CPU Port Contention Without SMT

Thomas Rokicki, Clémentine Maurice, Michael Schwarz

▶ To cite this version:

Thomas Rokicki, Clémentine Maurice, Michael Schwarz. CPU Port Contention Without SMT. 27th European Symposium on Research in Computer Security (ESORICS 2022), Sep 2022, Copenhagen, Denmark. pp.209-228, 10.1007/978-3-031-17143-7_11 . hal-03798342

HAL Id: hal-03798342

https://hal.science/hal-03798342

Submitted on 5 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CPU Port Contention Without SMT

Thomas Rokicki[1][0000−0002−7041−4300], Clémentine Maurice[2][0000−0002−8896−9494], and Michael Schwarz[3][0000−0001−6744−3410]

[1] Univ Rennes, CNRS, IRISA
[2] Univ Lille, CNRS, Inria
[3] CISPA Helmholtz Center for Information Security

Abstract. CPU port contention has been used in the last years as a stateless side channel to perform side-channel attacks and transient execution attacks. One drawback of this channel is that it heavily relies on simultaneous multi-threading, which can be absent from some CPUs or simply disabled by the OS.
In this paper, we present *sequential port contention*, which does not require SMT. It exploits sub-optimal scheduling to execution ports for instruction-level parallelization. As a result, specifically-crafted instruction sequences on a single thread suffer from an increased latency. We show that sequential port contention can be exploited from web browsers in WebAssembly. We present an automated framework to search for instruction sequences leading to sequential port contention for specific CPU generations, which we evaluated on 50 different CPUs. An attacker can use these sequences from the browser to determine the CPU generation within 12 s with a 95 % accuracy. This fingerprint is highly stable and resistant to system noise, and we show that mitigations are either expensive or only probabilistic.

Keywords: Side channels · CPU port contention · Browsers · Fingerprinting.

1 Introduction

Microarchitectural attacks exploit side effects of the CPU's internal implementation. These attacks have been shown to leak sensitive data even in the absence of software vulnerabilities. Many of these attacks exploit a microarchitectural state that depends on a secret value. A thoroughly studied state is the cache state, *i.e.*, if data resides in the cache or the main memory. This state can be observed using timing measurements when accessing data, as done in, e.g., cache attacks such as Flush+Reload [56]. Such *stateful* channels have the advantage that attacker and victim do not have to run in parallel. Recently, there were also advances in the exploitation of *stateless* channels. In such channels, the microarchitectural state change does not persist and can only be observed while the victim is running. As attacks using stateless channels require the attacker and victim to run in parallel while sharing the same hardware resources, they typically rely on simultaneous multi-threading (SMT), also known as hyper-threading

on Intel CPUs. An example of such a channel is port contention [3], where an attacker observes the latency of executed instructions caused by a victim on the hyper-thread that blocks resources necessary to execute the instructions.

The majority of microarchitectural attacks are initially shown in native code. Browsers specifically modified for prototyping microarchitectural attacks [39,11] can help port such attacks step-by-step to the browser, ultimately enabling these attacks from unmodified browsers. The number of such attacks that can be mounted in the browser is steadily increasing [39] even though the sandboxed environment in browsers prevents access to low-level functionality used in microarchitectural attacks, such as high-resolution timers [46,39] or control over the CPU-core placement [11,40]. Despite all these challenges, stateful channels [34,20] and stateless channels [40] have been exploited in browsers.

In this paper, we introduce a new stateless channel that can be exploited in the browser as well. We introduce *sequential port contention*, a novel form of contention on the CPU execution ports. With sequential port contention, we show that port contention [5,40] does not necessarily require SMT. Instead of exploiting *thread-level* parallelism, we exploit *instruction-level* parallelism. We exploit the limited look-ahead window of the instruction scheduler that results in sub-optimal scheduling–and, as a result, increased latency due to sub-optimal instruction-level parallelism–for specific instruction sequences on a single CPU core. We show that such sequences work in native code but also work reliably in WebAssembly. Our side channel works in unmodified off-the-shelf browsers, including the privacy-focused browsers Tor and Brave.

We present an automated framework to search for instructions sequences leading to sequential port contention in the browser. The framework supports Intel and AMD CPUs and works in WebAssembly. Our evaluation on 26 CPUs spanning 13 CPU generations discovers at least 36 instruction sequences causing sequential port contention.

In a case study, we demonstrate that the framework discovers sequences that allow distinguishing CPU generations. We use the results from our framework to automatically build a k-NN classifier that reliably detects the CPU generation based only on timing measurements. As we use a differential measuring approach based on sequential port contention, our results are independent of the overall CPU performance, *i.e.*, the CPU frequency and workload on other CPU cores do not impact our classification. We evaluate our classifier in the browser on 50 CPUs[1]. From within Chrome and Firefox, we classify the CPU generation with a very high accuracy of 95 % within, on average, 12 s.

Due to the robustness of our approach, we show that our side channel is hard to mitigate and is highly resistant to system noise. Moreover, proposals for preventing (hardware) fingerprinting [9,8,7] are ineffective, as we only require coarse-grained timing measurements. The CPU does not change often, and mitigations against this type of attack are difficult, granting the fingerprint a high stability over time. We show that our fingerprint is stable on all major

[1] Sources and evaluation data are available on https://github.com/MIAOUS-group/port-contention-without-smt.

releases of Chrome and Firefox in a year and, therefore could be used to link less stable fingerprints [26,53].

Hence, we stress that this new side channel is a real privacy risk, as it can be used to track users based on their CPU. Moreover, we show that sequential port contention is also possible in a virtualized environment but stops working in emulated environments. These results also indicate that the side channel is valuable for malware as an anti-emulation measure.

Our key contributions are:

- We introduce a CPU-port-contention primitive that relies on instruction-level parallelism instead of SMT and build a framework to automatically find WebAssembly instructions creating such sequential port contention.
- We use this primitive to fingerprint the CPU generation in WebAssembly in web browsers without any browser API.
- We evaluate our new fingerprinting method on 50 CPUs from 12 generations with an accuracy of 95 % with a runtime of only 12 s.
- We discuss that the fingerprint is highly stable over major releases of browsers, is robust against system noise, and mitigations are difficult.

