QUARANTINE: Mitigating Transient Execution Attacks with Physical Domain Isolation

Mathé Hertogh[*]
Vrije Universiteit Amsterdam
m.c.hertogh@vu.nl

Manuel Wiesinger[*][†]
Vrije Universiteit Amsterdam
m.wiesinger@vu.nl

Sebastian Österlund[‡]
Intel Corporation
sebastian.osterlund@intel.com

Marius Muench
Vrije Universiteit Amsterdam &
University of Birmingham
m.muench@bham.ac.uk

Nadav Amit
VMware Research
namit@vmware.com

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

ABSTRACT

Since the Spectre and Meltdown disclosure in 2018, the list of new transient execution vulnerabilities that abuse the shared nature of microarchitectural resources on CPU cores has been growing rapidly. In response, vendors keep deploying "spot" (per-variant) mitigations, which have become increasingly costly when combined against all the attacks—especially on older-generation processors. Indeed, some are so expensive that system administrators may not deploy them at all. Worse still, spot mitigations can only address *known* (N-day) attacks as they do not tackle the underlying problem: different security domains that run simultaneously on the same physical CPU cores and share their microarchitectural resources.

In this paper, we propose QUARANTINE, a principled, software-only approach to mitigate transient execution attacks by eliminating sharing of microarchitectural resources. QUARANTINE decouples privileged and unprivileged execution and physically isolates different security domains on different CPU cores. We apply QUARANTINE to the Linux/KVM boundary and show it offers the system and its users blanket protection against malicous VMs and (unikernel) applications. QUARANTINE mitigates 24 out of the 27 known transient execution attacks on Intel CPUs and provides strong security guarantees against future attacks. On LMbench, QUARANTINE incurs a geomean overhead of 11.2%, much lower than the default configuration of spot mitigations on Linux distros such as Ubuntu (even though the spot mitigations offer only partial protection).

[*]Both authors contributed equally to this research.
[†]Contributions were partly made while employed at SBA Research.
[‡]Contributions were made while employed at Vrije Universiteit Amsterdam.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**; • **Software and its engineering → Message passing**.

KEYWORDS

Operating systems, Transient execution attacks

ACM Reference Format:
Mathé Hertogh, Manuel Wiesinger, Sebastian Österlund, Marius Muench, Nadav Amit, Herbert Bos, and Cristiano Giuffrida. 2023. QUARANTINE: Mitigating Transient Execution Attacks with Physical Domain Isolation. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3607199.3607248

1 INTRODUCTION

After the initial Spectre [52] and Meltdown [65] disclosure, many other transient execution attacks have come to light [7, 8, 18, 21, 29, 51, 53, 71, 85, 88, 89, 92, 95–99, 101] and the end is not in sight—many new vulnerabilities and attack variants have been disclosed in this past year alone [3, 48, 84, 87]. Since the vulnerabilities are in the hardware of billions of devices, fixing them is complicated. To minimize disruption, software and hardware vendors keep releasing ad-hoc mitigations that stop specific (*known* or N-day) exploits, but fail to address their root cause and hence protect against future (*unknown* or zero-day) attacks. Moreover, such "spot" mitigations often incur high performance costs [60, 62], especially when used in combination. Such costs permanently affect current/old-generation processors and may be only alleviated (but not eliminated [4]) by upgrading to newer generations with in-silicon fixes—until the next vulnerability is disclosed and new costly N-day mitigations are needed. Moreover, to mitigate the costs, some mitigations are often disabled by default, leaving systems vulnerable.

In this paper, we propose to break the current N-day vulnerability/mitigation cycle and counter zero-day exploits by addressing their root cause: the fact that transient execution attacks are generally enabled by different security domains sharing microarchitectural resources. Exemplary here are MDS attacks, where an attacker can sample a plethora of microarchitectural buffers to disclose arbitrary data on the running core [8, 88, 98]. To this end, we present QUARANTINE, a principled approach to physically isolate different security domains on different CPU cores. Such *physical domain isolation* offers blanket protection against all core-local transient execution attacks across domains, including future ones. By furthermore partitioning the last level cache (LLC) among security domains, we harden the system against cross-core transient execution attacks. Moreover, we aim to investigate the costs of comprehensively addressing the root cause of transient execution attacks in software. We believe that this investigation is essential as baseline for follow-up research.

While physical separation as a mitigation against transient execution attacks applies to any set of security domains, full isolation is nontrivial to achieve. For instance, it is not sufficient to schedule unprivileged applications or VMs on a separate core, as the most security-sensitive domains (kernel/hypervisor), will still run on the same core [40]. If the attacker manages to disclose data from such domains, *all* security guarantees are void. Specifically, compromising the kernel/hypervisor also compromises all user applications/VMs.

Previous efforts to separate unprivileged domains while protecting the privileged domains from malicious users running on the same core were unsuccessful. For instance, in addition to a group scheduler that runs only mutually trusting threads on a single core, Intel proposed a design with synchronized kernel entry and exit—where no thread on the same core runs outside the kernel when another is in the kernel [40]. Given the nontrivial complexity and performance impact, such design was abandoned after initial evaluation by the Linux `coresched` team [14].

In contrast, QUARANTINE moves the privileged code to a separate core, leaving only minimal code performing non-sensitive core-local operations behind. Moreover, on commodity systems, physical separation between privileged and non-privileged code is possible not just between kernel and user, but also between host (hypervisor) and guest (VM). As such, we investigate both options and show that, while performing isolation at the user-kernel level is complex and arguably impractical, the guest-host interface provides a promising target. Indeed, we show that instantiating our design via virtualization-based isolation provides a practical solution to shield VMs but also (unikernel) applications from (un)known transient execution attacks.

We evaluated the resulting solution on kernel and server benchmarks. Our evaluation shows that QUARANTINE incurs a low performance cost, while providing strong (exploit-agnostic) security guarantees with an attack surface reduction of 97.5%.

Summarizing, our contributions are:

- QUARANTINE, a principled approach to mitigate transient attacks across privilege domains by means of physical domain isolation.
- An exploration of the design space, with cost/complexity analysis of applying physical domain isolation to kernel- and virtualization-based isolation.
- Implementations[1] of QUARANTINE for Linux/KVM, both kernel- and virtualization-based.

[1]available at https://github.com/vusec/quarantine

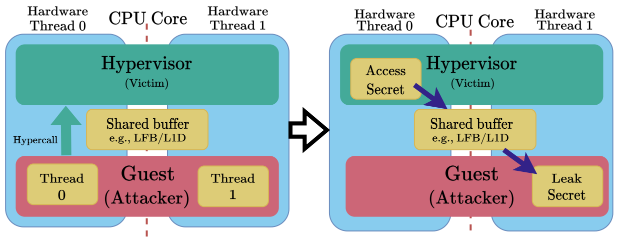


Figure 1: Transient execution attack against the hypervisor. An attacker with co-located SMT threads can steal secrets from the hypervisor by leaking data from shared buffers.

- An evaluation of QUARANTINE, to understand the real cost of a comprehensive solution and demonstrating its strong security guarantees and much lower cost than existing N-day mitigations combined.

2 BACKGROUND

Terminology. We reserve the term *core* to refer to a physical CPU core. By a *CPU* we shall mean a logical CPU core, i.e., a (hyper)thread. A *sibling CPU* is a CPU on the same core.

2.1 Transient Execution Attacks

Transient execution attacks exploit microarchitectural side channels to leak secret data. As such, deploying such attacks requires (i) access to secret data during transient execution that leave microarchitectural traces and (ii) the ability to turn such traces into observable secret-dependent behavior. As an example, Figure 1 depicts a Microarchitectural Data Sampling (MDS) [8, 88, 98] attack, where data is leaked via on-core buffers shared among sibling CPUs.

Attack classifications [7, 84] generally group attacks into two categories: *branch misprediction-based* (or Spectre-type [52]), which rely on speculative execution, and *machine clear-based* (or Meltdown-type [65]), which rely on plain out-of-order execution. Attacks either leak *on-core* or *off-core* data, depending on the location of the microarchitectural component that leaks. Whereas the above properties are unique to each attack, the same attack can be launched in up to three different attack scenarios. *In-domain* attacks leak data from the same domain (from the hardware perspective), circumventing software enforced boundaries, e.g., MDS from javascript against the sandbox. *Domain-bypass* attacks enable an adversary to directly leak data from other domains, e.g., a user process performing MDS to leak kernel data. *Cross-domain* attacks leak data from other domains in a confused deputy style, triggering transient leakage in the victim domain via gadgets inside the victim's code, e.g., a VM triggering hypervisor code that happens to perform MDS. Gadgets consist of two parts: the *trigger gadget* triggering transient execution and transiently leaking secret data, and the *disclosure gadget* encoding the secret into a covert channel. Trigger gadgets for some attacks, say Spectre, are more common to find in commodity software than for others, say MDS. Independent of the above attack scenarios, an attack can be mounted against a victim running on the attacker's core, or a different core, and we call the attack *core-local* or *cross-core* in those situations respectively.

2.2 Mitigating Transient Execution Attacks

Mitigating transient execution attacks typically involves different components, such as microcode, firmware, OS, hypervisor, and applications, thereby making mitigations complex and often brittle. Moreover, each mitigation tends to target only specific exploit variants and adds performance overhead due to the need to, e.g., flush caches [26], limit speculation [81, 94], or partition resources [25]. Hence the combined complexity and performance impact grows over time [4, 59, 60, 62, 78].

