Rain: Transiently Leaking Data from Public Clouds Using Old Vulnerabilities

Anonymous Authors

***Abstract*—Given their vital importance for governments and enterprises around the world, we need to trust public clouds to provide strong security guarantees even in the face of advanced attacks and hardware vulnerabilities. While transient execution vulnerabilities, such as Spectre, have been in the spotlight since 2018, until now there have been no reports of realistic attacks on real-world clouds, leading to an assumption that such attacks are not practical in noisy real-world settings and without knowledge about the (host or guest) victim. In particular, given that today's clouds have large fleets of older CPUs that lack comprehensive, in-silicon fixes to a variety of transient execution vulnerabilities, the question arises whether sufficient software-based defenses have been deployed to stop realistic attacks—especially those using older, supposedly mitigated vulnerabilities.**

In this paper, we answer this question in the negative. We show that the practice of mitigating vulnerabilities in isolation, without removing the root cause, leaves systems vulnerable. By combining such "mitigated" (and by themselves harmless) vulnerabilities, attackers may still craft an end-to-end attack that is more than the sum of its parts. In particular, we show that attackers can use L1TF, one of the oldest known transient execution vulnerabilities (discovered in January 2018), in combination with a simple speculative out-of-bounds load, to leak data from other guests in a commercial cloud computing platform. Moreover, with an average end-to-end duration of 15 hours to leak the TLS key of an Nginx server in a victim VM under noisy conditions, without detailed knowledge of either host or guest, the attack is realistic even in one of today's biggest and most important commercial clouds.

1. Introduction

As governments and enterprises around the world rely on proprietary public clouds for on-demand access to scalable computing, today's clouds have become a foundational component of our digital infrastructure. Since all multi-tenant cloud solutions imply that users may share physical resources with untrusted other tenants, they require implicit trust in the security guarantees of the underlying hardware and system software to keep data safe from adversarial co-tenants. Providing such guarantees is challenging, especially in the presence of the ever-expanding arsenal of transient-execution and side-channel exploits [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. While mitigations exist for each of these vulnerabilities, even for older and vulnerable hardware, they are often incomplete, especially as deployed. The reason is that comprehensive mitigation of transient execution vulnerabilities such as Spectre [1], L1TF [7], [13], and MDS [8], [9], [10] is very expensive [14]. Not surprisingly, industry deploys 'spot' mitigations instead—rendering the vulnerability too difficult to exploit using known techniques, but without removing the root cause.

Even so, there have been no reports of realistic attacks on real-world clouds with any of the older vulnerabilities. It appears as if the collection of mitigations put in place by cloud providers is sufficient to stop exploitation of these vulnerabilities in practice [15]. Indeed, some consider these attacks so complicated that it is questionable if they form a practical threat [16], [17], [18], [19], [20]. While new vulnerabilities still create media attention [21], [22] and often remain under embargo while mitigations are deployed [23], there is little concern about older vulnerabilities such as L1TF [7], [13] (discovered in January 2018).

In this paper, we question this lack of concern and show not only that practical attacks on modern clouds are possible, but that they are possible with vulnerabilities we considered mitigated 7 years ago. In particular, we use L1TF together with a speculative out-of-bounds load to overcome all relevant security measures and leak sensitive data from the hypervisor and even a co-tenant on the Google cloud. While others have argued that combining L1TF and half-Spectre was possible [24], [25], and some used synthetic gadgets in proof-of-concepts [26], we show that more than a theoretical possibility, this is a real-world threat in popular clouds. Using a novel technique based on pointer chasing through the host and guest, we leak all information required to manually perform two-dimensional page table walks (i.e., through the guest's page tables and extended page tables) in software; with this, we can translate arbitrary virtual guest addresses to host physical addresses, enabling the leakage of any byte in the memory of the victim via L1TF.

Our leakage primitive also works on the AWS cloud, but defenses in depth restrict leakage to non-sensitive host data. The attack assumes no knowledge of the software running on the victim, nor does it assume a prior leakage of the randomized address space layouts, and it is effective even in the presence of realistic noise. Our research provides further evidence that the common practice of mitigating a vulnerability in isolation is not sufficient if it fails to remove the root cause, as attackers may combine issues that are harmless in isolation to craft an end-to-end attack that is more than the sum of its parts.

We demonstrate our attack on KVM-based hypervisors, and believe the approach applies to other hypervisors as well. In particular, we evaluate the attack in three types of cloud environments that run on CPUs where L1TF was

mitigated in software: a local system running base KVM, Google cloud (GCE) instances of type N1 [27], and AWS cloud instances of type C5 [28]. On base KVM and GCE instances, we successfully extracted sensitive data, such as the TLS key from an Nginx server running on a guest VM. On AWS cloud instances, we were able to leak non-sensitive information from the host, but not any data from guest VMs, due to defenses in depth in the hypervisor that unmap sensitive guest data from it. Our attack is stealthy and explicitly detects whether it is running on a CPU that may be L1TF-vulnerable. We are not aware of any mechanism that reliably allows a hypervisor to detect this type of transient execution attack and have not received any indication that Google or AWS detected any of our exploits[1].

We summarize our contributions as follows:

- We show transient execution attacks: (a) are *practical* to leak sensitive data in *real-world clouds*, (b) using some of the *oldest vulnerabilities* available, and (c) *without prior knowledge* of the victims.
- We demonstrate that spot mitigations of transient execution vulnerabilities are not sufficient, as the combined exploitation of vulnerabilities deemed harmless individually can lead to high-impact attacks.
- We describe a novel attack technique based on aggressive pointer chasing and software-controlled two-dimensional page table walks to leak any byte in a VM running on the same host system in the GCE cloud.
- We showcase our findings by launching an end-to-end guest-to-guest attack on proprietary cloud environments and successfully leak cryptographic data from a victim VM in the GCE cloud.

Coordinated Disclosure. Following coordinated disclosure practices, we officially notified the relevant parties – Google and AWS – about the attacks in May, 2025. Both Google and AWS acknowledged the leakage primitive presented in this paper and confirmed our ability to read host memory. AWS further confirmed that deployed defense-in-depth mechanisms prevent us from leaking sensitive data (i.e., memory from guest VMs running on the same physical system). Google, being vulnerable to our end-to-end exploit, verified the attack and confirmed its severity; in response, they awarded a total bounty of over $150k as part of their Cloud Vulnerability Reward Program (VRP), the most that the Cloud VRP has ever given out. We strongly believe that this open attitude towards security is a good thing. Our conclusion is not that AWS's and Google's security was lacking, but that they are actively stimulating security improvements. Google and AWS have notified us that, in addition to patching the exploited half-Spectre gadget in their respective hypervisors, they plan to implement additional security measures to (further) improve their security. They aim to release security patches/updates accordingly.

[1]As we explain later, we did our research with explicit permission of the cloud providers, with safeguards to prevent leakage of third party data.

Ethical Considerations. To avoid any negative impact on the targeted cloud provider parties or any of their customers, we took the necessary precautions. Both Google and AWS were fully aware of our exploitation efforts, and we frequently discussed progress. All of our experiments were conducted on dedicated host systems in the respective provider's fleet. As a precaution, we informed each provider about the exact system we were going to attack and when. This guaranteed exploitation occurred in a realistic, representative environment with no risk to any of the cloud providers' customers. Lastly, we disclosed all findings in this paper to the affected parties in a coordinated disclosure, and discussed its contents prior to submission. Google and AWS acknowledge our findings and have agreed on a coordinated public disclosure.

2. Background

2.1. Transient Execution Attacks

For optimization purposes, modern processors implement out-of-order and prediction-based speculative execution to perform operations before they are needed or before the CPU knows if it should perform them at all. Doing so may lead to undesired behavior when the prediction is wrong or the processor is unable to handle execution faults (e.g., illegal data accesses) immediately. While the CPU will detect these errors eventually and roll back all changes to the architectural state (registers, memory), traces of such "transient execution" remain detectable in the microarchitectural state. For instance, data accessed by the transient instructions will now be in the cache. Moreover, if a memory access is secret dependent, attackers can often recover the secret data by means of a cache attack [29], [30], [31], [32]. Since the first disclosure of Meltdown [2] and Spectre [1] in 2018, researchers have discovered many new transient execution attacks [3], [4], [6], [7], [8], [9], [11], [13], [33], [34], [35], [36], [37].

2.2. L1 Terminal Fault

One of the earliest transient execution vulnerabilities, discovered in January 2018, is L1 Terminal Fault (L1TF [7], [13], [38], [39]). It allows unprivileged speculative access to data residing in the L1 data cache and affects Intel processors up to Coffee Lake [40]. The root cause of L1TF is the use of faulty address translations in speculative execution in the case of invalid page table entries (PTEs). For instance, accessing a virtual address for which the corresponding PTE's present bit is cleared (or its reserved bit is set), should stop all further address translation [38], including Extended Page Table (EPT) translation. However, speculative execution will not wait for such checks to resolve and, instead, load the referenced data, assuming it resides in the L1 data cache. Moreover, it will forward the data to subsequent instructions, enabling the leakage.