2 Background

2.1 CPU Port Contention

Simultaneous multithreading (SMT) allows parallelization by sharing the resources of one physical core across two or more logical cores. Intel's SMT implementation is called Hyper-Threading (HT). Typically, a pair of logical cores in the same physical core shares L1 and L2 caches, a branch prediction unit, and the execution engine, among other components.

Instructions are fetched from memory by the pipeline, which in a second step decomposes each instruction into smaller, atomic operations, called micro-operations or μops. The μops are then distributed to the execution engine by the scheduler through multiple CPU *execution ports* belonging to the *execution units*. Each execution unit is specialized to process precise types of instructions, e.g., arithmetic μops are distributed to port 0, 1, 5, or 6 (noted P0156). Abel et al. [1] have documented the port usage of instructions for various Intel CPU generations. The port usage can differ from one generation to another for the same instruction.

Since the execution engine is shared across two logical cores, two threads on the same physical core can create contention on this resource by executing instructions issued to the same port. Port contention has been used in side-channel attacks [3] and transient execution attacks [5]. Rokicki et al. [40] showed that this side channel is exploitable from web browsers using WebAssembly.

2.2 WebAssembly

WebAssembly [54] is a binary instruction format for a stack-based virtual machine. It is designed to be deployed on the web, on the client or server sides. It is a

portable compilation target from other languages, such as C, C++ or Rust, with the purpose of bringing native-like performance to the browser. Client-side WebAssembly is designed to run inside the JavaScript sandbox [18], hence it is heavily restricted for security purposes: among others, it cannot use native instructions or have access to arbitrary memory addresses. WebAssembly is built around a stack machine in a format resembling native assembly. WebAssembly offers more than 200 specified instructions [17], including SIMD operations. Although originally a binary language, it supports a human-readable text format, allowing reading and modifying compiled WebAssembly code at a low level.

2.3 Browser Fingerprinting

Browser fingerprinting is a stateless technique collecting data from the browser or machine configuration, usually from dedicated APIs [29]. It aims to construct a unique identifier, called a browser fingerprint, without storing any cookie. JavaScript APIs and HTTP headers give information such as the User Agent, screen resolution, and time zone, which, alone, are perfectly harmless and even help enhance user experience on websites. However, the combination of these attributes is often unique, and can therefore be used for either tracking or as another factor of authentication [27].

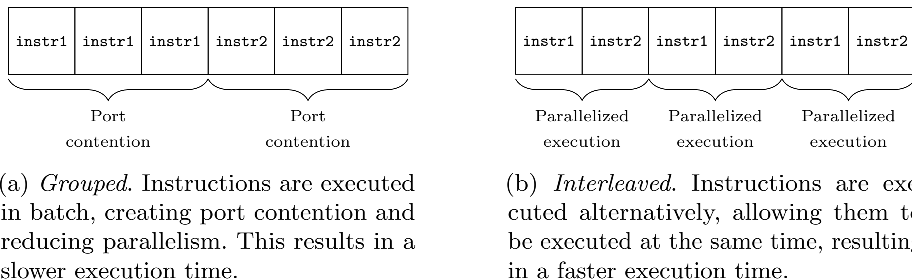
To be useful, a fingerprint should have the following properties. *Uniqueness*: a fingerprint should be able to uniquely identify users. This is obtained by collecting multiple attributes, rather than from a single unique attribute. *Stability*: any change in an attribute value changes the fingerprint and therefore breaks user identification. However, relying on software fingerprinting means that attributes are constantly changing (e.g., the browser version in the User Agent). Vastel et al. [53] showed that it is possible to link two fingerprints that are slightly different from each other through heuristics. Therefore, for a single attribute, uniqueness is less important than stability to link fingerprints.

3 Threat Model

Sequential port contention, as most microarchitectural attacks, requires code execution on the victim machine. We assume that the attacker either has native unprivileged code execution (native side channel) or can run WebAssembly in the victim's browser (browser-based side channel). The attacker does not rely on software vulnerabilities, does not require any permissions that have to be granted by the victim, or any particular setup such as SMT or a specific core assignment. We assume that the victim spends at least 15 s on the attacker's website, based on the average time of 20 s users dwell on an unknown website [32].

4 Port Contention Without SMT

Port contention, as described by Aldaya et al. [3], requires SMT. Both the attacker and the victim need to run on the same physical core for the attack to work, as



(a) *Grouped.* Instructions are executed in batch, creating port contention and reducing parallelism. This results in a slower execution time.

(b) *Interleaved.* Instructions are executed alternatively, allowing them to be executed at the same time, resulting in a faster execution time.

Fig. 1: Illustration of the differences in execution time based on the order of instructions, with a look-ahead window of size 1.

CPU ports are on-core resources. This prerequisite represents a challenge in some settings, as some systems do not have SMT or disable it [21], and it may become increasingly hard to fulfill as countermeasures to SMT attacks are being explored [52,51]. It also has severe implications in a web setting, where the attacker script, situated inside the JavaScript sandbox, cannot know nor control which core it is running on. In this section, we show port contention without requiring SMT, both in a native setting and in a browser sandbox.

4.1 Main Idea

The main idea of *sequential port contention* is to exploit the limited look-ahead window of the μop scheduler, leading to contention for well-chosen instruction pairs (`instr1`, `instr2`). Both instructions use different execution ports on the CPU. If the instructions are grouped, *i.e.*, if the instruction stream consists of n instructions `instr1`, followed by n instructions `instr2`, with n larger than the look-ahead window of the scheduler, parallelization is not possible (cf. Figure 1(a)). The scheduler cannot detect that some instructions later in the instruction stream could already be executed in parallel. However, if interleaved in an instruction stream of $2n$ instructions, they can be executed in parallel (cf. Figure 1(b)). As a result, the overall execution time of an instruction stream of the same length depends on the order of the two repeated instructions `instr1` and `instr2` if these instructions do not use the same ports.

Similar to port contention with SMT [3], the contending instructions `instr1` and `instr2` depend on the underlying microarchitecture. However, as this information is publicly available [1], sequential port contention is applicable to a wide range of microarchitectures. We show sequential port contention in native environments (Section 4.2) and demonstrate that it is also exploitable from off-the-shelf unmodified browsers (Section 4.3).

