Dynamic Process Isolation

Martin Schwarzl
Graz University of Technology
martin.schwarzl@iaik.tugraz.at

Pietro Borrello
Sapienza University of Rome
borrello@diag.uniroma1.it

Andreas Kogler
Graz University of Technology
andreas.kogler@iaik.tugraz.at

Kenton Varda
Cloudflare
kenton@cloudflare.com

Thomas Schuster
Graz University of Technology
thomas.schuster@student.tugraz.at

Daniel Gruss
Graz University of Technology
daniel.gruss@iaik.tugraz.at

Michael Schwarz
CISPA Helmholtz Center for Information Security
michael.schwarz@cispa.saarland

***Abstract*—In the quest for efficiency and performance, edge-computing providers eliminate isolation boundaries between tenants, such as strict process isolation, and instead let them compute in a more lightweight multi-threaded single-process design. Edge-computing providers support a high number of tenants per machine to reduce the physical distance to customers without requiring a large number of machines. Isolation is provided by sandboxing mechanisms, e.g., tenants can only run sandboxed V8 JavaScript code. While this is as secure as a sandbox for software vulnerabilities, microarchitectural attacks can bypass these sandboxes.**

In this paper, we show that it is possible to mount a Spectre attack on such a restricted environment, leaking secrets from co-located tenants. *Cloudflare Workers* is one of the top three edge-computing solutions and handles millions of HTTP requests per second worldwide across tens of thousands of web sites every day. We demonstrate a remote Spectre attack using amplification techniques in combination with a remote timing server, which is capable of leaking $120\,\mathrm{bit/h}$. This motivates our main contribution, *Dynamic Process Isolation*, a process-isolation mechanism that only isolates suspicious worker scripts following a detection mechanism. In the worst case of only false positives, Dynamic Process Isolation simply degrades to process isolation. Our proof-of-concept implementation augments a real-world cloud infrastructure framework, *Cloudflare Workers*, which is used in production at large scale.[1] With a false-positive rate of only $0.61\,\%$, we demonstrate that our solution vastly outperforms strict process isolation in terms of performance. In our security evaluation, we show that Dynamic Process Isolation statistically provides the same security guarantees as strict process isolation, fully mitigating Spectre attacks between multiple tenants.

I. Introduction

Cloud computing is a flexible and scalable approach to deploy applications without requiring dedicated resources. The demand for cloud computing is still unbroken, as users benefit from the scalability, availability, security, and performance of the cloud. The cloud provider benefits from dynamic resource allocation, maximizing usage of available resources. To ensure high performance, cloud providers often rely on relatively modern CPUs that constantly improve the performance over the previous CPU generations.

Many performance optimizations are done in the microarchitecture of the CPU, *i.e.*, the actual implementation of the instruction-set architecture (ISA). Especially data-dependent optimizations, such as caches [76], [39], [80], [117] or branch predictors [1], [2], [26], have been well-studied. These optimizations have been shown to leak meta-data, e.g., memory-access patterns of the processed data, via side channels such as timing differences [30]. Traditionally, such microarchitectural attacks, e.g., cache attacks [117], [76], were mainly used to attack cryptographic algorithms, where the processing of secrets led to secret-dependent memory accesses [59], [77], [98], [7], [80], [76], [120], [52]. As a result, many cryptographic libraries are nowadays resilient against side-channel attacks [7], [20].

With the recent discovery of transient-execution attacks [13], such as Spectre [58], Meltdown [64], or ZombieLoad [90], attackers gained a new powerful primitive. In contrast to side-channel attacks, transient-execution attacks leak data, not meta-data. As a result, algorithms cannot be implemented such that their secrets are not susceptible to these attacks. As most transient-execution attacks work across logical CPUs, *i.e.*, hyperthreads, many cloud providers do not assign logical CPUs to different tenants [114], reducing the attack surface to applications in the same virtual machine.

While isolating tenants through virtualization prevents exploitation of co-located tenants, it also increases the performance overhead. Hence, cloud providers introduced edge computing [18], [4], where resources are dynamically allocated by the cloud provider, on a machine that is close to the customer to ensure low latencies. By running the code of multiple users within the same virtual machine, the virtualization overhead is reduced, and the cloud provider's resource utilization increases. Increasing the number of tenants per server also reduces the number of servers required at different locations. To still ensure isolation among tenants, cloud providers either rely on strict process isolation [4], [71], *i.e.*, use one process per tenant, or on language-level isolation [18], [27], [24], *i.e.*, the code has to be written in a sandboxed language such as JavaScript.

While language-level isolation incurs the least overhead, it does not protect against Spectre. Although microcode updates prevent mistraining of branch predictors from other processes [47], there is no impediment for Spectre within the same process. Even worse, Spectre attacks have been demonstrated in JavaScript and WebAssembly [70], [58], [94]. To mitigate Spectre attacks within the same process, it is necessary to insert memory barriers for each conditional branch, and avoid indirect calls and jumps. However, these mitigations have a big impact on the performance of programs [48]. Reis et al. [82]

[1] *Cloudflare Workers* are currently used by multiple hundred thousand customers.

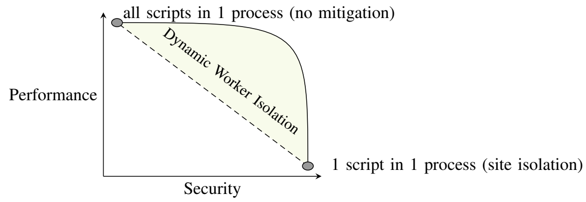

Fig. 1: For state-of-the-art strict process isolation, the trade-off between security and performance can be chosen by setting the number of scripts running inside the same process (dashed line). Dynamic Process Isolation uses a detection mechanism to improve this trade-off. Even in the worst case, *i.e.*, a $100\,\%$ false-positive rate of the detection, the performance is not worse than strict process isolation.

showed that an effective mitigation in Chrome for Spectre attacks is to perform site isolation and to ensure that no untrusted code runs within the same process. To avoid these costly countermeasures while still providing isolation among tenants, *Cloudflare Workers* rely on a modified JavaScript sandbox [18] that disables all known timers as well as primitives that can be abused to build timers [91]. This architecture allows it to support higher numbers of tenants per machine, which enables decentralized serving of applications around the world, instead of having a single centralized server per application. *Cloudflare* is one of the top three edge computing providers, *Cloudflare Workers* handles millions of HTTP requests daily. This raises an important scientific question:

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? Are the existing mitigations sufficient to prevent successful attacks?

In this paper, we analyze the mitigations used in *Cloudflare Workers*, which remove the possibility to create a local high-resolution timer and show that they are insufficient. *Cloudflare Workers* is a system which is kept up-to-date and relies on language-level isolation, thus we focus on microarchitectural attacks to attack it. We demonstrate that it is possible to slowly steal secrets using an amplified Spectre attack that acquires accurate timestamps from an external time server. However, as the resolution of the timer is not high enough, we use an amplification technique [95] that increases the timing difference between a cache hit and a miss. We achieve a leakage rate of $120\,\mathrm{bit/h}$ inside the restricted *Cloudflare Workers*. While this proof-of-concept attack does not pose an immediate threat, it shows that language-level isolation is not sufficient, and additional measures have to be taken. As already observed with other transient-execution attacks, a slow proof-of-concept implementation often just requires more engineering to make it practical [105], [94], [107].

To prevent the exploitation of Spectre in environments such as *Cloudflare Workers*, we propose a novel technique called *Dynamic Process Isolation*. This technique relies on a probabilistic detection mechanism that tries to detect ongoing Spectre attacks and isolates a suspicious workload into a separate process. With Dynamic Process Isolation we show a middle ground between the two extremes of full process isolation and language-level isolation. On the one hand, Dynamic Process Isolation keeps the performance benefits of language-level isolation for the majority of benign workloads. On the other hand, Dynamic Process Isolation provides the security guarantees of process isolation for malicious workloads, such as Spectre attacks. Even in the worst case, where every workload is classified as a Spectre attack, Dynamic Process Isolation simply degrades to strict process isolation plus the small overhead of $2\,\%$ for the detection, while, on average, it results in higher performance, as illustrated in Figure 1.

Our detection technique relies on hardware performance counters (HPC) normalized by iTLB accesses. We propose a novel probabilistic technique to detect Spectre attacks based on mispredicted and retired branches sampled via Intel Precise Event Based Sampling. We show that performance counters cannot simply be used in high-performance environments as suggested by previous work [122], [51], [79], [118], [16], [44], [73], as such a usage incurs non-negligible performance overheads. However, we demonstrate that even with a limited set of performance counters, we detect running Spectre attacks when accepting a small performance overhead of $2\,\%$.

In cooperation with *Cloudflare*, we evaluated our Dynamic Process Isolation on a production environment in the cloud. Even in such a diverse environment with different workloads, we see a false-positive rate of only $0.61\,\%$. We also mounted an attack in a production environment that we detected successfully. In all cases, Dynamic Process Isolation ensures that our exploit was blocked 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* environment, showing that the currently deployed mitigations are not sufficient.
2) We propose a novel probabilistic detection technique for Spectre attacks with low overhead.
3) We introduce Dynamic Process Isolation, a technique with, on average lower overhead than state-of-the-art strict process isolation.