In prior work [13], Weisse et al. outlined different methods to exploit L1TF. For instance, an unprivileged attacker

can wait for the operating system to clear the present bit of a PTE when swapping pages to disk and then access any cached data that maps to the same physical address as pointed to by the PTE, regardless of ownership. Better still, a malicious guest VM may itself clear the present bits in the guest page table to trigger terminal faults. Because of the fault, the CPU skips host address translation and passes the guest physical address immediately to the L1 data cache, enabling the attacker to read any cached physical memory on the system.

To combat such exploits, Intel provides in-silicon mitigations for newer generations of CPUs, and microcode updates and software-based measures for older CPUs [38]. Fully mitigating L1TF in software requires disabling SMT together with flushing the L1 data cache upon context switches, or disabling EPT in virtualized environments—measures that are so expensive that providers opt for more practical spot mitigations [18].

In particular, cloud providers use the combination of L1D flushing [38], [41] and core scheduling [15], [42]. L1D flushing consists of flushing the L1 data cache upon a context switch and prevents a task's data from leaking to another task scheduled after it on the same vCPU. In Linux on x86-64, this occurs upon `vmresume` instructions, either for every context switch or conditionally [41]. However, flushing the cache alone does not suffice, as another task or guest may run concurrently on another vCPU of the same physical core. Security-minded systems therefore combine flushing with core scheduling: ensuring that a task is only scheduled on a core if the other tasks on the core are also trusted. Combined, these mitigations prevent user-to-user or guest-to-guest attacks. In principle, user-to-kernel or guest-to-host attacks are still possible, as the kernel/hypervisor can still run on the same physical core with any process/guest at any time, but Linux kernel developers explicitly doubted the practical exploitability of the short vulnerable window [15]. Further, some kernel developers questioned whether the risk posed by vulnerabilities such as L1TF were worth the cost and complexity of implementing a more comprehensive mitigation such as address space isolation [43], [44], especially when VMs are running on dedicated physical cores and core scheduling is enabled [43].

2.3. Half-Spectre Gadgets

To launch a Spectre attack [1], an attacker steers speculative execution toward a particular code pattern, a so-called "gadget", that discloses and transmits secret data via a microarchitectural side channel (e.g., a cache attack). For instance, the Spectre-v1 gadget in Listing 1 consists of an out-of-bounds memory access that loads some secret data, and a subsequent secret-dependent load that transmits said data, following a mispredicted conditional branch. To exploit this gadget, the attacker lets the branch execute many times with a value for index that satisfies the condition on Line 1 to 'train' the CPU to predict that the branch will be taken upon the next iteration. By subsequently providing a value for index that does not satisfy the condition, the CPU speculatively accesses a value outside the array bounds (Line 2) and then leaks the data using a memory dereference through B, before recognizing its mistake.

```
1  if (index < ARRAY_SIZE)
2     x = A[index];  // access secret during speculation
3     y = B[4096*x]; // transmission of secret
4
5  if (index < ARRAY_SIZE)
6      x = A[index];  // access secret during speculation
```

Listing 1: Top: a classic Spectre-v1 gadget. Bottom: a half-Spectre gadget, an incomplete Spectre-v1 gadget where the transmission of the secret is absent.

To mitigate Spectre, software developers aggressively remove such code patterns from their programs. However, what about incomplete gadgets, such as the bottom code fragment in Listing 1? This "half-Spectre" gadget (sometimes referred to a prefetch gadget in literature [45]) is exactly like the top one, but without the transmission part (Line 3). These are harmless by themselves, and are still very common. While attackers may still leak the secret accessed in Line 2 using other transient execution attacks such as MDS [8], [9], [10], these attacks have been comprehensively mitigated [46], so the risk was considered limited. However, recent work has shown that, in theory at least, half-Spectre gadgets are still usable in combination with other vulnerabilities to leak sensitive data [24], [26]. The question is whether the spot mitigations deployed in modern clouds are sufficient to stop such attacks in practice.

3. Threat Model

We consider a malicious VM in a realistic cloud environment, in the presence of co-located workloads. Attackers have root access to their own virtual machine and may perform hypercalls to interact with the hypervisor. We assume that the host system runs the latest microcode version and is up-to-date with regard to all (default) mitigations against transient execution attacks, including L1D flushing [38], [41] and core scheduling [15], [42]. In addition, we assume the presence of all default mitigations against traditional software exploitation, such as stack canaries, W$\oplus$X and (K)ASLR. Further, we consider the attacker blind with respect not just to other guest VMs, but also to the host system, i.e., they do not have any in-depth knowledge of the (proprietary) hypervisor and the undocumented measures deployed against exploitation. The aim of our attacker is to perform a transient execution attack in order to leak sensitive data from the host and/or other VMs in the system.

4. Attack Overview

In this section, we provide an overview of L1TF Reloaded, in which an attacker combines L1TF with a half-Spectre gadget in the hypervisor to launch a guest-to-guest attack that achieves arbitrary data leakage from other guest VMs in the system.

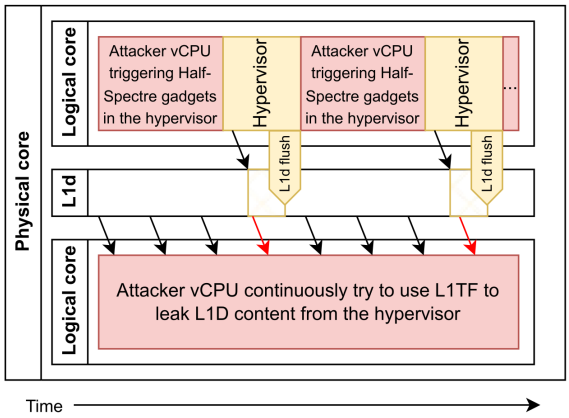

Figure 1: L1TF Reloaded combines L1TF with half-Spectre. On logical core 1, the attacker performs a hypercall, to make the hypervisor execute a half-Spectre gadget. By mistraining a prior bounds check, the attacker forces the hypervisor to speculatively access out-of-bounds memory, bringing hypervisor data into L1d which logical core 2 leaks using L1TF.

L1TF mitigations, such as L1D flushing and core scheduling, prevent guest-to-guest attacks, but not guest-to-host attacks—although Linux kernel developers were initially doubtful about their practical exploitability [15]. In this paper, we instead demonstrate the practicality of the attack. By means of a controllable half-Spectre gadget in the hypervisor, we trick it into speculatively loading arbitrary system memory into the L1 data cache. We then use L1TF to leak the contents of the L1 data cache, despite all mitigations present against L1TF. Figure 1 conceptually illustrates this combination of L1TF + half-Spectre.

Our end-to-end attack consists of six phases: (1) local attack preparation, (2) cloud host profiling, (3) gadget base discovery, (4) host targeting, (5) guest targeting, and (6) data extraction. Given the sophisticated nature of the attack, we first give a high-level overview of each of the steps and defer the detailed explanation to Section 5. We display this high-level overview in Figure 2.

Local Attack Preparation. In this phase, we lay the ground work for our subsequent attack steps on our own machine. In particular, we prepare a guest VM that is optimized for L1TF leakage and scan a recent version of the target hypervisor for an exploitable half-Spectre gadget. While we do not know if the version of the hypervisor in the public cloud is the same, we assume that it will be similar. For instance, after discovering an exploitable half-Spectre gadget in KVM through local gadget analysis, we make the educated guess that this gadget also exists in the hypervisors in public clouds that are based on KVM. While any controllable transient load primitive will do, we assume in the remainder of this paper that the gadget accesses an element in an array with an offset that is controlled by the attacker.

Cloud Host Profiling. Next, we determine whether the host system provided by the targeted public cloud is vulnerable to our attack. We do so by first profiling the CPU, to see if our guest VM is likely running on a CPU that is vulnerable to L1TF. If so, we perform a number of experiments that detect the mitigations that the host enables against L1TF, and check whether they impede our attack. If all indicators suggest that the host is vulnerable, we commence our exploit.

Gadget Base Discovery. Given the half-Spectre gadget in the hypervisor, we define x as the attacker-controllable offset used by the gadget to access an array element. We first consider the gadget's base—the address corresponding to offset zero of the array. This is an unknown virtual address v. Since we can only use L1TF to leak data from the cache corresponding to a particular physical host address, we find the physical host address p that corresponds to v.

Next, we are interested in the host kernel's direct (memory) map, where all physical memory of the host system is linearly mapped in virtual memory. Namely, if v points inside the direct map, we can use the gadget to load arbitrary physical host memory mapped at $v + x$ (with x potentially negative) into the L1 data cache, and extract it using L1TF on the linearly matched physical address $p + x$. In that way, we acquire an arbitrary physical memory read primitive for the host.

However, it is not guaranteed that the gadget's base v points inside the direct map. If not, we must find the value for x such that $v + x$ accesses the gadget's base v' in the direct map; with that, we can then add y to x such that $v + (x+y)$ loads arbitrary physical memory from the direct map that we can then leak using L1TF at the matching physical address $p + y$.