Listing 1.1: *Grouped.* Always creates contention.

```
1  grouped:
2    lfence
3    rdtsc # Timestamp
4    lfence
5    .rept $n # First loop
6      instr1 %reg, %reg
7    .endr
8    .rept $n # Second loop
9      instr2 %reg, %reg
10   .endr
11   lfence # Timestamp
12   rdtsc
```

Listing 1.2: *Interleaved.* Creates contention if the two instructions share a CPU port.

```
13 interleaved:
14   lfence
15   rdtsc # Timestamp
16   lfence
17
18   .rept $n # Single loop
19     instr1 %reg, %reg
20     instr2 %reg, %reg
21   .endr
22
23   lfence # Timestamp
24   rdtsc
```

4.2 Native Environment

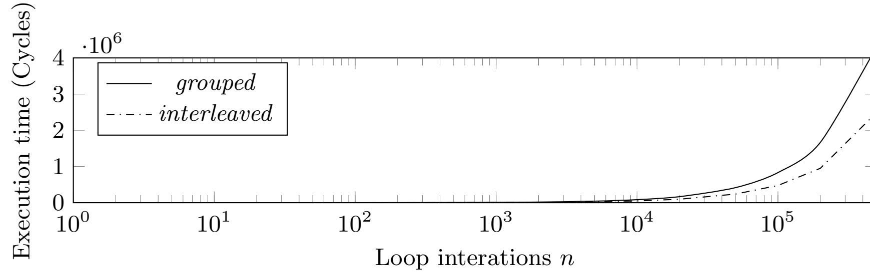
Proof of Concept Our proof of concept of sequential port contention is based on two experiments, illustrated in Listings 1.1 and 1.2. In these experiments, we evaluate two native x86 instructions, `instr1` and `instr2`.

The first experiment is a control experiment, *grouped*, which is composed of two loops, each calling an instruction n times. As the decomposition of instructions in μops is deterministic, the various calls to the same instructions have the same port usage. This means that during loop 1 (respectively loop 2), `instr1` (respectively `instr2`) always creates contention on its ports. The second experiment, *interleaved* is composed of a single loop with the same number of iterations. Instead of calling the same instructions in a row, it alternatively calls `instr1` and `instr2`. If `instr1` and `instr2` emit μops to the same port, it creates contention, resulting in a slower overall execution time. However, if they do not emit μops on the same port, the execution is parallelized due to instruction-level parallelization, resulting in a faster execution time.

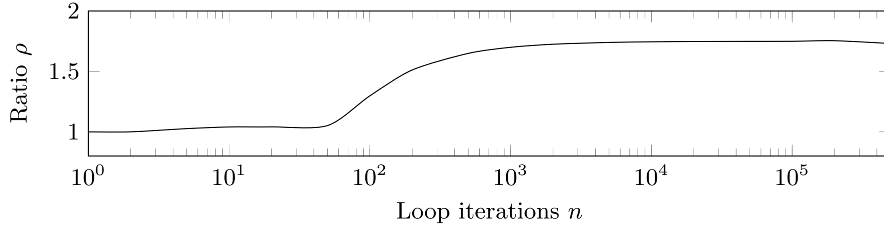
By computing $\rho = \frac{time(grouped)}{time(interleaved)}$, we know if *interleaved* creates contention. If $\rho \approx 1$, both experiments have a similar execution time, *i.e.*, the instructions share at least one port. If $\rho > 1$, *interleaved* has a shorter execution time than *grouped*, *i.e.*, the instructions do not share a common port.

Experiments We run this experiment on an Intel i5-8365U (Whiskey Lake), with TurboBoost enabled and without fixing the CPU frequency. First, we run it with `instr1` = `crc32`, which emits a single μop on execution port 1 (P1), and `instr2` = `aesdec`, which emits a single μop on execution port 0 (P0). Both instructions have the same throughput and latency.

Figure 2 illustrates the results of this experiment when we vary the number of loop iterations n. Figure 2(a) shows how the *grouped* execution time is systematically higher than the *interleaved* one. The gap between the two curves increases with the number of loops. Figure 2(b) shows that ρ quickly converges to 1.8 at $n = 1000$. It then remains constant when increasing the number of loop iterations. The inflection point situated around $n = 64$ is caused by the size of the



(a) Execution time of the experiments depending on the number of loop iterations n.



(b) Ratio ρ depending on the number of loop iterations n.

Fig. 2: Sequential port contention experiments for instructions (`crc32`, `aesdec`).

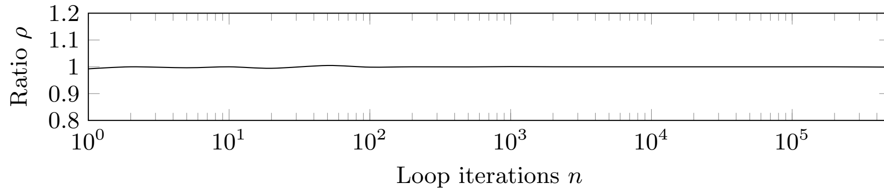


Fig. 3: Ratio ρ for (`crc32`, `popcnt`) depending on the loop iterations n.

look-ahead window of the scheduler. When instructions from both loops fit inside this window, the scheduler can rearrange instructions to execute them in the most optimized order, prioritizing parallelism and thus reducing port contention. When an `mfence` is added between the two loops (Lines 7-8 of Listing 1.1), this inflection point disappears, and the curve raises smoothly to 1.8.

We run the same experiment with `instr1` = `crc32` and `instr2` = `popcnt`. Both instructions emit a single P1 μop, and have the same throughput and latency. Figure 3 shows that ρ stays constant around 1. That is expected, as the contention is always the same on P1, independently of instruction order.

