Vulnerability Disclosure Report: "L1TF Reloaded"

Mathé Hertogh, Thijs Raymakers, Mahesh Hari Sarma,
Dave Quakkelaar, Marius Muench, Herbert Bos, Erik van der Kouwe
Vrije Universiteit Amsterdam

May 7, 2025

1 High Level Summary

This document discloses a microarchitectural security vulnerability enabling unauthorized data exfiltration from pubic clouds. We have implemented an attack, with which a malicious cloud customer can read data from the underlying cloud infrastructure, as well as from *other* customers using the same cloud infrastructure. The attacker does not need to know anything, a priori, about its victims: the attack itself can discover which other customers are using the same cloud infrastructure, list the programs they are running, and read their data. We show an example exploit, where a victim customer is hosting a website in a public cloud, and our attack leaks cryptographic data enabling decryption of the website's traffic, and impersonation of the website. While mitigating our particular example attack is easy – this report includes the fix – mitigating the attack *technique* in general is difficult to do effectively as well as efficiently. We encourage cloud providers to take more in depth measurements against this attack surface. The main takeaways are:

- Some microarchitectural mitigations deployed by cloud vendors are insufficient.

  GCE: customer data at direct risk.

  AWS: system security weakened, without direct risk to customer data.

- Open security research is the best way to ensure security. Google and Amazon actively enabled this research (cf. Section 3). The general conclusion should *not* be that Amazon's and Google's security was lacking, but that they are actively stimulating security improvements. We think cloud vendors with closed security policies are putting end users at more risk then, in this case, Google and Amazon are.

2 Technical Summary

In the scope of the ongoing research project "Rain", that investigates the practical exploitability of transient execution vulnerabilities in public clouds, we have found and successfully exploited a combination of two such vulnerabilities: L1TF and half-Spectre. The affected CPUs are therefore restricted to L1TF-vulnerable ones. The half-Spectre gadget we found is in Linux/KVM, reachable from a VM via a hypercall – restricting our PoC to target KVM-based hypervisors. Our PoC achieves arbitrary host memory leakage, as well as either restricted or arbitrary guest data leakage, depending on different cloud vendors their defense-in-depth deployments. On GCE VMs we showcase that, within hours, we can (1) leak what other VMs the host kernel is running; (2) list what processes are running within a victim VM; and (3) leak the private TSL key of an Nginx webserver running inside a victim VM. On AWS, only (1) succeeds. The main technical takeaways are:

- Transient execution attacks are real, and pose serious threats. Although understanding the underlying microarchitectural vulnerabilities requires expert knowledge, and the side-channel building blocks of the attacks are unreliable and slow, one can build reliable and robust exploits on top of them that find and leak critical secrets with perfect accuracy within hours.
- Transient execution attacks are stealthy. Neither Google nor Amazon detected any of our exploit versions, which had been running on their public cloud infrastructure for weeks on end.
- Spectre gadget removal provides *zero* security guarantees. Easy-to-spot gadgets are being (re-)inserted into critical software constantly, let alone deep and generic gadgets, whose detection is an unsolved problem.

3 Coordination & Ethics

Amazon and Google gave explicit permission for this research, provided free-of-charge access to their public clouds, and are participating in the responsible disclosure process. From the point of view of the research team (the authors of this report), the research was done black-box, without any company-insider information: no knowledge of the host kernel, no knowledge of deployed mitigations, etc. We did however get the upfront confirmation that dedicated-hosts, i.e., sole tenant machines, run in the same environment and configuration as shared hosts/machines. Therefore, we conducted all of our experiments and exploitation efforts on dedicated hosts, giving us the guarantee that no customer data could ever be leaked, while preserving a perfectly realistic exploit environment of real world public clouds.