Host Targeting. From the direct map, we leak a kernel pointer to easily recognizable data and search through physical memory for this data using our previously acquired primitive. To this end, we find the physical address that corresponds to the (virtual) kernel pointer. As the physical memory is linearly mapped in the direct map, subtracting the found physical address from the kernel pointer gives us the start of the direct map in virtual memory. With the start of the direct map, we can translate host virtual addresses in the direct map to physical addresses to leak them using L1TF, effectively breaking KASLR in the host.

We subsequently use our address translation ability to discover the location of the root page table of the host. We do so by chasing (i.e., repeatedly dereferencing) kernel pointers in the direct map through kernel structures and leaking their contents. By performing a page table walk with our leakage primitive, we can now translate arbitrary host virtual addresses to physical addresses (and with that, chase arbitrary host pointers).

Guest Targeting. Using our ability to translate arbitrary host virtual addresses to physical addresses, we chase pointers through kernel structures to find and leak the metadata of our victim VM in the host system. Doing so, we leak the root of its extended page tables and the value of its CR3 register (i.e., the root of its own page tables). Together, these enable

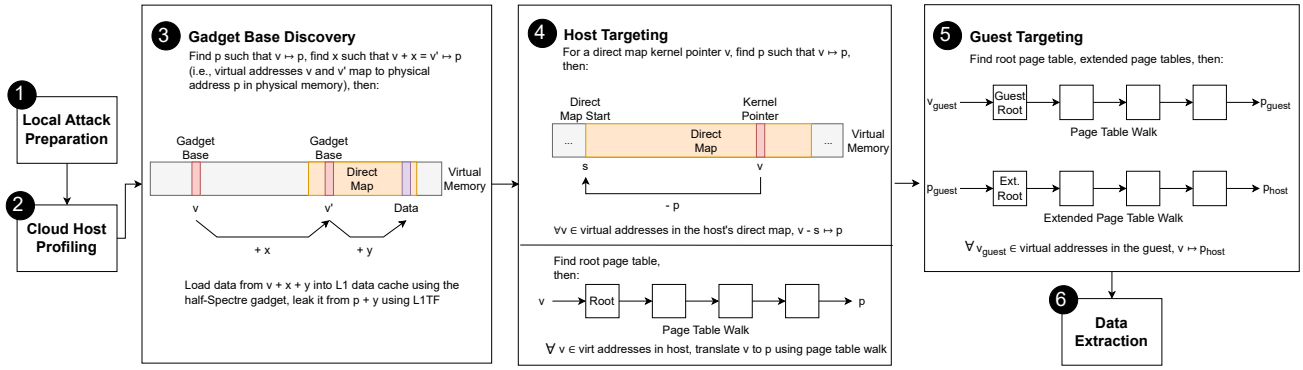

Figure 2: Overview of our attack strategy. After **(1)** preparing our attack locally and **(2)** checking whether our VM is likely running on a vulnerable CPU, we **(3)** find the physical address of the gadget's base and use the host kernel's direct map to establish an arbitrary host physical memory read primitive. Then, we **(4)** break KASLR by determining the start of the direct map, and by chasing kernel pointers in it locate the host's root page table to enable the translation of all host virtual addresses to host physical addresses. Using this translation ability, we further chase pointers through kernel structures to **(5)** find the victim VM's page tables and extended page tables, enabling virtual guest address to host physical address translation. With that, we can **(6)** chase arbitrary guest pointers to extract sensitive data from the victim VM guest.

us to perform two-dimensional page table walks, translating guest virtual addresses to host physical addresses.

Next, we evaluate the guest's page tables to determine its address space layout in order to break KASLR in the guest. With that, we have now acquired the ability to chase arbitrary pointers in the guest kernel, and know where to start searching for interesting processes to target.

Data Extraction. Finally, with knowledge of the guest's kernel layout, we leak the kernel structures necessary to discover all running tasks in the guest. For the target task, we chase pointers throughout its metadata structures to leak its root page table. From there, we search through the task's memory in the guest for any sensitive data that we desire.

5. L1TF Reloaded

L1TF Reloaded, our end-to-end transient execution attack on a (proprietary) public cloud, leaks sensitive data from a victim guest VM on a mainline Linux host and a GCE host, while on AWS, we leak (only) non-sensitive data from the host itself due to deployed defenses in depth. The main challenge for the attacker is finding the targeted secret data in physical memory – within the hundreds of gigabytes of RAM commonly available on cloud servers. We address this challenge by abusing the (meta)data resident in the host and victim guest kernels. In this section, we discuss the steps necessary to build our exploit and leak sensitive data from a victim guest VM.

5.1. Local Attack Preparation

Before launching the exploit, an attacker locally prepares a guest kernel and scans for usable half-Spectre gadgets in a host that is similar to the targeted cloud's hypervisor.

Exploiting L1TF. To leak data from arbitrary physical addresses using L1TF, the attacker needs (partial) control over the PTE that is used to cause a terminal fault, and ensure that the data is present in L1 data cache. In our threat model, the attacker has full control over a malicious VM, and thus has the ability to change their own PTEs. Specifically, they can choose the host physical address to leak from and set the present or reserved bits to trigger the terminal fault. For our attack, we use a modified version of PTEditor [47], which maps the page tables of our program into user space to enable the modification of PTEs without involving the kernel. Normally, the TLB must be flushed when the PTE is changed. However, as no TLB entry or paging-structure cache entry is created for non-present pages on Intel CPUs [48], we add this use case to PTEditor to improve our performance.

Upon accessing a non-present page, the CPU's MMU will raise a page fault exception that is handled by the kernel's page fault handler. Consequently, every address we attempt to leak causes a slow context switch to the kernel, as well as the execution of many instructions. This not only impedes our performance, but also affects our leakage results, as every instruction that is executed may influence the state of the L1 data cache and may even lead to the eviction of the target data. Since we have full control over the guest kernel, we apply a patch that suppresses a page fault in the kernel and bypasses the page fault handler. More specifically, instead of properly handling the L1TF-induced page fault, the interrupt handler returns immediately to a user-defined user address—executing only a few instructions in the kernel. In terms of performance, this approach yields an average 0.5 microsecond latency for a page fault as opposed to a 2.6 microsecond one for the base kernel (across 2 million measurement samples).

Half-Spectre in the Hypervisor. L1TF allows an attacker to leak data in the L1 data cache in a guest-to-host attack scenario. Thus, any data accessed by a hypervisor running on the same core as the attacker will be susceptible to leakage. As the hypervisor by itself is unlikely to (frequently) access sensitive data, we combine L1TF with a half-Spectre gadget to trick the hypervisor into loading sensitive data into the L1 data cache, and subsequently leak it using L1TF.

For a half-Spectre gadget in the hypervisor to be usable in our attack, the attacker must be able to control its execution reliably. In particular, to induce mispredictions on a conditional branch, the branch needs to be consistently taken to train the branch predictor. Secondly, the attacker should control the memory the gadget accesses, for instance, by means of an attacker-controlled offset. To find such gadgets, we focused our analysis on VM hypercalls and potential gadgets in their handlers, because a VM may use the privileged `vmcall` instruction to call a predefined function in the hypervisor—a hypercall. As these functions can be reached reliably by an attacker VM, and some hypercalls take attacker-controlled arguments, they are fertile ground to scan for exploitable gadgets.

Through manual code review, we found the half-Spectre gadget in the hypercall handling code for the `KVM_HC_SCHED_YIELD` hypercall shown in Listing 2. By manually tracing uses of this hypercall's user-controlled parameters, we find that one of these values is used to index into the `phys_map` array—an array located in the direct memory map of the Linux kernel. In the base, upstream Linux kernel, all available physical memory of the host is directly mapped into the kernel's virtual memory [49]. Thus, an out-of-bounds memory access in this array allows an attacker to transiently access all of the host's memory. In this gadget, the index value is checked before it is used, preventing an architectural out-of-bounds access. However, in-place branch mistraining of the bounds check on `dest_id`, and evicting `map→max_apic_id` from the cache to lengthen the speculative window, allows an attacker to force the host kernel to perform the speculative load `map→phys_map[dest_id]` on any (8-byte aligned) 64-bit virtual address.

A drawback of this particular half-Spectre gadget is that the throughput, even on a mostly idle system, is very low due to the single_task_running check (line 9) blocking access to the gadget most of the time. To quantify this, the gadget is essentially blocked 99.9% of the time. However, this is not a fundamental issue, and we will show in Section 5.3 that with various optimizations we can use the gadget to perform an end-to-end exploit.

5.2. Cloud Host Profiling

To perform the attack, the attacker must determine whether their malicious VM runs on a vulnerable host system in the targeted cloud.