A more principled way to address transient execution (and generally side-channel) attacks is to isolate mutually distrusting security domains [16]. However, in practice doing so is challenging, and production deployment has been limited to two use cases: *Site Isolation* [86], which browsers use to isolate web apps in separate processes; and *core scheduling*, which kernels/hypervisors use to isolate processes/VMs in separate cores [14]. However, neither of these techniques isolates the privileged domain (i.e., kernel or hypervisor), the most security critical component of the system. To protect the privileged domain, Intel proposed mode-switch rendezvous [40], ensuring no two sibling CPUs ever run in different privilege domains at the same time, but the performance overhead and complexity has been found to be excessive [9, 12].

2.3 Virtualization

To efficiently support virtualization, hardware extensions, such as VT-x and AMD-V, introduce a new processor mode—*guest mode*. In guest mode, some instructions cause a *VM-exit* (exit from guest mode) to yield control to the hypervisor.

The CPU can enter guest mode by means of a dedicated instruction, such as `VMRUN` on AMD or `VMLAUNCH` on Intel. This performs a *world switch* from host to guest, switching essential registers like the stack and instruction pointers. From that moment, the guest runs directly on the hardware. The host can predetermine what events cause the VM to VM-exit. Some commonly intercepted events are for example interrupts and writes to the CPU control registers. Upon a VM-exit, the hypervisor regains control and solves the exit reason, for example by handling the interrupt or emulating the write to the control register. Once the VM-exit has been handled, the hypervisor can start up the VM again.

The hypervisor maintains a memory resident data structure, called the VM control block (VMCB) on AMD or VM control structure (VMCS) on Intel, to control the VM. Among other things, it can be used to inject virtual interrupts into the guest, to configure which events cause a VM-exit, and to inspect what caused a VM-exit after it happened.

3 THREAT MODEL

We consider a modern system running a state-of-the-art OS/hypervisor, and an attacker seeking to leak confidential across hardware enforced security boundaries, without resorting to any software vulnerabilities. We assume the attacker is capable of running arbitrary unprivileged code directly on the system, either as part of a user application or a VM. The attacker is able to launch arbitrary transient execution attacks. Concretely, we consider application-to-application, application-to-OS, VM-to-VM, and VM-to-hypervisor attack scenario's. Note that this renders in-domain attacks (e.g., sandbox escapes) out of scope; the attacker will have to use either domain-bypass or cross-domain attacks.

4 MOTIVATION

To understand the problem that our work is tackling, we start by getting an overview of the currently known transient execution attacks, as well as their state-of-the-art (spot) mitigations before proposing our solution.

4.1 Spot Mitigations

We analyzed all known transient execution attacks on Intel CPUs and their mitigations. Table 1 provides an overview in chronological attack disclosure order. The next four paragraphs explain each of the four columns respectively.

Known Attacks. Distinguishing different attacks (or attack vectors) from each other is not straightforward. For example, Intel allocated a single CVE for Branch Target Injection (BTI), in which Intel includes both BTB- and RSB-misspeculation attacks, which are commonly classified as distinct attacks [7]. On the other hand, Intel considers Intra-mode BTI a separate attack from BTI, while others do not. Our classification follows Intel.

Default Spot Mitigations. For mitigations the situation is also complicated. Multiple spot mitigations may attempt to defend against the same attack, while others are effective against multiple attacks. As a reference for what the industry uses today, we picked the Linux kernel, as it is security sensitive, supports (almost) all modern CPUs, and is a central piece of modern computing infrastructure. The default spot mitigations applied by Linux depend on the CPU, as many mitigations require hardware (and microcode) support. For example, if eIBRS is not supported, Linux can fall back to retpolines [94] against BTI and call depth tracking [61] against Return Stack Buffer Underflow (RSBU). For clarity, we only listed the best combination of mitigations against each attack.

Full Spot Mitigations. The Full Mitigation column shows the additional mitigations needed, *on top* of the default mitigations, to fully protect against a specific attack. Here we also follow Intel's recommendations, coupled with Linux' needs. Due to the heavy performance overhead of these mitigations, they are not deployed by default.

We colored all of the mitigations based on how well they defend the system according to the vendor's statements. While this results into a consistent classification, we want to stress that the situation may be worse than depicted in Table 1. For example, Intel reports that the Spectre, L1TF and MDS mitigations are sufficient for mitigating LVI, as "an unprivileged adversary has few points of leverage to induce faults or assists into code executing at a higher privilege level" [35]. However, gadget scanners like Kasper [48], showed that users can induce such faults inside the Linux kernel. Mitigation would require buffer flushing upon kernel entry, on top of flushing upon kernel exit as required for mitigations against other attacks. Another example is SCSB, which is deemed harmless by Linux developers, while JITted BPF code or kernel module insertion may provide avenues for SCSB against Linux.

Attack	Default Spot Mitigation	Full Spot Mitigation	Leak Origin
BCB [23, 51, 52]	Bounds clipping, serialization	Full serialization	Mapped memory
BTI [24, 52, 53]	eIBRS, selective IBPB, RSB filling	IBPB always	Mapped memory
RDCL [28, 65]	KPTI		L1D
RSRR [29]	Microcode: stop speculation		System registers
SSB [30]	Selective SSBD	SSBD always	Mapped memory
LazyFP [27, 92]	Eager FPU restore		FPU
L1TF [26, 95, 101]	PTE inversion, conditional L1D flush	L1D flush always, no SMT	L1D cache
MFBDS [32, 88, 98]	Flush buffers	No SMT	FB
MSBDS [8, 32]	Flush buffers	No SMT	SB
MLPDS [32, 98]	Flush buffers	No SMT	LP
MDSUM [32]	Flush buffers	No SMT	FB, SB, LP
SWAPGS [33]	Serialization		Mapped memory
TAA [31, 98]	Flush uarch buffers	No TSX	FB, SB, LP
VRS [38]	Microcode: stop propagation		Vector registers
L1DES [34, 100]	Microcode: stop propagation		L1D cache
LVI [35, 96]			Mapped memory
SAL1DS [36]	No mitigation	L1D flushing, no SMT	L1D cache
SRBDS [37, 85]	Microcode: flush buffers and serialize		Uncore
FPVI [39, 84]			Mapped memory
SCSB [41, 84]			Mapped memory
FSFP [43]	Microcode: unshare FSFP, selective FSFD	FSFD	Mapped memory
BHI [3, 42]	BTI + no user BPF, selective serialization	Unshare BHB	Mapped memory
IMBTI [3, 42]	BTI + no user BPF, selective serialization	No speculation	Mapped memory
SLD [47, 84]	Serialize shared memory access	Serialize shared memory access	Mapped memory
SBDS [45]	Flush buffers	No SMT, flush buffers on MMIO	Uncore
RSBU [46, 102]	eIBRS	INT3 after RET	Mapped memory
PRSBP [44]	RSB filling		Mapped memory

Table 1: Known transient execution attacks on Intel CPUs and their corresponding spot mitigations, as well as their dependence on microarchitectural resources. Color legend: full mitigation , code audit dependent mitigation , partial mitigation , no mitigation , on-core leak origin , off-core leak origin .

Furthermore, is is questionable how future-proof the mitigations are, as many attacks in recent years comprised the "full" spot mitigations available at the time. Examples of previously compromised spot mitigations include eIBRS [3], retpoline and RSB filling [102], `VERW` buffer flushing [100], and `lfence/jmp` [74].

Leak Origin. For each attack, we also show the source from which data leaks in Table 1. Many of the attacks can leak any data that the victim core has legal access to. In particular, this memory must have been mapped by some CPU on the core at some point in time—possibly before the attack, as data may remain in caches even after unmapping—hence we specify "mapped memory" as leak origin for these attacks. For the other attacks, we can more precisely pinpoint the microarchitectural component from which data is leaked.

A color coding distinguishes on-core and off-core leak origins. Note that "mapped memory" leaks on-core data, as an attacker cannot leak data that was not architecturally accessible to the victim core in the first place, as described above. An adversary mounts an attack in either a domain-bypass or a cross-domain scenario—recall that in-domain attacks are out of scope. Attacks with on-core leak origins can only perform a domain-bypass, if the victim resides on the same core. Against victims running on separate cores, the adversary is therefore required to resort to cross-domain attacks.

Overhead. Let us examine the performance overhead of the default spot mitigations against transient execution attacks that are currently in widespread use. We run LMbench under Ubuntu 20.04 on an Intel(R) Xeon(R) Silver 4110 CPU @2.10GHz with 32 GB of RAM, which supports many spot mitigations.

Table 2 presents our results normalized to a configuration without mitigations (second column). The third column provides the normalized performance for default Ubuntu, while the other columns present results for specific mitigations disabled. As shown in the table, even enabling just the default mitigations results in a geomean slowdown of almost 2x, with BTI/Spectre-v2 mitigations, page table isolation and MDS incurring the highest costs. Going beyond the default mitigations adds so much overhead that kernel developers consider full spot mitigations impractically expensive (e.g., LVI-CFI [96]). The overhead on newer CPU models may be lower [4], but it is still substantial.

