Rethinking Isolation Mechanisms for Datacenter Multitenancy

Varun Gandhi James Mickens
Harvard University

1 Introduction

Multitenancy is the foundation of modern cloud computing: a single datacenter machine must run code from multiple customers. To safely expose such a machine to untrusted tenants, datacenters have traditionally leveraged virtualization [4, 9]. In this approach, privileged hypervisor software (provided by the datacenter operator) mediates tenant access to raw physical resources like RAM and IO devices.

A hypervisor isolates tenants from each other, and isolates the hypervisor from tenants. However, tenants are not isolated from the datacenter operator; the operator's hypervisor can arbitrarily manipulate tenant state, and the operator herself can physically inspect or modify the contents of server RAM. Intel's SGX-enabled processors [13] stop these attacks. Using hardware-enforced memory partitioning, SGX prevents a hypervisor from accessing secure tenant pages. SGX hardware also transparently encrypts and HMACs cache lines during eviction to RAM; thus, a datacenter operator with physical control of a machine cannot see cleartext tenant RAM, or undetectably tamper with the encrypted RAM that *is* visible. SGX is the foundation for a variety of software-level runtimes that isolate datacenter tenants from privileged management software [3, 5, 45, 58].

Unfortunately, SGX-based approaches have three important limitations.

- SGX gives a tenant the illusion of ISA-level isolation. However, tenants cohabitate at the *microarchitectural* level, resulting in side channel vulnerabilities that leak information from ostensibly secure computations (§2.1).
- SGX can cryptographically vouch for the initialization-time integrity of a secure computation. However, SGX has no way to attest a computation's *dynamic* (i.e., *post-load)* integrity. Both initial and post-load integrity are important (§2.2). Clients do not wish to exchange data with "secure" server-side code that was initialized correctly, but subsequently corrupted by a network attacker (e.g., via a ROP exploit [7, 43, 49] or type confusion [8, 30, 31, 35]).
- SGX is implemented using a combination of hardwired circuits and updatable microcode [13]. Unfortunately, Intel hides many implementation details that are necessary to fully understand SGX's security properties (§2.3). In response to high-profile security bugs involving SGX [10–12, 39, 46, 52], Intel has released a variety of microcode patches, many of which were incorrect or incomplete and required subsequent revision [21, 23, 34]. These episodes demonstrate that security researchers need more visibility into the silicon and the microcode that implement hardware-based isolation mechanisms.

Motivated by these problems, we propose a new isolation approach for datacenter multitenancy. As with SGX, we leverage trusted hardware to isolate tenants from each other and from the datacenter operator. However, our approach differs from SGX in three crucial ways.

- First, our trusted hardware strongly isolates each tenant's ISA-level state at the microarchitectural level, removing side channels involving other tenants or the hypervisor.
- Second, we allow a tenant to explicitly bind application code to *monitor code* that dynamically enforces runtime security invariants like control flow integrity. The monitor code runs in parallel with application code; however, the monitor runs on a different CPU pipeline (managed by trusted hardware) that receives a read-only stream of the register state from the application-level pipeline. With the exception of this register mirroring, the two pipelines are isolated at the microarchitectural level. This design prevents side channel leakages of monitor state to application code that might be under attack. Microarchitectural partitioning of an application and its monitor also eliminates more direct attacks that could occur if monitor state were located in the same address space as the application to protect.
- Third, our new trusted hardware uses an open microcode format, and exposes a software-readable description of microarchitectural-level hardware details. This approach allows tenants to independently verify the security properties of a server's hardware. This design also allows tenants to customize monitor code to fully exploit the microarchitectural affordances provided by a particular datacenter server.

In Section 2, we provide more background on SGX and related technologies. We then sketch a preliminary design for our new isolation hardware (§3). We conclude by describing some open research challenges (§4).

2 Background

Intel SGX [13] and ARM TrustZone [2] are the most popular hardware environments for secure execution. We explain their strengths and weaknesses to motivate our new proposal.