4 Threat Model

We consider the attacker to be a malicious public cloud user, renting VMs and trying to disclose confidential information from the cloud. The main target is obtaining data from other cloud customers. We do not assume the attacker to know who the other customers are, or where in the world they are running their VMs. The attacker will simply spawn VMs, discover what other customers they are colocated with on the same host, discover what they are doing, and leak sensitive data.

5 L1TF Mitigations

L1 Terminal Fault (L1TF) is a transient execution vulnerability affecting older-generation Intel processors. For details on L1TF, we refer to the Foreshadow discovering paper [1] and Intel's whitepaper [2]. L1TF allows a VM to read arbitrary host physical memory *residing in the L1 data cache*. Since L1TF cannot be fixed in microcode, older generation CPUs require software mitigations. The only mitigation preventing the read primitive itself is disabling extended paging, causing major performance overhead. The *goal* of all alternative mitigations is to ensure that, whenever an untrusted guest is running on a CPU core, that core's L1D cache does not contain secret data. To this end, the hypervisor can flush the L1D cache just before every VM-enter. Some kernels implement optimizations, e.g. Linux' "conditional flushing", but this is not important to our attack. Additionally, either SMT must be disabled –a major performance reduction– or the hypervisor must perform core-scheduling: distrusting parties may never run simultaneously on the same core [2,3]. Perfect enforcement of this scheduling constraint requires "kernel synchronization", i.e., one hyperthread's VM-exit forces its sibling to VM-exit as well, also resulting in high performance loss.

6 CPU Targeting & Mitigation Detection

An attacker's first aim is to get VMs running on L1TF-vulnerable CPUs in the cloud, and check whether (in)sufficient mitigations are deployed by the hypervisor. From inside a VM, one does not know a priori what CPU one is running on: the hypervisor can emulate `CPUID` to return a wrong answer. Similarly, despite what the hypervisor tells to the VM about its L1TF mitigation deployments, the truth may be different. Therefore, our attack does not rely on a truthful hypervisor, but seeks answers to these questions independently itself.

Both AWS and GCE allow a user to request VMs of specific instance/machine "types" [4,5]. On AWS and GCE we pick the `C5` and `N1` types respectively, which are both listed to be partly backed by Intel Skylake servers – vulnerable to L1TF. Our attacker VM verifies that it is running on the expected Skylake CPU, by measuring a microarchitectural characteristic of the CPU that varies across

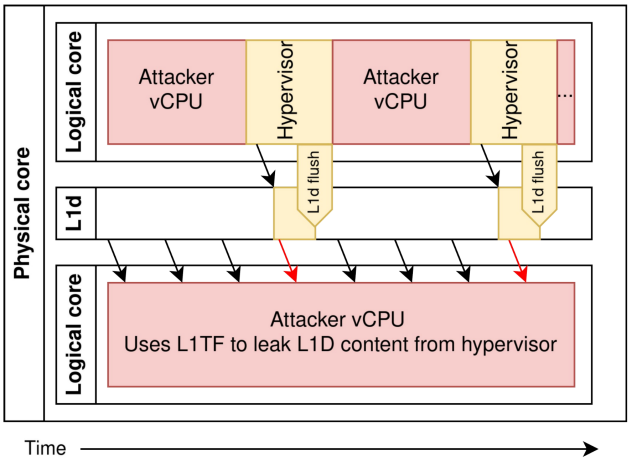

Figure 1: L1D Flushing and Core Scheduling without Kernel Synchronization leave the hypervisor's data vulnerable to be leaked.

different microarchitectures: the amount of global branch history memorized by its branch predictor [6].

Next, we designed timing side-channel techniques that let a VM detect side-effects of different L1TF mitigations. Using a cache side channel before and after VM-exits, we detected that both AWS and GCE deploy Linux' conditional L1D flushing on their Skylake servers. Via a port contention side-channel, we enable two vCPUs to detect their simultaneous co-location on the same physical core. From this, we detected SMT to be enabled, while core-scheduling with kernel synchronization was disabled, on both AWS and GCE.