Summary. Since the high overhead of applying *all* spot mitigations is impractical for real-world systems—even some single spot

	no-mitigations	default Ubuntu	l1tf_off	mds_off	no_stf_barrier	noibpb	noibrs	nopti	nosmt	nospec_store_bypass_disable	nospectre_v1	nospectre_v2
Simple syscall	1.0	8.00	8.06	7.90	8.19	8.20	7.89	4.42	7.91	7.98	7.95	7.80
Simple read	1.0	3.65	3.59	3.54	3.53	3.68	3.57	2.31	3.64	3.50	3.55	3.26
Simple write	1.0	5.21	5.21	5.14	5.17	5.13	5.21	3.25	5.01	5.24	5.15	4.71
Simple stat	1.0	1.84	1.78	1.74	1.79	1.81	1.82	1.44	1.78	1.82	1.80	1.72
Simple fstat	1.0	3.68	3.72	3.67	3.67	3.76	3.74	2.40	3.62	3.69	3.62	3.45
Simple open/close	1.0	1.90	1.86	1.80	1.89	1.84	1.88	1.44	1.91	1.86	1.85	1.72
Select on 10 fd's	1.0	2.49	2.47	2.42	2.51	2.49	2.54	1.72	2.40	2.47	2.46	2.43
Select on 100 fd's	1.0	1.38	1.41	1.32	1.37	1.38	1.39	1.17	1.39	1.36	1.37	1.35
Select on 250 fd's	1.0	1.15	1.18	1.12	1.15	1.14	1.16	1.07	1.16	1.15	1.16	1.16
Select on 500 fd's	1.0	1.09	1.11	1.06	1.08	1.09	1.09	1.05	1.10	1.09	1.10	1.08
Select on 10 tcp fd's	1.0	2.60	2.59	2.48	2.59	2.56	2.60	1.90	2.57	2.59	2.60	2.27
Select on 100 tcp fd's	1.0	2.91	2.96	2.83	2.90	2.91	2.93	2.82	2.89	2.91	2.93	1.16
Select on 250 tcp fd's	1.0	2.90	2.94	2.83	2.94	2.91	2.90	2.88	2.93	2.92	2.95	1.05
Select on 500 tcp fd's	1.0	2.97	2.98	2.87	2.94	2.94	2.95	2.94	2.96	2.93	2.96	1.04
Signal handler installation	1.0	3.63	3.74	3.61	3.73	3.72	3.61	2.38	3.78	3.65	3.73	3.58
Signal handler overhead	1.0	1.35	1.33	1.28	1.36	1.35	1.30	1.18	1.36	1.30	1.34	1.34
Protection fault	1.0	1.74	1.64	1.56	1.69	1.63	1.67	1.28	1.65	1.63	1.64	1.54
Pipe latency	1.0	1.23	1.23	1.20	1.24	1.23	1.24	1.15	1.23	1.23	1.19	1.13
AF_UNIX sock stream latency	1.0	1.60	1.68	1.62	1.60	1.59	1.64	1.28	1.61	1.67	1.60	1.57
Process fork+exit	1.0	1.16	1.05	1.13	1.13	1.14	1.14	1.05	1.01	1.14	1.08	1.09
Process fork+execve	1.0	1.08	1.11	1.12	1.12	1.09	1.11	1.04	0.99	1.15	1.11	1.09
Process fork+/bin/sh -c	1.0	1.12	1.08	1.09	1.13	1.11	1.11	1.04	0.94	1.10	1.09	1.08
Pagefaults on /tmp/lmbench/XXX	1.0	1.13	1.12	1.11	1.12	1.13	1.13	1.08	1.12	1.13	1.13	1.13
UDP latency using localhost	1.0	1.30	1.30	1.29	1.29	1.30	1.32	1.25	1.32	1.32	1.31	1.11
TCP latency using localhost	1.0	1.23	1.24	1.23	1.24	1.25	1.22	1.21	1.23	1.27	1.24	1.08
TCP/IP connection cost to localhost	1.0	1.22	1.24	1.23	1.25	1.22	1.23	1.20	1.28	1.22	1.24	1.06
mean	1.0	2.29	2.29	2.24	2.29	2.29	2.28	1.77	2.26	2.28	2.28	1.96
geomean	1.0	1.95	1.94	1.90	1.95	1.94	1.94	1.60	1.91	1.94	1.93	1.65

Table 2: LMBench results for different kernel mitigation configurations compared to a `mitigations=off` baseline.

mitigations alone are considered too costly [1, 9]—these systems are instead left (partially) vulnerable. Moreover, as we have seen, even the default spot mitigations may incur very high overheads. The complex system of transient execution attacks makes security analysis and picking spot mitigations difficult. Perhaps most importantly, spot mitigations are not future proof: every new attack requires additional spot mitigations, incurring even more complexity and performance overhead. In the meantime, systems are almost certainly vulnerable to yet undisclosed attacks.

4.2 Towards a Solution

Can we do better than spot mitigations? As we show in Table 1, 24 of the 27 known transient execution attacks depend on on-core leakage, which suggests cross-core attacks are inherently more difficult. Indeed, from a computer architecture perspective, essential ingredients of transient execution attacks are mostly core-local: faults, pipeline flushes, micro-optimizations, mispredictions, optimistically forwarding data across different buffers, etc. On the other hand, off-core events are much rarer and adhere to a well-defined interface, which enables better security analysis.

Furthermore, we note that previous work [105] analyzed transient execution attacks and their covert channels. Out of the 14 covert channel types analyzed, 9 are core-local. Moreover, the core-local ones are the most widely used in known attacks: 19 out of 20 analyzed attacks use a core-local covert channel, such as the L1 or L2 data cache. The only ones that generalize to cross-core covert channels are data cache covert channels, by instead using the LLC which is shared across cores.

Our approach. Based on these insights, we propose physical domain isolation (physically separating victim and attacker on different cores) as a principled defense against transient execution attacks. Domain-bypass attacks relying on on-core leakage are directly rendered impossible. In other words, on-core attacks are only possible in cross-domain fashion, requiring the victim to contain specific gadgets, to be triggered by the attacker across cores.

While trigger gadgets are attack specific (and hence not easily targeted by a blanket mitigation), disclosure gadgets only depend on the covert channel. Physical domain isolation ensures that the covert channel *must* be cross-core, a severe limitation as we saw above. Indeed, the only practical cross-core covert channel resource

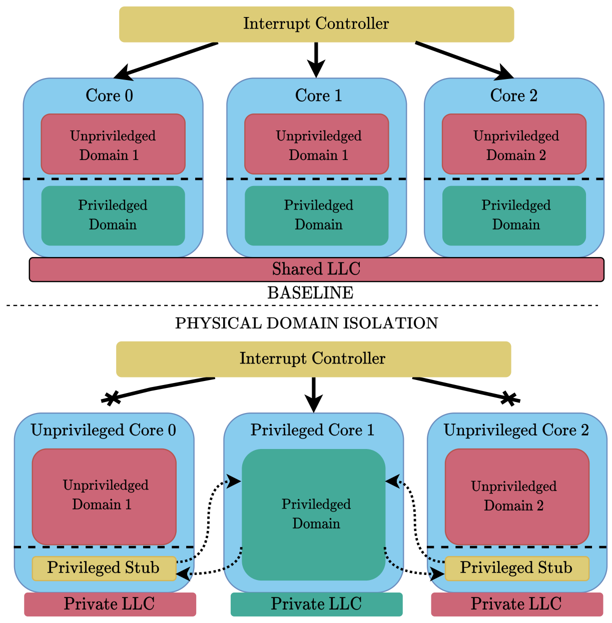


Figure 2: Physical domain isolation in QUARANTINE.

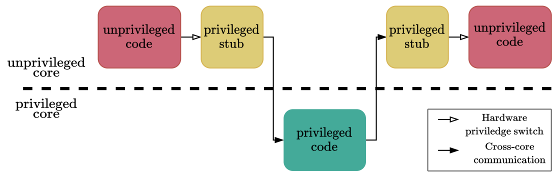


Figure 3: Privilege mode switching in QUARANTINE.

used in real-world cross-domain transient execution attacks so far is the LLC. By additionally partitioning the LLC between victim and attacker, physical domain isolation eliminates even this disclosure vector.

In the following, we present QUARANTINE, our approach for achieving physical domain isolation.

5 PHYSICAL DOMAIN ISOLATION

QUARANTINE's physical domain isolation isolates different security domains on separate cores to prevent them from sharing core-local microarchitectural resources. Moreover, it unshares the LLC, partitioning it among the security domains. In the following, we describe our design at a high level and explain our design choices, like control flow and interrupt rerouting, and potential problems, like breaking CPU-locality assumptions.

5.1 Core Partitioning

Using core scheduling or affinity pinning, modern systems already support isolation of different unprivileged domains on distinct cores. The design, depicted as the baseline situation in Figure 2, thwarts application-to-application and VM-to-VM attacks. However, it does not protect privileged domains (i.e., the OS or hypervisor) as they still share microarchitectural resources with untrusted domains.

In contrast, physical domain isolation strives to run *all* security domains on separate cores, including privileged domains. The exact accomplishment of this goal would result in a perfect defense against core-local transient execution attacks across security domains. Unfortunately, contemporary hardware does not allow exact separation of privileged and unprivileged code on separate cores. In particular, mode switching between privilege levels (e.g., by syscalls or VM-exits), always occurs on the same CPU.

QUARANTINE circumvents these hardware limitations by means of a *privileged stub*, as shown in the lower half of Figure 2. Conceptually, unprivileged domains and privileged domains execute exclusively on their own subset of cores, while privileged stubs support unprivileged domains on mode switches—providing core-local scheduling and redirecting control flow to and from the privileged cores for all other privileged functionality. These stubs are minimal in size and only access insensitive data. We analyze the remaining attack surface that these stubs introduce in Section 8.3.