4.3 Web Browsers

Challenges Porting these experiments to a browser sandbox introduces new challenges. First, neither JavaScript nor WebAssembly provides high-resolution

Listing 1.3: *Grouped* in WebAssembly. Always creates contention.

```
25 (module
26  (func $grouped
27  (param $p type)(result type)
28   (local.get $p)
29         (type.instr_1)
30         ... # Repeat $n
31         (type.instr_1)
32         (type.instr_2)
33         ... # Repeat $n
34         (type.instr_2)
35  )
36  (export "grouped" (func $grouped))
37 )
```

Listing 1.4: *Interleaved* in WebAssembly. Creates contention if the two instructions share a CPU port.

```
38 (module
39  (func $interleaved
40  (param $p type)(result type)
41   (local.get $p)
42         (type.instr_1)
43         (type.instr_2)
44         ... # Repeat $n
45         (type.instr_1)
46         (type.instr_2)
47  )
48  (export "interleaved" (func $interleaved))
49 )
```

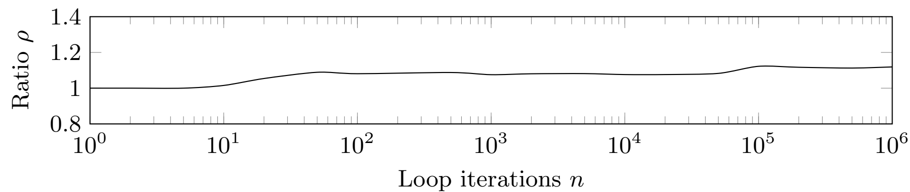


Fig. 4: Ratio ρ for the WebAssembly instructions (`i64.popcnt`, `i64.or`) depending on the number of loop iterations n.

timers. This comes from an effort from browser vendors to prevent timing attacks. However, it is still possible to create high-resolution auxiliary timers [46,39]. For all experiments in this section, we use a timer based on `SharedArrayBuffer`, defined by Schwarz et al. [46]. It uses a constant increment of a shared integer as a time unit and offers a resolution of 20 ns. However, this timer is still not as accurate as native cycle-accurate timers. Second, both JavaScript and WebAssembly are high-level languages, running inside a sandbox. There is no access to native instructions or arbitrary virtual addresses. Moreover, WebAssembly instructions are an abstraction of native instructions and thus do not directly map to execution ports. As WebAssembly is aimed at being portable, the translation of WebAssembly to native code depends on the browser's WebAssembly compiler and the targeted CPU. We can, however, empirically determine the port usage of these instructions for a system [40].

Proof of Concept Similar to native experiments, the sequential port contention in WebAssembly is composed of two different functions. Listing 1.3 shows the code for the *grouped* experiment, which results in a slow execution time as instructions are delayed by contention. Listing 1.4 shows the *interleaved* experiment. A low execution time indicates that the experiments were not slowed down by contention, whereas a high execution time means both instructions share at least one port.

Experiments We run this experiment on the same Intel i5-8365U CPU, with `instr1` = `i64.popcnt` and `instr2` = `i64.or`. Figure 4 illustrates how ρ also increases with the number of loops. On both Chrome 101 and Firefox 99, ρ stabilizes around 1.1 starting from $n = 100\,000$ loop iterations. This ratio is significantly lower than the native one. This stems from lower precision timers, as well as the stack structure of WebAssembly, where we need to add a value to the stack between instructions. Running the same experiment with `instr1` = `i64.popcnt` and `instr2` = `i64.ctz`, ρ remains constant around 1 when varying the number of loops. We devise a framework to isolate pairs of instructions that exhibit sequential contention in Section 5.2.

Privacy-oriented browsers are also vulnerable to sequential port contention. With 100 000 loop iterations in Brave 1.38, we obtain $\rho = 1.1$. In Tor Browser 11.0.11, `SharedArrayBuffer` is disabled by default to prevent timing attacks. However, we can still reproduce sequential port contention with the lower-resolution timer `performance.now()` by increasing the number of loop iterations n to 100 000 000. In that case, we obtain $\rho = 1.2$, but each experiment takes up to 1 sec, *i.e.*, 1000 times more than for other browsers.

5 Fingerprinting CPU Generations

In this section, we show how sequential port contention can be used to determine the CPU generation of the victim, even from inside the JavaScript sandbox.

5.1 Core Idea

The port usage of native instructions varies across generations of microarchitecture. As the number of execution units and CPU ports vary, the same instruction can emit P1 μops on a given generation and P0 on another generation. For instance, `VPBROADCASTD` emits one μop on P5 on both Haswell and Whiskey Lake microarchitectures, and `AESDEC` emits one μop on P1 on Haswell and one μop on P5 on Whiskey Lake. We computed ρ on an Intel i5-8365U (Whiskey Lake) and an Intel i3-4160T (Haswell). The frequency of these CPUs can vary. However, the base frequency does not impact our experiment as we compute a ratio. We found $\rho_{\text{WhiskeyLake}} = 1$ and $\rho_{\text{Haswell}} = 1.8$. This correlates with the documented port usage. Indeed, on Whiskey Lake, both instructions emit a μop on P5. Thus, both experiments are slowed down by contention. On Haswell, the two instructions do not share a common port. Thus, the *interleaved* experiment is not slowed down by port contention, resulting in a faster execution time and a ratio $\rho > 1$.

In summary, by finding pairs of instructions that create contention for some generations but not others, we can detect on which CPU generation the code is executed. As sequential contention is visible from a browser (cf. Section 4.3), we also aim to discover pairs of WebAssembly instructions that exhibit sequential port contention to fingerprint the CPU generation from a web page.

5.2 Framework

The port usage of the CPU-independent WebAssembly instructions cannot be determined from the WebAssembly source code. Thus, we build a framework based on PC-Detector [40] to automatically evaluate 458 pairs of WebAssembly instructions for contention on a specific CPU generation. Due to the nature of WebAssembly, it is highly portable and can be executed on any microarchitecture. This framework aims at isolating instruction pairs that can act as distinguishers. Such distinguishers have two major properties: 1) they exhibit different contention for different generations, and 2) they always exhibit the same contention for different CPUs of the same generation. The second property is essential, as other sources of contention that do not depend on the generation could yield false results. Changes in the microarchitecture, e.g., floating-point units, inside a CPU generation can cause changes in behaviors, thus preventing stable fingerprinting.

Using this framework, we collect the best distinguishers to create traces for each generation. To fingerprint generations, we create a k-NN-based classifier and train it with results from the framework. It represents traces as points in an l-dimensional space, where l is the length of the trace, *i.e.*, the number of distinguishers. Given a distance for each unknown execution trace, the classifier computes the k-nearest traces from our training dataset. A trace is classified in the most frequent class, *i.e.*, CPU generation, in the k-nearest-neighbors.

To collect evaluation traces for the two sequential port contention experiments, we use a simple web page (https://fp-cpu-gen.github.io/fp-cpu-gen). It works on the latest versions of Firefox and Chrome, on Linux, macOS and Windows.

5.3 Evaluation

This section presents the results of our classifier and the different parameters used. Our classifier presents a 95 % accuracy in a real-world threat model (cf. Section 3): a user visits a malicious website for a few seconds.

