

# Robust and Scalable Process Isolation against Spectre in the Cloud (Extended Version)

Martin Schwarzl<sup>1</sup>, Pietro Borrello<sup>2</sup>, Andreas Kogler<sup>1</sup>, Kenton Varda<sup>3</sup>, Thomas Schuster<sup>1</sup>, Michael Schwarz<sup>4</sup>, and Daniel Gruss<sup>1</sup>

<sup>1</sup>Graz University of Technology, Austria      <sup>1</sup>Sapienza University of Rome, Italy  
<sup>3</sup>Cloudflare Inc.      <sup>4</sup>CISPA Helmholtz Center for Information Security, Germany

**Abstract.** In the quest for efficiency and performance, edge-computing providers replace process isolation with sandboxes, to support a high number of tenants per machine. While secure against software vulnerabilities, microarchitectural attacks can bypass these sandboxes.

In this paper, we present a Spectre attack leaking secrets from co-located tenants in edge computing. Our remote Spectre attack, using amplification techniques and a remote timing server, leaks 2 bit/min. This motivates our main contribution, *DyPrIs*, a scalable process-isolation mechanism that only isolates suspicious worker scripts following a lightweight detection mechanism. In the worst case, DyPrIs boils down to process isolation. Our proof-of-concept implementation augments real-world cloud infrastructure used in production at large scale, *Cloudflare Workers*. With a false-positive rate of only 0.61%, we demonstrate that DyPrIs outperforms strict process isolation while statistically maintaining its security guarantees, fully mitigating cross-tenant Spectre attacks.

## 1 Introduction

With the recent discovery of transient-execution attacks [7], such as Spectre [34] or Meltdown [37], attackers even leak data, not only meta-data. As most transient-execution attacks work across logical CPUs, *i.e.*, hyperthreads, many cloud providers do not assign logical CPUs to different tenants. With the introduction of edge computing [9, 2], where resources are dynamically provided on a machine that is close to the customer, virtualization-based security was replaced by more efficient solutions. Cloud providers either rely on strict process isolation [2, 42], *i.e.*, one process per tenant, or language-level isolation [9, 17, 16], *i.e.*, code is written in a sandboxed language such as JavaScript. While language-level isolation has the least overhead [10], it does not protect against Spectre within the same process [30, 41, 34, 56], necessitating process or site isolation [48]. To avoid these costly countermeasures, *Cloudflare Workers* rely on a modified JavaScript sandbox [9] that disables all known timers and primitives that can be abused to build timers [54, 22]. A similar design using language-level isolation WebAssembly is used by Fastly [17]. As *Cloudflare* is one of the top three edge computing providers, with millions of requests daily, this raises the following scientific question:

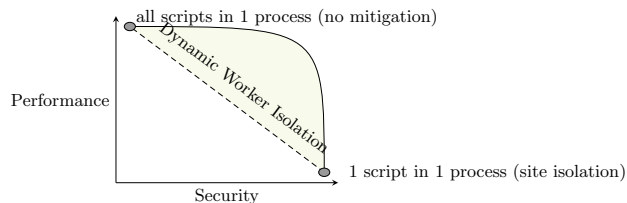


Fig. 1: Strict process isolation chooses the security and performance trade-off via the number of scripts inside one process (dashed line). DyPrIs improves this trade-off while never being worse than strict process isolation.

*Can edge computing without strict process isolation, as is already deployed and widely used today, offer the same security levels with respect to microarchitectural attacks as edge computing with strictly isolated processes?*

This paper has an offensive and a defensive contribution: First, we demonstrate that it is possible to steal secrets on *Cloudflare Workers* with 2 bit/min using an amplified Spectre attack [57] relying on an external time server. This proof-of-concept attack shows that language-level isolation is insufficient.

Second, we propose, *DyPrIs (Dynamic Process Isolation)*, a technique that relies on a probabilistic Spectre detection and process-isolates suspicious workloads. DyPrIs is a middle ground between the two extremes of strict process isolation and language-level isolation. Hence, DyPrIs keeps the performance benefits of language-level isolation for the majority of benign workloads and provides the security guarantees of process isolation against malicious workloads. Even if every workload was classified as Spectre, DyPrIs only boils down to strict process isolation with a the small overhead of 2% for the detection, but on average, it results in far higher performance (cf. Figure 1).

Our detection uses hardware performance counters (HPC) for mispredicted and retired branches. We show that HPC usage, as suggested in prior work [32, 46, 69, 43] has too much overhead for efficiency-driven edge systems. However, we demonstrate that even with a limited set of performance counters, we detect running Spectre attacks with a small performance overhead of 2%.

We evaluated DyPrIs in a production environment in the cloud. Our result is a false-positive rate of 0.61%, while detecting all attack attempts with all state-of-the-art techniques. DyPrIs blocks our attack without interrupting any of our own or other workloads.

**Contributions.** The main contributions of this work are:

1. We demonstrate a remote Spectre attack on the restricted *Cloudflare Workers*, showing that current mitigations are insufficient.
2. We propose a novel, low-overhead probabilistic detection for Spectre attacks.
3. We introduce DyPrIs, a technique with, on average, lower overhead than state-of-the-art strict process isolation.

## 2 Background and Related Work

In modern processors, instructions are divided into multiple micro-operations ( $\mu$ OPs) that are executed out of order. To improve the performance of branch instructions, CPUs leverage speculative execution. For example, the branch prediction unit (BPU) tries to predict whether a branch is taken or not using different data structures, e.g., the Pattern History Table (PHT) [34]. If the prediction was correct, the results of the execution are retired. Otherwise, the speculatively executed instructions are discarded, and the correct code path is executed. Mistakenly executed instructions are called *transient instructions* [37, 7]. They still have an effect on the microarchitecture, e.g., measurable timing differences in the cache that can be extracted with cache attacks [37, 7, 34]. Cache attacks are even possible in JavaScript [44].

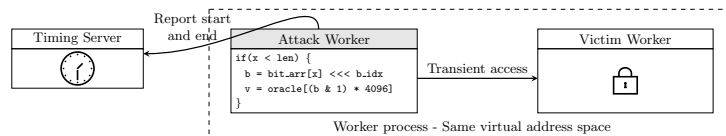
Spectre attacks [34] exploit speculative execution. Spectre-PHT [7] (also known as Spectre V1) exploits the Pattern History Table, which predicts the outcome of a conditional branch [34]. A typical Spectre-PHT gadget is a bounds check, e.g., `if (x < array1_size) y = array2[array1[x] * 4096];`. The attacker controls the index `x`, which is bounds-checked. By mistraining the branch prediction with in-bounds values, speculation follows the in-bounds path with out-of-bounds values, allowing out-of-bounds reads. Spectre variants exploit different prediction mechanisms, e.g., the Branch Target Buffer, memory disambiguation, or the Return-Stack Buffer [34, 29, 35, 38] and have been demonstrated over the network [55] and in JavaScript [34, 41, 56].