Outline. The remainder of the paper is organized as follows.

In Section II, we provide the background. In Section III, we explain the building blocks of our attack and demonstrate a remote Spectre attack on *Cloudflare Workers*. In Section IV, we present a dynamic process isolation approach to detect Spectre attacks and dynamically isolate malicious scripts into separate processes. In Section V, we evaluate our approach on our production system. In Section VI, we discuss its implications before we conclude in Section VII.

II. Background and Related Work

In this section, we provide background on microarchitectural attacks and the current mitigations of *Cloudflare Workers*.

A. Cache Attacks

Cache attacks exploit the access-time differences for cached and uncached data. Since their introduction in

1996 [59], they have been used for powerful attacks on cryptographic primitives [59], [77], [98], [7], [80], [76], user interactions [89], [62], [108], or as building blocks for transient-execution attacks [64], [58], [13].

Nowadays, mainly two techniques are used to perform cache attacks, namely Prime+Probe [76] and Flush+Reload [117]. Prime+Probe does not require shared memory and was used to create cross-core and cross-VM covert channels, attacks on co-location detection, cryptographic primitives, and SGX [84], [119], [65], [74], [62], [69], [87]. Flush+Reload [117] requires shared (read-only) memory between the attacker and the victim. Flush+Reload is more accurate and has been used to build local cross-core attacks on cryptographic primitives, and to spy on user behavior [117], [37], [41], [121], [53], [54].

B. Transient Execution

In modern processors, instructions are divided into multiple micro-operations (μOPs) [28] that are executed out of order. The reorder buffer ensures that the executed instructions retire in order. Typically, a program's control flow contains conditional branch instructions. To improve the performance of branch instructions, CPUs leverage speculative execution. The branch prediction unit (BPU) tries to predict whether a branch will be taken or not using different data structures, e.g., the Pattern History Table (PHT) [28], [58]. Based on the prediction, the predicted branch is executed speculatively. 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. The instructions that were mistakenly executed based on branch prediction or out-of-order execution are called *transient instructions* [64], [13]. Transient instructions still have an effect on the microarchitecture. They can, e.g., lead to measurable timing differences in the cache state. These timing differences can then be exploited using traditional side-channel attacks to create transient-execution attacks.

C. Transient-Execution Attacks & Defenses

We distinguish between Meltdown-type [64], [13] and Spectre-type attacks [58], [13]. A Meltdown-type attack exploits transient execution caused by out-of-order execution after a CPU exception occured, e.g., a page fault [64]. This enables an attacker to transiently access memory, which is architecturally inaccessible. While the original Meltdown attack [64] showed leakage from the cache, further Meltdown-type attacks such as Foreshadow [102], [114], RIDL [105], ZombieLoad [90], and Fallout [12] also leak data from various other internal CPU buffers. Load Value Injection (LVI) is a transient-execution attack that turns Meltdown around and transiently injects data [103]. Spectre-type type attacks [58] exploit speculative execution. Spectre-PHT [13] (also known as Spectre V1) exploits the Pattern History Table, which predicts the outcome of a conditional branch [58]. A typical Spectre-PHT gadget is listed in Listing 1, containing a conditional branch with a bounds check. The attacker controls the index variable x, which length is checked for in-bounds values. By mistraining the branch prediction with in-bounds values, the branch will be speculatively executed for out-of-bounds indices. The speculative execution causes a caching of the out-of-bounds accessed value. This cached value can then be verified via a Flush+Reload loop over the byte oracle (array2 in Listing 1) to see which value was cached. Multiple Spectre variants exist which exploit different prediction mechanisms in the CPU, for instance, the Branch Target Buffer, memory disambiguation, or the Return-Stack Buffer [58], [56], [45], [60], [66]. Spectre attacks were also shown by Chen et al. [15] on Intel SGX. Schwarz et al. [92] showed that it is possible to perform a Spectre attack over the network. To mitigate vulnerable conditional branches, Intel and AMD propose to use serializing instructions, *i.e.*, `lfence` on both sides of the branch [49], [3]. Intel proposed several mitigations to tackle the different Spectre variants [49]. Furthermore, several approaches were proposed using shadow structures for caches, adding protection domains, or to actively detect and patch Spectre gadgets in the compilation phase or with binary patching [116], [55], [14].

```
if (x < array1_size) y = array2[array1[x] * 4096];
```

Listing 1: Spectre-PHT gadget.

D. Detection of Spectre Attacks

Over the past decade, many cache side-channel defenses have been proposed [9], [17], with several approaches focusing on *detection* of cache-based side-channel attacks using HPCs [51], [79], [118], [16], [44], [111], [112], [110], [123]. To detect Spectre-type attacks, static code analysis and patching, taint tracking, symbolic execution, and also detection via HPCs were proposed [21], [43], [109], [38], [73], [40]. Mushtaq et al. [73], Gulmezoglu et al. [40] and Li et al. [61] propose sampling of HPC data in combination with machine learning techniques to detect attacks actively. Related to that, Mambretti et al. [68] showed that HPCs can be used to analyze and debug Spectre attacks. However, these proposals focus on the detection of attacks, but do not propose and evaluate mechanisms to respond to protected attacks. As all these detection methods suffer from false positives, simply terminating a detected attack is not acceptable in the scenario of *Cloudflare Workers*.

E. Microarchitectural Attacks in JavaScript

Oren et al. [74] showed the first practical JavaScript-based microarchitectural attack, *i.e.*, a cache attack, that did not require installing any additional software on the victim's machine. Kocher et al. [58] showed that it is possible to perform Spectre attacks in JavaScript. McIlroy et al. [70] demonstrated all Spectre variants in the V8 engine. Röttger and Janc et al. [94] demonstrated a high-performance Spectre attack in the browser exploiting L1 timers. The main requirements for such attacks in the browser are the possibility to accurately measure time and a Spectre gadget generated by the Just-In-Time (JIT) compiler. Since the Spectre mitigations reduce the accuracy of JavaScript timers [72], other timer techniques such as counting threads have to be used [91], [58].

F. Cloudflare Workers

Cloudflare Workers is a edge computing service to intercept web requests and modify their content using JavaScript. This service handles millions of HTTP requests per second

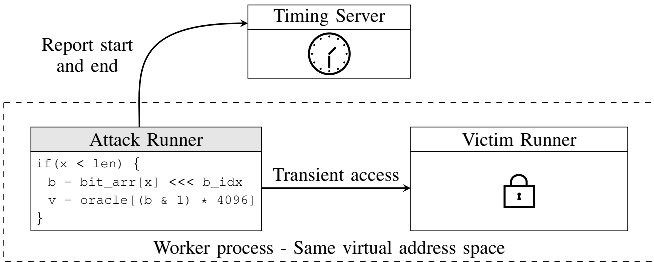

Fig. 2: Overview of the remote Spectre attack in the *Cloudflare Workers* environment.

worldwide across tens of thousands of web sites. In *Cloudflare Workers*, all worker threads run in the same process, thus sharing the same virtual address space. *Cloudflare Workers* support multiple thousand workers from up to 2000 tenants running inside the same process. Moreover, each worker is single-threaded and stateless. This design leads to a high-performant solution which is 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, mitigating counting threads [91], [34], [58]. With those restrictions in place, they claim that it is not possible to build accurate timers. In addition, the execution time and memory usage per worker can be restricted.

III. 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 is no local timing primitive, a common requirement for microarchitectural attacks [30], [88], we have to resort to a remote timing primitive. Our proof-of-concept implementation running on *Cloudflare Workers* leaks $120\,\mathrm{bit/h}$.

A. 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 co-located with the victim worker, e.g., by spawning multiple *Cloudflare Workers* and detecting co-location. As shown by Ristenpart et al. [84], an attacker spawning its instances close in time to the victim's one could maximize the probability of co-location. *Cloudflare Workers* architecture aims to serve the same application from every location, a high number of tenants per machine is possible. Co-location is not required for our attacker, 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 I, 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 to rely on 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 [33] 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, or target a certain range of the virtual memory, e.g., regions where V8 places similar objects [100].

For our attack, we rely on a Spectre-PHT [58] gadget, as this is the simplest gadget to introduce in JIT-compiled code. Moreover, Spectre-BTB [58] can be prevented by the JIT compiler [96]. In contrast to the original Spectre attack [58], we do not encode the data bytewise but bitwise. 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 [8], [92]. 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 use the amplification techniques proposed by McIlroy et al. [70],and combine them with the remote measurement methods proposed by Schwarz et al. [92]. 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 that does not require native code execution or a local timer, or an existing gadget, and that cannot be prevented in microcode (cf. Table I).

B. 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 [58], 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 [91]. We verified that, indeed, no technique from Schwarz et al. [91] resulted in a timer with a resolution higher than $100\,\mathrm{ms}$. 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 [58].