SGX: SGX refers to a secure computation as an "enclave." The memory pages for an enclave are stored in a protected region of physical RAM; however, those pages are embedded within the virtual address space of an untrusted host process. To invoke enclave functionality, untrusted code invokes the special `EENTER` instruction. The CPU responds by flipping the CPU's *isEnclave* bit to 1, and then jumping to enclave code. During enclave execution, SGX allows the MMU to access all pages (both enclave and non-enclave) in the un-

trusted host. Enclave code invokes the `EEXIT` instruction to flip `isEnclave` and return control to the untrusted host process. Since `isEnclave` is now set to 0, SGX will prevent the CPU from accessing enclave memory.

When cache lines from enclave memory are evicted to RAM, SGX transparently encrypts and HMACs them. Similarly, when enclave cache lines are pulled from RAM into the L3 cache, SGX transparently decrypts the lines and checks the HMACs to detect tampering. The encryption keys and HMAC keys never leave the SGX hardware; so, even if the datacenter operator physically probes SGX memory, the operator can only discover encrypted, integrity-protected data.

Enclave code runs at Ring 3 (i.e., the least privileged level) and cannot issue system calls. Thus, enclave code relies on the untrusted host process to issue IOs. Enclave code places IO requests in a non-enclave part of the untrusted host's address space. The untrusted host (hopefully) performs the desired IO, places the result in a non-enclave page, and then invokes `EENTER`. Enclave code and IO endpoints (e.g., a local disk, a remote network client) can use cryptography to detect if the untrusted host has tampered with IO. However, denial of service attacks are outside the scope of SGX.

TrustZone: TrustZone partitions memory into a "normal world" and a "secure world." When the CPU executes in secure mode, both secure memory and normal memory are accessible; when the CPU is in normal mode, only normal memory can be accessed. Both worlds support the user/supervisor privilege distinction. Thus, the normal world runs a privileged OS and unprivileged user-mode applications, while the secure world also executes kernel code and user-level code. Privileged code in the secure world determines how physical RAM is allocated to the two worlds. Privileged secure-world code also assigns each IO device to a world. So, privileged code in the secure world can receive device interrupts, use DMA to exchange data with devices, and expose devices to user-level secure-world code. This contrasts with SGX, where secure code is dependent upon untrusted software to perform IO.

In TrustZone, secure code executes in response to the firing of a secure-device interrupt, or the normal-world invocation of the `smc` ("secure monitor call") instruction. Since `smc` is similar to SGX's `EENTER`, it can be used to construct a secure application whose architecture uses an SGX-style split between an "untrusted host" and "secure component" [20, 24, 26].

Current TrustZone hardware does not encrypt or HMAC secure-world RAM. However, the required hardware [25] is not SGX-specific, and could be ported to TrustZone.

Attesting Secure Computations: Attestation [42, 50, 51] allows a secure computation to vouch for the integrity of its initialization-time code and data. Before a remote client exchanges sensitive data with the computation, the client will force the computation to attest.

At a high level, attestation leverages a unique public/private key pair that is possessed by each SGX or TrustZone instance. The public half is certified by the hardware vendor, and the private half is never exposed outside of the trusted hardware. As a secure computation is initialized with code pages and data pages, trusted hardware records the identity of those pages. For example, in SGX, when an untrusted host adds a new memory page to an enclave, the SGX hardware updates a cumulative hash over all such pages. Once enclave initialization is complete, the SGX hardware seals the enclave, preventing further modifications by untrusted code. When a client asks the enclave to attest, the enclave asks the SGX hardware for a signed copy of the cumulative hash. The enclave returns the signed hash to the client, who then verifies that the hash corresponds to a trusted initial enclave state.

2.1 Security challenge #1: Weak Isolation

SGX and TrustZone strongly partition the *ISA-visible* state belonging to different tenants. However, at the *microarchitectural* level, tenants share resources. For example, in SGX, a single hyperthreaded physical core may run enclave code on one logical core, and non-enclave code on another. This design allows enclaves to execute atop the highly-optimized pipelines that were designed in the pre-SGX era. Unfortunately, co-tenancy at the microarchitectural level exposes SGX and TrustZone to a variety of side channel attacks [1, 6, 10, 12, 13, 19, 22, 36, 47, 57]. For example, consider a hyperthreaded SGX core: if enclave code is co-resident with malicious code, then the malicious code can measure contention for shared functional units, and infer the instructions being issued by the enclave [13].