5.2 Isolating Privileged Execution

Upon a *synchronous* mode switch into a privileged domain, e.g., due to a system call or VM-exit, control flow enters the privileged stub on the unprivileged core, as depicted in Figure 3. The stub performs a minimal recovery from the mode switch, e.g., restoring its register state, and then sends a request to a privileged core. The privileged core handles the request and notifies the (unprivileged) stub, as soon as the request is completed. Upon notification, the stub immediately returns control to the unprivileged domain. For cross-core communication, we exchange data via shared memory. While we could, in principle, move some rerouting code from the stub to the unprivileged domain, much like the mode-switching optimizations for exceptionless syscalls [91], we favored simplicity and compatibility in our design.

External interrupts cause *asynchronous* privilege mode switches. To prevent (privileged) interrupt handlers from running on unprivileged cores, we programmed the system-wide interrupt controller to send core-independent interrupts to privileged core only. Indeed, the majority of interrupts (including all I/O interrupts), can be handled by any core. We leave only a small subset to be handled locally, most notably local timer interrupts for core-local scheduling.

5.3 Breaking Locality Assumptions

As modern hardware is designed to run privileged and unprivileged domains on the same CPU, so is modern software. Operating system kernels and hypervisors generally assume that they run on the same CPU as the process/VM that they service. Privileged code accessing an unprivileged address space or using CPU-local variables implicitly depends on such locality assumptions. By moving the code to a different core, physical domain isolation breaks many of the underlying locality assumptions, and hence its correctness. Care must be taken in resolving such locality issues on modern kernels and hypervisors, lest they lead to substantial additional complexity. Furthermore, the corresponding patches (e.g., address space switching on the privileged CPU), may well incur significant performance overhead.

5.4 Cache Partitioning

Core isolation, as described above, already stops the sharing of on-core caches, typically L1 and L2, between different security domains. But the LLC, typically L3, is normally shared among cores. Quarantine explicitly partitions the LLC to give every security domain exclusive access to a different part of the LLC. Doing so eliminates LLC covert channels, the cross-core alternative to widely used, but core-local, L1 data cache covert channels (cf. Section 4.2). As data caches are transparent to software, this does not require any software modifications, apart from the LLC configuration code.

5.5 Kernel- vs Virtualization-based Isolation

Our high-level design for physical domain isolation can, in principle, be applied either to the user-kernel interface to support *kernel-based isolation* or to the guest-host interface to support *virtualization-based isolation*. In the following, we explore both design options on commodity systems, using Linux/KVM as a reference. We first describe our unsuccessful efforts to implement kernel-based isolation and argue this option is impractical for operating systems such as Linux. Next, we present a virtualization-based instantiation of our design and provide concrete evidence of its practicality and show that it is able to mitigate transient execution attacks mounted by malicious VMs and (unikernel) applications.

6 (IM)PRACTICALITY OF KERNEL-BASED ISOLATION

Quarantine's kernel-based isolation prototype isolates unprivileged user processes from the privileged Linux kernel (v5.15), as well as from each other.

6.1 Core Partitioning

Kernel-based isolation redirects execution to an isolated *kernel core* whenever a user process, running on a distinct *user core*, traps into the kernel. As system calls are frequent events for many workloads, the performance of user applications depends directly on their latency. The ideal configuration for minimal latency would dedicate a kernel core to each user application, such that this core can immediately service system calls whenever they get executed. The downside of such a setup is that it removes one core for each isolated user process and, thus, does not scale to real-world workloads. Hence, we developed a kernel-based isolation prototype in which a kernel core can handle a configurable number of user cores.

6.2 Isolating the Kernel

On each kernel CPU we deploy a *service thread*, which services system calls coming from privileged stubs on user CPUs. To minimize overhead and attack surface, we redirect system calls from user to kernel CPUs as soon as the user CPU is ready to execute it, i.e., in `do_syscall_64`. To ensure minimal request latency, service threads and user processes poll a shared memory location.

6.3 Locality Problems

While *conceptually* simple, cross-core system calls on operating systems such as Linux are very complex in practice.

Syscall	Users	Calls	Baseline μs	Quarantine μs	Overhead x
`getppid()`	1	23200	0.04	0.72	18.51
	2	360614	0.04	1.43	36.76
	4	821871	0.04	2.46	63.18
	8	1384734	0.04	5.09	130.86
`read()`	1	12137	0.07	1.22	16.86
	2	342346	0.21	1.38	19.09
	4	735226	0.37	2.70	37.28
	8	1362904	0.41	5.96	82.46
`write()`	1	218210	0.06	0.91	15.43
	2	348516	0.15	1.19	20.12
	4	731462	0.22	2.48	41.84
	8	1340910	0.32	5.20	87.85

Table 3: Kernel-based isolation performance for LMbench's latsyscall microbenchmark.

User Context Switching. Frequently used system calls such as `read`, `write`, and `ioctl` require access to a process' context, e.g., its address space, locks and bookkeeping data. As a result, service threads servicing multiple user CPUs frequently switch between user contexts, which harms performance [64]. Moreover, without kernel-based isolation user space applications immediately trap into the kernel (e.g., via the syscall instruction on x86_64) and continue execution, whereas with kernel-based isolation processes may have to wait until the service thread finished servicing other processes.

Scheduling. Scheduling service threads using the existing Linux scheduler leads to unacceptable system call latencies. To improve response times, we perform direct context switching in and out of our service treads, circumventing the Linux scheduler. Unfortunately, such customizations are not very compatible with the rest of the kernel and require custom solutions for complicated scheduling decisions (e.g., whether to run a service thread or a normal kernel thread upon whose results the service thread may depend).

CPU-local Variables. Linux' system call handlers make heavy use of CPU-local variables, such as `current` (the currently running process) or RCU locks. Patching these handlers to become independent of such CPU-local variables requires pervasive changes to the kernel. The variable `current` alone is referenced thousands of times throughout system call handlers. After two person years' worth of implementation effort, our prototype reliably supports a few dozen system calls. Implementing enough system calls to run, say, a web server or browser, requires a major overhaul of the Linux kernel.

6.4 Performance

We evaluate the performance of different configurations for kernel-based isolation with the OS microbenchmark suite LMbench [73] and report results in Table 3. To highlight solely the system call rerouting overhead, we disabled interrupt and page fault rerouting, as well as LLC partioning. Despite our optimization efforts, the results suggest that kernel-based isolation is impractical. Even in an ideal one-on-one configuration, the system call latency is unacceptably high and further degrades once multiple users share one kernel CPU.

Conclusion. The relationship between the Linux kernel and its user processes is complex and the kernel assumes CPU-locality in many places. Supporting a significant number of system calls especially requires a heroic effort. Furthermore, the resulting performance overhead is unacceptable. We conclude that kernel-based isolation, while possible in theory, is not practical for operating systems such as Linux.

7 VIRTUALIZATION-BASED ISOLATION

Instead of targeting the kernel-user boundary, virtualization-based isolation physically separates the hypervisor (the *host*), from the VMs (the *guests*)—as well as the VMs from each other. Although *conceptually* Quarantine effectuates radical changes to the fundamental workings of the hypervisor, our patches are noninvasive and minimally impact the operation of the Linux kernel. We implemented Quarantine's virtualization-based isolation for AMD on top of Linux/KVM for kernel v5.15 in 523 lines of code, including changes in the architecture and vendor-specific subsystems.

7.1 Resource Partitioning

For simplicity, we partition the available cores during system initialization statically into *host* and *guest cores*, whose CPUs we call *host* and *guest CPUs* respectively. Furthermore the guest cores are distributed among different users, such that different users are also isolated from each other. We implemented the physical isolation of security domains using the topology and CPU affinity functionality of the Linux kernel. While dynamic host/guest core policies are possible, our experimental results confirm that the host domain is typically used sparingly and simple static policies, e.g., a single host core, are sufficient.

For LLC partitioning we also choose for simplicity: every domain gets a part of the LLC proportional to the number of cores it got assigned. For example, if a user runs on 2 of the 8 cores in total, then it will have access to a quarter of the LLC. LLC partitioning is implemented on top of Linux' resctrl functionality.

7.2 Isolating Hypervisor Execution

Under Linux/KVM, a VM is implemented as a user process. The hypervisor consists of both user space, e.g., QEMU [6], and the host kernel, in particular its KVM module. We call the user process of a VM its *owner*. Each VM is associated with a *runner*: a kernel thread on a guest CPU responsible for running its VM. Runners are part of the privileged stubs on guest CPUs (cf. Figure 2).

Rerouting VM-exits. Figure 4 illustrates the steps to run a VM under Quarantine. On a host CPU, an owner instructs KVM to run its VM via the `ioctl` system call. KVM almost entirely sets up the VM to run, and then sends a cross-core *VM-start message* to the VM's runner function on a guest CPU. In response, the runner performs the remaining CPU-local setup and executes the `VMRUN` instruction. From here on, the VM takes control and executes its code in guest mode until the occurrence of a VM-exit event. The latter returns control to the runner, which recovers from the VM-exit and as soon as possible sends a cross-core *VM-exit message* to the host CPU of the VM's owner. KVM and the owner process handle the VM-exit on the host CPU, after which the VM will be ready to run again.