However, we can use requests to an attacker-controlled remote server to measure the execution time of the worker

Spectre attack (variant)	Gadget	Native	HR Timer	Memory	Leakage Rate	Error	Channel
Kocher et al. [58] (PHT)	Yes	Yes	Yes (ns)	2.40 MB	4420.46 B/s ± 6.75 %	0.07 %	Cache-L3
Canella et al. [13] (PHT)	Yes	Yes	Yes (ns)	3.54 MB	3.13 B/s ± 113.79 %	0.00 %	Cache-L3
Safeside [32] (PHT)	Yes	Yes	Yes (ns)	7.00 MB	4384.03 B/s ± 7.75 %	0.00 %	Cache-L3
Canella et al. [13] (BTB)	Yes	Yes	Yes (ns)	6.91 MB	0.71 B/s ± 2.43 %	0.00 %	Cache-L3
SafeSide [32] (BTB)	Yes	Yes	Yes (ns)	7.01 MB	269.53 B/s ± 0.85 %	0.00 %	Cache-L3
Canella et al. [13] (STL)	Yes	Yes	Yes (ns)	3.54 MB	14.37 B/s ± 211.95 %	0.00 %	Cache-L3
Safeside [32] (STL)	Yes	Yes	Yes (ns)	7.00 MB	272.46 B/s ± 0.22 %	0.00 %	Cache-L3
Canella et al. [13] (RSB)	Yes	Yes	Yes (ns)	20.08 MB	30.67 B/s ± 195.59 %	0.00 %	Cache-L3
Safeside [32] (RSB)	Yes	Yes	Yes (ns)	7.00 MB	116.70 B/s ± 0.58 %	0.00 %	Cache-L3
Google [94] (PHT)	No	No	Yes (μs)	15.00 MB	335.02 B/s ± 23.50 %	0.26 %	Cache-L1
Google [94] (PHT)	No	No	Yes (ms)	15.00 MB	9.46 B/s ± 31.40 %	2.71 %	Cache-L1
Schwarz et al. [92] (PHT)	Yes	Yes	No	N/A	7.50 B/h ± N/A	0.58 %	AVX unit
Our work (PHT)	**No**	**No**	**No**	27.54 MB	15.00 B/h ± 2.67 %	0.00 %	Cache-L3

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

TABLE I: Requirements and leakage rate of Spectre attacks.

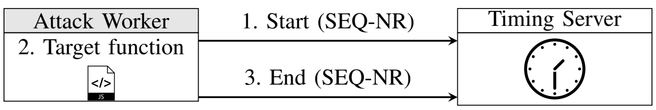

Fig. 3: Remote timing attack on *Cloudflare Workers*. We use a high-resolution timer on an external server to measure the runtime of the target function.

script, as illustrated in Figure 3. 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.*, if the remote server runs on the same physical machine, we achieve a resolution of $0.47\,\text{ns}$ on a 2.1GHz CPU, with a jitter of $1.67\,\%$. With a resolution of $0.47\,\text{ns}$, it is possible to distinguish a cache hit from a miss, as required by the cache covert channel. However, this case is not likely in reality, as the latency is typically in the microsecond range [104].

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 [70]. 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 also 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 [95]. Listing 2 illustrates an *arbitrary amplification* [95] 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. By using

```
//transiently leak bit value
if (secret_bit) { read A; } else { read B; }
read A; //perform architectural access of 1-bit
```

Listing 2: Amplified Spectre-PHT gadget [70].

a loop over the gadget, we can create arbitrarily large timing differences between cache 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. Figure 4 shows the latency between a leaked '0' and '1' bit when increasing the number of amplification iterations using an amplified Spectre-PHT gadget. As expected, there is a linear growth with the increase of the amplification iteration. 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 simply increase the amplification, which only results in a reduced leakage rate, but does not prevent the attack, as also shown in related work [94].

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 [74], [35], [106]. 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 $100\,\text{ms}$ [106]. In our remote scenario, this would require a lot of network requests to be performed 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 sizes of the machine) in cache-line steps (64 byte) and access the values. If there are enough addresses accessed, the cached value is evicted [42], [35], [58].

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) as follows. First, we access a certain index v of a large array such that it gets cached. Then, we perform the eviction loop and afterward verify whether v is still cached. To detect the optimal size of the eviction set, we

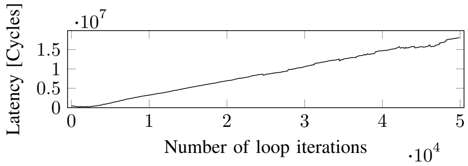

Fig. 4: Latency between a leaked '0' and '1' bit using an amplified Spectre-PHT attack. The latency increases with the number of loop iterations.

increase the size of our eviction array on each iteration until we do not observe any hits *v*. For each eviction size, we repeat our measurements 1000 times. We observe that an eviction array of 2 MB always evicts *v* on our Intel Xeon Silver 4208.

Bypassing Memory Randomization The multi-threaded execution model of *Cloudflare Workers* uses the V8 engine to provide each worker with an isolated environment, including a dedicated heap for JavaScript objects. The V8 engine randomizes the location of each heap space independently. Every pointer in each heap space is compressed, such that a pointer cannot refer to a location in a different heap space [100].

An exception is how V8 manages `ArrayBuffers`: Their backing store, *i.e.*, the buffer that contains the data, is allocated in the thread's native heap. The backing store is referred to using non-compressed pointers [100]. Therefore, the offset between two `ArrayBuffers` can be predicted and influenced by classic heap-grooming techniques [33]. Even if the two buffers are allocated in two different thread heap areas, their relative offset is constant across executions. We verified this experimentally. We thus leveraged the predictability of such allocations to leak data across *Cloudflare Workers*' `ArrayBuffers`. We evaluated the heap layout given a fixed sequence of `ArrayBuffers` allocations by inspecting the V8 process memory, and confirmed that the offset is completely deterministic as long as the attacker ensures that the victim worker maintains the same allocation sequence.

However, such a Spectre attack is not limited to `ArrayBuffers`. As the index to the `ArrayBuffer` is attacker-controlled, an attacker can provide any value to target an arbitrary virtual address in the address space. In contrast to memory-corruption attacks, an attacker can use Spectre to safely probe the address space, as invalid accessses do not raise an exception [58], [31]. Göktas et al. [31] introduced the concept of a speculative probing primitive that leverages Spectre to break classical and fine-grained address space layout randomization. Gras et al. [34], Schwarz et al. [86], and Lipp et al. [63] also demonstrated that microarchitectural attacks in JavaScript can break memory randomization. Hence, memory randomization is only a small obstacle that can be deterministically circumvented using engineering.

C. Attack on Cloudflare Workers

Using the discussed building blocks, we mount an attack on *Cloudflare Workers* to extract secret bits from a worker under the assumption that the location of the targeted byte is known to estimate the best possible attack. We mount our remote JavaScript attack as follows.

1) We send an initial request with a sequence number to a timing server. The timing server stores the current local, high-resolution timestamp when receiving the request.
2) We perform a Spectre attack on a target address.
3) We send another request to the timing server. The timing server calculates the delta between the current and the stored timestamp to distinguish between a cache hit or miss.

The timing server infers the cache state from the delta between timestamps, *i.e.*, if two requests with the same sequence number have a small delta, a cache hit was observed, otherwise, a cache miss was observed. Note that the cache state corresponds to the leaked secret bit. To initially get the timing difference between an amplified '0' and '1' bit, the attacker can setup an experiment where the JavaScript architecturally produces two cache hits ('1' bit) versus a cache hit and a miss ('0' bit). Based on the delta, the attacker chooses a threshold to distinguish the bit. Thus, the remote timing server infers the secret bit from the timings. 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 tries to emit optimized code for a function. Turbofan, the V8 optimizing compiler, uses assumptions gathered from the V8 interpreter to emit optimized code tailored to the observed inputs. If such assumptions are invalidated, the function is de-optimized. Moreover, during the garbage collection phase, objects are moved between different heap spaces of the same worker to reduce the memory footprint of the code. Such dynamic code changes impact the success rate of a PoC.

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). To speed up exploit development, we do not set the `--untrusted-code-mitigations` runtime flag, which enables array index masking in V8 [99]. Previous work already showed that these mitigations can be partially circumvented with the use of large and small JavaScript objects [42]. Furthermore, the developers of V8 found no mitigation to mitigate Spectre V4 [70], [95].

We create a Spectre-PHT PoC that leaks bits from a victim `ArrayBuffer` by transiently reading out-of-bounds. As discussed in Section III-B, using `ArrayBuffers` allows an attacker to statically compute the relative offset at which the secret is located. We describe the optimizations to increase the leakage rate in Section B.

We call the function performing a Spectre attack 10 000 times and repeat the experiment 1000 times, observing a success rate of 1 % (n = 1000, σ = 1.25%) for an unoptimized version in terms of successfully fetching an out-of-bounds value into the cache. We repeat the same experiment with our optimized version (cf. Section B) and observe a success rate of 54.31 % (n = 1000, σ = 23.16%). We assume that the attacker is capable of creating a completely stable exploit with 100 %

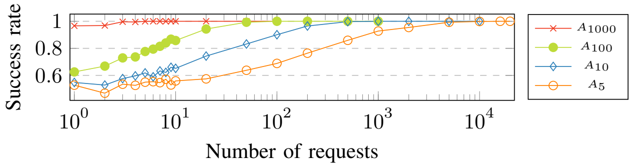

Fig. 5: Success rate over the number of requests per number of amplification factor.