For the host system, this means:

1) The host must run on an L1TF-vulnerable CPU.

```
1  static void kvm_sched_yield(struct kvm_vcpu *vcpu,
2                              unsigned long dest_id)
3  {
4      struct kvm_vcpu *target = NULL;
5      struct kvm_apic_map *map;
6
7      vcpu->stat.directed_yield_attempted++;
8
9      if (single_task_running())
10         goto no_yield;
11
12     rcu_read_lock();
13     map = rcu_dereference(vcpu->kvm->arch.apic_map);
14
15     if (likely(map) && dest_id <= map->max_apic_id
16             && map->phys_map[dest_id])
17         target = map->phys_map[dest_id]->vcpu;
18
19     ...
```

Listing 2: The half-Spectre gadget we exploit in Linux/KVM. Via a hypercall, a VM can trigger this KVM function with full control over the hypercall argument `dest_id`.

2) The hypervisor must deploy insufficiently comprehensive mitigations in software.

In a virtualized environment, the hypervisor may emulate CPUID to hide the true CPU microarchitecture, and in addition, attackers do not know the exact defenses deployed. Therefore, we consider an attack scenario where we automatically detect whether a system is likely to be vulnerable from inside a guest, without relying on the hypervisor.

Vulnerable CPU. Intel server CPUs of the Skylake microarchitecture and earlier are vulnerable to L1TF [40]. Hence, to address the first requirement, we assess whether the attacker VM runs on such a system. Unfortunately, we cannot assume that self-reporting by means of the `CPUID` instruction is accurate, as the instruction may be emulated. Instead, we measure the size of the Path History Register (PHR), which in Skylake and older microarchitectures is only 93 branches, while on recent CPUs such as Ice Lake have PHRs it is as large as 194 [50]. The PHR is a shift register that records the global history of recently executed branches (and whether they were taken) to help predict correlated branch patterns. Thus, we can determine the PHR size by testing how many dummy branches we can insert between branches while still capturing the correlation between them—using an algorithm from prior work [50] adapted to a virtualized environment.

A PHR size equal to 93 does not *guarantee* that the CPU is vulnerable to L1TF. For instance, the Cascade Lake microarchitecture has the same PHR size, but is not vulnerable to L1TF by virtue of hardware mitigations. Even so, PHR size estimation provides attackers with a quick test to narrow down the search for vulnerable systems, without wasting cycles on CPUs that are known not to be vulnerable.

Deployed Hypervisor Mitigations. As the only mitigations that tackle the root of the issue in software are prohibitively expensive, the default configuration on operating systems such as Linux relies on spot mitigations instead. We address the second requirement by checking the inventory

of software-based mitigations against L1TF present on the system via timing side channels.

First, we measure memory access latency before and after a VM guest-to-host context switch, to determine whether the hypervisor implements L1D flushing. We prime the L1 data cache by accessing a set of addresses that neither evict other addresses in the same cache set nor incur prefetching by the memory controller. By triggering a VM exit path that does not trash the L1 data cache and measuring the addresses' subsequent access latency, we can tell whether the cache was flushed.

Next, we build upon prior work [51] to detect SMT using a port-contention side channel. Modern CPUs contain a number of execution ports that process micro-operations that originate from instructions executed on either of the CPU's logical cores (vCPUs). As the ports are shared by the vCPUs, contention can lead to an increased instruction execution latency for one vCPU if the other vCPU occupies the same ports. Thus, we determine whether SMT is enabled by measuring the instruction latency of a particular contention-sensitive workload that runs on both vCPUs of a physical core against a single-vCPU baseline.

Concerning core scheduling, in our experiments on proprietary cloud environments, we found that it is common to implement a scheduling policy that dictates only processes of the same VM can co-locate on the same physical core. As two different VMs will never be co-located, we decide not to test for basic core scheduling.

5.3. Gadget Base Discovery

L1TF can leak data from any physical address as long as the data at that address resides in the L1 data cache. To make sure sensitive data from the hypervisor is loaded into the L1 data cache, we want to use the half-Spectre gadget in Listing 2 to speculatively perform an out-of-bounds memory access. However, since the physical address of the data is unknown, we cannot extract it from the cache. We now explain how we solve this and build our arbitrary host physical memory read primitive.

The half-Spectre gadget accesses an array in memory by taking the address of index (offset) zero of the array (i.e., the gadget's base) v and adding an attacker-controlled offset (index) x. While v is unknown, we know that v = `&map→phys_map`. By making use of the page alignment of the `map` object, which is dynamically allocated upon the creation of our VM, we elect to brute force its physical address. To do so, we continually trigger the half-Spectre gadget with offset zero (i.e., `dest_id` = 0) to bring the start of the `phys_map` structure into the L1 data cache. Using L1TF on the sibling vCPU, we stride at page granularity through physical memory, guessing the base's physical address p. The beginning of the `phys_map` array holds a pointer to a `kvm_lapic` structure for each vCPU of the VM, which in our case is two pointers. From here, we know we have correctly guessed the physical address of the two pointers if we leak them using L1TF.

```
1  static int __pv_send_ipi(unsigned long *ipi_bitmap,
2                           struct kvm_apic_map *map,
3                           struct kvm_lapic_irq *irq,
4                           u32 min)
5  {
6      int i, count = 0;
7      struct kvm_vcpu *vcpu;
8
9      if (min > map->max_apic_id)
10         return 0;
11
12     for_each_set_bit(i, ipi_bitmap,
13         min((u32)BITS_PER_LONG,
14                 (map->max_apic_id - min + 1))) {
15         if (map->phys_map[min + i]) {
16             ...
```

Listing 3: The architectural gadget we use to help locate the half-Spectre gadget's base.

The discovery of the physical address p of our gadget's base yields an arbitrary host physical memory read primitive. Since the base is `kmalloc`'d and hence points inside the host's direct map, passing a (possibly) negative offset x to the half-Spectre gadget lets it speculatively load from address $p+x$ through the direct map (as the linear mapping of the direct map means it matches $v+x$).

While the accessed array is located in the host kernel's direct memory map for this particular half-Spectre gadget, for another it may not be. In that case, we must find the gadget's base mapped in the direct map at v' by speculatively trying different values of x for our out-of-bounds memory access at $v+x$. We can verify whether we encounter the base in the direct map at v' by checking the leaked data using L1TF and seeing whether it corresponds to the leaked data from v. Using the knowledge of the base's physical address, the attacker would know the base's offset in the physical memory page, allowing them to search through memory at a page-sized stride; regardless, this step requires additional optimization to be practical.

As we discussed prior in Section 5.1, the throughput of the half-Spectre gadget is very low. To compensate, we optimize our approach by firstly not leaking the entirety of the pointers, and secondly by employing an alternative half-Spectre gadget that, while not exploitable due to its particular data flow, can consistently load our actually targeted gadget's base address into cache. We show this architectural gadget in Listing 3.

5.4. Host Targeting

Next, using the arbitrary physical memory read primitive we acquired, we fully leak the two previouly mentioned kernel pointers that are at the beginning of the `phys_map`. We know from manual analysis that these two pointers point to structures that start with easily recognizable data in the host kernel's direct map; we refer to them as direct map pointers. Now, we aim to break KASLR in the host kernel by finding the physical address of the data one of the pointers points to.

As the direct map is backed by 1 GB pages in memory, we know the lower 30 bits of the physical addresses of

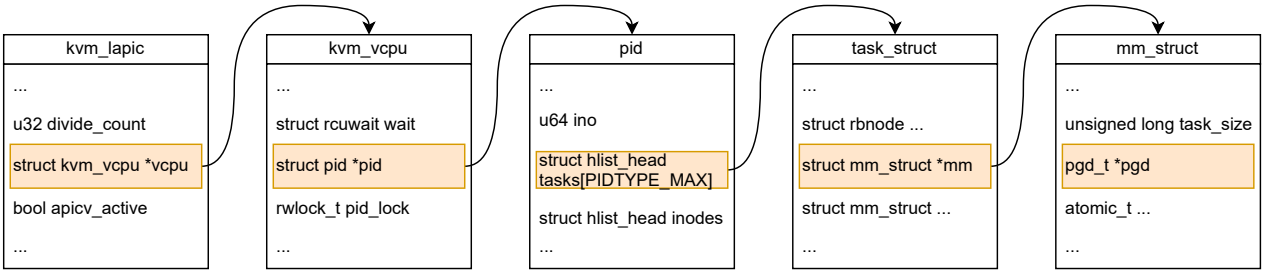

Figure 3: Example of a pointer chase in our exploit chain. As we can translate a (virtual) pointer to a physical address, we leak the data to which the pointer points, essentially dereferencing it. From the `kvm_lapic` struct, we leak consecutive pointers to other structs until we find the host's root page table (`pgd`, Page Global Directory).

the pointers (because the pointer is a virtual address, and the direct map is linearly mapped into virtual memory). We can brute-force the remaining few physical address bits by leaking from every possible physical address and checking whether the leaked data matches the data we are looking for. Once we have found the physical address that corresponds to one of the pointers, we can subtract the physical address from the pointer to find the virtual address of the start of the host's direct map; this gives us the ability to translate direct map pointers into physical addresses—allowing us to leak data from any kernel direct map pointer using L1TF.