The *training set* is composed of 26 different CPUs spanning 13 different generations. It is composed of both AMD and Intel CPUs, including server and standard desktop CPUs. Table 1 in Appendix A presents the training set. The *test set* is composed of a subset of traces from the training set. It contains 13 different CPUs. The *evaluation set* is composed of traces from our website. These traces come from an uncontrolled environment since the web script cannot control or quantify the system noise. It contains 50 CPUs from 12 different generations.

Training and Testing We train our model using data from our training set. The CPUs used in our training set are not balanced in terms of CPU generations, some being more represented than others. We therefore include the same number of traces for each generation to compensate this. Our framework finds 36 pairs of instructions acting as distinguishers between the CPU generations. We use the traces from these distinguishers to train our k-NN classifier. Our model shows a 96 % accuracy on the test set, using $k = 5$ neighbors and majority voting.

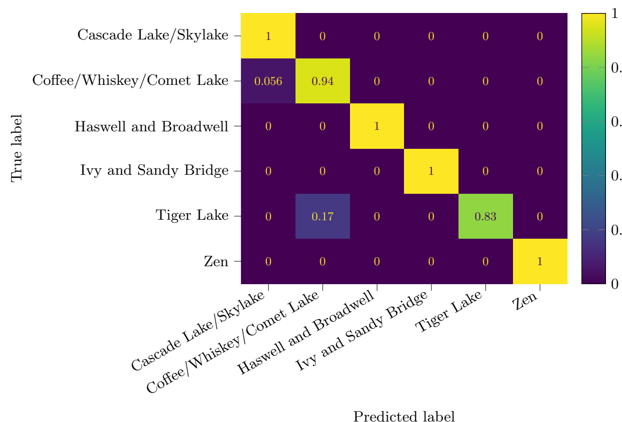


Fig. 5: Confusion matrix for the evaluation of the k-NN classifier with grouped generations, $k = 5$ and majority voting on 10 traces.

Accuracy Figure 5 illustrates the results of our classifier on the evaluation set. It has a balanced accuracy of 95 %. We use $k = 5$ as the number of neighbors. We gather 10 traces and classify each one independently. The class for the experiment is determined using majority voting on the 10 classified traces. Due to the lack of microarchitectural changes between closely-related generations, some generations have the same assignment of execution ports for all instructions. As a consequence, some generations are indistinguishable using sequential port contention. We grouped such generations in the classes of our classifier. This includes the Bridge (Ivy Bridge, Sandy Bridge), Well (Haswell, Broadwell), Skylake (Skylake, Cascade Lake), and Coffee-Lake group (Coffee Lake, Whiskey Lake, Comet Lake). AMD CPUs are distinguishable from Intel ones, but the generations are grouped in the Zen group (Zen 1, Zen 2, and Zen 3).

Execution Time The total execution time is composed of the offline and the online execution time. The *one-time offline execution time* is composed of the framework execution time and k-NN training time. On average, testing all pairs of instructions in the framework takes 4 h, with a standard deviation of 1 hour and 7 minutes. Training the k-NN model takes 5 s on an i5-8365U. The *online execution time* is composed of the data collection time, *i.e.*, the time taken on the website to gather the 10 traces, plus the prediction time, *i.e.*, the execution time of predicting the class of the trace. The data collection time is the most critical factor, as it represents the duration the victim has to remain on the web page. The client sends the traces to the server that then computes the prediction, so the victim can then close the web page. Data collection takes 12 s on average, with a standard deviation of 6 s. The data collection time is faster on average on Google Chrome (10 s) compared to Firefox (13 s). On both browsers, the data collection time is in the range of the average visit time on a website [32]. The

prediction time is on average of 40 ms for 10 traces on an i5-8365U, which is negligible compared to the data collection time.

Impact of Majority Voting System noise can decrease the accuracy of a single trace. In particular, the first traces gathered when launching the script are the most prone to these misclassifications. On our evaluation set, the first trace for each experiment has a 30 % chance of being mispredicted, the second 12 %, and then the misclassification rate goes down with repetitions to 6 %. There are multiple reasons, including the power saving policy of the system, where by default, the CPU does not use its maximum frequency to save energy, and cold caches. The first traces act as a "warm-up" of the CPU, before reaching maximum frequency. To compensate for this phenomenon, we implement majority voting. With majority voting, we gather data on v traces and classify the experiment based on the most common classification of these traces. This improves accuracy at the cost of a higher execution time. Without majority voting, our evaluation set shows an accuracy of 70 % with a data collection time of 5 s. With $v = 5$, the accuracy increases to 86 % and a data collection time of 9 s on average. Starting from $v = 10$, the accuracy peaks at 95 % with a data collection time of 12 s.

Impact of the Number of Neighbors The number of neighbors k is a significant factor in the classifier's efficiency. A small number renders the classification vulnerable to noise, as a single noisy trace in the training set can lead to mispredicting many evaluation traces. A higher number tends to increase the impact of densely grouped traces, as well as increase the computation costs.

For instance, when using $k = 1$ on the evaluation dataset, the classifier accuracy is reduced to 85 %. We found that $k = 5$ grants a higher accuracy for our testing and evaluation sets. Higher values of k tend to yield a lower accuracy. This comes from the similarity of traces between closely-related generation groups, e.g., Skylake and Coffee-Lake groups.

Time Stability Time stability is an essential feature, as hardware is seldomly changed by users, compared to software that has regular updates. We evaluate the stability of the classifier on an Intel i5-8365U on each major version of Firefox and Chrome covering about a year. The generation is correctly classified from Chrome 91 (released in May 2021) to 101 and Firefox 89 (released in June 2021) to 100. Prior versions did not support WebAssembly SIMD instructions, which are part of the distinguishers in our traces. Our CPU-generation fingerprints have been stable for a year. This represents a high time stability for a browser fingerprint compared to ever-changing browser APIs and other hardware-based approaches, such as DrawnApart [26], where the fingerprint may change with browsers' major releases, resulting in a median tracking time of 28 days.