Many cache side-channel defenses have been proposed, e.g., focusing on *detection* using HPCs [32, 46, 69, 8, 28, 65, 66, 70]. To detect Spectre-type attacks, static code analysis and patching, taint tracking, symbolic execution, and detection via HPCs were proposed [13, 27, 64, 25, 43, 26, 40]. However, these proposals focus on attack detection but do not propose and evaluate mechanisms to respond to detected attacks. Detection methods suffer from false positives but terminating a detected attack is not acceptable for *Cloudflare Workers*.

*Cloudflare Workers* is an edge computing service to intercept web requests and modify their content using JavaScript, handling millions of HTTP requests per second across tens of thousands of web sites. *Cloudflare Workers* support multiple thousand workers from up to 2000 tenants running inside the same process. Each worker is single-threaded and stateless. This design leads to a high-performant solution based on language-level isolation. To impede microarchitectural attacks, *Cloudflare Workers* restricts the available JavaScript timing functions to only update after a request is performed. Additionally, JavaScript worker threads are disabled to prevent counting threads [54, 22, 34].

## 3 Remote Spectre Attacks on *Cloudflare Workers*

In this section, we show that the single-address-space design of *Cloudflare Workers* enables remote Spectre attacks. First, we define the Spectre building blocks and overview how a remote adversary can mount a Spectre attack. Since there

Fig. 2: Overview of the *Cloudflare Workers* remote Spectre attack.

is no local timing primitive, a common requirement for microarchitectural attacks [18, 53], we have to resort to a remote timing primitive. Our proof-of-concept implementation running on *Cloudflare Workers* leaks 2 bit/min, even if address space layout randomization (ASLR) is active.

### 3.1 Threat Model & Attack Overview

In our threat model, the attacker can run *Cloudflare Workers* executing JavaScript code but no native code. Furthermore, the attacker controls a remote server to record high-resolution timestamps, e.g., using `rdtsc`, and a low-latency network connection. We also assume a powerful attacker with a worker co-located with the victim worker, e.g., by spawning multiple *Cloudflare Workers* and detecting co-location. An attacker spawning its instances close in time to the victim’s one can maximize the probability of co-location [49]. *Cloudflare Workers* architecture aims to serve the same application from every location. A high number of tenants per machine is possible. Physical co-location of the attacker server is not required. However, this leads to the strongest possible attacker. We assume no exploitable software bugs, e.g., memory safety violations, in the JavaScript engine and no sandbox escapes. Thus, *architectural* exploits to leak data from other tenants or processes are not possible.

The typical requirements for state-of-the-art Spectre attacks on the timer and memory are listed in Table 1, showing the differences to our attack. Figure 2 provides an overview of our attack. In the *Cloudflare Workers* setup, each worker runs in the same process, and thus, shares the virtual address space. The attacker runs a malicious JavaScript file containing a self-crafted Spectre-PHT gadget that performs a Spectre attack on its own process. As the victim and attacker share the same process, the attacker can leak sensitive data from a victim worker, without having an existing Spectre gadget in the victim.

Spectre attacks in JavaScript rely on speculative out-of-bounds accesses of objects. Assuming the attacker can either trigger a victim worker’s secret allocation, delay it, or just manages to execute before the victim, we can use heap-grooming techniques [21] to bring the process memory into a predictable state before both the leaking object and the victim data are allocated. Alternatively, the attacker worker can predict the offset between the leaking object and the victim worker’s data, target a certain range of the virtual memory, e.g., regions where V8 places similar objects [60], or break ASLR using speculative probing [19]. Hence, ASLR does not mitigate the attack. Furthermore, Agar-

Table 1: Requirements and leakage rate of Spectre attacks.

Spectre attack (variant)	Gadget	Native	HR Timer	Memory	Leakage	Rate	Error	Channel
Kocher et al. [34] (PHT)	Yes	Yes	Yes (ns)	2.40 MB	4420.46 B/s $\pm$	6.75 %	0.07 %	Cache-L3
Canella et al. [7] (PHT)	Yes	Yes	Yes (ns)	3.54 MB	3.13 B/s $\pm$	113.79 %	0.00 %	Cache-L3
Safeside [20] (PHT)	Yes	Yes	Yes (ns)	7.00 MB	4384.03 B/s $\pm$	7.75 %	0.00 %	Cache-L3
Canella et al. [7] (BTB)	Yes	Yes	Yes (ns)	6.91 MB	0.71 B/s $\pm$	2.43 %	0.00 %	Cache-L3
SafeSide [20] (BTB)	Yes	Yes	Yes (ns)	7.01 MB	269.53 B/s $\pm$	0.85 %	0.00 %	Cache-L3
Canella et al. [7] (STL)	Yes	Yes	Yes (ns)	3.54 MB	14.37 B/s $\pm$	211.95 %	0.00 %	Cache-L3
Safeside [20] (STL)	Yes	Yes	Yes (ns)	7.00 MB	272.46 B/s $\pm$	0.22 %	0.00 %	Cache-L3
Canella et al. [7] (RSB)	Yes	Yes	Yes (ns)	20.08 MB	30.67 B/s $\pm$	195.59 %	0.00 %	Cache-L3
Safeside [20] (RSB)	Yes	Yes	Yes (ns)	7.00 MB	116.70 B/s $\pm$	0.58 %	0.00 %	Cache-L3
Google [56] (PHT)	No	No	Yes ( $\mu$ s)	15.00 MB	335.02 B/s $\pm$	23.50 %	0.26 %	Cache-L1
Google [56] (PHT)	No	No	Yes (ms)	15.00 MB	9.46 B/s $\pm$	31.40 %	2.71 %	Cache-L1
Agarwal et al. [1] (PHT)	No	No	Yes ( $\mu$ s)	N/A	533.00 B/s $\pm$	N/A	0.32 %	Cache-L3
Schwarz et al. [55] (PHT)	Yes	Yes	No	N/A	7.50 B/h $\pm$	N/A	0.58 %	AVX unit
<b>Our work</b> (PHT)	<b>No</b>	<b>No</b>	<b>No</b>	27.54 MB	15.00 B/h $\pm$	2.67 %	0.00 %	Cache-L3

Gadget: Spectre gadget must be in victim; Native: native code execution; HR Timer: High-resolution timer

wal [1] demonstrated that it is possible to leak over the full address space using a JavaScript Spectre attack in the V8 engine.

For our attack, we rely on a Spectre-PHT [34] gadget, as this is the simplest gadget to introduce in JIT-compiled code. Moreover, Spectre-BTB [34] can be prevented by the JIT compiler [58]. In contrast to the original Spectre attack [34], we do not encode the data bitwise but bit-wise. The advantage of such a *binary Spectre gadget* is that it is easier to distinguish two states compared to 256 states using a side channel [4, 55]. While such a gadget might not be commonly *found* in real applications, it is easy to *introduce*.