Though we now have the ability to leak data from any kernel direct map pointer, we do not have any knowledge about the layout of the host's memory, and thus do not know the location of interesting data to leak. Hence, we choose to perform *pointer chasing* on the pointers we have access to through kernel structures in search of the desired sensitive data. By continuously dereferencing pointers and finding new pointers to chase to more data structures in the host kernel, we can eventually end up at valuable structures that contain e.g., root page tables, heap memory mappings, etc. One could, in essence, view the majority of the exploit chain as one long pointer chase throughout the host's physical memory.

As we show in Figure 3, we take our own vCPU's `kvm_lapic` structure (which we know exists at the beginning of `phys_map`) and do a pointer chase from it through the host kernel, only using direct map pointers, in search of interesting structures to leak. In particular, we discover or own `kvm_vcpu` structure, which represents our vCPU, our `task_struct`, describing the host process running our VM, and the VM's address space information in its `mm_struct`. From the latter structure, we leak the location of our task's root page table. With it, we can translate arbitrary host virtual addresses to physical addresses by performing a page table walk using our leakage primitive. Now, rather than being limited to direct map pointers, we are able to leak data from any kernel pointer in the host system.

5.5. Guest Targeting

With the ability to chase any kernel pointer in the host and leak data from it, we now aim to find a victim VM. We start by looking at our own `task_struct`, which is part of a linked list of all tasks in the host; from it, we can leak the process ID and command, i.e., the name of the task, for all processes running on the host. From the task name, we can make an educated guess on which process is in charge of running another VM. Having selected a task that runs a VM, we locate the structures that represent its (open) files, and find the file that KVM hands out to represent the VM. Using it, we discover the victim VM's metadata: the `kvm` and `kvm_vcpu` structs.

From the metadata, we leak the root of the victim VM's extended page tables, as well as the current value of its CR3 register, which is a physical address at which the root page table used by the guest itself resides; this root page table can thus be of any given process currently running in the guest.

Together, these enable us to perform a two-dimensional page table walk, translating guest virtual addresses to host physical addresses. With that, we can now chase pointers within the guest VM's kernel and leak their data via L1TF.

However, we do not yet know where to start our pointer chasing to discover interesting structures. For that, we need to break KASLR in the guest kernel. Using the root page table we leaked earlier, we read the kernel's address space layout, which is possible because every process maps the upper kernel range of the virtual address space, and find the start of the kernel's text and the start of the guest kernel's direct map. With the former, we can start our pointer chase in the guest kernel, while with the latter we can skip most of the costly page table walks in the case of (common) guest pointers inside the direct map.

5.6. Data Extraction

Armed with knowledge of the victim guest's kernel layout, we start by leaking its `init_task` struct, and from there traverse its tasks to find a promising victim process handling sensitive data. If the targeted victim process runs as a system service, we can apply a further optimization: We

first find `systemd` as the first child of `swapper`, and then only traverse the children of `systemd` to locate the victim process. We note that a similar optimization is possible for the prior phase in the attack where we locate the victim VM: by abusing the tree structure of the process tree, one could search for the victim VM inside the host kernel more efficiently. However, we did not implement this.

After we located the victim process, we leverage knowledge about the target's process layout to continue the attack. Consider Nginx as example. By inspecting its `mm_struct`, we recover its root page table and the virtual address of the start of its heap memory; using the former, we determine the physical address of the latter to enable data leakage from Nginx's heap. Using the knowledge that Nginx stores its private key at a static location on its heap, we leak the two prime numbers that comprise the RSA key and reconstruct the private key using the RSA protocol. Note that, even if we did not know the key's location, it can be brute-forced since the two prime numbers are surrounded by two "magic" numbers (PEM-format tags), making it easy for us to recognize them.

For the purpose of reliability, we implement an additional check to ensure that we leak all 2048 bits of Nginx's key correctly. To do so, we exploit the sparsity of prime numbers: if we make an error during key leakage, the resulting (large) number will with a very high likelihood not be prime. Hence, by checking if the leaked number is a prime number, we are able to verify our leakage results.

6. End-to-End Exploit

6.1. Experimental Setup

To evaluate our L1TF Reloaded exploit, we select three platforms as our testbeds. First, we use a 6.13 host base Linux kernel on a local Skylake CPU system to host our malicious VM. Second, we rent a dedicated host system of the N1 instance type from the Google Compute Engine (GCE) part of the Google Cloud Platform (GCP) [27]; these instances are reported to run on Sandy Bridge, Ivy Bridge, Haswell, Broadwell, or Skylake CPUs—which are all vulnerable to L1TF. Third, we rent a dedicated host system of the C5 instance type at Amazon Web Services (AWS) [28]; these instances are reported to run on Skylake and Cascade Lake CPUs—of which only the former is vulnerable to L1TF. We note that we did not have any issue acquiring a machine backed by a vulnerable Skylake CPU on AWS for our experiments.

Next, for the proprietary cloud instances, we determined that these systems are vulnerable to L1TF by first selecting the instance type corresponding to a vulnerable Intel CPU (e.g., Skylake) as per the above, and then verifying the processor type through microarchitectural measurements. We then profile the host system for present mitigations, and find that L1D flushing and SMT are enabled. This means that our attack will not be impeded by any mitigations present, as SMT is not disabled, and we can assume that EPT is also available (as disabling it is not cost effective in cloud environments).

Our end-to-end attack is specifically tailored to target KVM-based hypervisors, which we know are active in the GCE and AWS clouds. However, since the L1TF vulnerability is a CPU bug (and is thus present on all systems running on such CPUs), and a half-Spectre gadget is a common code pattern that is not unique to Linux, the L1TF Reloaded attack technique should work on any host system with a controllable hypervisor gadget. That is, as long as the host CPU is vulnerable to L1TF, and the attacker can find and control a usable half-Spectre gadget in the hypervisor to load any desired data into the L1 data cache.

Given that our attack evaluation occurs on a dedicated host system (to ensure no actual cloud customer is at risk), we ourselves must create a victim VM to perform our exploitation efforts against. To establish a representative target, we spawn a victim VM on the host system that runs an Nginx web server, and aim for our attacker to leak the victim's private RSA key of its TLS certificate. With this certificate, the attacker would be able to impersonate the web server to any desired malicious ends. Note that the attack works against any in-memory guest secret; we picked this particular one to show concretely how we can leak a high-value target.

6.2. Host Kernel Reverse Engineering

To discover the data we need to leak to implement our end-to-end exploit, we perform pointer chasing through kernel structures on the host kernel. However, we as an attacker have no a priori knowledge concerning the exact host kernel version used, its configuration, and possible patches that a cloud vendor may have implemented in the production environment. In order to perform the required pointer chase through the host kernel, we need to determine the offsets of some specific fields in the structures we traverse. This requires some manual reverse-engineering efforts.

On an up-and-running mainline base Linux host kernel, we dump all the relevant data structures at runtime. Next, on an unknown proprietary cloud version of Linux, we use our leakage primitive to recover (parts of) the same data structures. Since we have acquired the offsets for the structures in the base Linux kernel already, we can compare the leaked data and make educated guesses about where the unknown kernel's data resides. We illustrate the aforementioned with a simple example in Figure 4.

Not all structures are straightforward to reverse engineer, however. Some structures have fields we aim to target that are thousands of bytes removed from their position in the base Linux kernel, and others are annotated with randomization compiler attributes; these require more manual effort to reverse and in some cases pointer chases to discover. Regardless, we were able to determine all offsets required to start the attack on both the GCE and AWS clouds.

	mainline Linux	GCE	AWS
kvm_lapic + 78	3e8	695	7ba
kvm_lapic + 80	1	8000000000000000	8000000000000000
kvm_lapic + 88	2	100000000	1
kvm_lapic + 90	ffffa03509eec600	10	2
kvm_lapic + 98	100010101	ffff9352eff70e40	ffff93e461d18000
kvm_lapic + a0	ffffffff	100000101	100000101
kvm_lapic + a8	ffffa035f6c5e000	ffffffff	ffffffff
kvm_lapic + b0	0	ffff934164541000	ffff93e461274000
kvm_lapic + b8	0	0	0

Figure 4: Example of reversing the offset of `struct kvm_lapic`'s `vcpu` field (highlighted for mainline Linux), by leaking the `kvm_lapic`'s data on GCE and AWS. The obvious (and correct) guess here for both AWS and GCE is 0x98.

6.3. Chasing-and-Checking Pointers

Potentially, the leakage primitive may be unreliable, and the leaked values could contain errors. As a result, if only a single error transpires during the exploit's chain of pointer chases, the attack fails. To overcome this issue, we do a sanity check after every a few pointer chases, where we consider whether the new location we arrived at is consistent with what we expect; we refer to this as chasing-and-checking. A simple example can be given by chasing down a doubly linked list: after every traversal of the next pointer, you also leak the previous pointer, and check that it indeed points to your previous location. We do this throughout the exploit, with different checks tailored to different places throughout the exploit's long chase.