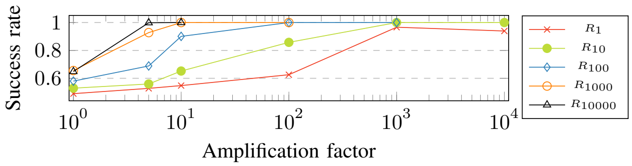

Fig. 6: Success rate of different amplification factors for different number of requests.

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.

According to NetSpectre [92], to clearly distinguish between cached and uncached data on a local network, $1\,000\,000$ measurements (1 measurement is the difference between two timing requests) are required. However, if amplification is used, the number of required requests can be reduced. 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 [22] to determine the number of required requests [104], [11], [22]. Based on the sampled data, we create histograms and choose the threshold. We determine the required number of requests to achieve the maximum success rate of $100\,\%$.

Figure 5 illustrates the number of requests required to achieve a certain success rate for different amplification factors. As can be seen, the higher the amplification factor is, the fewer requests are required to achieve high success rates. We are also in a similar range for the success rate of a sequential timing attack on the localhost [104]. In addition, we evaluate the success rate of different amplification factors based on a different number of requests. As Figure 6 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. [104] and Schwarz et al. [92] 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. The optimal attacker would choose the number with the highest success rate and the lowest number of requests required, resulting in the lowest 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 $2.5\,\mathrm{s}$ (n = 100, $\sigma_{\bar{x}}$ = $0.05\,\%$). We repeat the experiment 100 times and observe a leakage rate of $23\,\mathrm{bit/s}$ (n = 100, $\sigma_{\bar{x}}$ = $2.8\,\%$). Using an outlier filter, this error can be reduced towards 0. We measure an average distance between the means of the '1's and '0's of $3\,434\,697$ cycles. As these values are from a native-code attack, we consider these numbers as the maximum achievable leakage rate for JavaScript. The reason for that is that 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, and thus, the attacker requires a warmup to stabilize the JIT-compiled code before performing the attack.

We first evaluate the amplification in JavaScript directly in the V8 engine. We use an amplification factor of $250\,000$ and use a native timestamp counter to measure the response times. We run a script for a uniformly distributed secret with $16\,\mathrm{bit}$ length. For the first 11 bits, we observe an average timing difference of $21\,779\,307\,\mathrm{cycles}$. However, we observe that the execution timings of the function shift over time, even on an isolated core with a fixed CPU frequency (cf. Figure 17 Section C). We measure the runtime of our amplified JavaScript attack and estimate the maximum achievable leakage rate for our script. All evaluated numbers are provided in Table III (Section A). One script execution takes about $30\,\mathrm{s}$, and with a success rate of $100\,\%$ we determine an optimal leakage rate of $2\,\mathrm{bit/min}$ leading to a leakage rate of $120\,\mathrm{bit/h}$.

The current default settings for *Cloudflare Workers* are provided in Table II (Section A). With these settings, we cannot mount a successful attack due to the short runtime ($50\,\mathrm{ms}$). However, *Cloudflare Workers* consider higher runtimes and a larger amount of subrequests per intercepted request in the future. We leave it as future work to additionally improve the attack performance by leveraging parallelism in the attack [104]. Generally, our attack is not limited to *Cloudflare Workers*. It shows that an attacker does not need access to a local timer to mount microarchitectural attacks. Hence, this complements the results from Schwarz et al. [92] showing that an attacker does not require code execution, and McIlroy et al. [70] showing that microarchitectural timings can be amplified and exploited with a low-resolution timer.

IV. Dynamic Process Isolation

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. While a detection mechanism typically suffers from false positives, Dynamic Process Isolation 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 also 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. If the false positive rate is below $100\,\%$, the overall performance is better than for strict process isolation.

First, we discuss how to reliably detect Spectre attacks using performance counters (cf. Section IV-A). Second, we perform a microbenchmark on the `perf` interface to evaluate its overhead on different CPUs with different kernel boot

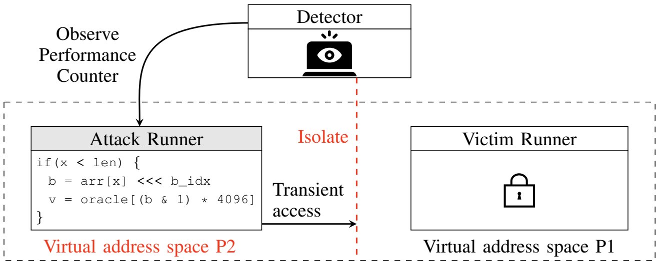

Fig. 7: Dynamic Process Isolation isolating a malicious worker based on the performance-counter reading.

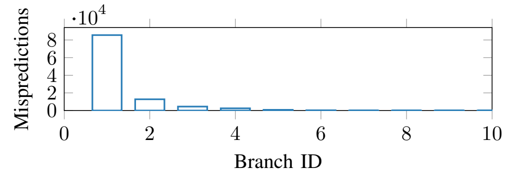

Fig. 8: Distribution of mispredicted and retired branches in a Spectre-PHT attack sorted by the number of mispredictions. The branch ID uniquely identifies a branch at a specific virtual address and is assigned based on the number of mispredictions, *i.e.*, , branch ID 1 refers to the branch with the highest number of mispredictions.

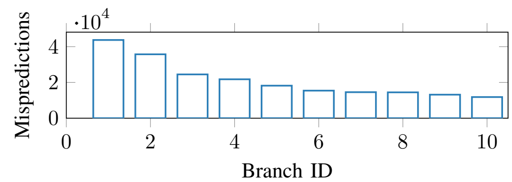

Fig. 9: Distribution of mispredicted and retired branches for the Nero2D program [81].

parameters and microcode versions (cf. Section IV-B). Finally, we integrate our approach into *Cloudflare Workers* and measure the performance overhead of reading out performance counters on a real-world cloud system (cf. Section IV-C). We show that there is a negligible performance overhead of 2 % for reading out performance counters.

A. 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 [113], [115], [67], [125], [44], [10], [36], [16]. For Dynamic Process Isolation, we develop two approaches based on HPCs. The first approach (cf. Section IV-A1) samples the mispredicted and retired branches and compares them to the distribution of a template attack using a two-sided Kolmogorov-Smirnov (KS) test [83]. Our second, faster approach uses the number of retired branches normalized with the iTLB accesses, similarly to the approach used by Gruss et al. [36] to detect cache attacks.

1) Detecting Attacks using PEBS: Our first approach uses Precise Event-Based Sampling (PEBS) on all retired and mispredicted branches (`BR_MISP_RETIRED.ALL_BRANCHES_PEBS`). Based on the samples acquired using PEBS, we build a simple pattern detection for Spectre-PHT gadgets with in-place mistraining [13]. In such a scenario, an attacker first executes the bounds check multiple times with an in-bounds index to mistrain the branch predictor. Then, for the actual transient out-of-bounds access, the branch predictor predicts that the index is in-bounds with a high probability. We only focus on the in-place variant of Spectre-PHT, as it is the most stable variant that cannot be mitigated in microcode.

We use `BR_MISP_RETIRED.ALL_BRANCHES_PEBS` (`EventSel=C5H`, `UMask=04H`) as a counter, and set the event count to 1 to sample every mispredicted and retired branch. Each sample contains the virtual address of a branch that was initially mispredicted but retired later on. Hence, this approach provides the per-branch information of the number of mispredictions. Figure 8 illustrates the histogram of mispredicted and retired branches of a Spectre gadget running a Spectre-PHT attack for a minute. As shown in Figure 9, there is a strong difference in the distribution of the mispredicted branches for a benign program. We use the two-sided KS-test for discrete distributions to compare the distribution of a sample Spectre attack and sample programs. This test verifies the hypothesis of whether two independent samples are drawn from the same independent distribution. To reject the hypothesis that the two distributions are equal, the p-value can be chosen to a certain value. The smaller the p-value, the higher the confidence, typically it is chosen to be between 0.01 and 0.1 to achieve a higher confidence. We observe that a typical Spectre attack only has a few highly mispredicted branches. Contrary to our expectations, the most mispredicted branch is not the branch exploited by the Spectre attack, *i.e.*, the bounds check. Instead, it is the delay loop used to increase the transient window (cf. Appendix C, line 55 in Kocher et al. [58]). On the second rank is the branch that is mistrained. This delay loop or similar functions are required and important for the success rate of the Spectre attack. If a delay loop which counts up to 100 is used, the success rate of the Spectre attack is at 99.76 % (n = 1000, $\sigma_{\bar{x}}$=0.53 %). If omitted, the success rate of the Spectre goes down to nearly zero (0.64 %, n = 1000, $\sigma_{\bar{x}}$ = 1.47 %). Moreover, the history of recently taken/non-taken branches affects branch prediction. Hence, adding a fixed-size delay loop helps controlling the branch-predictor state, ensuring consistent conditions for the mistrained branch [94]. Thus, also in JavaScript, this delay loop is required for a successful attack. We evaluate this approach in Section V-A.

2) 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 (iTLB accesses) which was also used by Gruss et al. [36] 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 per default 50 ms, cf. Table II Section A). 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 V-B and discuss further detection methods in Section B-A.