Contention-based side channels also exist in TrustZone. For example, each cache line in TrustZone is tagged with a bit that indicates whether the line belongs to the secure world. Using this bit, the cache controller prevents normal-world code from accessing secure-world cache lines. However, cache lines from both worlds share a unified cache. Thus, malicious code in the normal world can use cache contention as a side channel, e.g., to steal secure-world encryption keys [36, 57].

The TrustZone model has an additional challenge: the secure world can access state in both worlds. Thus, the normal world cannot protect itself from secure-world code that is curious or malicious. Datacenters need the ability to enforce "all-pairs" distrust, such that tenants and privileged management software are all isolated from each other.

2.2 Security challenge #2: Post-load Integrity

The goal of trusted hardware in the cloud is to expose a secure service to remote clients. However, some clients may be evil; network-based attacks are endemic to the modern Internet. What happens if a malicious client successfully corrupts a "trusted" datacenter service? Remote attestation unfortunately provides no guarantees about the dynamic, post-load integrity of a secure computation. Thus, clients have no way to determine whether remote code has control flow integrity [7, 43, 49], type safety [8, 30, 31, 35], or other runtime invariants.

A secure computation can obviously try to enforce those properties itself. For example, a computation might use the HexType runtime [31] to dynamically check C++ casts for type safety. To thwart control-flow hijacks, a secure computation might use Shuffler [54] to dynamically rerandomize code locations. The computation might also use compile-time frameworks for information flow control [40] or page fault masking [48] to eliminate vectors for data leakage. However, in all of these cases, the code and data which enforce security *reside within the address space to protect*. This is troubling, since memory disclosure vulnerabilities (which are the foundation of many attacks [15, 38, 44]) allow process-local, security-sensitive state to be discovered and subverted. For example, managed languages like Go and JavaScript rely on strong types and no-execute page protections to prevent attacker-induced code injection. However, the type system relies on hiding low-level, type-enforcing information from the attacker. If attackers can use a read vulnerability to discover `this` pointers and `vtables`, attackers can escape the type system using carefully-crafted managed code which artfully manipulates low-level type information [16, 44, 49].

Ideally, a trusted service's post-load integrity would be verified by an *external* monitor. Such a monitor would live outside of the service's address space, to provide isolation from the potentially-exploited service. The monitor would also require isolation from the datacenter operator; otherwise, a malicious operator could corrupt the monitor, and then corrupt the trusted service without detection.

2.3 Security challenge #3: Opaqueness

SGX adds a minimal amount of new hardware to a stock x86 chip. The biggest change is the addition of the memory encryption engine which sits between the L3 cache and RAM. However, the bulk of SGX's functionality is implemented via microcode [13]. Microcode instructions are the instructions that are actually executed by the guts of an x86 pipeline. When an ISA-level instruction hits the decode stage, the decoder uses a translation table to map the ISA-level instruction to a corresponding sequence of microcode instructions; the microcode sequence is then released into the downstream pipeline. A processor also uses microcode to automate low-level behaviors like the pushing of registers onto a kernel stack when an exception occurs [33].

Microcode enables two nice features. First, for a given chip design, support for a new instruction like `EENTER` can mostly or fully be added merely by updating the microcode translation table in ROM. Second, by augmenting the translation table ROM with some updatable memory, the behavior of ISA-level instructions can be changed after a chip has already left the factory; this ability is helpful if an instruction's factory-preset microcode translation is later found to contain bugs. Indeed, Intel has issued microcode patches in response to recent processor vulnerabilities like Spectre [32] and Meltdown [37].

Unfortunately, Intel's microcode is encrypted, which prevents outside security experts from vetting patches. The microarchitecture that is controlled by microcode is also partially opaque; Intel-published optimization manuals [28] are selective in revealing low-level details. As a result, datacenter tenants have a difficult time reasoning about the security properties of SGX. Tenants also lack deep insights into the behavior of patches for SGX microcode. Thus, tenants must rely on Intel to properly implement the SGX specification.