As there are no high-resolution timers to distinguish microarchitectural states directly, we have to amplify the timing difference between a cache hit and a miss, *i.e.*, between a leaked ‘0’ and ‘1’ bit. We combine the amplification techniques by McIlroy et al. [41] with the remote measurement methods by Schwarz et al. [55]. With this semi-remote Spectre attack, we show that it is indeed feasible to leak data from co-located *Cloudflare Workers* in such a restricted setting. Our Spectre attack is the only one not requiring native code execution, a local timer, or an existing gadget. Moreover, microcode cannot prevent it (cf. Table 1).

### 3.2 Building Blocks

As our attack uses the cache as the covert-channel part of the Spectre attack, we require building blocks for measuring the timing of cache accesses in JavaScript. While this can be done using a high-resolution timer in some browsers [34], the required primitives are not available on *Cloudflare Workers*. Hence, in addition to a different timing primitive with a lower resolution, we have to amplify the signal such that we can reliably distinguish ‘0’ and ‘1’ bits.

**Remote Timer** On *Cloudflare Workers*, there are no local timers or known primitives to build timers [54]. We verified that, indeed, no technique from Schwarz et al. [54] resulted in a timer with a resolution higher than 100 ms. Thus, there is no possibility to accurately measure the time directly in JavaScript, and, therefore, it is not possible to perform a local Spectre attack [34].

```

if (secret_bit) { read A; } else { read B; } //transiently leak bit
read A; //perform architectural access

```

Listing 1.1: Amplified Spectre-PHT gadget [41].

In this setup, the attacker sends a network request to a remote server to start a timing measurement. The remote server stores a local high-resolution timestamp, e.g., using `rdtsc`, associated with the request. To stop the timing measurement and receive the time delta, the attacker sends another request to the remote server, which sends back the time difference from the current to the stored timestamp. Hence, the attacker has a high-resolution time difference that is only impacted by the network latency between the attacker’s worker and the remote server. We evaluated this timing primitive on *Cloudflare Workers*. For the best case, *i.e.*, same physical machine, we achieve a resolution of 0.47 ns on a 2.1GHz CPU, with a jitter of 1.67%. With a resolution of 0.47 ns, we can distinguish a cache hit from a miss for the cache covert channel. However, this case is unlikely in reality, as the latency is typically in the microsecond range [62].

**Amplification** In our attack scenario, the attacker has no high-resolution timer but full control over the Spectre gadget. Hence, to mount a successful attack with the remote timer, we have to rely on amplification techniques that amplify the latency between a cache hit and miss [41]. One such technique is to transiently access multiple cache lines for a single bit instead of a single cache line and probe over these to increase the latency between a cache hit and a miss. However, this technique is quite memory-consuming and limited by the number of cache lines.

A way to arbitrarily amplify the latency between cache hits and misses is to either access a memory location which encodes a ‘0’ or ‘1’ bit transiently and then accesses the memory location for a ‘1’ again architecturally [57]. Listing 1.1 illustrates an *arbitrary amplification* [57] gadget. If the Spectre gadget is optimal in terms of mistraining, we have twice as many cache misses for a ‘0’ bit as for a ‘1’ bit. With a loop over the gadget, we can create arbitrarily large timing differences between hits and misses. We evaluate the amplification idea on an Intel Xeon Silver 4208, running Ubuntu 20.04 (kernel 5.4.0) in native code. We increase the number of amplification iterations and run each iteration 1000 times to get stable results. This leads to a linear growth with the increase of the number of loop iterations (amplification factor). Depending on how much runtime is given to the worker, it is possible to arbitrarily increase the delay. Hence, we can also see that there are no strict requirements for the resolution of the remote timer. For lower resolutions, we can increase the amplification, resulting in a reduced leakage rate, no prevention of the attack, as also shown in related work [56].

**Eviction** To repeat our amplification and reset the cache state, cache eviction is required. One way to evict certain addresses from the cache is by building eviction sets [44, 23, 63]. While a targeted eviction set leads to a fast eviction, building the eviction set is costly. Even with a local timer, the currently fastest approach takes more than 100 ms [63]. In our remote scenario, this would require a lot of network requests to find the eviction set for our encoding oracle, as building the eviction set requires constant timing measurements. Furthermore,

eviction sets cannot be reused due to address-space-layout randomization on each run. Instead of using eviction sets, we iterate over a large eviction array (multiple MB, depending on the cache size) in cache-line steps (64 byte) and access the values. If enough addresses are accessed, the cached value is evicted [23, 34].

We evaluate the eviction directly on the V8 engine used in *Cloudflare Workers* on an Intel Xeon Silver 4208, running Ubuntu 20.04 (kernel 5.4.0). We access a certain index  $v$  of a large array to cache it, iterate over the eviction set, and verify if  $v$  is still cached. We observe that an eviction array of 2 MB always evicts  $v$  on our Intel Xeon Silver 4208 ( $n = 1000$ ).

Note that address randomization can be deterministically circumvented using engineering. Göktas et al. [19] introduced the concept of a speculative probing primitive that leverages Spectre to break classical and fine-grained ASLR. Gras et al. [22], Schwarz et al. [51], and Lipp et al. [36] demonstrated that microarchitectural attacks in JavaScript can break memory randomization.

### 3.3 Attack on *Cloudflare Workers*

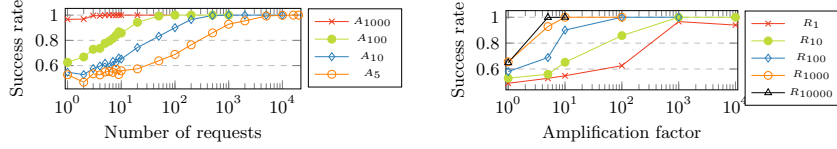
Using the building blocks, we mount an attack on *Cloudflare Workers* to extract secret bits from a worker at a known location to estimate the best possible attack. For that, we send an initial request with a sequence number to a timing server. The timing server stores a local, high-resolution timestamp on this request. We perform a Spectre attack on a target address and send another request to the server. The timing server computes the delta between the current and the stored timestamp to distinguish between a cache hit or miss. As the attacker controls both the attacking worker and the timing server, there is no need to send the leaked information back to the worker.

There are different challenges when creating a JavaScript Spectre PoC, as the V8 JIT compiler optimizes code based on assumptions. If such assumptions are invalidated, the function is de-optimized. We thus avoid triggering any de-optimization points in our generated code, as that ruins the training achieved. Therefore, we place the out-of-bound access behind a mispredicted guard branch, preventing the JIT compiler from de-optimizing the code when detecting out-of-bound accesses. Moreover, during the garbage collection phase, objects move between different heap spaces of the same worker to reduce the memory footprint. By forcing garbage collection phases, we stabilize an object’s location.