Although this mitigation configuration is Linux' default, its documentation explicitly states "The administrators of cloud and hosting setups have to carefully analyze the risk for their scenarios and make the appropriate mitigation choices, ..." [7]. The deployed mitigations allow an attack scenario as depicted in Figure 1, as also discussed in Linux' core-scheduling documentation [3], although its practical exploitability is stated to be "debatable".

7 Half Spectre

Under normal circumstances the hypervisor accesses only non-sensitive data, such as the attacker VM's own metadata. Therefore, an attacker as in Figure 1 will have a hard time exploiting this vulnerability beyond much more than for example a host KASLR break. Extraordinary circumstances can however be created by abusing half-Spectre gadgets in the hypervisor. By triggering a half-Spectre gadget in the hypervisor, the attacker may trick the hypervisor into speculatively fetching *any mapped data* into its L1D cache, as depicted in Figure 2.

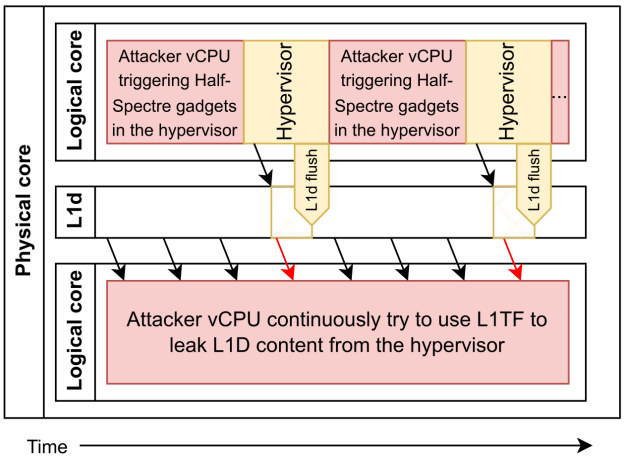

Figure 2: For a short duration of time, in between the hypervisor's speculative load and the next VM-enter of the half-Spectre hyperthread, the secret data will reside in the attacker core's L1D cache. Within this short window, the L1TF hyperthread leaks the secret data out of the shared L1D cache.

With manual code review of Linux' KVM subsystem, and in particular its hypercall handlers, we found the half-Spectre gadget listed in Figure 3. This gadget gives an attacker the primitive to perform a speculative load within the hypervisor's address space. With 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, we can force the host kernel to perform the speculative load `map->phys_map[dest_id]` on *any* (8-byte aligned) 64-bit virtual address.

8 Leak Primitive

The attacker spawns a VM on one physical core, with two vCPUs, corresponding to the core's two hyperthreads – the default on public cloud setups, due to core-scheduling. One vCPU, i.e., one hyperthread, continuously triggers the half-Spectre gadget, while its sibling performs L1TF to leak the contents of the speculatively loaded cacheline, as depicted in Figure 2.

This results in a leak primitive allowing the disclosure of any data that can be speculatively loaded by the half-Spectre gadget, i.e., any memory mapped into KVM's address space during execution of `kvm_sched_yield`. For mainline Linux, as well as GCE's host kernel, this is *all physical memory*, due to the kernel's direct map. On AWS, limitations apply as we will see in Section 9.

Our implementation successfully leaks data on a local Skylake server running a mainline 6.13 host Linux kernel, as well as on a AWS `C5` node, and a GCE `N1` node. We setup an experiment that measures the leakage rate and accuracy of this leak primitive, for leaking chunks of both 8 bytes and 256 bytes, represen-

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

Figure 3: 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`.

tative of leaking pointers and leaking a 2048 bit private RSA key. The results are listed in Figure 4. The virtual line indicates the setting used in our exploit: about 1B/s, resulting in roughly 95% on mainline Linux and AWS, and 50% accuracy on GCE.

