `stress -c 3` case. In that case, the lost frame rate increases drastically because of lower resolution from our timer. Introducing specific error-correcting codes to correct

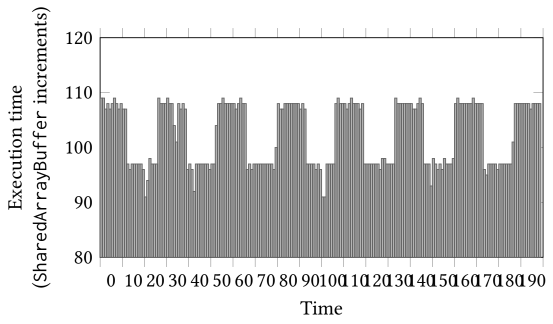


Figure 7: Transmitted square signal from Firefox 90 to Chrome 94 with t_{bit} =1 ms

insertion or deletion errors could greatly improve the performance of the channel in noisy conditions.

Virtualized sender. We also evaluate our channel in a virtualized setup. In this scenario, the native sender runs inside of a virtual machine running Ubuntu. The browser runs in the standard OS. The main change in the threat model is that the native sender has no control or knowledge of cores, physical or logical. However, by creating multiple sender threads and not pinning them, we managed to force at least one sender thread to run on a physical core shared with a receiver. In this setup, our covert channel has a bit rate of 80 bit/s. This bit rate is still higher than that of many browser covert channels [21, 31, 33, 38, 40], and even equivalent to some native covert channels in the same setup [34].

6.3 Cross-Browser Covert Channel Bandwidth Estimation

Our covert channel can be extended to a cross-browser setup. As we can create and detect contention on the browser, we can replace the native sender with a JavaScript sender. This has two major impacts on the effectiveness of the attack. First off, the web-based sender loses access to powerful native timers, potentially creating new errors on the request frames. Most importantly, the browser has no knowledge of physical or logical cores. It cannot know nor control on which core it is running. To circumvent this difficulty, the web-based sender creates a number of Web Workers equal to the number of physical cores of the machine. By doing so, the scheduler will spread these senders on different physical cores. When launching the receiver, however, the senders are not the only processes using a high workload, and we have noticed that launching the receiver and the `SharedArrayBuffer` clock after the sender results in a physical core running both the clock and the receiver, and the senders sharing the remaining core. We overcome this limitation by initializing the clock and receiver before the sender. As a result, the scheduler assigns a physical core shared by a receiver and a sender, effectively allowing the implementation of our covert channel.

Using this technique, we were able to transmit bits of information across browsers through port contention with t_{bit} = 1 ms. *i.e.*, conditions equivalent to the native-to-web covert channel. We were able to demonstrate data transfer at the physical layer from Chrome to Firefox, from Firefox to Chrome, and between two instances of the same browser. Figure 7 shows the transmitted square signal from Firefox to Chrome. We did not re-implement the data-link layer to this threat model, as it represents significant engineering work, and leave it to future work. However, this proof of concept solves all scientific and technical challenges, including the most difficult, *i.e.*, core management (**C2**), by its ability to transmit bits. As the physical layer offers similar bit and error rates to the native sender, even for a long duration of transmission, it is safe to estimate that this cross-browser covert channel can reach a final bandwidth on-par with the native-to-web covert channel, *i.e.*, in the order of 200 bit/s.

7 Discussion

In this section, we discuss the limitations of our approach, potential countermeasures, as well as future work.

7.1 Limitations

The WebAssembly implementation of port contention offers a lower spatial resolution than the native PortSmash attack proposed by Aldaya et al. [3]. Most of this performance loss originates from the challenges introduced by the JavaScript sandbox. In particular, **C3** is the most challenging aspect. Although auxiliary timers offer a very high resolution, they are still inaccurate compared to native cycle-accurate timers. This difference particularly impacts the attack's spatial resolution, as timer imprecision prevents us from measuring small time differences.