As the VM is controlled via the memory resident VMCB, VM-exits can be handled from any CPU. To illustrate this, let us consider the example of a guest VM-exiting due to a page fault. The VMCB

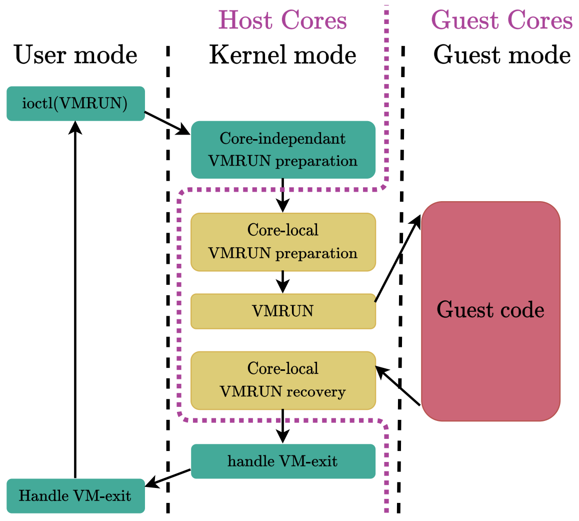


Figure 4: KVM execution under Quarantine. CPU-independent operations are bound to host CPUs, while CPU-local operations are executed on guest CPUs.

contains an exit code that tells the host CPU that a page fault happened, as well as information such as the faulting address. The host CPU determines whether it involved a host- or a guest-side page fault, i.e., if it was caused by host page swapping or not. In the former case, KVM swaps the page back in. In the latter case, KVM injects a page fault into the guest by setting a flag in the VMCB, prompting the guest OS to execute its own page fault handler upon the next VMRUN.

Cross-core Communication. Besides the general execution flow, Figure 4 also shows the physical isolation boundary. The red line separates execution performed on host vs. guest CPUs. We cross this physical isolation boundary by sending VM-start and VM-exit messages between host and guest CPUs via shared memory. Runners and host CPUs receive these messages by performing *collaborative polling*: iteratively checking for a message and invoking the scheduler if there is no message yet. This method allows us to multiplex CPUs between multiple runners or owners, while keeping the latency for VM-start and VM-exit messages low.

Shrinking Runners. The yellow blocks in Figure 4 highlight the runner's code in the stub, which Quarantine seeks to minimize to provide strong isolation guarantees. To this end, we thoroughly analyzed KVM's code paths concerned with `VMRUN` handling and determined which parts of the code are CPU-independent and which need to run on the guest CPUs. Listing 1 shows the critical section around a `VMRUN`, delimited by en/disabling interrupts and preemption. We concluded that this entire critical section must be run on the guest CPU. The only addition is that KVM may need to handle four special time management requests just before the critical section, which we determined to be CPU-local as well and hence also execute on guest CPUs. Based on our analysis, we conclude that the resulting amount of privileged code of runners

```
vcpu->srcu_idx = srcu_read_lock(&vcpu->kvm->srcu);
preempt_disable();
static_call(kvm_x86_prepare_guest_switch)(vcpu);
local_irq_disable();
...
VMRUN
...
local_irq_enable();
preempt_enable();
```

Listing 1: KVM's critical section around a `VMRUN`.

on guest CPUs is minimal. We refer to Section 8.3 for a quantitative analysis of the whole host stub.

Interrupt Rerouting. Quarantine relies on the Linux' SMP IRQ affinity interface [76] to redirect all architecture-independent interrupt requests (IRQs) to host CPUs while allowing that minimal set of interrupts (such as timer interrupts) that is necessary for CPU-local functionality to reach the guests.

7.3 Locality Problems

Cross-core Control Flow Diversion. As VM-start and VM-exit messages effectively cause control flow to switch to a different CPU and lock-ownership is CPU-bounded in Linux, Quarantine ensures that senders release any acquired locks before sending a message and corresponding receivers acquire these locks again. In addition, to avoid KVM issues at the VM-start and VM-exit boundaries resulting from variables on the CPU-local kernel stack, we replaced these with per-VM variables on the heap such that their state persists independently of CPU-local function call stacks.

Owner Context Switching. An important locality assumption KVM makes is that it runs in the user context of the current VM's owner. This for example required us to let each owner share its address space with its runner. On the host CPU, it requires us to switch between owner contexts upon serving VM-exits from different VMs. As we will see in Section 8, this does not result in bad performance, as it did for kernel-based isolation (cf. Section 6.3). We expect this difference to stem from VM-exits happening less frequently and being more expensive to handle than system calls.

Scheduling. This prototype uses the unmodified Linux scheduler, contrary to kernel-based isolation (cf. Section 6.3).

CPU-local Variables. In order to get rid of all problematic CPU-local variables, we only had to replace six references to `current` by a pointer to the owner inside the host stubs. This was a very low engineering effort compared to similar problems for kernel-based isolation (cf. Section 6.3). As opposed to virtualization-based isolation breaking locality assumptions only on the by design small host stubs, kernel-based isolation does so for kernel code running on kernel CPUs, i.e., almost the entire kernel. This discrepancy in complexity represents a major practical advantage of hypervisor-based over kernel-based isolation. For the former, two man years of effort led to the support of a few dozen system calls, while the latter offers the same functionality as unmodified KVM.

7.4 Isolating User Applications

So far, we presented a virtualization-based Quarantine prototype to protect and isolate VMs, but the same design can be used to protect user applications by adopting a unikernel architecture [56]. Striking a balance between performance and application compatibility is notoriously challenging for unikernels [54]. However, recent solutions show that the Kernel Mode Linux (KML) [70] can be used to implement highly efficient and compatible unikernels, by simply folding existing application code into the Linux kernel [56].

In Quarantine, we adopt this approach to turn unmodified Linux applications into "VMs" which can be isolated with Quarantine. This approach eliminates the need to partition kernel-side application execution at the OS level (which is challenging, as previously discussed) and even provides opportunities for unikernel optimizations. In particular, recent work shows that even straightforward KML-based optimizations (e.g., running a minimal, optimized Linux kernel) can lead to impressive speedups [56].

Nonetheless, for a fair evaluation, we enabled no special unikernel optimizations for our experimental analysis, using the same kernel for the KML guest and the host. As such, despite some intrinsic KML optimizations (i.e., the syscall interface being reduced to a lighter library call interface in the guest [56]), we observed essentially identical performance for our benchmarks running in VMs vs. virtualized unikernels. As a result, for simplicity, we only present results for the VM-based configuration of Quarantine in our evaluation. Similarly, we only consider a VM-based baseline for our benchmarks, even to evaluate the impact of our design on baseline (non-virtualized) user applications. While virtualization does add a cost compared to native execution, we observed relatively low overhead for our benchmarks (e.g., around 5% Nginx saturated throughput degradation), which can easily be more than amortized by the speedups provided by unikernel optimizations (e.g., over 30% Nginx saturated throughput improvement with Lupine optimizations [56]).

8 EVALUATION

We evaluate Quarantine in terms of performance, engineering complexity, and security guarantees.

8.1 Performance Evaluation

Setup. We evaluated our Quarantine prototype on a *test machine* with an AMD Ryzen Threadripper PRO 5995WX 64-Core Processor with 2 CPUs per core, 512 GB of RAM, and an Aquantia AQC107 NIC. The test machine runs Ubuntu 22.04.1 using Linux kernel 5.15 with Quarantine enabled or disabled (baseline). To reduce noise, we used the *performance* scaling governor and disabled KASLR, THP, and KSM. As Quarantine is an alternative to deployed spot mitigations against transient execution attacks, we also disabled all spot mitigations that can be disabled without source code modification[2].

We ran all benchmarks inside a lightweight Alpine Linux 3.15 (running kernel 5.15.12) *test VM* on the test machine. The test VM

[2]Note, that even when disabling all mitigations the Linux kernel protects against Spectre-v1 "on a case by case base with explicit pointer sanitation and usercopy LFENCE barriers." [13]. To compare against an unmodified ("vanilla") baseline, we chose to keep these mitigations in-place.

Benchmark	Baseline	Quarantine	Overhead
Simple syscall	0.09 μs	0.09 μs	5.94 %
Simple read	0.11 μs	0.13 μs	13.49 %
Simple write	0.11 μs	0.12 μs	10.75 %
Simple stat	0.42 μs	0.46 μs	10.00 %
Simple fstat	0.17 μs	0.17 μs	0.69 %
Simple open/close	0.66 μs	0.69 μs	5.13 %
Select on 10 fd's	0.27 μs	0.27 μs	-1.65 %
Select on 100 fd's	0.63 μs	0.82 μs	29.98 %
Select on 250 fd's	1.22 μs	1.72 μs	40.95 %
Select on 500 fd's	2.27 μs	3.27 μs	44.15 %
Select on 10 tcp fd's	0.29 μs	0.29 μs	1.42 %
Select on 100 tcp fd's	1.33 μs	1.72 μs	30.00 %
Select on 250 tcp fd's	3.06 μs	4.12 μs	34.52 %
Select on 500 tcp fd's	6.01 μs	8.16 μs	35.69 %
Signal install	0.13 μs	0.14 μs	10.93 %
Signal overhead	0.63 μs	0.48 μs	-23.92 %
Protection fault	0.23 μs	0.26 μs	11.67 %
Pipe latency	1.97 μs	1.99 μs	0.94 %
AF_UNIX sock stream	3.23 μs	3.48 μs	7.60 %
Process fork+exit	21.65 μs	23.43 μs	8.23 %
Process fork+execve	61.69 μs	66.38 μs	7.59 %
Process fork+/bin/sh	150.24 μs	165.09 μs	9.88 %
Pagefaults	0.09 μs	0.10 μs	13.31 %
UDP latency localhost	3.36 μs	3.48 μs	3.62 %
TCP latency localhost	4.22 μs	4.37 μs	3.65 %
Local TCP/IP connect	11.76 μs	11.91 μs	1.27 %