Doing so is hard, given that Intel has been forced to revise incorrect security patches. A particular patch has sometimes been revised multiple times. For example, in 2018, various security researchers discovered a family of attacks which Intel calls "microarchitectural data sampling" [29]. The attacks target store buffers (which hold queued memory writes), fill buffers (which are used to manage L1 cache misses), and load ports (which are functional units that fetch data from memory). At a high level, the attacks exploit (1) out-of-order execution and (2) microarchitectural co-tenancy of code from different trust domains. For example, if kernel code pulls a kernel cache line into a fill buffer, then malicious code from a co-resident, unprivileged tenant can try to read the line. Due to out-of-order execution, there is a race condition between the access check on the malicious memory load, and the execution of other malicious instructions which update a microarchitectural side channel based on the value that was loaded. The processor will squash any *architectural* side effects of the malicious instruction stream, but, depending on the outcome of the race condition, data may have already leaked through the microarchitectural side channel.

In response to this family of problems, Intel released a microcode patch in May 2019. However, the patch still allowed some known attacks in the family. Security researchers agitated, and Intel released a new version in November 2019. Alas, some attacks were still possible, and Intel released yet another revision of the patch in January 2020 [23].

At the time of this paper's writing, security researchers were still determining whether the latest patch is sufficient. However, the history of the patch is a troubling one for datacenter tenants who cannot inspect the cleartext contents of Intel-provided patches, and do not have full visibility into the microarchitecture that patches modify. In fairness to Intel, we believe that Intel *wants* to make their chips secure. However, recent hardware exploits suggest that vetting a microarchitecture is too complex for a single company to do well. This implies that microarchitectures (and microcode patches) should be opened up for vetting by external parties.

3 Our Proposal: Isolated Monitor Execution

Figure 1 illustrates our high-level proposal. A developer writes application code to deploy on a remote machine; concurrently (and perhaps with assistance from automated tooling), the developer creates a *dynamic integrity checker* for the application. The integrity checker examines the instruction

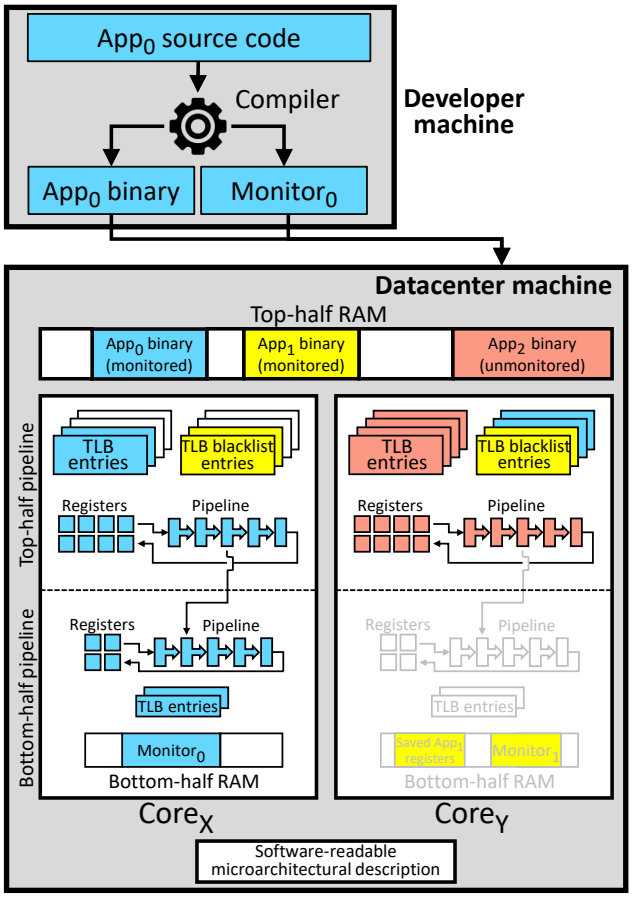

Figure 1: High-level overview of our proposed IME system. We extend each core on a datacenter machine to include a "bottom-half" pipeline. This pipeline, managed by IME's trusted hardware, runs monitor code on behalf of the secure application that runs on the traditional "top-half" pipeline. Monitor code runs integrity checks over a read-only stream of instructions, operands, and results from the top-half pipeline; otherwise, the bottom-half pipeline is microarchitecturally isolated from all top-half software (including management software run by the datacenter operator). SGX-style TLB blacklisting and memory encryption protect top-half pages belonging to secure applications.

stream and register state of a running application to ensure that the application has not been compromised. In the rest of this paper, we refer to the integrity checker as the "monitor."