9 Additional Cloud Vendor Specific Mitigations

The difference between GCE and mainline Linux/AWS in Figure 4 hints at something additional measures on GCE hampering the leakage signal. Google has acknowledged that they implemented extra, custom mitigations against L1TF, explaining this behavior. We do not know any details of these additional mitigations – neither did we try to find out. The conclusion is however clear: although this may slow down an attack, it will not prevent one, as proven by this research.

During the exploitation of our leakage primitive on AWS, we discovered that the AWS host kernel does not have all of physical memory mapped into its direct map of physical memory. We attribute this to a defense-in-depth, that unmaps physical memory of other guests from the direct map, à la `memfd_secret` [8], and this was confirmed by AWS. Since our leak primitive allows data exfiltration of any memory *mapped* into KVM's address space, during execution of the half-Spectre gadget on our own logical core, this prevents us from directly leaking other guest memory.

This leaves open the question of whether some other guest memory is still mapped into the address space, but outside of the direct map. For example, although an example victim VM's metadata (`struct kvm`) in AWS's host kernel was not mapped in the direct map, and hence our initial attempts at leaking it from another VM failed, it *was* still mapped in the host kernel's vmalloc

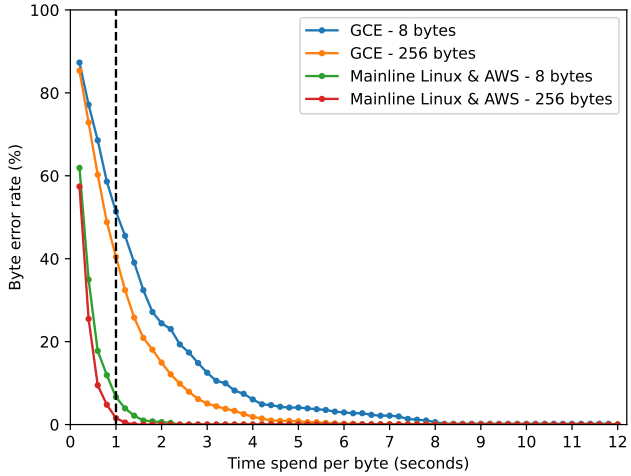

Figure 4: The byte error rate, i.e., the chance of a byte being leaked incorrectly, as a function of the time we spend leaking per byte. Our measurements on mainline Linux and AWS were so similar that we merged their data points.

region, eventually allowing its cross-VM leakage. We leave further analysis of the residual mappings of guest memory in host address space for future research, and encourage Amazon to openly publish the details of this mitigation, as their security hinges on such details.

Lastly, even in the best-case scenario were no other guest memory is mapped in at all, this vulnerability still achieves arbitrary host kernel and host userspace data leakage. This is still a very powerful primitive for an attacker, which, as part of a bigger exploit chain, could enable the exploitation of other vulnerabilities that enable full system compromise but might be hard to exploit, e.g., memory safety bugs.

10 Exploit Overview

Despite the slow and somewhat unreliable leakage primitive, we build a proof-of-concept exploit that reliably leaks a web server's private TSL certificate within hours. The main challenge is *finding* the secret data (e.g., the TSL key) in physical memory – within the hundreds of gigabytes of RAM on cloud servers. We tackle this challenge by abusing the (meta)data resident in the host and victim guest kernels. On a high level, the exploit consists of the following steps:

1. Locate the half-Spectre gadget's base address;
2. Break host KASLR;
3. Gain host address translation capability;

4. List processes running on the host, finding other (victim) VM(s);
5. Gain guest address translation capability;
6. Break guest KASLR;
7. List processes running in the guest, finding Nginx;
8. Leak Nginx's private TSL certificate from its heap.

The whole exploit chain succeeds on mainline Linux and GCE, whereas only steps 1, 2, 3 and 4 succeed on AWS (cf. Section 9). Below we describe each step in more detail.