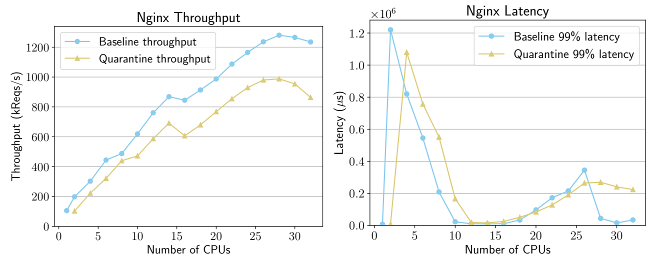
Table 4: LMBench performance: microbenchmark latencies for baseline vs. Quarantine.

runs on top of KVM and QEMU 4.2.1 with 64 GB of hugepage backed memory. We pass through the host's Aquantia NIC to the test VM via `vfio`. For server benchmarks, we generate a load from a separate *client machine* with an AMD Ryzen 5 5600X 6-Core Processor, 16 GB of RAM, and an Aquantia AQC107 NIC, running Ubuntu 20.04.4.

We ran all our experiments 11 times and report the median. During our experiments, we varied the host/guest CPU configuration to understand the impact of CPU count. To fairly compare Quarantine against the baseline, both configurations use always an equal number N of CPUs on the physical test machine. In an N-CPU configuration, the baseline runs the test VM with N virtual CPUs (vCPUs). In contrast, Quarantine always uses a single host CPU and $N-1$ guest CPUs, and therefore runs the test VM with $N-1$ vCPUs. As Quarantine needs both a host and a guest CPU, the minimal configuration it can run on is the 2-CPU one.

LMbench. We first evaluated Quarantine on the LMbench benchmark suite to stress-test the guest kernel. As LMbench is a single-threaded workload, we ran LMbench in the test VM under the minimal 2-CPU configuration. Table 4 presents our results.

As shown in the table, the performance overhead varies greatly across microbenchmarks. The overhead is more prominent for the `select` benchmarks, presumably due to the growing number of VM-exits as one increases the number of monitored file descriptors.



(a) Throughput **(b) 99% Latency**

Figure 5: Performance impact of Quarantine for Nginx. Both baseline and Quarantine have the same total number of CPUs available during the experiments.

Overall, Quarantine suffers a 11.2% geomean overhead compared to the baseline. This is better than spot mitigation performance.

Nginx. To evaluate the impact of our design on real-world user applications, we evaluated the Nginx web server running in the test VM on Quarantine. In particular, we ran experiments on Nginx 1.20.2, serving a 64 byte file over 4,096 concurrent connections. We benchmark Nginx using the `wrk` [17] benchmarking tool on the client machine, using 32 client `wrk` threads for 30 seconds. The client machine is not capable of fully saturating nginx for large numbers of cores. Therefore, although we ran our experiments up to 128 CPUs, we only included the results where the CPU saturation was more than 99%, i.e., up to the 30-CPU configuration.

Figure 5a displays Nginx's throughput for varying configurations. In a 2-CPU configuration, the baseline has approximately double the throughput of Quarantine. This is expected, as Quarantine can only use a single guest CPU to run the VM with Nginx, as opposed to the baseline using two. The cost of Quarantine due to dedicating a CPU to run the host gets amortized as we move to bigger CPU counts. Quarantine's relative throughput degradation compared to the baseline becomes 23.9% at the 10-CPU configuration and stays stable until the 28-CPU configuration, listing 22.8% degradation. For even bigger configurations, Quarantine's performance starts to plummet due to the single host CPU bottlenecking the system.

The throughput degradation is caused by two factors: (1) not running the VM/Nginx on the host CPU, and (2) the performance impact of Quarantine, due to, e.g., VM-exit and interrupt rerouting. For the 10-CPU and 28-CPU configurations, (1) contributes 10.0% and 3.6% throughput degradation respectively, and hence (2) accounts for 15.4% and 19.9% respectively. We suspect the increasing load on the host CPU, as we scale up, to lead to longer VM-exit latencies, which cause the increase in overhead of type (2).

We also measured the 99% tail latency experienced of Nginx during the execution of our benchmarks. Figure 5b presents our results. Running Nginx on only a single vCPU, i.e., in the 1-CPU configuration for the baseline and in the 2-CPU configuration for Quarantine, results in very low tail latencies—not unexpected in a single-worker-process configuration of Nginx.

On higher-CPU-count configurations, the tail latency of the baseline and Quarantine become similar. Both baseline and Quarantine have minimal 99% tail latency in the 14-CPU config, with

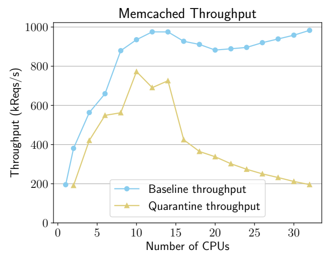


Figure 6: Memcached throughput.

QUARANTINE's latency being 73.2% longer than the baseline's. From the 26-CPU configuration onwards, the baseline's tail latency stays low, while QUARANTINE's steadily grows. This is again because the host CPU start bottlenecking the system for bigger configurations.

Memcached. We ran Memcached 1.6.12 in the exact same setup as Nginx in our test VM. The client machine runs memaslap v1.0 for 30 seconds with 20 threads and a concurrency of 140 to generate the workload. The results are listed in Figure 6. The CPU utilization is again more than 99% for every config with at most 30 CPUs.

QUARANTINE has peak throughput on the 10-CPU configuration, reporting a throughput degradation of 17.4% compared to the baseline. The figure also emphasizes that Quarantine's core configuration is workload dependent. Memcached spends more time in the host and therefore needs more host cores per guest core on average, compared to Nginx.

Interrupt Rerouting and Cache Partitioning. We also ran the benchmarks above with interrupt rerouting and/or LLC partitioning disabled. This did not result in any significant changes in performance. We think this is because induced extra communication overhead is compensated by better locality. We conclude that these security enhancing measures do not affect performance.

8.2 Engineering Effort

We now compare the engineering effort of our virtualization-based isolation QUARANTINE prototype. We measured 523 lines of code[3] over 16 files for the full prototype, which was designed and engineered in only 5 person months. Such low effort is in contrast to kernel-based isolation as well as other existing blanket protections against transient execution attacks, namely Address Space Isolation (ASI) [90] and the Secret-Free hypervisor (SF) [103]. ASI and SF provide MMU-based isolation (as oposed to our core-based isolation). This requires to have explicit knowledge of "secrets" (or "non-secrets"), introducing additional complexity. ASI was developed by multiple teams from multiple companies for over 3 years, with the most recent patch changing 189 files and 4,229 lines of code[3] [90]. SF's engineering effort was not detailed by the authors, however they report 2,415 lines of code changed for Secret-Free Xen. We conclude that QUARANTINE's engineering (and maintenance) effort is far lower than competing solutions.

[3]measured using CLOC v1.92 [11]

8.3 Security Evaluation

Remaining Attack Surface. In this section, we evaluate how effective QUARANTINE is in reducing the attack surface of core-local transient execution attacks. Recall that, although an ideal design would achieve perfect isolation, i.e., 100% attack surface reduction, such perfect physical domain isolation strategy is infeasible in practice due to the constraints imposed by virtualization hardware. As such, QUARANTINE strives to minimize the privileged (host) stub of code run on unprivileged (guest) CPUs. To quantify such residual attack surface, we measure the number of hypervisor functions run on guest CPUs compared to the baseline. We focus on the number of functions since the number of potential transient execution gadgets is approximately proportional to the number of vulnerable functions an attacker has access to.

Our hypervisor consists of two components: QEMU and KVM. QEMU contains 9,831 functions in total. This static set over approximates all the functions (and gadgets) an attacker can possibly reach, targeting specific virtual devices, etc.—providing an indication of the baseline attack surface for QEMU. A similar estimate is much harder for KVM, due to its tight integration inside the Linux kernel. As a more realistic but also pessimistic proxy, we used Linux' function tracing capabilities to dynamically trace the set of KVM functions executed during the execution of saturated Nginx. We adopted a similar tracing-based approach to identify the set of KVM functions QUARANTINE needs to run on guest CPUs—and manually checked the code to ensure our approximation was sound. Our analysis reported 2,113 KVM functions executed by the baseline and 297 KVM functions executed by QUARANTINE's guest CPUs. As our combined results show, the baseline (QEMU+KVM) attack surface consists of 11,944 host functions, while QUARANTINE's residual attack surface consists of only 297 functions, 2.5% of the baseline.

To understand the nature of the residual attack surface, we inspected the 297 remaining host functions. The purposely chosen code for our runners contributes 48 of the functions, while the scheduler contributes most of the code, namely 167 functions. The other 87 functions have a variety of origins, such as CPU-local interrupt handlers, wait queues, watchdogs, the `eventfd` subsystem, and the RCU subsystem.

We conclude that QUARANTINE significantly reduces the transient execution attack surface in practice: 97.5% for VMs and even more (over 99.5%) for unikernel applications, given that the *entire* Linux kernel and its many gadgets [48] are part of the baseline attack surface.