B. Overhead of Hardware Performance Counters

In this section, we evaluate the real-world performance overhead of using performance counters as a detection mechanism. Most previous works use smaller microbenchmarks or smaller real-world applications [122], [51], [79], [118], [16], [44], [73]. In addition to microbenchmarks, we also evaluate the performance on *Cloudflare Workers* to show that the overheads of many performance counters are so large that they cannot be used in a real-world cloud system.

Setup. We run a C microbenchmark [29] computing points on a Mandelbrot set. Such a CPU-bound benchmark is similar to a worker workload, as they cannot directly issue syscalls or interact with I/O devices. First, we set up the performance counters and measure the overhead of the setup. In a second test, we activate one performance counter and read out the values. Most Intel architectures are limited to program 4 HPCs per core, or 8 HPCs per core if hyperthreading is disabled [46]. Therefore, we activate up to four different counters and measure their performance overhead. In addition, we run our programs multi-threaded to see the additional overhead. We use `PERF_SAMPLE_READ` as sample type for our test cases such that all counters in a group are read. We repeat the microbenchmark 100 times and measure the average execution time. Additionally, we evaluate the impact of different transient-execution-attack countermeasures on the overhead of performance counters. We evaluate 5 different microcode versions on Intel CPUs that introduced mitigations, ranging from January 2018 to April 2020. In addition, we compare the impact of different active transient-execution-attack mitigations supported by the Linux kernel by successively disabling them.[2] We run our benchmark on a Xeon E5-1630.

Experimental Results. To mitigate the Rogue System Register Read vulnerability (CVE-2018-3640), a microcode update ensures that `rdmsr` cannot be used transiently [50]. Hence, we expect a measurable difference caused by the microcode when reading performance counters using `rdmsr`. In a microbenchmark, we measure the time it takes to read a performance counter 1 000 000 times, once using `rdmsr` from user space via the `msr` module, and once using the user-space-enabled `rdpmc` instruction. For the `rdmsr` benchmark, we measure the time to perform the syscall `pread` whichs reads out the msr value. Before the patched microcode (Ngovember 2017), the reading takes 37 cycles for `rdpmc` and 1291 cycles for `rdmsr`. Note that `rdmsr` is handled by the kernel, hence the larger overhead compared to `rdpmc`. Since the microcode update containing the mitigation (October 2019 update), the reading takes on average 37 cycles for `rdpmc` and 2003 cycles with `rdmsr`. Hence, the mitigation has indeed slowed down the `rdmsr` instruction, while it did not affect `rdpmc`. As all other mitigations are disabled during this test, the overhead can be attributed to the changed microcode. However, for the microbenchmark where we sample the performance counters only once every 50 ms, the performance overhead is amortized for all microcode versions. The overhead of reading the performance counters before and after the benchmark compared to normally running the benchmark was on average 0.04 % (n = 100, $\sigma_{\bar{x}}$ = 0.011 %).

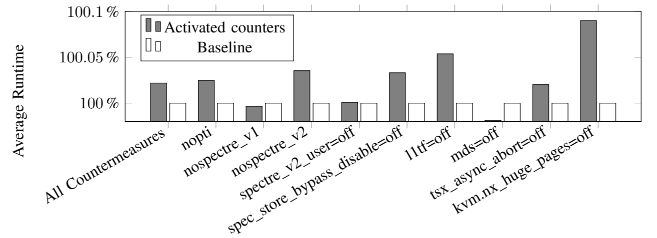

Fig. 10: HPC microbenchmark with different mitigations. The grey bars show the scaled performance overhead in comparison to the baseline programs (white bars).

In our second experiment, we selectively disable kernel mitigations for transient-execution attacks and evaluate their impact on the performance of reading performance counters. We use the microcode version from January 2020. We observe an additional average overhead of 0.08 % when running with all Spectre mitigations enabled, as shown in Figure 10. When reading the same performance counters directly in the kernel, we observe a small averaged performance overhead of 2.2 % (n = 1000, $\sigma_{\bar{x}}$ = 0.5 %) on our test devices.

We integrate the performance counter setup into *Cloudflare Workers* and run a JavaScript benchmark to evaluate its performance in a real-world cloud scenario. Reading the performance-counter values from the `perf` subsystem results in a high performance overhead of 22 % (n =10 000). If the counter is additionally enabled and disabled before and after the script execution, respectively, the overhead increases to 33 % (n =10 000). Hence, using the `perf` subsystem for performance-counter-based detection mechanisms is too expensive for *Cloudflare Workers*. With always-enabled counters and `rdpmc`, we measure a performance overhead of 2 % (n =10 000). Thus, low-overhead detection with HPCs for a production system requires obtaining measurements directly from the kernel or the user-space-enabled `rdpmc` instruction.

C. Process Isolation

For Dynamic Process Isolation, we fundamentally rely on process isolation. A well-known implementation of process isolation is site isolation, where every page in a browser

[2] https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html

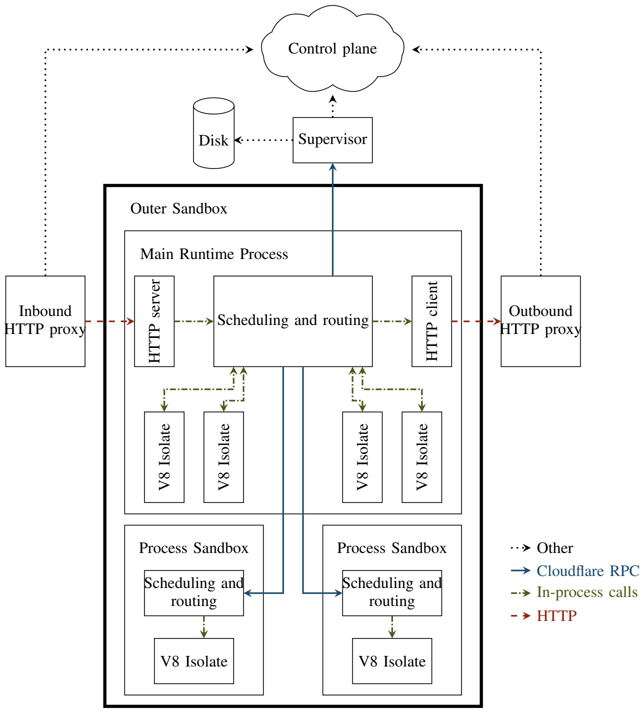

Fig. 11: System overview for Dynamic Process Isolation.

runs in its own process to prevent memory safety violations as well as Spectre attacks [82]. However, in contrast to full site isolation, we only isolate malicious *Cloudflare Workers* if the Spectre detection mechanism flags them. Hence, Dynamic Process Isolation only falls back to full site isolation in the worst case, while reducing the overhead caused by process isolation in the average case where only suspicious *Cloudflare Workers* are isolated.

Related work proposes efficient in-process isolation mechanisms using Intel Memory Protection Keys (MPK) [101], [78], [85]. However, Intel MPK is only available on selected CPUs since Skylake-SP, limited to 16 protection keys and thus not practical for *Cloudflare Workers* [101], running multiple thousand workers per process. Furthermore, the threat model of these approaches does not include side-channel or transient-execution attacks. To perform dynamic process isolation, 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, as illustrated in Figure 11. For that, we start process sandboxes by forking from a zygote process, and talk to the new process over an RPC protocol [5], [19]. All communication between the main process and the isolated process are over this RPC connection, communications between the process sandbox and the outside world have to go back 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 calculates a moving average. From our results in Section V, 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 the per-CPU-core performance-counter data of real scripts running on the production system. Based on our evaluation in Section V-B, we use a threshold of 4096 retired branches per iTLB access to distinguish between a suspicious and a benign script.

If a script exceeds this threshold, we flag it as a potential Spectre attack. Such a worker is isolated 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 still continue running, e.g., in case the detection was a false positive, while it cannot access data of any other worker.

V. Evaluation

In this section, we evaluate the accuracy and performance overhead of our detection methodologies. First, we evaluate our PEBS-based approach (cf. Section IV-A1). 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 [57], 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 %. However, in our production environment, we observe a performance overhead of up to 50 %. In contrast, for our normalized counters approach, we observe a negligible overhead of 2 %.

A. PEBS-based Approach

We evaluate the PEBS-based approach using 128 programs from the Phoronix benchmark suite [81] (cf. Table IV Section D). The test suite already supports the perf interface. We only have to add the `BR_MISP_RETIRED.ALL_BRANCHES_PEBS` event.