10.1 Find Gadget's Base in Physical Memory

The gadget's *base address* is `&map->phys_map`. Making use of the page-alignment of the `map` object, that is dynamically allocated upon creation of our VM, we brute force it's physical address as follows. We constantly trigger the half-Spectre gadget with offset zero (i.e., `dest_id` = 0), bringing the start of the `phys_map` array into the L1D cache. On the L1TF sibling, we stride at page granularity through physical memory, guessing the base's physical address. The `phys_map` array holds one pointer to a `kvm_lapic` structure for each vCPU of the VM, in our case 2. A correct guess is indicated by the L1TF-leakage of these two kernel pointers.

This gives us the physical address p of the gadget's base, which enables the arbitrary physical memory read primitive: as the base is `kmalloc`ed and hence points inside the host's direct map, passing a (possibly negative) offset of x to the half-Spectre gadget lets it speculatively load at physical address $p + x$ through the direct map.

As an optimization, we don't actually leak the pointers, but only scan for their top two bytes being all ones. Another optimization is that we don't actually use the half-Spectre gadget listed above, but the "architectural gadget" listed in Figure 5. The half-Spectre gadget's throughput is, on a mostly idle system, very bad due to the `single_task_running` check blocking access to the gadget most of the time – in the order of 99.9%. The architectural gadget is, due to the particular data-flow, not an exploitable half-Spectre gadget, but *is* consistently reachable.

10.2 Break Host KASLR

Next, we do actually leak the full two pointers at the start of `phys_map`. The structures they point to start with easily recognizable data. These are `kmalloc`-pointers, inside the host kernel's direct map, which is backed by 1GB pages. Therefore, we know the lower 30 bits of their physical address. The remaining few physical address bits we brute force: at every possible physical address location, we leak the data, and check for the expected recognizable data. Upon a match, we have found a kernel pointer in the direct map, as well as its physical

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

Figure 5: The architectural gadget used to locate the half-Spectre gadget's base.

address. Subtracting them from each other gives us the start of the host's direct map. Therefore, from now on, we can translate direct map pointers into physical addresses, a crucial capability going forward.

As an aside, knowing start of the direct map also tells us the *virtual* address of the gadget's base. This can be important for using the half-Spectre gadget to target parts of the host kernel's virtual address space *outside* of its direct map, as well see later.

10.3 Gain Host Address Translation Capability

We already know the location of our own vCPU's `kvm_lapic` structure. From here, we do a pointer chase through the host kernel, using *only* direct map pointers, to find some more useful structures. In particular, we find our own `kvm_vcpu` structure, representing our vCPU, our own `task_struct`, describing the host process running our VM, and its address space information in its `mm_struct`. From this last structure, we leak the location of our own task's root page table. This enables us to translate *arbitrary* host virtual addresses, by performing a page table walk with our leakage primitive. In turn, this allows us to chase arbitrary host kernel pointers.

10.4 Find Victim VM

With full host pointer chasing capability, we are ready to find a victim VM. Our own `task_struct` is part of a linked list of *all* host tasks. We simply chase this linked list, leaking the process ID and command, i.e., name of the task, of all processes running on the host. Names such as "qemu", "nanny", "VCPU-", or "dom-", reveal the processes in charge of running (other) VMs.

Given a task running a VM, we locate the structures representing its (open) files. We find the file handed out by KVM that represents the VM, via which we locate the victim VM's `kvm` and `kvm_vcpu` structures.

10.5 Gain Guest Address Translation Capability

From the victim VM's metadata, we leak the root of its extended page tables, as well as its current `cr3` value, pointing to the root page table in use by the guest. Together, these enable us to perform two-dimensional page table walks, translating guest virtual addresses to host physical addresses. In other words, we can now start pointer chasing within the *guest kernel*.

10.6 Break guest KASLR

Where do we start chasing? Well first, we break the guest's KASLR. This boils down to reading its address space layout from its page tables. In particular, we find the start of the guest kernel's text section, as well as the start of the guest kernel's direct map. The former allows us to start our pointer chase in the guest kernel. The latter we merely use to skip most of the costly 2D-page table walks in case of (very common) guest pointers inside the direct map.