Another limitation, inherent to port contention and SMT attacks, is that this attack cannot run in a cross-core setting. We can effectively circumvent **C2** by creating more threads to share a core with the victim, but the attack still depends on the OS scheduler. If the attacker cannot run code on the victim's physical core, the attack does not succeed.

7.2 Countermeasures

Many countermeasures have been proposed to mitigate microarchitectural attacks. However, most of these propositions are heavily focused on cache-based side channels. In this section, we provide an overview on existing academic work or other suggestions that focus on mitigating contention-based side channels.

Hardware. One pre-requisite of port contention attacks is sharing CPU ports between a victim and an attacker. SMT is therefore at the core of the attack. To prevent SMT-based side channels, including port contention, some have suggested disabling SMT altogether. For instance, SMT is disabled by default in OpenBSD [20] or Google's ChromeOS [12]. However, this proposition represents a major performance degradation of up to 15% [5], as SMT allows for a highly efficient use of hardware resources.

Townley and Ponomarev [37] proposed SMT-COP, a hybrid approach based on partitioning the use of resources between threads. This partitioning could be either temporal, each thread accessing the resource after the other, or spatial, each thread having their execution units. Their approach must be supported by the hardware and introduces a performance overhead of 8% compared to standard SMT, while preventing most contention-based side channels on the execution units or ports.

More recently, Taram et al. proposed SecSMT [36], focusing on more secured shared resources against contention-based side channels. Their approach introduces, at the hardware level, different ways to share resources. In a static partitioning, the resources are statically shared between logical cores. In an adaptive partitioning, the partition of resources evolves according to the workload of logical cores to enhance parallelization. However, the resources are never used by both cores at the same time. More interestingly, asymmetric partitioning relies on different levels of trust. This model gains even more performance by letting a low level security thread leak information to a high-security thread, but not letting high-security information leak to other threads. This is particularly interesting in a browser-based scenario. It is unsafe to leak information to the sandbox, whereas leaking sandboxed information to other threads presents less threats. Their asymmetric partitioning presents almost no overhead compared to traditional SMT.

OS and applications. Software mitigations, outside of the browser, have also been suggested. First, similarly to cache-attacks mitigations, static or dynamic analysis has been suggested in the original PortSmash article [3]. In particular, a process could try to differentiate malicious port usage from legitimate usage by using Hardware Performance Counters. However, to the best of our knowledge, static or dynamic analysis of contention-based side channels has not been studied in the literature.

Port-independent code has also been suggested [3]. If the port usage does not vary accordingly to the secret information, then port-contention-based side channel attacks are ineffective. However, such a solution requires to detect and correct all sensitive code in existing sensible implementations, and does not apply to covert channels.

At the operating system level, the scheduler can be aware of SMT attacks, and provide more isolation between processes. For instance, allowing highly sensitive operations, such as computations depending on a secret, to run on a different physical core than other applications could reduce the risk of leaking private information in a side-channel attack. Similarly, only sharing hardware resources between processes owned by the same user could provide more isolation, especially in cloud environments.

Browsers. After the publications of microarchitectural attacks inside the JavaScript sandbox [18, 27], browser vendors studied mitigations against timing attacks. A popular solution in browsers is to remove access to high-resolution timers. By not granting access to a timer able to identify port contention, the side channel would be mitigated. In particular, by disabling `SharedArrayBuffer`, the threat posed by port contention side channels would be diminished. However, this would only reduce the resolution of the side channel and lower the bitrate of our covert channel, but not fully prevent attacks, as other high-resolution timers have been implemented [30, 33]. Browser vendors recently shifted their mitigation paradigm from timer-based countermeasures to isolation-based countermeasures. However, proposed isolation-based countermeasures [29] focus on memory isolation, and therefore do not apply to port contention side channels.