Security Guarantees. Recall that our threat model considers user-to-user, user-to-kernel, VM-to-VM, and VM-to-hypervisor attack scenarios. Within this context, QUARANTINE effectively mitigates a whole *class* of attacks, namely transient execution attacks that leak on-core data. As physical domain isolation separates attacker and victim on separate cores, QUARANTINE forces an attacker to use a cross-core attack. Cross-core domain-bypass attacks are inherently impossible using on-core leakage. Cross-core cross-domain attacks require the victim to contain a trigger gadget reachable across cores, as well as a disclosure gadget for a non-LLC cross-core covert channel, making such attacks infeasible in practice.

This class of on-core leaking transient execution attacks includes 24 of out the 27 known transient execution attacks on Intel CPUs (cf. Table 1). Possibly even more importantly: QUARANTINE is more future proof than the plethora of spot mitigations. QUARANTINE guarantees to defend against any on-core leaking transient execution attack, including future ones. Spot mitigations do not give similar guarantees whatsoever. In contrast, historically spot mitigations have time and time again been circumvented [3, 74, 100, 102].

9 DISCUSSION

Our evaluation demonstrates that physical domain isolation is feasible and can be implemented using reasonable performance penalties. We believe that our prototype provides a solid base to demonstrate this, however we believe there is room for additional improvements.

Scaling via Multiple Host Cores. Currently, our prototype only supports one host CPU. Consequently, we did not evaluate the effect of several host cores, but we believe that allowing for multiple host cores will improve QUARANTINE's performance after the first host core is saturated. However, given that our evaluation showed that even a single host core can handle huge and realistic workloads, we leave this to future work.

Hardening of Privileged Stubs. Although privileged stubs architecturally only access security insensitive data, they might speculatively access secret data, which would then become leakable on-core. To this end, we minimized the size of the privileged stubs (by 97.5%), ensuring no such gadgets remain in the stub. In order to systematically ensure this, future work could use modern gadget scanners [48, 79, 83], which are effective since the stub's code is small and frequently executed. Another option would be to unmap all memory on guest cores, and just-in-time map pages whenever the stub needs access (meaning the data is insensitive).

Alternative Covert Channels. QUARANTINE mitigates cross-core covert channels used by transient execution attacks, by partitioning the LLC. Practical exploits have so far only used data cache covert channels, which could generalize to a cross-core setting using the LLC. Other cross-core covert channels do exist, e.g., DRAM row buffer [82], though there usability in practical transient execution attacks has never been shown. In particular, there granularity is much higher and no practical disclosure gadgets in real-world software has been found so far. The same holds for covert channels abusing imperfect LLC partitioning implementations [50, 80]. If, in the future, another covert channel does pose a threat, QUARANTINE could be extended to also mitigate it, e.g., a DRAM row buffer aware allocator to isolate different security domains on different DRAM banks.

Hardware/Software Co-design. Although QUARANTINE is currently a software-only mitigation, the approach could benefit from a hardware/software co-design. Heterogeneous multicore processors [75] allow host and guest cores to be mapped on different microarchitectures, potentially improving overall efficiency. QUARANTINE would also benefit from lower latency cross-core communication primitives.

10 RELATED WORK

Isolation for Performance. There is a large body of work on system design using isolation to improve performance, with an extensive focus on virtualized environments and clouds. Some efforts rely on commodity hardware features such as Intel CAT to partition shared microarchitectural state such as LLCs and study the resulting performance isolation guarantees on otherwise unmodified virtualized systems [104, 107]. Other efforts focus on rethinking the virtualization stack to improve performance isolation and specialization.

FlexSC [91] is an early example for for separating kernel and userspace. It reduces the cost of system calls by sending system call requests and replies via a shared page. This way, the authors could run the kernel on one core and the user process on another. FlexSC does not focus on fully synchronizing the two domains (e.g., it does not reroute interrupts). Later work showed that spreading the components of a small (research) OS across separate cores improves reliability [22].

Kumar et al. [15, 55] conducted early work on the idea of *sidecores*: cores dedicated to performing specific hypervisor functionality. Since then, much research has focused on improving virtualized I/O performance by using I/O sidecores [2, 19, 20, 57, 63, 67, 106]. The main idea is to move the part of the hypervisor responsible for I/O to dedicated cores, in order to minimize the number of I/O-induced VM-exits and hence optimize performance. In contrast to these solutions, QUARANTINE seeks to isolate as much privileged (hypervisor) code as possible to specific cores for security.

Landau et al. [58] previously explored the idea of splitting guest and hypervisor execution on separate cores to improve performance. They argue that their split hypervisor/guest execution design is infeasible on commodity hardware and describe a hardware model which would make the design practical. QUARANTINE can be seen as a practical approximation of "split execution" for security on contemporary hardware and hypervisors.

Finally, unikernels [54, 56, 68, 69, 72] isolate application code from the rest of the system using virtualization. This paradigm has emerged as the golden standard for performance specialization in virtualized environments, with impressive speedups even when simply folding unmodified applications and the Linux kernel into a unikernel using KML—as suggested by Lupine [56] and Unikraft [54]. With QUARANTINE, we suggest unikernels can also serve as a convenient way to transform unmodified applications into portable security domains for security isolation.

Isolation for Security. Apart from domain-specific variants (e.g., Site Isolation in web browsers [86]), much research on isolation against side-channel and transient execution attacks focuses on OS- or hypervisor-level isolation. Similar to performance isolation, prior research has suggested using commodity hardware features (e.g., Intel CAT [66]) or system design to counter side-channel attacks. In the latter category, early work [49, 93] suggested hypervisor-less cloud architectures, which, however, lack many of the modern virtualization features. More recent work focuses on commodity virtualization stacks, with solutions such as moving target defenses to periodically randomize VM placements and minimize attack exposure [77]. Unlike QUARANTINE, all these solutions target traditional cross-VM side-channel attacks, but are not concerned with transient execution attacks leaking on-core data from other unprivileged/privileged security domains.

Even more recently, researchers have suggested isolation primitives such as USC to mitigate transient execution attacks [5]. To minimize the attack surface, USC requires the kernel/hypervisor to map the bare minimum amount of memory while serving user requests. While this design has been demonstrated on research kernels [5], it is challenging to implement on commodity kernels, as it requires pervasive kernel/hypervisor changes. Indeed, there are concurrent proposals to implement similar solutions such as ASI on Linux/KVM [90] and SF [103] on Xen, which are considerably more complex than Quarantine, as discussed in Section 8.2. A simpler option is to allow users to annotate sensitive memory regions and prevent any non-user accesses [10]. However, this strategy can only protect predetermined data. Moreover, unlike Quarantine, all these solutions assume that the necessary kernel-mapped data contains no relevant secret, a property which is nontrivial to verify in practice. Finally, another ongoing proposal is for `coresched` [14] to re-enable its *kernel-protection* mechanism to mitigate on-core unprivileged-to-privileged domain attacks, but developers have reported "*abyssal*" performance due to the strict kernel entry/exit synchronization between sibling CPU threads [9].

11 CONCLUSION

Domain isolation is a well-established systems security principle and its applicability has transferred to the ongoing transient execution era. Unfortunately, the ability of a transient execution attacker to leak data across security domains such as concurrently running kernel code makes it challenging to implement isolation on commodity systems. In this paper, we showed that, by targeting the *guest-host* (rather than much more complex user-kernel) interface, it is feasible to move the privileged *hypervisor* domain to an entirely separate core at a low complexity cost. This design provides strong security guarantees against a broad spectrum of both known and unknown transient execution attacks. To substantiate our claims, we presented a Quarantine prototype for Linux/KVM, empirically showing that *physical domain isolation* is efficient and has less overhead than the combination of state-of-the-art spot mitigations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work was supported by Intel Corporation through the "Allocamelus" project, the Dutch Science Organization (NWO) through project "Intersect", and VMWare through an "Early Career Faculty" award. SBA Research (SBA-K1) funded this work within the framework of COMET–Competence Centers for Excellent Technologies by the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Austrian Federal Ministry of Labour and Economy (BMDW), and the federal state of Vienna, managed by the The Austrian Research Promotion Agency (FFG).

REFERENCES

[1] Amazon. 2020. Flushing L1d On Context Switches. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Blasts-L1d-Flushing.

[2] Nadav Amit, Muli Ben-Yehuda, IBM Research, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU Emulation. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/usenixatc11/viommu-efficient-iommu-emulation

[3] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 971–988. https://www.usenix.org/conference/usenixsecurity22/presentation/barberis

[4] Jonathan Behrens, Adam Belay, and M. Frans Kaashoek. 2022. Performance Evolution of Mitigating Transient Execution Attacks. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 15 pages. https://doi.org/10.1145/3492321.3519559

[5] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. 2020. Efficiently Mitigating Transient Execution Attacks using the Unmapped Speculation Contract. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1139–1154. https://www.usenix.org/conference/osdi20/presentation/behrens

[6] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[9] Alexandre Chartre. 2022. Address Space Isolation for KVM. https://lore.kernel.org/lkml/91dd5f0a-61da-074d-42ed-bf0886f617d9@oracle.com/

[10] Jonathan Corbet. 2021. memfd_secret() in 5.14. https://lwn.net/Articles/837595/

[11] Albert Danial. 2021. *cloc: v1.92.* https://doi.org/10.5281/zenodo.5760077

[12] Linux Developers. 2020. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html#protecting-the-kernel-irq-syscall-vmexit

[13] Linux Kernel Developers. 2019. Spectre Side Channels - Linux Kernel Documentation. https://www.kernel.org/doc/html/v5.15/admin-guide/hw-vuln/spectre.html

[14] Joel Fernandes. 2020. Core scheduling (v9). (Nov 2020). https://lore.kernel.org/all/20201117232003.3580179-1-joel@joelfernandes.org/

[15] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. 2007. High-performance hypervisor architectures: Virtualization in hpc systems. In *Workshop on system-level virtualization for HPC (HPCVirt)*. Citeseer.