**Evaluation.** To develop and evaluate a proof-of-concept attack, we obtained a local developer copy of *Cloudflare Workers* to not interfere with any worker of other customers. We ensured that the configuration on our local system is identical to the configuration running on the cloud. As *Cloudflare Workers* mostly use server CPUs, we also focus our attack on an Intel server CPU, specifically an Intel Xeon Silver 4208, running Ubuntu 20.04 (kernel 5.4.0).

We create a Spectre-PHT PoC that leaks bits from a victim `ArrayBuffer` by transiently reading out-of-bounds. We describe the technical implementation details for optimal leakage in Appendix B.

We call the function performing a Spectre attack 10 000 times and repeat the experiment 1000 times, observing a success rate of 54.31% ( $n = 1000$ ,  $\sigma =$



(a) Success over the number of requests (b) Success of different amplification factors for different number of requests.

23.16 %). We assume that the attacker is capable of creating a stable exploit with 100 % success rate. From now on, we evaluate our metrics with a 100 % success rate to estimate the best possible attack, where the attacker knows where the secret array is located.

We evaluate a set of different amplification factors (number of loop iterations) in native code between 1 and 1000, and sample each loop length 100 000. We implement the box test [14] to determine the number of required requests [62, 6, 14]. Figure 3a illustrates the number of requests required to achieve a certain success rate for different amplification factors. The higher the amplification factor is, the fewer requests are required to achieve high success rates. As Figure 3b illustrates, with small amplification factors but enough requests, we can also achieve a high success rate of more than 95 %. We refer to the work of Van Goethem et al. [62] and Schwarz et al. [55] for the required requests in a network with multiple hops.

We evaluate our attack locally, *i.e.*, with a timing server on the same machine. We first evaluate an optimal attack in native code. Ideally, an attacker chooses the number with the highest success rate and the lowest number of requests required, minimizing the execution time. We choose a random 16-bit secret. As amplification factor, we choose 100 000 loop iterations and perform just one request. With this setup, leaking one bit takes on average 2.5 s ( $n = 100$ ,  $\sigma_{\bar{x}} = 0.05\%$ ). We repeat the experiment 100 times and observe a leakage rate of 23 bit/s ( $n = 100$ ,  $\sigma_{\bar{x}} = 2.8\%$ ). Using an outlier filter, this error can be reduced towards 0. As these values are from a native-code attack, we consider these numbers as the maximum achievable leakage rate for JavaScript. A JavaScript attacker is more restricted in terms of evicting certain addresses from the cache and thus requires additional time for the eviction. Furthermore, the code is JIT-compiled, requiring a warmup to stabilize the JIT-compiled code. We evaluate the amplification in JavaScript in the V8 engine with an amplification factor of 250 000, a native timestamp counter to measure the response times, and a random 16 bit secret. One script execution takes about 30 s, which is the maximum execution time for *Cloudflare Workers* [12]. All evaluated numbers are shown in Table 2 (Appendix A). With a success rate of 100 % we determine an optimal leakage rate of 2 bit/min leading to a leakage rate of 120 bit/h.

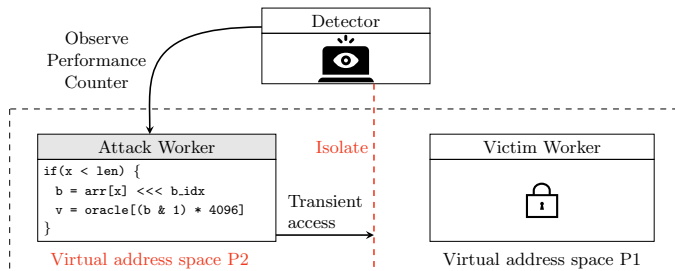


Fig. 4: DyPrIs isolating a malicious worker based on performance counters.

## 4 DyPrIs

In this section, we present an approach to dynamically isolate malicious *Cloudflare Workers* to benefit both from the security of process isolation and the performance of language-level isolation. The basic idea is to use HPCs to detect potential Spectre attacks and isolate suspicious *Cloudflare Workers* using process isolation (Figure 4). While a detection mechanism typically suffers from false positives, DyPrIs can cope even with high false-positive rates. In the worst case, a Spectre attack is detected for every worker, leading to the worst-case scenario of one worker per process, *i.e.*, strict process isolation, as currently used in browsers plus the 2% detection overhead. As workers are stateless, they can also be suspended or migrated at any time. Thus, even if many worker are considered malicious, the resources of *Cloudflare* are not exhausted. Every false-positive rate below 100% performs better than strict process isolation.

We discuss how to reliably detect Spectre attacks using performance counters (cf. Section 4.1). We integrate our approach into *Cloudflare Workers* and measure the performance overhead of reading performance counters on a real-world cloud system (cf. Section 4.2). We show that there is a small performance overhead of 2% for reading performance counters.

### 4.1 Detecting Spectre Attacks

In this section, we discuss the detection of Spectre attacks using HPCs. While the common use of HPCs is finding bottlenecks, researchers used HPCs for detecting malware, rootkits, CFI violations, ROP, Rowhammer, or cache-side channel attacks [67, 68, 39, 72, 28, 5, 24, 8].

**Detecting Attacks using Normalized Performance Counters** Our second approach tries to detect Spectre attacks using normalized performance counters. At first we collect data from different performance counters. We collect the following hardware events (PERF\_COUNT\_HW\_\*): CACHE iTLB, BRANCH\_MISSES, BRANCH\_INSTRUCTIONS, CACHE\_REFERENCES, CACHE\_MISSES, CACHE\_L1D/READ\_MISSES and CACHE\_L1D/READ\_ACCESSES. We normalize the values using iTLB performance counters (iTTLB accesses) which was also used by

Gruss et al. [24] to detect Rowhammer and cache attacks. Similarly to Rowhammer and cache attacks, the main attack code for Spectre has a small code footprint with a high activity in the branch-prediction unit.

The iTLB counter normalizes the branch-prediction events with respect to the code size by dividing the performance counter value by the number of iTLB accesses. We integrate the monitor into *Cloudflare Workers*, to read the performance counters before and after each script execution. The averaged per-execution numbers are updated in a 1-second interval (Note that a single script runs up to 30 s [12]). While reducing the interval does not directly impact the performance of a worker, it potentially leads to more false positives as outliers are not filtered. We collect data from the benign workload and compare it to a worker executing a Spectre attack. Based on the performance numbers, we find a threshold to distinguish between an attack and normal workload. We evaluate this approach in Section 5.1.

## 4.2 Process Isolation

For DyPrIs, we fundamentally rely on process isolation. A well-known implementation of process isolation is site isolation, where every page in a browser runs in its own process to prevent memory safety violations as well as Spectre attacks [48]. However, in contrast to full site isolation, we only isolate potentially malicious *Cloudflare Workers* if the Spectre detection mechanism flags them. Hence, DyPrIs only falls back to full site isolation in the worst case, while reducing the overhead caused by process isolation in the average case.