7.3 Future Work

This work paves the way to future work on the threat posed by contention-based side channels in the browser. First, the security implications of WebAssembly are not properly evaluated yet, especially in the field of microarchitectural attacks. Studying the compilation of WebAssembly and the resulting threats on microarchitecture would bring a more systematized approach to this field. A benchmark, similar to Abel and Reineke's `uops.info` [2] could clarify the execution pipeline, from high level JavaScript code to native code, including WebAssembly instructions. Moreover, a more generic study of contention-based side channel in the browser, not only ports, could widen the attack surface to other types of victims or other threat models. Finally, we presented a covert channel and an artificial example exploiting port contention. Since our attack has a temporal resolution at least in the order of Prime+Probe, we infer it can be used as the fundamental building block of many future attacks, e.g., on cryptographic implementations or monitoring.

8 Conclusion

We presented the first implementation of port contention in the browser. We showed that port contention side channels have a performance on-par or better than previous microarchitectural side channels in the browser, and a more generic threat model. We demonstrated the genericity of this attack by building several types of exploits, including a 200 bit/s covert channel, as well as a concrete example illustrating a side-channel attack with a spatial resolution of 1024 instructions. We further demonstrated the portability of web-based port contention by testing instructions on different Intel CPUs, and we showed that our attack also works in cross-browser and Host-to-VM settings, while being more resilient to noise than cache attacks. We consider port contention side channels, and hardware contention side channels in general, to be a generic class of attacks that can be used as a building block for future microarchitectural attacks in the browser. This work illustrates the difficulty to isolate the JavaScript sandbox from microarchitectural attacks, as currently deployed countermeasures fail to mitigate contention-based side channels.

Acknowledgments

This work benefited from the support of Intel and of the project ANR-19-CE39-0007 MIAOUS of the French National Research Agency (ANR).

Artifact Availability

To ensure the repeatability of our findings and assist defensive research, we will publicly release all developed code and data artifacts. This includes the documented source code of PC-detector, our covert channel, and the artificial example, as well as the data and results of our experiments. In particular, we hope that public access to PC-detector on a larger scale will help assess the whole picture of the threat posed by port contention side channels.

References

[1] Andreas Abel and Jan Reineke. Tzcnt uops.info page. https://uops.info/html-instr/TZCNT_R16_R16.html. Accessed: 2021-11-11.

[2] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, 2019.

[3] 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.

[4] Jay M Berger. A note on error detection codes for asymmetric channels. *Information and control*, 4(1):68–73, 1961.

[5] 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.

[6] Feng Cao, Martin Estert, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *Proceedings of the 2006 SIAM international conference on data mining*, 2006.

[7] MDN contributors. Cross-origin-embedder-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy. Accessed: 2021-19-11.

[8] MDN contributors. Cross-origin-opener-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy. Accessed: 2021-19-11.

[9] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security Symposium*, 2020.

[10] ECMA. Standard ecma-262. https://www.ecma-international.org/publications/standards/Ecma-262.htm. Accessed: 2021-10-11.

[11] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, 2018.

[12] Google. Product status: Microarchitectural data sampling (mds). https://support.google.com/faqs/answer/9330250?hl=en. Accessed: 2021-19-11.

[13] Google. V8 javascript engine. https://v8.dev/. Accessed: 2021-10-11.

[14] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[15] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, 2017.

[16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *DIMVA*, 2016.

[17] Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[18] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.

[19] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *HPCA*, 2009.

[20] Michael Larabel. Openbsd disabling smt / hyper threading due to security concerns. https://www.phoronix.com/scan.php?page=news_item&px=OpenBSD-Disabling-SMT. Accessed: 2021-19-11.

[21] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *ESORICS*, 2017.

[22] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *MICRO*, 2014.

[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, 2015.

[24] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*, 2017.

[25] MDN. Navigator.hardwareconcurrency. https://developer.mozilla.org/en-US/docs/Web/API/Navigator/hardwareConcurrency. Accessed: 2021-19-11.

[26] Mozilla. Spidermonkey javascript engine. https://spidermofnkey.dev/. Accessed: 2021-10-11.

[27] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *CCS*, 2015.

[28] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *S&P*, 2021.

[29] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security Symposium*, 2019.

[30] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *EuroS&P*, 2021.

[31] Michael Rushanan, David Russell, and Aviel D Rubin. Malloryworker: stealthy computation and covert channels using web workers. In *International Workshop on Security and Trust Management*. Springer, 2016.