The developer uploads the application and its monitor to a datacenter machine. The machine has our new trusted hardware, which we call *Isolated Monitor Execution* (IME). The application runs on the machine's normal CPU pipeline. However, the monitor runs on a separate pipeline managed by IME. This "bottom-half" pipeline executes the monitor in parallel with the monitored application. However, the bottom half shares almost no microarchitectural state with the top half, eliminating most side channels that would expose monitor state to untrusted code running on the normal pipeline. The bottom-half pipeline is limited to accessing a read-only stream of the executed top-half instructions; the stream includes the register values and memory values that are read and written by each instruction. Monitor code defines security invariants with respect to this input stream. If a monitor detects that a retiring top-half instruction would break a security invariant, the monitor asks IME to kill the top-half application. IME will then flush the top-half microarchitectural state belonging to the application. IME will also zero-fill the top-half, ISA-visible state associated with the application, e.g., resetting registers and memory pages.

To create a monitored application, untrusted management software uses IME instructions to inform IME about the initial state in the application and its monitor. Initial application state is added to top-half RAM, and initial monitor state is added to bottom-half RAM. Once initialization is complete, the application+monitor pair (which we call a *dyad*) can be executed using an instruction akin to SGX's `EENTER`.

To isolate the application's top-half memory from other top-half software, IME uses TLB blacklists and strict cache partitioning. TLB blacklists prevent a core from accessing any top-half RAM that belongs to a monitored application. Importantly, blacklists are managed by IME hardware, not by untrusted top-half management software. When a core explicitly asks IME to context switch to dyad d_i, IME temporarily removes the blacklist entries for d_i's top-half memory pages; those entries are restored if the application code willingly yields, or is forced off the CPU due to a hardware interrupt. Upon either type of yield event, IME stores the top-half register state in protected bottom-half memory before vectoring control to untrusted system software. Upon resuming a dyad, IME pushes the application's saved register context onto the top-half pipeline, and restarts that pipeline.

Monitor code is event-driven. The only event type is the incrementing of the top-half program counter. A monitor associates top-half PC values of interest with bottom-half monitor routines. A single routine can be mapped to multiple top-half instructions. For example, a naïve CFI monitor would map each indirect jump to a monitor routine that checks whether the attempted control transfer is a valid one. We discuss other potential monitors in Section 4.

The bottom-half RAM for a core may contain state for multiple monitors, each one bound to a different dyad. When dyad d_i runs, IME uses bottom-half TLB banks to ensure that d_i's monitor can only access d_i's bottom-half RAM. IME flushes bottom-half microarchitectural state when context-switching away from a dyad. Since at most one monitor is active on a core at any given time, flushing is sufficient to eliminate side channels in the bottom-half microarchitecture.

Note that there is no privileged software that runs on bottom-half hardware. Trusted IME hardware manages the context switches between monitors, and the enforcement of monitor isolation via TLBs. So, unlike the top-half pipeline, the bottom half pipeline does not require hardware-managed

TLB blacklists, because there is no bottom-half OS that might try to install inappropriate memory mappings.

An IME-compatible processor uses an open microcode format. As shown in Figure 1, an IME-compatible processor also exports a software-readable description of its microarchitectural details. These two features enable detailed offline vetting of IME's security properties by independent researchers. These features also make attestation more useful. When IME generates an attestation message for a dyad, the signed hash covers the microcode ROM, any microcode patches, the processor's microarchitectural description, and the memory pages in the dyad's initial state. This extended attestation message provides additional context for a client's evaluation of server-side dyad security.