Impact of Noise on Classification As the attacker resides inside a sandbox, they cannot know nor control noise created by other processes or tabs. Such noise

could deteriorate the performance of our classifier by creating wrong results in the data collection process. We run the data collection process in the website on Firefox 100 and Chrome 101 on a quadcore i5-8365U, while artificially creating noise with the `stress` command. The stress threads create noise, disturbing either the sequential port contention or the clock thread. Fewer noise sources, *i.e.*, `stress -c {1..4}`, result in 93 % accuracy. That is because the OS's scheduler balances the workload, and the attack physical core is not affected by the noise. A higher count of stress threads, i.e., 5 to 8, still yields an accuracy of 75 %.

6 Discussion

In this section, we discuss the practical use of CPU generation fingerprinting (Section 6.1), its limitations (Section 6.2), the effects of virtualization and emulation (Section 6.3) as well as possible mitigations (Section 6.4).

6.1 Practical Use of CPU-Generation Fingerprinting

The CPU-generation attribute does not have a high uniqueness, as even with a bigger training set, there are a limited number of CPU generations. The relevant feature here is its *stability*. We envision using this new fingerprinting attribute in combination with existing attributes. Its stability can be used as a linking factor to better link fingerprints to enhance tracking time [53] or use fingerprinting as a second authentication factor [27]. Hardware-based fingerprinting attributes are ideal candidates, as hardware is updated less often than software, and software updates usually lead to changes in fingerprints. However, even robust hardware-based methods can break with browser internal changes [26]. We have shown that our method is robust to major version changes of browsers over a year.

6.2 Limitations

For CPU generations with major changes, sequential port contention is a highly reliable method to fingerprint the CPU generation. Such changes are found on Intel CPUs between the Bridge (e.g., Sandy Bridge, Ivy Bridge), the Well (e.g., Haswell, Broadwell), and the Lake (e.g., Skylake, Coffee Lake, Whiskey Lake, Comet Lake) microarchitectures. However, starting with the successful Lake microarchitecture, changes between new versions are smaller, making it harder to detect the specific microarchitecture. For example, Coffee Lake, Whiskey Lake, and Comet Lake are based on the nearly identical designs of the execution units. Only Ice Lake introduced changes again, specifically with an additional store unit [55] which subsequently led to changes in the port assignment for several instructions. Hence, the detection of the CPU generation cannot differentiate names for essentially the same generation.

Due to lack of access, some generations are not included in the training set, e.g., Nehalem or Ice Lake. Thus, they cannot be correctly predicted by our proof-of-concept model and are not included in the evaluation set. This could be

easily corrected by extending our study and running the framework on a larger range of CPUs. CPUs with significant microarchitectural changes are potentially highly identifiable, e.g., Ice Lake with its addition of new store units.

6.3 Virtualization and Emulation

Sequential port contention is not limited to bare-metal code execution but also works from inside virtual machines if the guest is virtualized and not emulated.

Virtualization As all involved instructions are unprivileged and not emulated by the hypervisor, there is no difference in the execution stream to a bare-metal execution. Hence, the measured effects are also the same. Moreover, as only a single CPU core is required, the scheduler of the hypervisor does not affect the contention. We verify on Ubuntu 20.04 (kernel 5.13) with QEMU KVM 4.2.1 that we measure the same effect of sequential port contention within a virtual machine (Ubuntu 20.04, kernel 5.4).

Emulation Sequential port contention requires that the specifically-crafted instruction stream is executed without modifications on the CPU. For emulation, this is not the case if instructions are interpreted or translated just in time with potential additional instructions in the instruction stream. For example, when running the guest operating system (Ubuntu 20.04, kernel 5.4) in QEMU 4.2.1 with full system emulation (TCG), we are unable to measure the effect of sequential port contention. In this setup, the instruction stream with and without contention have the same execution time.

Based on this observation, sequential port contention can detect emulation, e.g., if the code is analyzed via a malware-analysis emulator [25,6]. Hence, sequential port contention provides malware with another trick to detect such environments. As discussed in Section 6.4, mitigating sequential port contention is difficult. Likewise, sequential port contention is likely infeasible to emulate, making it difficult to prevent malware from detecting the presence of an emulator.

6.4 Mitigation

Sequential port contention does not require any operating-system interface or particular setup, such as SMT (cf. Section 3). Hence, this side channel cannot be prevented on the operating-system level but potentially on the browser level. As previous work on microarchitectural attack detection [19,35,22,47], we show that this side channel can also be detected using hardware performance counters.

Browser Mitigation Existing browser mitigations against side-channel attacks are only effective against sequential port contention if they block access to timing sources [24,33,45] or entirely prevent the execution of active content [13,37]. However, while effective, these methods also impact the usability of all websites.

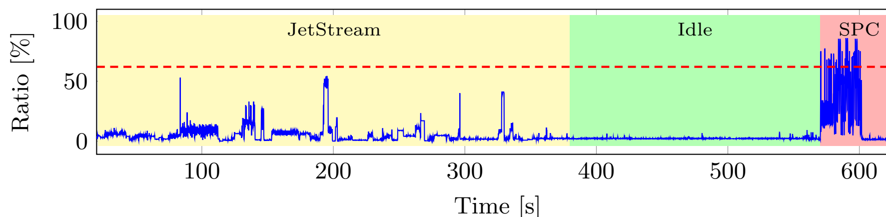


Fig. 6: Ratio of backend-bound to misprediction-bound execution when unning the JetStream JavaScript and WebAssembly benchmark (left), nothing (middle), and our website for generating the browser fingerprint (right) in Firefox 100.0.2.

The browser can interleave the generated instruction stream with memory fences, effectively preventing out-of-order execution. Theoretically, to fully mitigate the side channel, a browser has to emit a memory fence after every assembly instruction. However, this leads to unacceptable performance penalties, as it effectively prevents out-of-order execution while additionally adding the overhead of the fence (multiple cycles) after every instruction. A trade-off between the number of inserted fences and signal strength might be feasible, though. We leave an evaluation to future work.

Alternatively, the browser can reorder the instruction stream while keeping its functionality. Such reordering can be part of existing compiler optimizations, such as loop optimizations. Software-diversification approaches have also been shown as mitigation against side-channel attacks [10,36]. As the code required for sequential port contention requires precise control over the instruction sequence, any diversification likely breaks the side channel. We leave the evaluation of software-diversification methods applied by the browser to future work.