Related work proposes efficient in-process isolation mechanisms using Intel Memory Protection Keys (MPK) [61, 45, 50]. However, Intel MPK is only available on selected CPUs since Skylake-SP, limited to 16 protection keys and thus not practical for *Cloudflare Workers* [61], running multiple thousand workers per process. Furthermore, the threat model of these approaches does not include side-channel or transient-execution attacks. For DyPrIs, we modify the *Cloudflare Workers* software to isolate a potentially malicious worker, *i.e.*, a worker that was flagged by the performance-counter-based detection, into a separate process. We implement process isolation in *Cloudflare Workers* from scratch (cf. Figure 5). For that, we start process sandboxes by forking from a zygote process, and talk to the new process over an RPC protocol [3, 11]. All communication between the main process and the isolated process are over this RPC connection, communications between the process sandbox and the outside have to go through the main process. Since the runtime of a worker is, on average, less than 1 ms, the isolation must not introduce a high performance overhead. Thus, one instance of a worker frequently reads out the performance counters per script execution and computes a moving average. From our results in Section 5, we observed that the normalized iTLB performance provides the best detection tradeoff in terms of performance overhead and accuracy. We first run an attack and collect its performance-counter data. Additionally, we collect anonymized per-CPU-core performance-counter data of real scripts running in production. Based on our evaluation in Section 5.1, we use a threshold of 4096 retired branches per iTLB

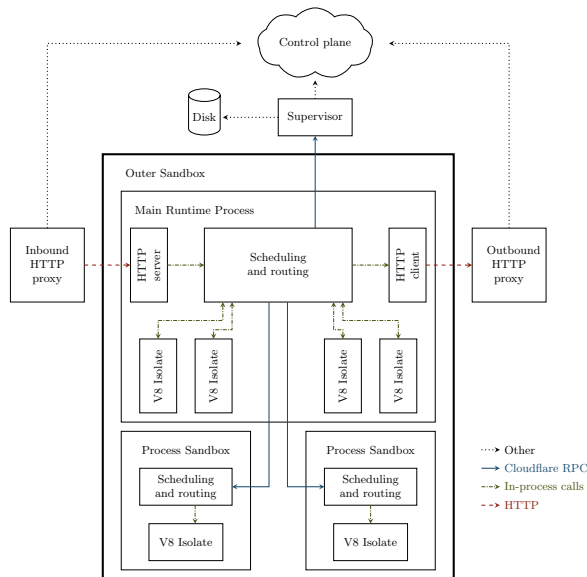


Fig. 5: DyPrIs overview.

access to distinguish between a suspicious and a benign script. If a script exceeds this threshold, we flag it as a potential Spectre attack and isolate it into a separate process. In contrast to, e.g., browser tabs, worker are stateless. Thus, a worker can simply be migrated. Isolating instead of terminating ensures that the worker can continue running, e.g., in case the detection was a false positive, while it cannot access data of any other worker.

## 5 Evaluation

In this section, we evaluate the accuracy and performance overhead of our detection methodology. We choose a threshold of 4096 branch accesses per iTLB access, which allows distinguishing a Spectre attack from a benign script. We use a large set of different programs to sample the number of mispredicted and retired branches. For our set, we observe that out of 141 programs, which includes the 13 Spectre gadgets from Kocher [33], we cannot distinguish 4 benign programs from a Spectre gadget, resulting in a false-positive rate of 2.83% with a small performance overhead of 2%. Using our normalized counters approach, we observe a negligible overhead of 2% in our production environment.

### 5.1 Normalized Performance Counters

We evaluated our approach on 5 Intel Xeon server CPUs (Broadwell, Skylake 4116, Skylake 6162, Skylake 6162, Cascade Lake 6262) and one AMD Epyc Rome

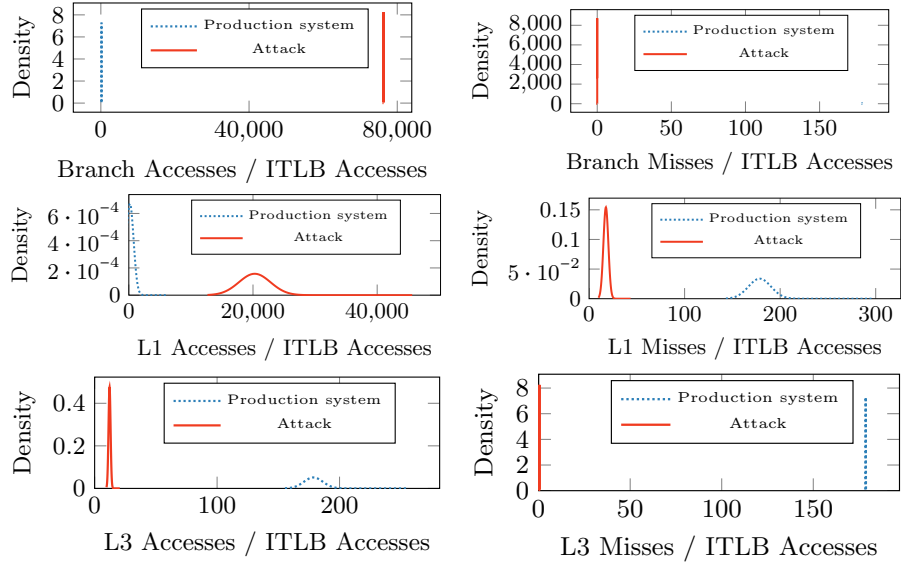


Fig. 6: Performance counters of average *Cloudflare Workers* and a Spectre attack on the production system.

CPU. To decide whether a script is susceptible or not, we collect performance data from the production system running our Spectre attack. We recorded the performance counters on the production environment and sampled over 50 000 times as a baseline. Figure 6 shows the normalized performance counters of our cloud machines. For last-level-cache accesses, misses, and branch misses, the numbers of the attack script are below the average script. For the number of L1-cache accesses and retired branches, we can clearly distinguish average script from attack. Especially for the retired branches, the distance between an attack script and the average regular script is 34 times the standard deviation of a benign script. We collected our numbers from real-world worker production machines to calculate the false-positive rate. We choose the number of normalized retired branch instructions as an indicator for a Spectre attack and run it on our cloud machines. First, we run a Spectre attack to verify whether their number is in a similar range on each test machine. We then evaluate different threshold boundaries for the number of normalized retired branch instructions and report the number of false positives. Figure 7 shows the number of false positives depending on the threshold on our cloud machines in the production environment. For a strict threshold, *i.e.*, 1024, the false-positive rate is 21.41%. However, this threshold is set higher to reduce the number of false positives. The numbers of false positives are in a similar range on each of the tested machines. Setting the threshold to 4096, results in an average false positive rate of 0.61% on our devices. For a threshold of 8192 the average false-positive rate decreases to 0.26%, and at a threshold of 65 536, we do not observe any false positives.