[32] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: mitigating conflict-based cache attacks with a practical fully-associative design. In *USENIX Security Symposium*, 2021.

[33] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data*

Security, 2017.

[34] Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. Leaky controller: cross-vm memory controller covert channel on multi-core systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2020.

[35] 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 *USENIX Security Symposium*, 2021.

[36] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. Secsmt: Securing SMT processors against contention-based covert channels. In *USENIX Security Symposium*, 2022.

[37] Daniel Townley and Dmitry Ponomarev. SMT-COP: defeating side-channel attacks on execution units in SMT processors. In *PACT*, 2019.

[38] Tom van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[39] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *S&P*, 2019.

[40] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security Symposium*, 2017.

[41] W3C. Index of standardized webassembly instructions. https://webassembly.github.io/spec/core/appendix/index-instructions.html. Accessed: 2021-19-11.

[42] W3C. Webassembly. https://webassembly.org/. Accessed: 2021-10-11.

[43] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[44] Yinqian Zhang and Michael K. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

A Port Contention on Other WebAssembly Instructions

Figures 8 to 10 show port contention on the following WebAssembly instructions: `f64.floor`, the pair `f32.convert_i32_u` and `i32.trunc_f32_u`, and `i64.rem_u`. We can clearly distinguish the three outcomes of a PC-detector usage:

- Figure 8 illustrates an instruction that do not cause contention. The P1 and P5 distributions have a similar mean and standard deviation, making them difficult to distinguish. However, they are still distinguishable from the control experiment.
- Figure 10 illustrates a pair of instructions causing contention on P5. The distribution P5 has a higher mean than P1 and the control experiment.
- Figure 9 illustrates an instruction causing contention on P1. The distribution P1 has a higher mean than P5 and the control experiment.

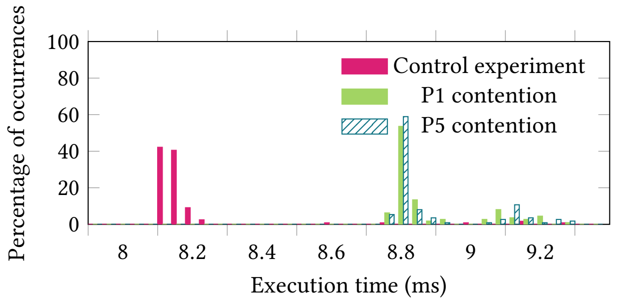


Figure 8: P1 contention experiment on `f64.floor` for 1 000 000 instructions.

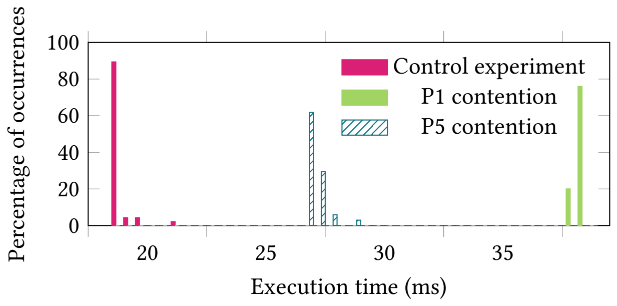


Figure 9: P1 contention experiment on `i64.rem_u` for 1 000 000 instructions.

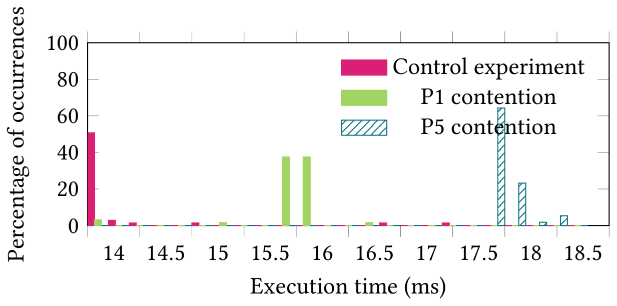


Figure 10: P5 contention experiment on paired `f32.convert_i32_u` and `i32.trunc_f32_u` for 1 000 000 instructions.