We use the two-sided KS-test for discrete distributions [83] to compare the distribution of mispredictions. In our scenario, we compare the distribution of mispredictions for benign and malicious scripts. We compare the programs from Table IV (Section D) to our Spectre attack. In addition, we integrate 13 sample Spectre gadgets from Kocher [57] and use it to evaluate the detection.[3]

We use the top 100 branches of the programs as input to the KS-statistic. If the p-value is high or the KS-statistic is low, we cannot reject the hypothesis that the two distributions are the same. We choose the p-value to be 0.1 to achieve a 90 % confidence level. Every script with a p-value lower than this threshold is rejected. The p-value seems to be high, however, as the structure of certain test programs have certain similarities to a Spectre attack, this filters out most of the programs with a small similarity. Thus, if the p-value is below this threshold, we can reject the hypothesis that the two samples are drawn from the same distribution. Conversely, if the p-value is above the threshold, we cannot reject the hypothesis and thus count it as a false positive. We evaluate all 141 test programs using the KS-test. We detect all 13 test

[3] Only 13 out of 15 test gadgets leaked data on our evaluated system.

programs and achieve a true positive rate of 100 %. Using this approach, we achieve a false positive rate of 2.83 %. In total, we achieve an F-Score of 0.87. We integrate this approach into *Cloudflare Workers* system. We sample the PEBS events on each occurrence while the script is running to get an exact distribution. To evaluate the performance overhead, we first run this sampling approach on a single-threaded Spectre-PHT attack proof of concept. For the small proof of concept, we measure a performance overhead of 2.3 % (n=1000, $\sigma_{\bar{x}}$=0.5 %) However, on a production system, we get an overhead of 50 % (n=100, $\sigma_{\bar{x}}$=8.13 %). While the approach works well in smaller non-multithreaded environments, an overhead of 50 % is too costly to run it on the production environment.

Copying the data in batches or directly mapping the PEBS buffer into user space[4] could drastically improve the performance of PEBS sampling, decreasing the performance overhead of this approach. However, we did not evaluate this on the *Cloudflare Workers* production system, as modifying the kernel on a production system would be a huge engineering and testing effort to ensure that no customer data is at risk.

B. Normalized Performance Counters

The PEBS-based approach is too expensive on a real-world cloud system. However, as shown in Section V, the performance overhead is only 2 % when using the `rdpmc` instruction for reading raw performance-counter values. We verified this on 5 Intel Xeon server CPUs (Broadwell, Skylake 4116, Skylake 6162, Skylake 6162, Cascade Lake 6262) and one AMD Epyc Rome CPU running in the cloud. 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 12 shows the normalized performance-counter values of our cloud machines. In the case of last-level-cache accesses, last-level-cache 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 between an average script and the 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 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 13 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 can be 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.

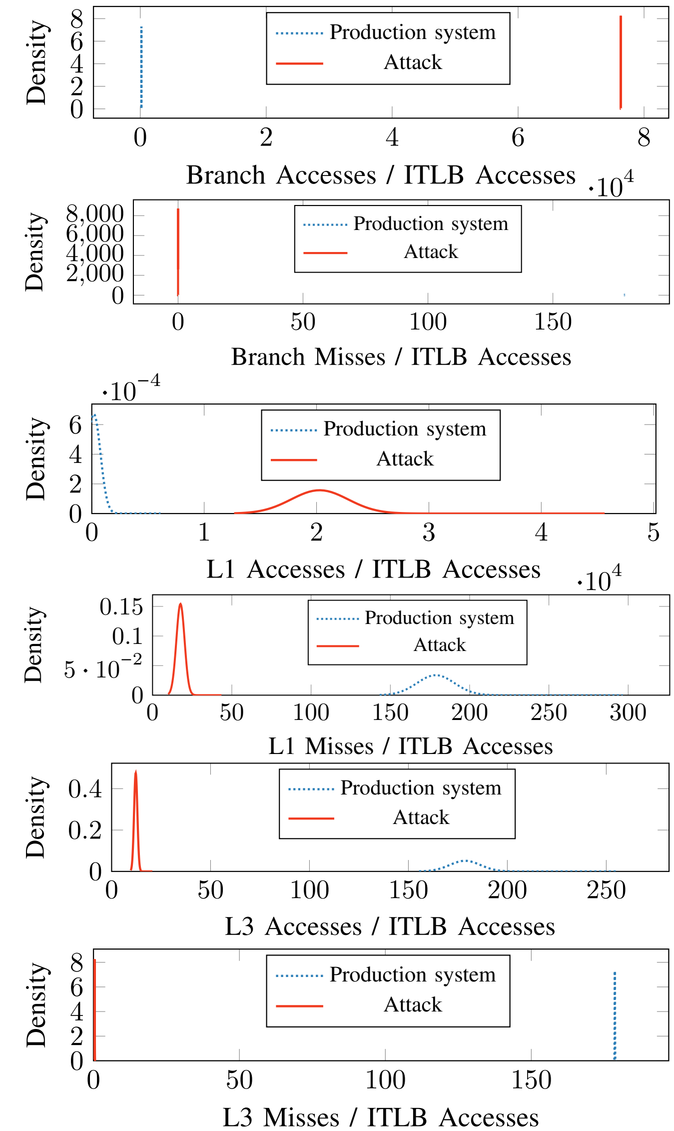

Fig. 12: Performance counters of the average *Cloudflare Workers* script and a Spectre attack script on the production system.

Next, we look at the performance overhead of our attack in the case where 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, the more requests, *i.e.*, samples, are required to clearly distinguish between a cache hit and miss (cf. Figures 5 and 6). We evaluate the best possible attacker in native code who only mistrains one branch. By omitting amplification or by applying 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, we observe a

[4] https://elixir.bootlin.com/linux/v5.8/source/arch/x86/events/intel/ds.c#L1754

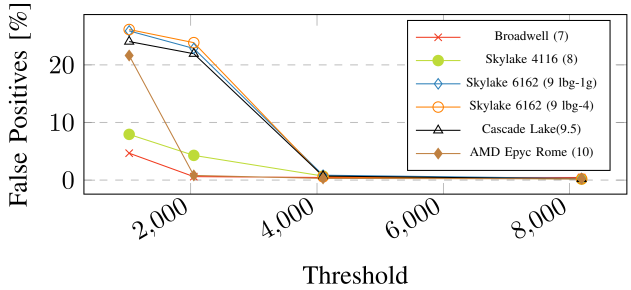

Fig. 13: Number of false positives depending on the normalized iTLB threshold.

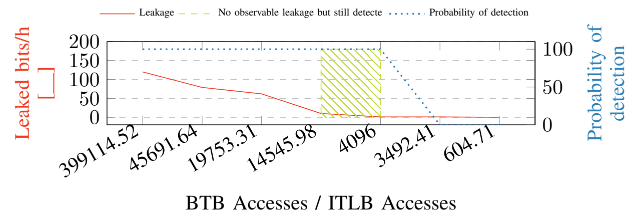

Fig. 14: Reducing the branch accesses / iTLB accesses and the corresponding leakage rat.

leakage rate of 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 14 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 III-C). 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 detect a slowed-down attack using our threshold-based approach. Additionally, if an attacker can get below the threshold, the number of requests on the same worker is limited.

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 13 Spectre samples provided by Kocher [57]. In addition, we evaluated and analyzed the new and larger Spectre-PHT gadgets generated by FastSpec [97]. 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. This is due to the fact that the mistraining for those gadgets is similar, leading to a similar range of branch accesses per iTLB access. We also evaluated the detection on the Spectre JavaScript PoC from Röttger and Janc [94]. Even with the low amplification factor of 4000 used in this PoC, we reliably detect the attack (n = 500, μ =19 253.73).

Spectre-BTB, Spectre-RSB and Spectre-STL. In addition to Spectre-PHT we also run our performance counter analysis on the other Spectre variants exploiting the branch-target buffer (BTB), return-stack buffer (RSB) and store-to-load (STL) forwarding. We create native code proof-of-concepts for these variants which execute each gadget 10 000 times on a Xeon Silver 4208. We ran the PoCs 500 times and collected the number of branch accesses and iTLB accesses. The numbers for Spectre-BTB and RSP are on order of magnitued lower compared to Spectre-PHT ($\mu_{b\bar{t}b}$ = 423171.54). However, they are still detected with the same metric (n = 500): Spectre-BTB ($\mu_{b\bar{t}b}$ = 23401.20), Spectre-RSB ($\mu_{r\bar{s}b}$ = 38369.17), Spectre-STL ($\mu_{s\bar{t}l}$ = 982.20)). The metric for Spectre-STL is far below the threshold of 4096. However, the performance counter values for `memory_disambiguation.history_reset` are significantly higher on average if the store-to-load logic is exploited in Spectre-STL (n = 500, $\mu_{s\bar{t}l}$ = 8993.98, $\mu_{no\bar{s}tl}$ = 2644.73). Thus, we additionally use this counter to detect potential Spectre-STL attacks.

C. Dynamic Process Isolation

We integrate Dynamic Process Isolation 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 will vary depending on the workload [82]. *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, dynamic process isolation 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, $\sigma_{r\bar{p}s}$ = 3.87%, $\sigma_{m\bar{e}m}$ = 0.23%). Figure 15 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 worker known to be unusually resource hungry in both CPU and memory usage. In this case, we found memory overhead to be 20 %-70 % worse with dynamic process isolation, and CPU time to be 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 [5] 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

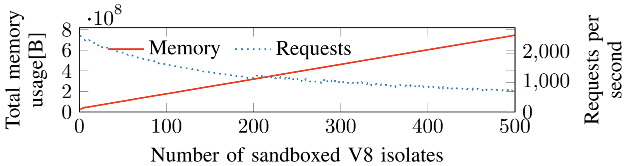

Fig. 15: Performance overhead in terms of requests per second and total memory consumption of process isolation

FUTEX_SWAP feature proposed by Google [75]. However, while especially the CPU overhead could be reduced, there will always be significant cost incurred by context switching and marshalling needed 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 additional 2 % overhead for the detection. However, due to the low false-positive rate, this is extremely unlikely.