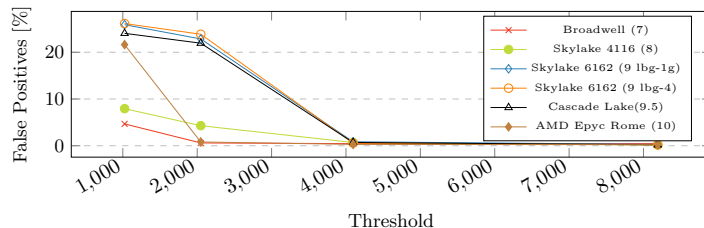


Fig. 7: Number of false positives depending on the normalized iTLB threshold.

Next, we look at the performance overhead of our attack when the attacker tries to get below the detection thresholds. Getting below this threshold requires the attacker to significantly slow down the amplified Spectre attack. Since the attacker cannot get rid of the cache eviction, the number of amplification iterations has to be reduced. Consequently, if the number of amplification iterations is reduced, more requests, *i.e.*, samples, are required to clearly distinguish cache hits and misses (cf. Figures 3a and 3b). We evaluate the best possible attacker in native code who only mistrains one branch. By omitting amplification or with a small factor of 10, we can reduce the number of retired branch instructions / iTLB accesses on our test devices to 604.71 and 3492.41, respectively, which is in the ranges of an average script. However, with the latter, the leakage rate is 1 bit/h. Thus, we set the threshold to 4096 and receive an average false positive rate of 0.61% on our tested devices. Figure 8 illustrates the decrease in leakage if the attack degrades from an amplified Spectre attack to a sequential attack. Using a non-amplified approach, about 250 000 requests are required (Section 3.3). We achieve a leakage of 1 bit/h in a local-network scenario. Hence, as an additional security margin, we limit the number of subsequent requests per worker to 10 000 on the same machine. If more than 10 000 requests are issued, we redirect the request to a different machine. Thus, we can still prevent leakage from a slowed-down attack using our threshold-based approach. We assume that there are no attacks running on the production system, thus we cannot measure the number of false negatives. Our own attack is detected by the threshold, as well as the 15 Spectre samples provided by Kocher [33]. In addition, we evaluated and analyzed the new and larger Spectre-PHT gadgets generated by FastSpec [59]. The gadgets are based on the the 15 variants, and we observe that the generated gadgets are quite similar. We evaluated 100 random gadgets from FastSpec and did not observe any false negatives with our detection. As the mistraining for those gadgets is similar, the branch accesses per iTLB access are in a similar range. We also evaluated the detection on the Spectre JavaScript PoC from Röttger and Janc [56]. Even with the low amplification factor of 4000 in this PoC, we reliably detect the attack ( $n = 500$ ,  $\mu = 19\,253.73$ ).

*Spectre-BTB, Spectre-RSB and Spectre-STL.* In addition to Spectre-PHT we also run our performance counter analysis on other Spectre variants exploiting the branch-target buffer (BTB), return-stack buffer (RSB) and store-to-load

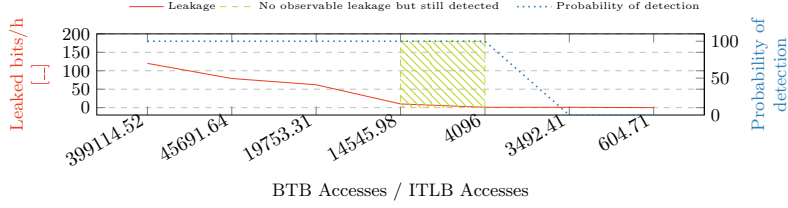


Fig. 8: Branch accesses / iTLB accesses and the corresponding leakage rate.

(STL) forwarding. We create native code proof-of-concepts for these variants executing each gadget 10 000 times on a Xeon Silver 4208. We ran the PoCs 500 times and collected the number of branch and iTLB accesses. The numbers for Spectre-BTB and RSP are an order of magnitude lower than for Spectre-PHT ( $\mu_{btb} = 423171.54$ ). However, they are still detected with the same metric ( $n = 500$ ): Spectre-BTB ( $\mu_{btb} = 23401.20$ ), Spectre-RSB ( $\mu_{rsb} = 38369.17$ ), Spectre-STL ( $\mu_{stl} = 982.20$ ). The metric for Spectre-STL is far below the threshold of 4096. However, the values for `memory_disambiguation.history_reset` are significantly higher on average if the store-to-load logic is exploited in Spectre-STL ( $n = 500$ ,  $\mu_{stl} = 8993.98$ ,  $\mu_{nostl} = 2644.73$ ). Thus, we also use this counter to detect potential Spectre-STL attacks.

## 5.2 DyPrIs

We integrate DyPrIs in *Cloudflare Workers*, which requires modifications of 6459 lines of code, not including the Spectre detection mechanism. As with any isolation technology, the performance overhead varies depending on the workload [48]. *Cloudflare Workers* is an environment where typical guest workloads use very little memory and spend very little CPU time responding to any particular event. As a result, in this environment, DyPrIs’s overhead is expected to be large compared to the underlying workload. In a first test, we evaluate the overhead for a test script by increasing the number of isolated processes, *i.e.*, the number of sandboxed V8 isolates, up to 500. We measure the overhead in terms of executed scripts per second, *i.e.*, the requests executed per second from the localhost and the total amount of consumed main memory. The execution is repeated 10 times per isolation level with 2000 requests ( $n = 20000$ ,  $\sigma_{rps} = 3.87\%$ ,  $\sigma_{mem} = 0.23\%$ ). Figure 9 shows the requests per second and the total memory consumption based on the number of isolated V8 processes. As expected, we observe a linear decrease in the possible number of requests per second and a linear increase in the memory consumption. Further, we performed a load test of *Cloudflare Workers* runtime using a selection of sample guest workers simulating a heavy-load machine. They mostly respond to I/O in under a millisecond and allocate little memory. By forcing process isolation on the workers, the memory overhead of each guest was 2x-5x higher, and CPU time was 8x higher, compared to a worker using a single process. We performed a second test using a real-world

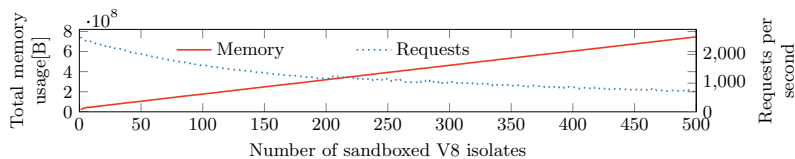


Fig. 9: Requests per second and memory consumption of process isolation.

worker known to be unusually resource hungry in both CPU and memory usage. In this case, memory overhead is 20%-70% worse with DyPrIs, and CPU time about 60% worse. These numbers appear to be high, but when only 0.61% of workers are isolated, the overhead is negligible. As our proof of concept was not optimized, it still has big potential for optimizations. For example, it currently uses an RPC protocol [3] to communicate between processes, but does so over a Unix domain socket. This protocol is designed in such a way that it could be communicated in shared memory, reducing communication overhead. The implementation could also use OS primitives for faster context switching, such as the FUTEX\_SWAP feature proposed by Google. However, while especially the CPU overhead could be reduced, there is always a significant cost incurred by context switching and marshalling to communicate between processes. The total overhead on all machines can only be estimated as it depends on the workload. The detection overhead is 2%. In the worst case, we are slightly worse than full process isolation due to the 2% detection overhead.