6.4. Implementation

We implemented a prototype of our end-to-end exploit in C. On the host system, it runs in a VM with Linux kernel 6.12.4 with the patch from Section 5.1 applied. Our prototype, at a high level, consists of two separate components: one that triggers the half-Spectre gadget to load data into L1 data cache, and one that utilizes L1TF to leak the data from the same L1 data cache on a sibling SMT core. Both parts run simultaneously on different vCPUs of the same physical core in the host system. This follows from the scheduling policy we found is active for both GCE and AWS hosts, where a guest VM occupies all vCPUs of the physical core it is scheduled on.

We will release the code of our prototype in correspondence to the coordinated disclosure with the targeted cloud provider parties – Google and AWS.

6.5. Performance Evaluation

Leakage Primitive. Using our leakage primitive, we successfully leak bytes of memory on a local Skylake server running mainline 6.13 host Linux as its host kernel, as well as on a AWS C5 node and a GCE N1 node. To evaluate the reliability of our leakage primitive, we set up an experiment that measures its leakage rate and accuracy, for leaking chunks of both 8 bytes and 256 bytes, representative of

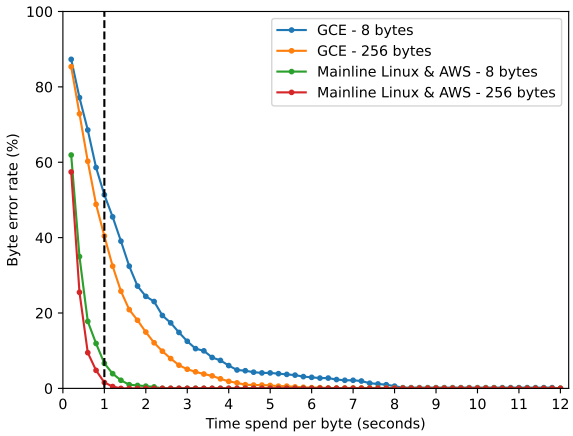

Figure 5: The byte error rate, i.e., the chance of a byte being leaked incorrectly, as a function of the time we spend leaking each byte. Our measurements on mainline Linux and AWS were so similar that we merged their data points.

leaking pointers and leaking a 2048 bit private RSA key. The results are shown in Figure 5. The virtual line indicates the setting used in our exploit: about 1 B/s, resulting in roughly a 95% accuracy on mainline Linux and AWS, and a 50% accuracy on GCE. We confirm with Google that this reduction in accuracy on GCE machine instances is caused by a defense they implement that is not public; at the time of writing, they were not able to share any further information about said defense. As we show in this paper, this reduced accuracy is not a fundamental problem, as we are still able to launch our end-to-end attack and leak sensitive guest data successfully.

Experimental Results. The full exploit chain only works on mainline Linux in a local setup and machine instances in the Google cloud (GCE) of type N1. On the C5 machine instances of AWS, defenses in depth ensure that sensitive data is unmapped from the host system, meaning we can only leak non-sensitive data from the hypervisor. To evaluate our exploit in the most realistic and representative setting, we assess its operation on GCE specifically. In our setup, one dedicated host runs two VMs: one (idle) 2-vCPU victim VM running Nginx, and one 2-vCPU attacker VM. Whereas we developed the exploit on one GCE dedicated host, attacking a Ubuntu 24.04 victim running the corresponding default Nginx web server, for the evaluation we spawned an additional five dedicated hosts with the default VM GCE suggests (Debian 12 Bookworm) as a victim, and its default Nginx.

We ran the exploit for total of 28 runs on the aforementioned six different physical hosts, out of which 25 completed successfully. Each successful exploit run leaked the entire private key correctly.

More specifically, among the successful exploit runs, the average run time was 14.2 hours (standard deviation: 16.2 hours), which was spent as follows:

- Find Gadget Base: 10.9h (76.3%)
- Find Victim VM in Host: 2.6h (18.3%)
- Find Victim Nginx in Guest: 0.3h (2.2%)
- Leak Nginx's TLS key: 0.4h (3.2%)

Note the large standard deviation: the initial step of finding the gadget's base can take half an hour if we are lucky, or 3 days if we are not.

The aforementioned evaluation of the exploit chain was performed on dedicated hosts that were mostly idle. Now, we aim to verify whether the attack also works as effectively when large amounts of system noise are present, as it would be on a real shared cloud. To do so, we must put the system under extreme memory (i.e., cache) pressure, and perform intensive I/O operations continuously. Note that CPU-intensive workloads on other cores do not influence the attack, whereas one could possibly make the argument that many interrupts from I/O could disrupt the exploit's flow/synchronization, or that cache pressure could disrupt its cache side channel used to exploit L1TF.

For this experiment, we fill one entire dedicated host with the maximum number of vCPUs, out of which a third is constantly copying hundreds of gigabytes of files (disk I/O), a third is constantly downloading huge files from an external server (network I/O), and the last third is pressuring the memory subsystem. In particular, the last one third consists of 32 vCPUs, of which 16 are trashing the (last-level) cache by traversing their own 1 GB of memory in a tight loop, and 16 are triggering large amounts of cache-coherency traffic by simultaneously reading and writing to 128 shared cachelines in a tight loop. Lastly, from an external server, we access the victim Nginx web server 100 times per second.

First, we repeated the experiment that produced the results in Figure 5, and found no significant differences. Next, we repeated the exploit on GCE 10 times, which succeeded all 10 runs by leaking the key correctly every time. For these runs, the average run-time was 15.2 hours (with a standard deviation of 9.9 hours) – showing no significant change from the runs on the idle system. From this, we conclude that our exploit is robust under extreme system noise.

7. Mitigations

With the L1TF Reloaded attack described in this paper, we demonstrated that L1TF—7 years after its discovery—can still be used to leak sensitive data from other VMs in a host system, despite the commonly deployed software mitigations in software.

7.1. Deployed Mitigations

Fully mitigating L1TF on vulnerable CPUs, as described earlier (Section 2.2) requires disabling either SMT or EPT, in combination with L1D flushing on every `vmresume`. However, both the former mechanisms remain active by default in Linux, as they can significantly affect the performance of a system [52]. The commonly enabled software countermeasures are L1D flushing and core scheduling, which, as mentioned previously, do not mitigate the guest-to-host attack scenario, and thereby do not impede our attack by themselves if we assume that the hypervisor can be made to load sensitive data (as we did using a half-Spectre gadget).

Our evaluation of the attack on the AWS cloud showed that some cloud vendors implement defenses in depth that mitigate our guest-to-guest exploit. Namely, we discover that the hypervisor of the AWS machine instances implements security countermeasures that ensure any sensitive guest data is not present in the host. These include, as per AWS, eXclusive Page Frame Ownership (XPFO) [53], [54] and process-local memory [55]. This, in combination with the existing L1D flushing and core scheduling mitigations, leaves us unable to leak any guest data from the system using L1TF and the half-Spectre gadget, only allowing us to leak non-sensitive data from the host.

As long as transient execution vulnerabilities are not addressed comprehensively (in software), novel or yet-to-be-practical combinations of their exploitation may still be discovered, and when they do the risk remains that the defenses deployed by a cloud provider are not sufficient to stop them (as was the case here for GCE).

7.2. Possible Defenses

The guest-to-host attack scenario has been known to be theoretically possible, as shown by prior acknowledgements in the Linux kernel documentation [52]. For this reason, several defensive measures have been proposed that range from highly targeted to more generalized approaches. Since replacing all vulnerable CPUs and hardening our microarchitectures against transient execution attacks fully is unrealistic, we discuss some of these approaches as mitigations against our attack.

Half-Spectre Gadget Scanning. One straightforward strategy is to remove all half-Spectre gadgets from the host kernel. If no gadgets exist, the attack we describe in this paper is no longer possible. However, as the code pattern that constitutes them is simply an array access with a bounds check on its index—an extremely common operation—it is not likely that one can completely avoid introducing them into code. In addition, automated Spectre gadget scanners [12], [56], [57], [58], [59], [60] cannot guarantee that all potential gadgets can be found, and in many cases do not offer an analysis of their potential exploitability. Therefore, to safeguard against our guest-to-host attack, we need to guarantee sensitive data is inaccessible to the hypervisor despite the presence of half-Spectre gadgets in the host.

Strict Core Scheduling. To block guest-to-host attacks, a hypervisor could implement a stricter form of core scheduling. This defensive measure entails effectively disabling SMT during critical sections of execution. In our case, that would mean that whenever a process on one vCPU enters the hypervisor context, its sibling vCPU is suspended; only when the former exits the hypervisor context will the latter return to its prior guest execution. This avoids the situation

in which SMT sibling cores run in different security domains, preventing vulnerabilities such as L1TF from leaking sensitive data when combined with L1D flushing [61]. Unfortunately, this strategy suffers from poor performance [62].