VI. Discussion

Mitigation versus Detection. The current mitigations for Spectre are costly [13], [58], [49]. Especially in high-performance scenarios, such as cloud systems, the mitigations result in high power consumption. Hence, instead of paying the constant costs of mitigations, it can be desirable to reduce the costs by relying on the detection of attacks. 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. Our detection relies on performance counters, and, thus, false positives will occur [23], [124]. The advantage of Dynamic Process Isolation is that the detection mechanism can tolerate false positives. In comparison to machine learning approaches that try to reduce the false-positive rate [61], [40], we propose a simple threshold to detect potential Spectre attacks with a false-positive rate of 0.61 %. This approach is lightweight, easy to integrate into production software, and has a small performance overhead. Our approach can be extended to more sophisticated machine learning approaches as proposed in related work [61], [40] to reduce the false-positive rate, as long as the false-negative rate does not increase.

False Negatives. Our detection relies on performance counters, and, thus, false negatives will occur [23], [124]. This is similar to other detection and mitigation techniques, such as oo7 [109] or Spectector [38], which suffer from false negatives for novel forms of Spectre gadgets [97]. Nevertheless, with the current limits given in Table II (Section A), the attack would always exceed the CPU runtime constraint. As *Cloudflare Workers* considers increasing the limits in terms of execution time, an attack could be feasible. Therefore we evaluated the attack in terms of the best possible attack we found. The results of Table III (Section A) show that it is possible to get below the threshold, leading to false negatives. But with an amplification factor of 10, the leakage rate is only about 1 bit/h and requires about 25 000 requests. By limiting the number of subrequests to 10 000, we mitigate this special case where the attacker slows down the attack.

Adding additional code pages might allow an attacker to get below the proposed thresholds. We evaluated the influence of the number of additional code pages on the detection factor in Figure 16 (Section A). To "hide" the native attack, we access 125 additional code pages per bit to get the branch accesses / iTLB accesses below the threshold. In total, this adds 500 kB code. While feasible in native code, the JavaScript environment is more constrained. We tried different methods to inflate the binary size. In all cases, the resulting code size causes V8 to abort the optimization phase, stopping the attack.

PEBS-based Detection. PEBS-based detection mechanisms seem to be an interesting research topic for future work, especially with the precise timestamps provided by PEBS. However, while working with PEBS, we discovered a bug in the Linux perf implementation, where Linux accidentally overwrites the precise timestamp. We reported this bug and submitted a patch to LKML [6].

Reliability of HPCs In Dynamic Process Isolation As Zhou et al. [124] and Das et al. [23] discuss, using HPCs for detection of microarchitectural attacks can lead to flaws caused by non-determinism and overcounting. Our approach uses the `perf` interface to setup the HPCs. However, by leveraging the `rdpmc` instruction, the counter values are read out directly from the model-specific registers. We consider non-determinism and overcounting effects by averaging multiple times over the performance counter results for each script.

Alternative Spectre JS attacks Concurrent work [94] has demonstrated an amplifiable Spectre exploit on V8, able to leak up to 60 B/s using timers with a precision of 1 ms or worse through a side channel on the L1 cache. 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. During our evaluation, we extended the published PoC with two additional steps. We first leverage the leaks inside the isolate to disambiguate the layout of the neighboring objects and locating the address of a controlled array. Then we leverage a custom Spectre V1 gadget to perform a *speculative type confusion* between two large objects that span multiple cache lines so that the type information of both objects are on different cache lines than the field target of the type confusion. Speculating past the type check dereferences an attacker-provided integer. Thus, the Spectre gadget can load data from an arbitrary 64-bit address. However, our enhanced PoC requires two memory dereferences during the speculation window. We found they can be completed reliably only if the type information of the offending object is not cached while the integer containing the address is cached. Evicting only the type information from the cache requires an eviction set, and we failed to obtain that using only a remote timer.

VII. Conclusion

In this paper, we presented Dynamic Process Isolation, 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 120 bit/h. We proposed a novel approach of actively detecting Spectre attacks by sampling mispredicted and retired branches, which achieves a false positive rate of $2.83\,\%$. However, due to the large number of samples required, it leads to a total overhead of $50\,\%$ on the production environment. 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, while still providing the security guarantees of process isolation.

References

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *AsiaCCS*, 2007.

[2] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[3] Advanced Micro Devices Inc. Software Techniques for Managing Speculation on AMD Processors, 2018. Revison 7.10.18.

[4] Amazon AWS. AWS Lambda@Edge, 2019. URL: https://aws.amazon.com/lambda/edge/.

[5] Anonymous. Anonymized for Double Blind Submission, 2019.

[6] Anonymous. Anonymized for Double Blind Submission, 2019.

[7] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[8] 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.

[9] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR. SGX: Automated and adjustable side-channel protection for SGX using data location randomization. In *ACSAC*, 2019.

[10] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *CODASPY*, 2018.

[11] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[12] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, 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/.

[14] Chandler Carruth. Speculative load hardening. URL: https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6.

[15] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, 2019.

[16] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.

[17] Haehyun Cho, Jinbum Park, Donguk Kim, Ziming Zhao, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. SmokeBomb: effective mitigation against cache side-channel attacks on the ARM architecture. In *MobiSys*, 2020.

[18] Cloudflare. Cloudflare Workers, 2019. URL: https://www.cloudflare.com/products/cloudflare-workers/.

[19] Cloudflare. Anonymized for Double Blind Submission, 2020.

[20] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P*, 2009.

[21] Jonathan Corbet. Finding Spectre vulnerabilities with smatch, April 2018. URL: https://lwn.net/Articles/752408/.

[22] 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.

[23] 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*, pages 20–38. IEEE, 2019.

[24] Deno. A Globally Distributed JavaScript VM, 2021. URL: https://deno.com/deploy.

[25] Elastic Blog. Detecting Spectre and Meltdown Using Hardware Performance Counters, 2018. URL: https://www.elastic.co/blog/detecting-spectre-and-meltdown-using-hardware-performance-counters.

[26] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[27] Fastly. Serverless Compute Environment - Fastly Compute@Edge, 2021. URL: https://www.fastly.com/products/edge-compute/serverless.

[28] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers, 2016.

[29] The Computer Language Benchmarks Game. C Benchmarks Game, 2020. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/c.html.

[30] 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.

[31] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, 2020.

[32] Google. SafeSide: Understand and mitigate software-observable side-channels, 2019. URL: https://github.com/google/safeside.

[33] Google Project Zero. What is a "good" memory corruption vulnerability?, 2015. URL: https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html.

[34] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, 2017.

[35] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[36] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[37] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[38] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled Detection of Speculative Information Flows. In *S&P*, 2020.

[39] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*, 2011.

[40] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. *arXiv:1907.03651*, 2019.

[41] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In *COSADE*, 2015.

[42] Noam Hadad and Jonathan Afek. Overcoming (some) Spectre browser mitigations, 2018. URL: https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/.

[43] Red Hat. Spectre And Meltdown Detector, 2018. URL: https://access.redhat.com/labsinfo/speculativeexecution.

[44] 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.

[45] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[46] Intel. Intel 64 and IA32 Architectures Performance Monitoring Events, 2017. Revision 1.0.

[47] Intel. Intel Analysis of Speculative Execution Side Channels, 2018. Revision 4.0.

[48] Intel. Intel analysis of speculative execution side channels, 2018. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[49] Intel. Speculative Execution Side Channel Mitigations, 2018. Revision 3.0.

[50] Intel. Intel-SA-00115 Q2 2018 Speculative Execution Side Channel Update, 2019. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html.

[51] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Preventing microarchitectural attacks before distribution. In *CODASPY*, 2018.

[52] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *RAID*, 2014.

[53] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Know thy neighbor: Crypto library detection in cloud. *PETS*, 2015.

[54] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *AsiaCCS*, 2015.

[55] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO*, 2018.

[56] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[57] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler, 2018.

[58] 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.

[59] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[60] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[61] Congmiao Li and Jean-Luc Gaudiot. Online detection of spectre attacks using microarchitectural traces from performance counters. In *Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018.

[62] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[63] 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.

[64] 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.

[65] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[66] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[67] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *STC*, 2011.

[68] 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.

[69] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, 2017.

[70] 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.

[71] Microsoft. Azure serverless computing, 2019. URL: https://azure.microsoft.com/en-us/overview/serverless-computing/.

[72] Mozilla. performance.now resolution, 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

[73] 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.

[74] 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.

[75] Peter Oskolkov. [PATCH for 5.9 0/3] FUTEX_SWAP (tip/locking/-core), 2019. URL: https://lkml.org/lkml/2020/7/22/1202.

[76] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[77] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *Cryptology ePrint Archive, Report 2002/169*, 2002.

[78] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys. *arXiv:1811.07276*, 2018.

[79] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.

[80] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[81] Phoronix. Linux 4.12 To Enable KASLR By Default, 2020. URL: https://www.phoronix-test-suite.com/.

[82] Charlie Reis. Mitigating spectre with site isolation in chrome, 2018. URL: https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html.