## 6 Discussion

### Comparison between *Cloudflare Workers* and competing approaches.

The main challenge of edge computing is to run various applications of numerous tenants efficiently. Approaches like AWS Lambda and Azure Functions rely on containers to achieve this [2, 42]. While their design strictly prevents Spectre attacks on other tenants, the performance overhead is higher for the use case of edge computing than *Cloudflare Workers* [10]. The *Cloudflare Workers* architecture is stateless in a sense that every worker in any data centre can process any request, *i.e.*, the request is processed by the worker with the lowest latency. *Cloudflare Workers* rely on a single-process architecture with language-level isolation to isolate their tenants architecturally. However, as we showed, this design leads to potential Spectre attacks. A similar design with language-level isolation of WebAssembly code from different tenants is used by Fastly [17]. Therefore, Fastly also needs to consider Spectre attacks within the same process by either applying DyPrIs or switching to full isolation via processes or containers.

**Mitigation versus Detection.** Especially in high-performance scenarios, such as cloud systems, Spectre mitigations [7, 34, 31] result in high power consumption. Hence, instead of paying the constant costs of mitigations, detecting attack can reduce the costs. However, the problem of detecting side-channel and

transient-execution attacks is still an open research problem. There is no universal solution that covers all different types of attacks.

**False Positives and Negatives.** DyPrIs suffers from false positives and false negatives [15, 71], similar to other detection and mitigation techniques [64, 25]. False positives only impact the performance and not the security. False negatives occur when slowing down attacks to 1 bit/h (cf. Table 2 in Appendix A). Therefore, the maximum execution time is restricted to 30 s, far from 1 h. Using the machine learning approach of Gulmezoglu [26], the false positive rate could be reduced further. However, this approach would require a re-training with real-world data of *Cloudflare Workers* and a frequent re-updating of the training set. Adding additional code pages also allows getting below the thresholds. To “hide” the native attack, we access 125 additional code pages (500 kB) per bit to get the branch accesses / iTLB accesses below the threshold (Figure 10 in Appendix A). While feasible in native code, the resulting code size causes V8 to abort the optimization phase, stopping the attack.

**Comparison to existing detections** Besides full site isolation, prior work discusses detection but not how to stop attacks once they are detected. Existing static analysis approaches [27, 13] on binaries are not applicable to the use case of *Cloudflare Workers*. Approaches that perform taint tracking and fuzzing on binaries to dynamically detect gadgets [25, 64, 52, 47] are infeasible for the high-performance requirements of *Cloudflare Workers*. The approach of Mambretti et al. [40] does not evaluate real-world workloads and cannot distinguish the different workloads of *Cloudflare Workers* from an external process.

**Reliability of HPCs In DyPrIs** As Zhou et al. [71] and Das et al. [15] discuss, using HPCs for detection of cache attacks can lead to flaws caused by non-determinism and overcounting. We showed that in our statistical approach both only marginally reduce the performance of DyPrIs not the security.

**Alternative Spectre JS attacks** Concurrent work [56] has demonstrated a Spectre exploit on V8, leaking up to 60 B/s using timers with a precision of 1 ms or worse through a L1 covert channel. Similarly to our PoC, it uses a Spectre-PHT gadget to read out-of-bound from a JavaScript TypedArray, giving an attacker access to the entire address space. The PoC uses small-sized TypedArrays for which the backing store is allocated in the isolate itself. Thus, it leaks data inside the same isolate. In concurrent work, Agarwal et al. [1] has extended the PoC from Röttger and Janc [56] to leak data using 64-bit addresses using a local timing source. They use speculative type confusion between an ArrayBuffer and a custom object that should be properly aligned across two cache lines.

## 7 Conclusion

In this paper, we presented DyPrIs, a practical low-overhead solution to actively detect and mitigate Spectre attacks. We first presented an amplified JavaScript remote attack on *Cloudflare Workers*, which leaks 2 bit/min, *i.e.*, 1 bit per worker invocation. We proposed a practical approach for actively detecting and mitigat-

ing Spectre attacks. We show that it is still possible to efficiently detect Spectre attacks using performance counters with a false-positive rate of 0.61 % at the cost of 2 % overhead for the detection. We demonstrate that conditionally applying process isolation based on a detection mechanism has a better performance than full process isolation, under the same security guarantees.

## Acknowledgments

We want to thank our anonymous reviewers and in particular our shepherds Roberto Di Pietro and Vijayalakshmi Atluri. This work was supported by generous gifts from Cloudflare. We want to especially thank Harris Hancock, Claudio Canella and Moritz Lipp for valuable feedback on this work. Any opinions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the funding parties.

## References

1. Ayush Agarwal, Sioli O’Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution. In *S&P*, 2022.
2. Amazon. AWS Lambda@Edge, 2019. URL: <https://aws.amazon.com/lambda/edge/>.
3. Anonymous. Anonymized for Double Blind Submission, 2019.
4. Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandt ner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMOtherSpectre: exploiting speculative execution through port contention. In *CCS*, 2019.
5. Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *CODASPY*, 2018.
6. David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
7. Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019. Extended classification tree and PoCs at <https://transient.fail/>.
8. Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.
9. Cloudflare. Cloudflare Workers, 2019. URL: <https://www.cloudflare.com/products/cloudflare-workers/>.
10. Cloudflare. Cloudflare Workers, 2019. URL: <https://blog.cloudflare.com/cloud-computing-without-containers/>.
11. Cloudflare. Anonymized for Double Blind Submission, 2020.
12. Cloudflare. Limits - Cloudflare Workers, 2021. URL: <https://developers.cloudflare.com/workers/platform/limits>.
13. Jonathan Corbet. Finding Spectre vulnerabilities with smatch, April 2018. URL: <https://lwn.net/Articles/752408/>.