10.7 Find Nginx Within Guest

With knowledge of the guest's kernel layout, we leak its `init_task`, and from there, traverse its tasks to find Nginx. As an optimization, we first find `systemd` as the first child of `swapper`, and then only traverse the childeren of `systemd` to find Nginx. A similar optimization, abusing the tree-structure of the process tree, could speed up the search of the victim VM inside the host kernel, but we never implemented that.

10.8 Leak Nginx's Private TSL Certificate

From Nginx's `mm_struct` we read find its root page table. Note that the previous guest root page table we leaked was just some random one that happened to be loaded at leak time. Since any process maps the entire upper kernel part of virtual address space, this sufficed until now. But now, we read the location of Nginx's heap from its `mm_struct`, and we actually need Nginx's page tables to translate that user space address into a host physical address. Nginx stores the private key on a static location on its heap. But even without this: the two prime numbers from which the RSA key is built up, are surrounded by two "magic" numbers (PEM-format tags), making it easy to recognize them. We leak the two prime numbers off of Nginx's heap, and use them to reconstruct the private key (following the RSA protocol).

11 Exploit Reliability: Chase & Check

Ignoring the abstraction levels we're piercing for a moment, one can view the exploit chain as one long pointer chase through host physical memory. This constitutes a problem given that our leakage primitive is unreliable: a *single* error anywhere in the pointer chase and we lose the trail. The classic reliability

solution for side-channel attacks is to trade performance for accuracy: take more side-channel measurements. But if your pointer chase doubles in depth, you need to double your amount of measurements *per pointer* to compensate for that, resulting in quadratic run-time complexity relative to the depth of the pointer chase. As our exploit involves a very deep chase, this becomes infeasible.

Instead, we opt for a strategy which we call "Chase & Check". After every (few) pointer chase(s), we do a sanity check, whether the new location we arrived is consistent with what we expect. A simple example can be given by chasing down a doubly linked list: after every traversal of the `next` pointer, you also leak the `prev` 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.

Although the final data we leak, Nginx's private key, is not part of the pointer chase, we still implement a "check" to ensure we leak all 2048 bits of the key correctly. For this, we abuse the sparsity of prime numbers: if we make an error during key leakage, the resulting (big) number will with very high likelihood not be prime.

12 Host Kernel Reversing

One last technicality we skipped over so far is the fact that, a priori, we as an attacker do not know the exact host kernel version, configuration, and possible patches, that Google and Amazon use in production. In order to perform a pointer chase through the host kernel data structures, we need to know the offsets of specific fields in the structures we are traversing. This requires some reverse engineering efforts.

On an up and running mainline Linux host kernel, we dump all the relevant data structures at run time. Next, on an unknown cloud version of Linux, we use our leak primitive to also leak (parts of) the same data structures. Since for mainline, we know the offsets already, we can compare the leaked data and make educated guesses about where the unknown kernel's data resides. An easy example is given in Figure 6.

	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 6: 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.

Not all structures are as easy as this one of course. We encountered structures where the field we are looking for is thousands of bytes apart from the position on mainline Linux. Also, structures annotated with the `randomize_layout` compiler attribute are harder to reverse, and require some chasing of pointer before conclusions can be drawn. But after a bit of manual labor (which we are quite sure of could be automatized as well), the attacker knows all the required offsets, and is ready for exploitation. We noticed that these offsets stay constant between different hosts on both GCE and AWS, leaving this to be a one-time effort.