[83] Simard Richard and Pierre L'Ecuyer. Computing the two-sided kolmogorov-smirnov distribution. *Journal of Statistical Software*, 01 2011.

[84] 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.

[85] 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.

[86] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725*, 2019.

[87] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[88] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS*, 2018.

[89] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*, 2018.

[90] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[91] 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.

[92] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[93] Mary Lou Soffa, Kristen R. Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging (nier track). In *International Conference on Software Engineering (ICSE)*, 2011.

[94] Stephen Röttger 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.

[95] Ben Titzer. What spectre means for language implementers, 2019. URL: https://pliss2019.github.io/ben_titzer_spectre_slides.pdf.

[96] Titzer, Ben L. and Sevcik, Jaroslav. A year with Spectre: a V8 perspective, 2019. URL: https://v8.dev/blog/spectre.

[97] 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.

[98] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In *CHES*, 2003.

[99] v8 developer blog, 2019. URL: https://v8.dev/docs.

[100] v8 developer blog, 2020. URL: https://v8.dev/blog/v8-release-83.

[101] 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.

[102] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[103] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[104] 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.

[105] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[106] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In *S&P*, 2019.

[107] Julien Voisin. Spectre exploits in the "wild", 2021. URL: https://dustri.org/b/spectre-exploits-in-the-wild.html.

[108] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In *NDSS*, 2019.

[109] 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.

[110] H. Wang, H. Sayadi, S. Rafatirad, A. Sasan, and H. Homayoun. SCARF: Detecting Side-Channel Attacks at Real-time using Low-level Hardware Features. In *IOLTS*, 2020.

[111] 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.

[112] 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.

[113] X. Wang and R. Karri. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *DAC*, 2013.

[114] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution, 2018. URL: https://foreshadowattack.eu/foreshadow-NG.pdf.

[115] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *DSN*, 2012.

[116] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*, 2018.

[117] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[118] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *RAID*, 2016.

[119] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *S&P*, 2011.

[120] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*, 2012.

[121] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*, 2014.

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

[123] 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.

[124] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *AsiaCCS*, 2018.

[125] Hongwei Zhou, Xin Wu, Wenchang Shi, Jinhui Yuan, and Bin Liang. Hdrop: Detecting rop attacks using performance monitoring counters. In *ISPEC*, 2014.

Appendix A
Current limits of default worker and JS attack numbers

The state-of-the art limits of the *Cloudflare Workers* setup for a single instance is given in Table II. We provide the leakage rate depending on the amplification factor in Table III.

Appendix B
Optimizations of Spectre attack

The function containing the Spectre gadget accesses the attacker's `ArrayBuffer` through differently-sized `TypedArray`s. 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 `TypedArray`s 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 specifically avoid triggering any de-optimization points in our generated code (*i.e.*, breaking assumptions of the JIT compiler), as that ruins the predictor's training. Therefore, we place the out-of-bound access of the `TypedArray` behind a mispredicted guard branch that depends on a variable with a value strictly lower than the size. This prevents the JIT compiler from de-optimizing the code when detecting out-of-bound accesses. We additionally increase the size of the attacker function to avoid it being inlined, which would cause de-optimization if the calling function is de-optimized, and would prevent training the predictors if called from different call sites. The function takes the offset to access as a parameter

TABLE II: Current limits of worker.

Configuration	Value
Allowed Consumed Memory	64 MB
Compile Time	200 ms
CPU runtime per request	50 ms
# of subrequests per request	50 requests

TABLE III: 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

and is called in a loop with a branch-less code sequence 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 using an out-of-bound index that is higher than the one in the guard branch, but in-bound with respect to the `TypedArray`, preventing the JIT compiler from making assumptions on the provided value. We additionally found that executing a different number of taken conditional branches before executing the target function affected the leakage rate considerably. The taken branches seem to affect code alignment and predictor states. By automatically tuning the number of such branches, we empirically verified that 70 is the optimal number for our PoC.

A. Other Detection Methodologies

On Skylake and newer generations, PEBS provides a precise timestamp of the sampled events [46]. As the `BR_INST_RETIRED.NOT_TAKEN` event does not support PEBS, we use the conditional mispredicted branches (`BR_MISP_RETIRED.CONDITIONAL`) and retired near taken call (`BR_INST_RETIRED.NEAR_TAKEN`) events. By sampling the timestamps, it is possible to timely interleave the two events. As a result, it is possible to record a Spectre-PHT in-place exploitation pattern, *i.e.*, multiple correct predictions and then one misprediction. Using this setup, we successfully detect a Spectre-PHT in-place exploitation pattern. While this approach detects a Spectre-PHT attack, it has a high false-positive rate, as other structures result in a similar pattern, such as, e.g., loops. Moreover, PEBS with precise timestamps is only available since Skylake, which limits its applicability in real-world scenarios.

Another approach proposed to detect Spectre-attack patterns is to use the Last Branch Record (LBR) [25]. While this approach counts how often a basic block is executed, it does not provide information on whether a branch to a basic block was correctly predicted. Furthermore, we found that the resolution of the LBR (32 entries on Skylake) is too small to reliably detect attacks. Alternatively, the Branch Trace Store (BTS) collects exact data on branch mispredictions [46], but introduces large performance overheads of up to 40 % [93].

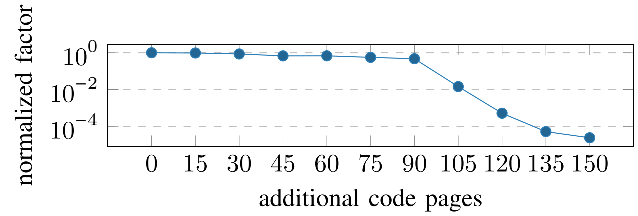

Fig. 16: Influence of the number of additional code pages on the branch accesses / iTLB accesses.

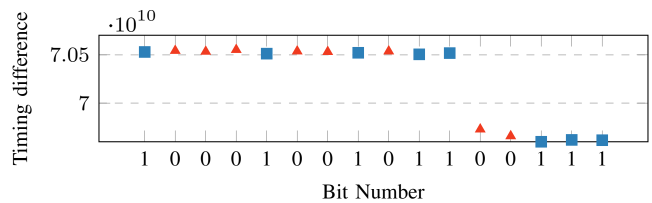

Fig. 17: Time shifting of local time stamp counter.

Appendix C
Time Shifting in V8

We observe a time shifting of the time stamp counter when leaking several bits after each other. After a random amount of time, the execution time of the function shifts to another range. We assume that some parts of the V8 engine might cause this effect, which we leave for future work to analyze.

Appendix D
Phoronix Benchmarks

Table IV lists all evaluated programs from the Phoronix benchmark suite for the PEBS-based approach (cf. Section V-A) as well as the 13 Spectre gadgets from Kocher [57].

TABLE IV: Evaluated programs of the Phoronix benchmark suite [81].

System Tests	Disk Tests	Processor Tests			Kocher Tests
apache	aio-stress	aircrack-ng	gmpbench	noise-level	Spectre V01
caffe	sqlite	amg	gnupg	npb	Spectre V02
compress-lzma		aobench	go-benchmark	n-queens	Spectre V03
compress-pbzip2		aom-av1	gpu-residency	oidn	Spectre V04
compress-zstd		blosc	graphics-magick	openssl	Spectre V07
gnupg		bork	hackbench	openvkl	Spectre V08
openssl		botan	himeno	ospray	Spectre V09
redis		bullet	hmmer	parboil	Spectre V10
battery-power-usage		byte	hpcg	perl-benchmark	Spectre V11
		cachebench	ipc-benchmark	polybench-c	Spectre V12
		clomp	java-gradle-perf	polyhedron	Spectre V13
		cloverleaf	java-scimark2	povray	Spectre V14
		compress-7zip	john-the-ripper	primesieve	Spectre V15
		compress-gzip	lammps	psstop	FastSpec gadgets
		compress-xz	lczero	qmcpack	
		core-latency	libgav1	radiance	
		coremark	llvm-test-suite	rays1bench	
		cp2k	luajit	redis	
		cpuminer-opt	lulesh	renaissance	
		crafty	luxcorerender	rodinia	
		c-ray	mafft	scimark2	
		cython-bench	mencoder	smallpt	
		dacapobench	minife	sqlite	
		dav1d	minion	stockfish	
		dcraw	mkl-dnn	sudoku	
		deepspeech	mpcbench	svt-av1	
		dolfyn	m-queens	svt-hevc	
		ebizzy	mrbayes	svt-vp9	
		embree	java-gradle-perf	swet	
		encode-flac	java-scimark2	system-decompress-bzip2	
		encode-mp3	john-the-ripper	system-decompress-tiff	
		encode-wavpack	mt-dgemm	system-decompress-xz	
		espeak	multichase	system-libjpeg	
		ffmpeg	namd	system-libxml2	
		ffte	neat	tachyon	
		fftw	nero2d	toybrot	
		fhourstones	nettle	tscp	
		glibc-bench	node-express-loadtest	ttsiod-renderer	
		tachyon	node-octane	tungsten	
		toybrot	ttsiod-renderer	vpxenc	
		tscp	x264	x265	
		yafaray	y-cruncher		