Detection via Performance Counters To detect sequential port contention, we propose a metric based on the topdown bottleneck decomposition [58]. Previous work focused mostly on cache-based performance counters for detecting microarchitectural attacks [19,35,22,47]. However, for sequential port contention, the cache activity is indistinguishable from typical workloads. The bottleneck exploited in sequential port contention is the execution unit in the backend. As the instruction stream is entirely linear, we use the ratio of backend-bound execution divided by misprediction-bound execution. Hence, the more often the bottleneck is in the backend, combined with next to no mispredictions, the higher the likelihood that the monitored snippet uses sequential port contention.

Figure 6 shows the evaluation of this metric in Firefox while running the JetStream JavaScript and WebAssembly benchmark (left), nothing (middle), and our website for generating the browser fingerprint (right). Our tests do not show any workload where this metric is as high as for sequential port contention, allowing detection of this side channel using a simple threshold (dashed line).

7 Related Work

7.1 SMT Side-Channel Attacks

Aldaya et al. [3] introduced the first SMT side-channel attack based on port contention. Their native implementation allowed inferring private keys from OpenSSL's TLS. Bhattacharyya et al. [5] exploited port contention to create a covert channel in a speculative execution attack. Other on-core resources can be targeted by SMT side-channel attacks: the Translation Lookaside Buffer [15], L1 data cache [57], L1 way predictor [31], or the μop-cache [38]. In a more systematized approach, ABSynthe [14] is a black-box framework to automatically detect on-core contention sources. The contention source is not documented by the framework, but is leveraged in a side-channel attack to recover EdDSA keys.

7.2 Side-Channel Attacks in Browsers

With the Prime+Probe cache attack in JavaScript, Oren et al. [34] proposed the first microarchitectural side-channel attack in the browser. Cache occupancy was also used to monitor opened websites in the browser [48]. DRAM has also been targeted to reproduce Rowhammer [20] or create a covert channel [46] in the browser. Gras et al. [16] demonstrated that an attacker in the JavaScript sandbox can reverse ASLR and de-randomize virtual addresses. Microarchitectural side-channels also let an attacker track user's browsing history through Floating-Point Units [4]. Transient execution attacks have also been shown in the browser, including Spectre [23], ZombieLoad [11], and RIDL [43].

7.3 Browser Fingerprinting

The first attempt to use fingerprints to de-anonymize web clients was introduced by Eckersley [12]. Laperdrix et al. [28] presented an overview of existing browser fingerprinting techniques and applications. Most fingerprints rely on software attributes, such as HTTP headers and user agents [12], Canvas API [50,2], or browser extensions [49,41,30]. Hardware features have also been used to create fingerprints. Sanchez-Rola et al. [42] demonstrated how imperfections in computer internal clocks can be used to fingerprint unique machines. JavaScript template attacks [44] were applied to fingerprinting, retrieving the instruction-set architecture, and used memory allocator from the JavaScript sandbox. Laor et al. [26] identified devices based on unique properties in the GPU stack.

8 Conclusion

We introduced sequential port contention, a new side channel based on port contention that does not require SMT. We proposed a WebAssembly framework to automatically determine instruction sequences creating sequential port contention. We demonstrated that an attacker can exploit sequential port contention to

determine the CPU generation of a victim from the browser within 12 s. This information is highly stable, and the attack works correctly even under heavy system noise. This new side-channel is privacy threatening, as it is hard to mitigate and can be used for improving the stability of fingerprints.

Acknowledgments This work benefited from the support of the project ANR-19-CE39-0007 MIAOUS of the French National Research Agency (ANR), and ANR-21-CE39-0019/Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) 491039149 FACADES.

References

1. Abel, A., Reineke, J.: uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In: ASPLOS (2019)
2. Acar, G., Eubank, C., Englehardt, S., Juárez, M., Narayanan, A., Díaz, C.: The web never forgets: Persistent tracking mechanisms in the wild. In: CCS (2014)
3. Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port contention for fun and profit. In: S&P (2019)
4. Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H.: On subnormal floating point and abnormal timing. In: S&P (2015)
5. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: Smotherspectre: Exploiting speculative execution through port contention. In: CCS (2019)
6. Brengel, M., Backes, M., Rossow, C.: Detecting hardware-assisted virtualization. In: DIMVA (2016)
7. Bugzilla: Check crossoriginisolated for all nsrfpservice::reducetimeprecision* callers. https://bugzilla.mozilla.org/show_bug.cgi?id=1586761, accessed: 2022-05-20
8. contributors, M.: Cross-origin-embedder-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy, accessed: 2021-19-11
9. contributors, M.: Cross-origin-opener-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy, accessed: 2021-19-11
10. Crane, S., Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Thwarting cache side-channel attacks through dynamic software diversity. In: NDSS (2015)
11. Easdon, C., Schwarz, M., Schwarzl, M., Gruss, D.: Rapid prototyping for microarchitectural attacks. In: USENIX Security Symposium (2022)
12. Eckersley, P.: How unique is your web browser? In: Privacy Enhancing Technologies (2010)
13. Giorgio Maone: NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience! (July 2017), https://noscript.net
14. Gras, B., Giuffrida, C., Kurth, M., Bos, H., Razavi, K.: Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In: NDSS (2020)
15. Gras, B., Razavi, K., Bos, H., Giuffrida, C.: Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In: USENIX (2018)
16. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: Aslr on the line: Practical cache attacks on the mmu. In: NDSS (2017)
17. Group, W.C.: Index of instructions webassembly 2.0. https://webassembly.github.io/spec/core/appendix/index-instructions.html, accessed: 2022-05-20