13 Exploit Success Rate & Run Time

As noted before, the entire exploit chain only works on mainline Linux and GCE. We evaluate the exploit in the most realistic setting: GCE. The exploit is even faster/more accurate on both mainline Linux (and AWS for the first part of the exploit), cf. Section [**?**]. In our setup, one dedicated host runs two VMs: one (idle) 2-vCPU victim VM 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 webserver, for the evaluation we spawned an additional five dedicated hosts, with the default VM GCE suggests as victim (Debian 12 Bookworm), and its default Nginx. We ran the exploit for multiple days on the six different physical hosts. This resulted in 28 exploit runs, out of which 25 successfully completed. Each successful exploit run perfectly leaked the entire private key correctly. 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 TSL key: 0.4h (3.2%)

Note the big standard deviation: finding the gadget's base can take half an hour if you are lucky, or 3 days if you are unlucky.

The above evaluation was run on dedicated hosts that were mostly idle. Lastly, to validate that this attack would work just as well with lots of system noise, we put the system under extreme memory (i.e., cache) pressure, as well as a lot of I/O. Note that CPU intensive workloads on other cores do not influence our 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-covert channel used to exploit L1TF.

We fill one entire dedicated host with the maximum amount of vCPUs, out of which 1/3 is constantly copying hundreds of gigabytes of files (disk I/O), 1/3 is constantly downloading huge files from an external server (network I/O), and

1/3 is pressuring the memory subsystem. In particular, the last 1/3 consists of 32 vCPUs: 16 are trashing the (last level) cache, by each traversing their own 1GB of memory in a tight loop; and 16 are triggering lots of cache coherency traffic, by all simultaneously reading and writing to 128 shared cachelines in a tight loop. Lastly, from an external server, we access the victim Nginx webserver 100 times per second.

Rerunning the experiment from Figure 4 did not give significantly different results. Running the exploit 10 times, it succeeded 10 times, leaking the key correctly every time. The average exploit run time was 15.2 hours (standard deviation: 9.9 hours) – again, no significant changes. Therefore, we conclude that the exploit is very robust under extreme system noise.

14 Mitigation

Cloud vendors should reconsider which mitigations against L1TF they deploy. Possible options include the mitigations mentioned in Section 5, as well as AWS-like approaches such as `memfd_secret` or Address Space Isolation (ASI) [9]. The last two options are limited in their effectiveness, as they still allow host memory reads, and only protect other guest data. Also, for ASI, one should carefully consider the attack surface left open by ASI-exits. Lastly, there is the strategy of eliminating (half-)Spectre gadgets from the hypervisor. We think this is not a durable solution, as Spectre gadget scanning is an unsolved problem, for which no sound solution is in sight. Or put even worse: there is not even consensus on the definition of a (half-)Spectre gadget yet.

Since the Linux kernel's current mitigation strategy *does* rely on the removal of Spectre gadgets, we will submit a patch to the mainline kernel to remove the half-Spectre gadget used in this exploit, by inserting an `array_index_nospec`, which we expect to have zero performance cost.

References

[1] 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. 5

[2] Intel, "L1 terminal fault," https://www.intel.com/content/www/us/en/developer/articles/technical/softwa security-guidance/technical-documentation/intel-analysis-l1-terminal-fault.html. 5

[3] The Linux kernel development community, "Core scheduling," https://docs.kernel.org/admin-guide/hw-vuln/core-scheduling.html. 5, 6

[4] AWS, "Amazon EC2 Instance types," https://aws.amazon.com/ec2/instance-types. 6

[5] G. Cloud, "General-purpose machine family for Compute Engine," https://cloud.google.com/compute/docs/general-purpose-machines. 6

[6] 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. 6

[7] Linux kernel documentation, "L1TF – L1 Terminal Fault," https://docs.kernel.org/admin-guide/hw-vuln/l1tf.html. 6

[8] Jonathan Corbet, "Two address-space-isolation patches get closer," in *LWN*, 2020, https://lwn.net/Articles/835342. 9

[9] J. Shahid, "Address space isolation for kvm," Feb. 2022. [Online]. Available: https://lore.kernel.org/lkml/20220223052223.1202152-1-junaids@google.com 14