[16] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 17 pages. https://doi.org/10.1145/3302424.3303976

[17] Will Glozer. 2012. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk

[18] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*.

[19] Abel Gordon, Nadav Har'El, Alex Landau, Muli Ben-Yehuda, and Avishay Traeger. 2012. Towards Exitless and Efficient Paravirtual I/O. In *Proceedings of the 5th Annual International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '12)*. Association for Computing Machinery, New York, NY, USA, Article 10, 6 pages. https://doi.org/10.1145/2367589.2367593

[20] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. 2013. Efficient and Scalable Paravirtual I/O System. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 231–242.

[21] Jann Horn. 2018. Speculative Store Bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[22] Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S. Tanenbaum. 2012. Keep Net Working - on a Dependable and Fast Networking Stack. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '12)*. IEEE Computer Society, USA, 12 pages.

[23] Intel. 2018. Bounds Check Bypass. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/bounds-check-bypass.html

[24] Intel. 2018. Branch Target Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html

[25] Intel. 2018. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html

[26] Intel. 2018. L1 Terminal Fault. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-

guidance/l1-terminal-fault.html
[27] Intel. 2018. Lazy FP state restore. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00145.html
[28] Intel. 2018. Rogue Data Cache Load. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/rogue-data-cache-load.html
[29] Intel. 2018. Rogue System Register Read. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/rogue-system-register-read.html
[30] Intel. 2018. Speculative Store Bypass. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html
[31] Intel. 2019. Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/intel-tsx-asynchronous-abort.html
[32] Intel. 2019. Microarchitectural Data Sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/microarchitectural-data-sampling.html
[33] Intel. 2019. Speculative Behavior of SWAPGS and Segment Registers. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-behavior-swapgs-and-segment-registers.html
[34] Intel. 2020. L1D Eviction Sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1d-eviction-sampling.html
[35] Intel. 2020. Load Value Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/load-value-injection.html
[36] Intel. 2020. Snoop-assisted L1 Data Sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/snoop-assisted-l1-data-sampling.html
[37] Intel. 2020. Special Register Buffer Data Sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/special-register-buffer-data-sampling.html
[38] Intel. 2020. Vector Register Sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/vector-register-sampling.html
[39] Intel. 2021. Floating Point Value Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/floating-point-value-injection.html
[40] Intel. 2021. Microarchitectural Data Sampling (MDS), Version: 3.0. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html.
[41] Intel. 2021. Speculative Code Store Bypass. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-code-store-bypass.html
[42] Intel. 2022. Branch History Injection and Intra-mode Branch Target Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-history-injection.html
[43] Intel. 2022. Fast Store Forwarding Predictor. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/fast-store-forwarding-predictor.html
[44] Intel. 2022. Post-barrier Return Stack Buffer Predictions. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/post-barrier-return-stack-buffer-predictions.html
[45] Intel. 2022. Processor MMIO Stale Data Vulnerabilities. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/processor-mmio-stale-data-vulnerabilities.html
[46] Intel. 2022. Return Stack Buffer Underflow. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/return-stack-buffer-underflow.html
[47] Intel. 2022. Speculative Load Disordering. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-load-disordering.html
[48] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2022. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. In *NDSS*.
[49] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. 2010. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*.
[50] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 974–987.
[51] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. arXiv:1807.03757 [cs.CR]
[52] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002
[53] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/koruyeh
[54] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248
[55] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. 2007. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*. Citeseer.
[56] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 15 pages. https://doi.org/10.1145/3342195.3387526
[57] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. 2016. Paravirtual remote i/o. *ACM SIGARCH Computer Architecture News* 44, 2 (2016).
[58] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. 2011. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In *3rd Workshop on I/O Virtualization (WIOV 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/wiov11/splitx-split-guesthypervisor-execution-multi-core
[59] Michael Larabel. 2018. Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower. https://www.phoronix.com/scan.php?page=article&item=linux-420-bisect.
[60] Michael Larabel. 2020. The Brutal Performance Impact From Mitigating The LVI Vulnerability. https://www.phoronix.com/review/lvi-attack-perf.
[61] Michael Larabel. 2022. Call Depth Tracking For Less Costly Retbleed Mitigation Hopes To Land Soon. https://www.phoronix.com/news/Call-Depth-Tracking-Hope-Soon
[62] Michael Larabel. 2022. In Light Of Spectre BHI, The Performance Impact For Retpolines On Modern Intel CPUs. https://www.phoronix.com/scan.php?page=article&item=spectre-bhi-retpoline&num=1.
[63] Dongwoo Lee, Changwoo Min, and Young Ik Eom. 2016. vCanal: Paravirtual Socket Library towards Fast Networking in Virtualized Environment. *IEICE TRANSACTIONS on Information and Systems* 99, 2 (2016).
[64] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the Cost of Context Switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (San Diego, California) *(ExpCS '07)*. Association for Computing Machinery, New York, NY, USA, 2–es. https://doi.org/10.1145/1281700.1281702
[65] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: Reading Kernel Memory from User Space. *Commun. ACM* 63, 6 (may 2020), 46–56. https://doi.org/10.1145/3357033
[66] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 406–418. https://doi.org/10.1109/HPCA.2016.7446082
[67] Jiuxing Liu and Bulent Abali. 2009. Virtualization polling engine (VPE) using dedicated CPU cores to accelerate I/O virtualization. In *Proceedings of the 23rd international conference on Supercomputing*.
[68] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu:Just-In-Time Summoning of Unikernels. In *NSDI*.
[69] Anil Madhavapeddy and David J Scott. 2013. Unikernels: Rise of the Virtual Library Operating System. *Queue* 11, 11 (2013).
[70] Toshiyuki Maeda and Akinori Yonezawa. 2003. Kernel Mode Linux: Toward an Operating System Protected by a Type Theory. In *Advances in Computing Science – ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation*, Vijay A. Saraswat (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–17.
[71] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution using Return Stack Buffers. (2018).

[72] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *SOSP*.

[73] Larry W McVoy, Carl Staelin, et al. 1996. LMbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA.

[74] Alyssa Milburn, Ke Sun, and Henrique Kawakami. 2022. You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection. *arXiv preprint arXiv:2203.04277* (2022).

[75] Sparsh Mittal. 2016. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. *ACM Comput. Surv.* 48, 3, Article 45 (feb 2016), 38 pages. https://doi.org/10.1145/2856125

[76] Ingo Molnar and Max Krasnyansky. 2020. SMP IRQ affinity. https://docs.kernel.org/core-api/irq/irq-affinity.html.

[77] Soo-Jin Moon, Vyas Sekar, and Michael K Reiter. 2015. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *CCS*.

[78] Tsing Mui. 2022. Intel CPUs Lose Up to 36% Performance with New Spectre Patch. *The FPS Review* (Mar 2022).

[79] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1481–1498. https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko

[80] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical.. In *USENIX Security Symposium*. 645–662.

[81] Salvador Palanca, Stephen A. Fischer, Subramaniam Maiyuran, and Shekoufeh Qawami. 2002. MFENCE and LFENCE Micro-Architectural Implementation Method and System. US Patent 6,651,151.

[82] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.. In *USENIX Security Symposium*. 565–581.

[83] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. 2021. SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets.. In *NDSS*.

[84] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security*.

[85] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*. Intel Bounty Reward.

[86] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: Process separation for web sites within the browser. In *USENIX Security*.

[87] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. 2021. I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA52012.2021.00036

[88] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354252

[89] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. Springer.

[90] Junaid Shahid. 2022. Address Space Isolation for KVM. https://lore.kernel.org/lkml/20220223052223.1202152-1-junaids@google.com

[91] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *OSDI*, Vol. 10.

[92] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. (2018).

[93] Jakub Szefer, Eric Keller, Ruby B Lee, and Jennifer Rexford. 2011. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*.

[94] Paul Turner. 2018. Retpoline: a Software Construct for Preventing Branch Target Injection. https://support.google.com/faqs/answer/7625886.

[95] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association.

[96] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[97] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. Addendum 1 to RIDL: Rogue In-flight Data Load. In *S&P*.

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

[99] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Addendum 2 to RIDL: Rogue In-flight Data Load. In *S&P*.

[100] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *S&P*.

[101] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).

[102] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3825–3842. https://www.usenix.org/conference/usenixsecurity22/presentation/wikner

[103] Hongyan Xia, David Zhang, Wei Liu, Istvan Haller, Bruce Sherwin, and David Chisnall. 2022. A Secret-Free Hypervisor: Rethinking Isolation in the Age of Speculative Vulnerabilities. In *IEEE S&P*.

[104] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic cache allocation with partial sharing. In *EuroSys*.

[105] Wenjie Xiong and Jakub Szefer. 2021. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys (CSUR)* 54, 3 (2021).

[106] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. 2013. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*.

[107] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. 2018. dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *EuroSys*.