Compared to Sanctum-based RISC-V processors [14], IME provides more radical features at the cost of additional die area. For example, Sanctum implements enclave isolation via open-source monitor code that runs at RISC-V's machine level [53]; however, Sanctum does not provide software-readable descriptions of the underlying microarchitecture. Like IME, Sanctum uses cache partitioning to eliminate cache-based side channels involving secure code; however, Sanctum does not eliminate side channels via other shared microarchitectural resources like hyperthreaded functional units. Most importantly, Sanctum provides no way to run integrity checks in an environment that is isolated from the application to be checked.

4 Open Challenges

A variety of research challenges must be solved to make IME practical. The challenges span topics in systems, hardware, and programming languages.

High-performance networking: An SGX enclave relies on the untrusted OS to perform IO. This approach adds latency and enables denial-of-service attacks. In TrustZone, secure-world code can directly handle interrupts and otherwise service devices; however, since the secure world is all-powerful, applications in the normal world are not protected from misbehaving secure-world code. Ideally, IME could virtualize network cards, e.g., by extending SR-IOV [27]; the goal would be to provide each dyad with an isolated virtual NIC that could be accessed directly. Enabling DMA between a monitored top-half application and an SR-IOV-style NIC would require careful synchronization between top-half TLB blacklists and the IOMMU settings of the NIC. A commodity SR-IOV NIC may also have side channels (e.g., via the internal L2 switch) that must be eliminated.

Useful, implementable policies: Co-tenancy in memory hardware is a common source of side channels (§2.1). Thus, IME isolates the memory hierarchies in the top half and the bottom half.[1] An important consequence is that monitor code in the bottom half cannot independently read or write top-half memory. Monitor code *does* receive a read-only stream of top-half instructions (and the inputs and outputs of those instructions). So, an important research question is "what useful security policies can be enforced solely with a trace of a program's instruction stream, register values, memory loads, and memory stores?" Control flow integrity, software fault isolation [55], and seccomp-style system call filtering [17] are easily supported by IME, because these policies are naturally expressed in terms of instruction stream values at certain moments in program execution. Taint tracking [18] is possible, since a monitor can store shadow bytes in bottom-half monitor pages; however, bottom-half RAM will likely be much smaller than top-half RAM, preventing large shadow maps.

Multi-threaded applications: Suppose that a single application process has multiple kernel-visible threads. What is the best way to monitor such a process? In the tentative design from Section 3, bottom-half pipelines do not share a memory hierarchy. This design eliminates cross-monitor side channels, but forces each thread in a multithreaded process to receive an isolated monitor instance—the monitor instance cannot share data with other monitors belonging to sibling application threads. Thus, IME could not enforce cross-thread security invariants, e.g., to ensure that thread activity conforms to desired state machine transitions [41, 56]. Compilers may be able to reflect such cross-thread behaviors into thread state that is eventually reflected into a thread's top-half registers (providing visibility to the thread's monitor). However, the specifics of this approach require further investigation. An alternative solution is for IME to provide a shared bottom-half memory hierarchy that partitions bottom-half RAM and cache lines in the same way that top-half RAM and cache lines are partitioned. This approach eliminates side channels, but requires more die area and energy consumption.

Dyad migration: A single datacenter machine has multiple cores. To manage power consumption and balance load, a datacenter operator would like to spin those cores up and down as necessary. However, if a machine lacks a shared bottom-half memory hierarchy, dyad migration across local cores becomes more expensive—an application thread must tear down its monitor on core X, and then reinitialize a new monitor on core Y, copying memory pages and making several context switches into and out of the kernel. Even if a shared bottom-half memory hierarchy exists, synchronizing migration between two cores is nontrivial if done purely in hardware (e.g., via microcode). Further research is needed to understand the expected frequency of dyad migration, and the best way to accomplish it.

Note that dyad migration across different *machines* is also important. IME will require hardware support and new cryptographic network protocols to securely snapshot, transport, and install dyad state. The network protocols will likely extend traditional approaches for remote attestation (§3).

[1] To enable monitor initialization, IME provides a dedicated, page-sized hardware buffer that untrusted top-half code can use to initialize a bottom-half monitor page.