14. Scott A Crosby, Dan S Wallach, and Rudolf H Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):17, 2009.
15. Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *S&P*, 2019.
16. Deno. A Globally Distributed JavaScript VM, 2021. URL: <https://deno.com/deploy>.
17. Fastly. Serverless Compute Environment - Fastly Compute@Edge, 2021. URL: <https://www.fastly.com/products/edge-compute/serverless>.
18. Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 2016.
19. Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, 2020.
20. Google. SafeSide: Understand and mitigate software-observable side-channels, 2019. URL: <https://github.com/google/safeside>.
21. Google Project Zero. What is a "good" memory corruption vulnerability?, 2015. URL: <https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html>.
22. Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, 2017.
23. Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.
24. Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.
25. Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled Detection of Speculative Information Flows. In *S&P*, 2020.
26. Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. *arXiv:1907.03651*, 2019.
27. Red Hat. Spectre And Meltdown Detector, 2018. URL: <https://access.redhat.com/labsinfo/speculativeexecution>.
28. Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings*, 2015.
29. Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.
30. Intel. Intel Analysis of Speculative Execution Side Channels, 2018. Revision 4.0.
31. Intel. Speculative Execution Side Channel Mitigations, 2018. Revision 3.0.
32. Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascot: Preventing microarchitectural attacks before distribution. In *CODASPY*, 2018.
33. Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler, 2018.
34. Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.
35. Esmail Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

36. Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD’s Cache Way Predictors. In *AsiaCCS*, 2020.
37. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.
38. G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.
39. Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *STC*, 2011.
40. Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations. In *ACM ACSAC*, 2019.
41. Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178*, 2019.
42. Microsoft. Azure serverless computing, 2019. URL: <https://azure.microsoft.com/en-us/overview/serverless-computing/>.
43. Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. WHISPER: A Tool for Run-time Detection of Side-Channel Attacks. *IEEE Access*, 2020.
44. 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.
45. Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys. *arXiv:1811.07276*, 2018.
46. Matthias Payer. HexPADS: a platform to detect “stealth” attacks. In *ESSoS*, 2016.
47. Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. In *NDSS*, 2021.
48. Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security Symposium*, 2019.
49. Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*, 2009.
50. David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security Symposium*, 2020.
51. Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725*, 2019.
52. Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTEXT: A Generic Approach for Mitigating Spectre. In *NDSS*, 2020.
53. Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS*, 2018.
54. Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*, 2017.

55. Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.
56. Stephen Roettger and Artur Janc. A Spectre proof-of-concept for a Spectre-proof web, 2021. URL: <https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html>.
57. Ben Titzer. What Spectre means for Language Implementers, 2019. URL: [https://pliss2019.github.io/ben\\_titzer\\_spectre\\_slides.pdf](https://pliss2019.github.io/ben_titzer_spectre_slides.pdf).
58. Ben L. Titzer and Jaroslav Sevcik. A year with Spectre: a V8 perspective, 2019. URL: <https://v8.dev/blog/spectre>.
59. M Caner Tol, Koray Yurtseven, Berk Gulmezoglu, and Berk Sunar. FastSpec: Scalable Generation and Detection of Spectre Gadgets Using Neural Embeddings. *arXiv:2006.14147*, 2020.
60. v8 developer blog, 2020. URL: <https://v8.dev/blog/v8-release-83>.
61. Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. In *USENIX Security Symposium*, 2019.
62. Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security Symposium*, 2020.
63. Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In *S&P*, 2019.
64. Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *Transactions on Software Engineering*, 2019.
65. Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Hybrid-shield: Accurate and efficient cross-layer countermeasure for run-time detection and mitigation of cache-based side-channel attacks. In *ICCAD*, 2020.
66. Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, Tinoosh Mohsenin, and Houman Homayoun. Comprehensive Evaluation of Machine Learning Countermeasures for Detecting Microarchitectural Side-Channel Attacks. In *GLSVLSI*, 2020.
67. X. Wang and R. Karri. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *DAC*, 2013.
68. Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *DSN*, 2012.
69. Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *RAID*, 2016.
70. Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, and Xiaoning Li. See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer. In *AsiaCCS*, 2021.
71. Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *AsiaCCS*, 2018.
72. Hongwei Zhou, Xin Wu, Wenchang Shi, Jinhui Yuan, and Bin Liang. Hdrop: Detecting rop attacks using performance monitoring counters. In *ISPEC*, 2014.

## A Current Worker Limits and JS Attack Numbers

The state-of-the art limits of the *Cloudflare Workers* setup for a single instance is 128 MB memory, and 30 s runtime. We provide the leakage rate depending on the amplification factor in Table 2.

Table 2: Attack results of our JS attack.

Amplification	Required requests	Script runtime	Leaked bits/hour
1	250 000	118 ms	0 bit/h
10	25 000	123 ms	1 bit/h
100	2500	137 ms	10 bit/h
1000	250	231 ms	62 bit/h
10 000	25	1813 ms	79 bit/h
250 000	1	30 000 ms	120 bit/h

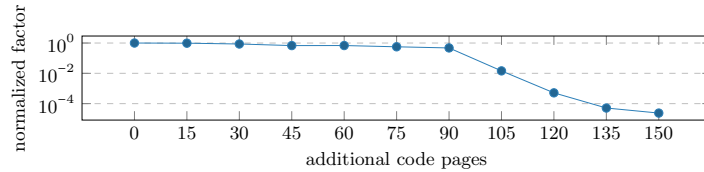


Fig. 10: Influence of the number of additional code pages on the branch accesses / iTLB accesses.

## B Spectre-Attack Optimizations

The function containing the Spectre gadget accesses the attacker’s `ArrayBuffer` through differently-sized `TypedArrays`. This prevents the JIT compiler from making assumptions on the memory accesses on the `ArrayBuffer`. Otherwise, the JIT compiler hard-codes the size for the bounds check, which significantly decreases the success probability for the Spectre attack. As the garbage collector moves objects around, using multiple `TypedArrays` increases the probability of having correctly aligned objects, such that the backing store pointer and the size of the `ArrayBuffer` lie on different cache lines. We also increase the size of the attacker function to avoid it being inlined, which would cause de-optimization if the calling function is de-optimized. The function takes the offset to access as a parameter and is called in a loop with a branch-less code that feeds it with four in-bound offsets and an out-of-bound offset to access.

We warm-up the JIT compilation by repeatedly calling our function with an out-of-bound index higher than the one in the guard branch, but in-bound of the `TypedArray`, preventing the JIT compiler from making assumptions on the provided value. We also found that executing a different number of taken conditional branches before executing the target function affected the leakage rate considerably. By automatically tuning the number of such branches, we verified that 70 is the optimal number for our PoC.

## C Noise on Cache Attacks

We evaluated the attack PoC of Röttger and Janc [56] with respect to memory-intense system workloads. We pin the attack PoC program to a specific hyper-thread and perform memory pressure on the sibling hyperthread. To simulate

such a behaviour, we use the `stress` tool with `-m 10` option on the sibling hyperthread and other cores. As the PoC of Röttger relies on amplification using the L1 replacement policy, the attack is practically mitigated by the additional cache activity. For our optimal attack, we observe a decrease of the success rate to 75 % for the time the other hyperthread creates memory pressure.