Kernel Core Isolation. Next, we consider kernel core isolation as proposed by Quarantine [63]. The authors describe isolating privileged and unprivileged execution on different physical cores, which become privileged and unprivileged cores. As a result, guests will run only on their own unprivileged cores, and will never be co-located with the kernel; any privileged operation, such as a VMEXIT, will be delegated to a privileged core. This prevents both the guest-to-guest and guest-to-host attack scenarios, as different security domains are always isolated to separate physical cores. To circumvent this measure, a cross-core transient execution attack is required. Though Quarantine is a comprehensive mitigation against all on-core leaking transient execution attacks, it is not reported how it performs when the hypervisor runs more than a single virtual machine.

Address Space Isolation. By default, the Linux kernel contains a direct map: a section in virtual memory that is a linear one-to-one mapping to physical memory. Because all memory is accessible via the kernel's direct map, all guest VM memory is thus accessible to the hypervisor and vulnerable to guest-to-host attacks, as we showed with our attack. Address space isolation (ASI) is a defense-in-depth that tackles this issue by creating restricted address spaces that unmap pages that could contain sensitive data, such as the memory of a VM; when a guest VM's memory is no longer mapped in the host kernel, then the hypervisor cannot access it either architecturally or speculatively. Compared to the similar defense XPFO [53], [54], ASI is more generic, as the former exclusively unmaps in-use user memory pages from the kernel's direct map.

Taking the different proposals and implementations of ASI for KVM [62], [64], [65] as an example, kernel address spaces under ASI are restricted to only contain mappings of code and data that are not sensitive. When the kernel does need to access sensitive data, a page fault is triggered in which it switches back to the full kernel address space, enabling the application of (expensive) mitigations only when necessary. In the case of L1TF, the kernel can suspend all sibling vCPUs to ensure no untrusted guest code runs on the same core while it switches to the full kernel address space and loads sensitive data into the L1 data cache; in combination with core scheduling, this protects against guest-to-guest and guest-to-host attacks. Currently, development is ongoing for an ASI implementation in the Linux kernel [65], and it is already partially deployed on Google Cloud's hypervisor and is planned to be deployed fleet-wide from 2025 onward [65], [66].

Secret-Free Hypervisor. Going a step further than ASI, a secret-free hypervisor creates several address spaces with different levels of secrecy, which are combined into a minimal per-vCPU address space that does not contain any secrets [61]. Implementing a hypervisor with this defense-in-depth makes the guest-to-host attack pointless, as the hypervisor no longer has access to secret data. As this approach is similar to address space isolation, it still requires core scheduling to mitigate guest-to-guest attacks.

The reported overhead of the existing secret-free hypervisor work is comparable to a baseline without any mitigations, only encountering a significant performance pentalty when a core has to switch context to a different CPU. However, the performance impact of enabling core scheduling on a secret-free hypervisor is not discussed.

8. Discussion

Applicability to Other Cloud Environments. With our end-to-end attack, we show that a combined exploitation of L1TF and a half-Spectre gadget in the hypervisor can leak data from guest VMs on a host system, and is practical in realistic commercial cloud settings. In particular, our attack can leak non-sensitive host data from C5 cloud instances of AWS, and that it can leak sensitive guest data from all VMs in the system on N1 cloud instances of GCE. While the gadgets used in this paper are specific to KVM (and thus KVM-based proprietary hypervisors), other hypervisors are likely to expose similar half-Spectre gadgets, enabling this attack. As long as L1TF is not mitigated in hardware, and the hypervisor can be made to load arbitrary data into the L1 data cache, an attacker can leak arbitrary host and/or guest VM data. As older CPUs vulnerable to L1TF are still part of the fleets of many commercial cloud providers and half-Spectre gadgets consist of code patterns common in virtually all software, cloud providers other than AWS and Google could be at risk.

Prevalence of L1TF Vulnerable CPUs. A main requirement for our attack is that the host system's CPU is affected by L1TF, a vulnerability known for 7 years with hardware mitigations implemented in newer CPUs. Yet, cloud providers expect long life times of their hardware, leading to a substantial number of hosts in their fleet where L1TF requires mitigation in software. Unfortunately, public data on fleet composition or instance popularity is not available for major cloud providers. Thus, we cannot accurately determine how many systems would be affected by our attack.

However, to estimate the prevalence of vulnerable systems, we inspect the specification of rentable instance types in public clouds as proxy. For AWS general purpose EC2 instances, out of 10 Intel-based instance types, 4 may be backed by L1TF affected CPUs [28]. Similar, out of 20 Intel-based instance types available in GCE, 4 may use according CPUs [27]. Hence, we expect that a substantial amount of L1TF-affected CPUs are actively in-use to this day.

9. Related Work

In this section, we discuss work related to the profiling of CPU microachitectures and mitigations, and the use of L1TF in transient execution attacks.

9.1. Profiling CPUs and Detecting Mitigations

Prior work has shown the possibilities of identifiying CPU microarchitectures and information about microarchitectural components in non-native attack scenarios, such as web browsers. For instance, Trampert et al. [67] present side-channel-related benchmarks that reveal CPU properties relevant to transient execution attacks, such as cache sizes or cache associativities, from the browser. Similarly, Saito et al. [68] show methods that help determine, among others, the number of CPU cores, presence of SMT, and CPU family type. While the aforementioned work focuses on browsers, we in our work target a virtualized attack scenario.

Rather than investigate the microarchitectural properties of a CPU directly, other work aims to directly determine a system's susceptibility to particular transient execution attacks. GhostBuster [69], for example, evaluates existing tools that check a system's vulnerability against transient execution attacks. The authors classify such tools into two categories: information gathering tools, which rely on system information sources such as CPUID and MSRs to map available mitigations [70], [71], and empirical tools, which emulate specific attacks to determine whether the transient execution attack is possible (and thus not mitigated) [45], [72]. For tools in the latter class, they note that one can use either performance monitoring counters [72] or side-channel attacks [45] to do so. In general, both classes of tools do not consider the virtualized attack scenario, as there CPUID and MSR access can be emulated and performance monitoring counters are rarely virtualized.

9.2. L1TF

The Foreshadow attack [7], [13], later described by Intel as L1 Terminal Fault (or L1TF) [38], [39], demonstrated that transient execution attacks could not only breach the user-kernel boundary [2], but also security boundaries such as Intel SGX [7] and the separation between virtual machines [13], [73]. This led to the implementation of mitigations such as core scheduling [15], [42] and L1D flushing [38], [41] to prevent L1TF as it was originally described.

Prior work showed how one could exploit L1TF despite the aforementioned mitigations [24], [25], [26]. Manthey et al. found half-Spectre gadgets in the Xen hypervisor, and described how they could potentially be exploited in tandem with L1TF [24]; specifically, to use the gadget to bring data into the cache on one hyperthread while L1TF is used on another to read the cached data. As a follow-up to the aforementioned, Stecklina demonstrated this attack by introducing an artifical half-Spectre gadget, showing how it bypasses the mitigations against L1TF in the Linux kernel [26]. While this was a synthetic attack that served to validate the deployed defenses, we implement the first end-to-end attack that exploits this combination of transient execution vulnerabilities on unmodified hypervisors. Next, Schwarzl et al. [25] show an attack on a Linux kernel without retpoline [74], where the attacker exploits attacker-controlled addresses remaining in general-purpose registers and their speculative dereferences in the kernel upon interrupt-induced context switches. In this manner, data from the L3 data cache is loaded into the L1 data cache, and can be leaked using L1TF. Whereas we in our attack trick the hypervisor into loading any desired sensitive data, the attacker here has no control over what data the host accesses, and relies on the host continuously accessing sensitive data to ensure it remains in the shared L3 cache.

10. Conclusion

In this paper, we have shown how two different transient execution vulnerabilities, L1TF and half-Spectre gadgets, can be combined into a powerful attack primitive. A malicious VM can bypass existing mitigations against guest-to-guest leakage attacks by using a half-Spectre gadget in the hypervisor. The gadget in the hypervisor lets it act as a confused deputy, loading host physical memory into the L1 data cache. On a sibling SMT core, the malicious VM can then use L1TF to leak the host physical memory from L1 data cache, potentially leaking memory from other guests on the system. With that, we demonstrate an end-to-end, guest-to-guest attack on base Linux and GCP cloud instance machines that leaks a private key from an Nginx web server running in a victim VM in an average time of 14.2 hours.

With our attack, we demonstrate that mitigating transient execution vulnerabilities in isolation is not effective when their exploitation can be combined to not only circumvent existing defenses but yield powerful attack primitives. Mitigations such as XPFO [53], [54] and process-local memory [55] (as shown by AWS), and proposed mitigations such as address space isolation [62], [64], [65], [75] or a secret-free hypervisor [61], would have prevented this attack from occurring. However, not comprehensively addressing transient execution vulnerabilities leaves room for novel combinations of exploitation to rear their heads.

Availability. We will make the prototype of our attack publically upon acceptance of this paper.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: reading kernel memory from user space," in *USENIX Security*, 2018.

[3] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS*, 2018.

[4] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *IEEE S&P*, 2020.

[5] J. Wikner and K. Razavi, "Retbleed: Arbitrary speculative code execution with return instructions," in *USENIX Security*, 2022.