18. Group, W.C.: Security - webassembly. https://webassembly.org/docs/security/, accessed: 2022-05-20
19. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA (2016)
20. Gruss, D., Maurice, C., Mangard, S.: Rowhammer. js: A remote software-induced fault attack in javascript. In: DIMVA (2016)
21. Hat, R.: Disabling smt to prevent cpu security issues using the web console. https://access.redhat.com/documentation/en-us/red-hat-enterprise-linux/8/topic/f1d65124-781b-4543-a51a-d2bf9fa794ac, accessed: 2022-05-10
22. Irazoqui, G., Eisenbarth, T., Sunar, B.: Mascat: Preventing microarchitectural attacks before distribution. In: CODASPY (2018)
23. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. In: S&P (2019)
24. Kohlbrenner, D., Shacham, H.: Trusted browsers for uncertain times. In: USENIX Security Symposium (2016)
25. Kruegel, C.: Full system emulation: Achieving successful automated dynamic analysis of evasive malware. In: BlackHat USA (2014)
26. Laor, T., Mehanna, N., Durey, A., Dyadyuk, V., Laperdrix, P., Maurice, C., Yossi Oren, R.R., Rudametkin, W., Yarom, Y.: DrawnApart: A Device Identification Technique based on Remote GPU Fingerprinting. In: NDSS (2022)
27. Laperdrix, P., Avoine, G., Baudry, B., Nikiforakis, N.: Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting. In: DIMVA (2019)
28. Laperdrix, P., Bielova, N., Baudry, B., Avoine, G.: Browser fingerprinting: A survey. ACM Trans. Web **14**(2), 8:1–8:33 (2020)
29. Laperdrix, P., Rudametkin, W., Baudry, B.: Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In: S&P (2016)
30. Laperdrix, P., Starov, O., Chen, Q., Kapravelos, A., Nikiforakis, N.: Fingerprinting in style: Detecting browser extensions via injected style sheets. In: USENIX Security Symposium (2021)
31. Lipp, M., Hadžić, V., Schwarz, M., Perais, A., Maurice, C., Gruss, D.: Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: AsiaCCS (2020)
32. Liu, C., White, R.W., Dumais, S.T.: Understanding web browsing behaviors through weibull analysis of dwell time. In: SIGIR (2010)
33. Mao, J., Chen, Y., Shi, F., Jia, Y., Liang, Z.: Toward Exposing Timing-Based Probing Attacks in Web Applications. In: International Conference on Wireless Algorithms, Systems, and Applications (2016)
34. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The spy in the sandbox: Practical cache attacks in javascript and their implications. In: CCS (2015)
35. Payer, M.: HexPADS: a platform to detect "stealth" attacks. In: ESSoS (2016)
36. Rane, A., Lin, C., Tiwari, M.: Raccoon: Closing digital {Side-Channels} through obfuscated execution. In: USENIX Security Symposium (2015)
37. Raymond Hill: uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean. (July 2017), https://github.com/gorhill/uBlock
38. Ren, X., Moody, L., Taram, M., Jordan, M., Tullsen, D.M., Venkat, A.: I see dead μops: Leaking secrets via intel/amd micro-op caches. In: ISCA (2021)
39. Rokicki, T., Maurice, C., Laperdrix, P.: Sok: In search of lost time: A review of javascript timers in browsers. In: EuroS&P (2021)

40. Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y.: Port contention goes portable: Port contention side channels in web browsers. In: ASIACCS (2022)
41. Sánchez-Rola, I., Santos, I., Balzarotti, D.: Extension breakdown: Security analysis of browsers extension resources control policies. In: USENIX Security Symposium (2017)
42. Sánchez-Rola, I., Santos, I., Balzarotti, D.: Clock around the clock: Time-based device fingerprinting. In: CCS (2018)
43. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: rogue in-flight data load. In: S&P (2019)
44. Schwarz, M., Lackner, F., Gruss, D.: Javascript template attacks: Automatically inferring host information for targeted exploits. In: NDSS (2019)
45. Schwarz, M., Lipp, M., Gruss, D.: JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: NDSS (2018)
46. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In: International Conference on Financial Cryptography and Data Security (2017)
47. Schwarzl, M., Borrello, P., Kogler, A., Varda, K., Schuster, T., Gruss, D., Schwarz, M.: Dynamic process isolation. arXiv:2110.04751 (2021)
48. Shusterman, A., Kang, L., Haskal, Y., Meltser, Y., Mittal, P., Oren, Y., Yarom, Y.: Robust website fingerprinting through the cache occupancy channel. In: USENIX Security Symposium (2019)
49. Starov, O., Laperdrix, P., Kapravelos, A., Nikiforakis, N.: Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In: WWW (2019)
50. Stone, P.: Pixel perfect timing attacks with HTML5 (2013)
51. Taram, M., Ren, X., Venkat, A., Tullsen, D.: Secsmt: Securing SMT processors against contention-based covert channels. In: USENIX Security Symposium (2022)
52. Townley, D., Ponomarev, D.: SMT-COP: defeating side-channel attacks on execution units in SMT processors. In: PACT (2019)
53. Vastel, A., Laperdrix, P., Rudametkin, W., Rouvoy, R.: FP-STALKER: tracking browser fingerprint evolutions. In: S&P (2018)
54. W3C: Webassembly. https://webassembly.org/, accessed: 2022-05-20
55. WikiChip: Sunny cove - microarchitectures - intel - wikichip. https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove, accessed: 2022-05-20
56. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security Symposium (2014)
57. Yarom, Y., Genkin, D., Heninger, N.: Cachebleed: A timing attack on openssl constant time RSA. In: CHES (2016)
58. Yasin, A.: A top-down method for performance analysis and counters architecture. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2014)

A Training set

Table 1: CPUs used in our training set

CPU	Vendor	Generation
Xeon X5670	Intel	Westmere
Xeon E5-2620	Intel	Sandy Bridge
Xeon E5-2630	Intel	Sandy Bridge
Xeon E5-2630L	Intel	Sandy Bridge
Xeon E5-2650 0	Intel	Sandy Bridge
Xeon E5-2660 0	Intel	Sandy Bridge
Core i5-2520M	Intel	Sandy Bridge
Xeon E5-2660 v2	Intel	Ivy Bridge
Xeon E5-2630 v3	Intel	Haswell
Core i3-4160T	Intel	Haswell
Xeon E5-2620 v4	Intel	Broadwell
Xeon E5-2630 v4	Intel	Broadwell
Xeon E5-2680 v4	Intel	Broadwell
Core i3-5010U	Intel	Broadwell
Xeon Gold 6126	Intel	Skylake
Xeon Gold 6130	Intel	Skylake
Core i9-9980HK	Intel	Coffee Lake
Core i5-8365U	Intel	Whiskey Lake
Xeon Gold 5218	Intel	Cascade Lake SP
Xeon Gold 5220	Intel	Cascade Lake SP
Core i7-10510U	Intel	Comet Lake
Core i7-10710U	Intel	Comet Lake
Core i5-1135G7	Intel	Tiger Lake
EPYC 7301	AMD	Zen
Ryzen 5 2500U	AMD	Zen
Ryzen 9 5900HX	AMD	Zen 3