[6] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the Spectre era," in *CCS*, 2020.

[7] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[8] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *IEEE S&P*, 2019.

[9] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on Meltdown-resistant CPUs," in *CCS*, 2019.

[10] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[11] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks," in *USENIX Security*, 2022.

[12] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, "InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2," in *USENIX Security*, 2024.

[13] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution," 2018.

[14] M. Larabel, "The brutal performance impact from mitigating the LVI vulnerability," https://www.phoronix.com/review/lvi-attack-perf, 2020.

[15] The Linux kernel development community, "Core scheduling," https://docs.kernel.org/admin-guide/hw-vuln/core-scheduling.html.

[16] Kasperksy, "Spectre vulnerability: 4 years after discovery," https://www.kaspersky.com/blog/spectre-meltdown-in-practice/43525/, 2022.

[17] L. Torvalds, "Linus torvalds growing frustrated by buggy hardware, theoretical cpu attacks," https://linux.slashdot.org/story/24/10/21/1533228/linus-torvalds-growing-frustrated-by-buggy-hardware-theoretical-cpu-attacks, 2024.

[18] P. Turner and P. G. Parseghian, "Protecting against the new "l1tf" speculative vulnerabilities," https://cloud.google.com/blog/products/gcp/protecting-against-the-new-l1tf-speculative-vulnerabilities, August 2018.

[19] L. Torvalds, "Linus torvalds on current meltdown/spectre patches—"somebody is pushing complete garbage for unclear reasons."," https://www.reddit.com/r/Amd/comments/7s3rnr/linus_torvalds_on_current_meltdownspectre_patches/, 2018.

[20] ——, "Linux kernel mailing list [patch 0/7] ibrs patch series," https://lkml.org/lkml/2018/1/4/720, January 2018.

[21] B. Toulas, "New apple cpu side-channel attacks steal data from browsers," https://www.bleepingcomputer.com/news/security/new-apple-cpu-side-channel-attack-steals-data-from-browsers/.

[22] R. Lakshmanan, "Researchers expose new intel cpu flaws enabling memory leaks and spectre v2 attacks," https://thehackernews.com/2025/05/researchers-expose-new-intel-cpu-flaws.html.

[23] K. Zetter, "Intel fixes a security flaw it said was repaired six months ago, asks researchers to not reveal flaws despite grace period," *New York Times*, November 2019.

[24] Manthey, Norbert and Stecklina, Julian and Wieczorkiewicz, Pawel, "XSA-289 - Xen Security Advisories," https://xenbits.xen.org/xsa/advisory-289.html, 2019.

[25] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss, "Speculative Dereferencing of Registers: Reviving Foreshadow," 2021.

[26] J. Stecklina, "l1tf-demo," https://github.com/blitz/l1tf-demo, 2019.

[27] G. Cloud, "General-purpose machine family for Compute Engine," https://cloud.google.com/compute/docs/machine-resource.

[28] AWS, "Amazon EC2 Instance types," https://aws.amazon.com/ec2/instance-types.

[29] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.

[30] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE S&P*, 2019.

[31] C. Percival, "Cache Missing for Fun and Profit," http://www.daemonology.net/papers/htt.pdf, 2005.

[32] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *Proceedings of the 24th USENIX Security Symposium*, ser. SEC '15. USENIX Association, 2015, p. 897–912.

[33] E. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks Using the Return Stack Buffer," in *WOOT*, 2018.

[34] MITRE, "CVE-2018-3639, Spectre Variant 4: Speculative Store Bypass (SSB)," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639, 2018.

[35] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *ESORICS*, 2019.

[36] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. Lai, "Sgxpectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution," in *EuroS&P*, 2019.

[37] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[38] Intel, "L1 terminal fault," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-l1-terminal-fault.html.

[39] ——, "L1 terminal fault," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1-terminal-fault.html.

[40] Intel, "Affected Processors: Guidance for Security Issues on Intel Processors," https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html.

[41] The Linux kernel development community, "L1D Flushing," https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1d_flush.html.

[42] J. Corbet, "Core scheduling," https://lwn.net/Articles/780703/, 2019.

[43] A. Chartre, "Re: [RFC PATCH 00/47] Address Space Isolation for KVM," https://lore.kernel.org/lkml/91dd5f0a-61da-074d-42ed-bf0886f617d9@oracle.com/, 2022.

[44] B. Jackman, "Re: [PATCH v2 19/35] Documentation/x86: Document the new attack vector controls," https://lore.kernel.org/lkml/CA+i-1C1zN_GcLagTRgfJqT6uFoZaMZj1NUfxkvP7eG=VGQ0GGQ@mail.gmail.com/, 2024.

[45] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019, Extended classification tree and PoCs at https://transient.fail/.

[46] Intel, "Microarchitectural Data Sampling (Disclosure Documentation 758366, v3)," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html, 2021.

[47] M. Schwarz, "Pteditor," https://github.com/misc0110/PTEditor.

[48] Intel, "Intel. 2023. Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4," 2023.

[49] The Linux kernel development community, "Linux Memory Management," https://www.kernel.org/doc/html/latest/arch/x86/x86_64/mm.html.

[50] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution," in *IEEE S&P*, 2023.

[51] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port Contention for Fun and Profit," in *IEEE S&P*, 2019.

[52] Linux kernel documentation, "L1TF – L1 Terminal Fault," https://docs.kernel.org/admin-guide/hw-vuln/l1tf.html.

[53] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *USENIX Security*, 2014.

[54] J. Corbet, "Exclusive page-frame ownership," https://lwn.net/Articles/700647/, 2016.

[55] M. Hillenbrand, "Process-local memory allocations for hiding KVM secrets," urlhttps://lwn.net/Articles/791069/, 2024.

[56] J. Zomer and A. Sandulescu, "Finding gadgets for CPU side-channels with static analysis tools," https://github.com/google/security-research/blob/master/pocs/cpus/spectre-gadgets/README.md.

[57] D. Carpenter, "Smatch check for Spectre stuff," https://lwn.net/Articles/752409/.

[58] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing spectre-type vulnerabilities to the surface," in *USENIX Security*, 2020.

[59] Z. Qi, Q. Feng, Y. Cheng, M. Yan, P. Li, H. Yin, and T. Wei, "SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets," in *NDSS*, 2021.

[60] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: scanning for generalized transient execution gadgets in the Linux kernel," in *NDSS*, 2022.

[61] H. Xia, D. Zhang, W. Liu, I. Haller, B. Sherwin, and D. Chisnall, "A Secret-Free Hypervisor: Rethinking Isolation in the Age of Speculative Vulnerabilities," in *IEEE S&P*, 2022.

[62] A. Chartre and C. Wilk, "Improve Security with Address Space Isolation (ASI)," https://blogs.oracle.com/linux/post/improve-security-with-address-space-isolation-asi, 2019.

[63] M. Hertogh, M. Wiesinger, S. Österlund, M. Muench, N. Amit, H. Bos, and C. Giuffrida, "Quarantine: Mitigating Transient Execution Attacks with Physical Domain Isolation," in *RAID*, 2023.

[64] J. Shahid, "[RFC PATCH 00/47] Address Space Isolation for KVM," https://lore.kernel.org/lkml/20220223052223.1202152-1-junaids@google.com/, 2022.

[65] B. Jackman, "[PATCH 00/26] Address Space Isolation (ASI)," https://lore.kernel.org/lkml/20240712-asi-rfc-24-v1-0-144b319a40d8@google.com/, 2024.

[66] T. L. Foundation, "Address Space Isolation - Brendan Jackman," https://www.youtube.com/watch?v=DxaN6X_fdlI, 2024.

[67] L. Trampert, C. Rossow, and M. Schwarz, "Browser-Based CPU Fingerprinting," in *ESORICS*, 2022.

[68] T. Saito, K. Yasuda, K. Tanabe, and K. Takahashi, "Web Browser Tampering: Inspecting CPU Features from Side-Channel Information," in *Broadband and Wireless Computing, Communication and Applications*, 2017.

[69] A. Mambretti, P. Convertini, A. Sorniotti, A. Sandulescu, E. Kirda, and A. Kurmus, "GhostBuster: understanding and overcoming the pitfalls of transient execution vulnerability checkers," in *SANER*, 2021.

[70] S. Lesimple, "spectre-meltdown-checker," https://github.com/speed47/spectre-meltdown-checker.

[71] VUSec, "mdstool-cli," https://github.com/vusec/ridl.

[72] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: a tool to analyze speculative execution attacks and mitigations," in *ACSAC*, 2019.

[73] M. S. Brunella, G. Bianchi, S. Turco, F. Quaglia, and N. Blefari-Melazzi, "Foreshadow-VMM: Feasibility and Network Perspective," in *IEEE NetSoft*, 2019.

[74] P. Turner, "Retpoline: A Software Construct for Preventing Branch-Target-Injection," https://support.google.com/faqs/answer/7625886, 2018.

[75] J. Corbet, "Another try for address-space isolation," https://lwn.net/Articles/974390/, 2024.