5 Desired Feedback

We hope that our paper will inspire discussion about what the next generation of trusted datacenter hardware should look like. The security problems of TrustZone (and particularly SGX) are well-publicized. However, the next steps forward are unclear. Various research projects have focused on attacking SGX and TrustZone, or defending against those attacks at a software level; unfortunately, much of this work assumes that SGX and TrustZone will not be changing in fundamental ways. At HotCloud, we hope to spark conversations about what fundamental change might look like.

Section 4 describes some known challenges for our proposed IME architecture. We would appreciate feedback on those specific topics. Below, we briefly discuss additional interesting problems.

The costs of strong isolation: IME introduces trade-offs between computational performance, side-channel isolation, and die area. For example, how much RAM should be allocated to the bottom half? How complex should a bottom-half pipeline be—is a simple five-stage pipeline acceptable, or is a complex, out-of-order pipeline necessary? Provisioning too few bottom-half resources will induce stalls in the top-half pipeline, as retiring top-half instructions block on the verdict of the associated monitor code. This blocking will become a side-channel unless monitors are carefully designed! Providing copious computational resources to the bottom half would reduce top-half stalls, but increase overall hardware budgets.

System calls: In SGX, an enclave cannot make system calls. Instead, the enclave relies on the untrusted host process to invoke system calls on the enclave's behalf; system call arguments and results are placed in an untrusted host page. This approach incurs an extra context switch for every system call invocation and return. Ideally, IME would allow a dyad's top-half code to directly invoke system calls and examine the results. A possible design is for the dyad's top-half code to designate a memory page as a "system call page." Top-half code would register the page with both the top-half OS and IME's bottom-half hardware. The OS must be informed so that, when the top-half code issues a system call, the OS knows where system call arguments and results should be. IME's bottom-half hardware must be informed about the page so that, when context-switching to the associated dyad, the bottom-half can mark the system call page as readable and writable by privileged software. When context switching *away* from the dyad, the bottom-half must add the system call page to the global TLB blacklist.

Datacenter-specific IME hardware: Datacenter operators have recently begun to design custom server hardware that is tailored to operator-specific workloads. To what extent can IME do the same, e.g., to optimize bottom-half hardware for monitor checks that are particularly important for datacenter operators?

References

[1] A. Aldaya, B. Brumley, S. ul Hassan, C. Garcia, and N. Tuveri. Port Contention for Fun and Profit, November 1, 2018. Cryptology ePrint Archive: Version 20181106:170703. https://eprint.iacr.org/2018/1060.

[2] ARM. ARM Security Technology: Building a Secure System using TrustZone Technology, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M.L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of OSDI*, pages 689–703, November 2016.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of SOSP*, October 2003.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of OSDI*, pages 267–283, October 2014.

[6] J. Bech, A. Biesheuvel, M. Brown, and D. Thompson. Implications of Meltdown and Spectre : Part 2, February 7, 2018. Linaro blog. https://www.linaro.org/blog/meltdown-spectre-2/.

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking Blind. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2014.

[8] F. Brown, S. Narayan, R.S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and Preventing Bugs in JavaScript Bindings. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 559–578, May 2017.

[9] E. Bugnion, J. Nieh, and D. Tsafrir. Hardware and Software Support for Virtualization. *Synthesis Lectures on Computer Architecture*, 38, 2017.

[10] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T.F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of USENIX Security*, pages 991–1008, August 2018.

[11] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss,

and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020.

[12] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T.H. Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution, June 3, 2018. arXiv: Version 1802.09085v3. https://arxiv.org/abs/1802.09085.

[13] V. Costan and S. Devadas. Intel SGX Explained, February 20, 2017. Cryptology ePrint Archive: Version 20170221:054353. https://eprint.iacr.org/2016/086.pdf.

[14] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of USENIX Security*, pages 857–874, August 2016.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 763–780, May 2015.

[16] S.J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, and B. De Sutter. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of CCS*, pages 243–255, October 2015.

[17] J. Edge. A seccomp overview, September 2, 2015. LWN. https://lwn.net/Articles/656307/.

[18] W. Enck, P. Gilbert, B.-Y. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI*, pages 393–407, October 2010.

[19] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of ASPLOS*, pages 693–707, March 2018.

[20] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software. In *Proceedings of SOSP*, pages 287–305, October 2017.

[21] J. Fingas. Intel fixes CPU security flaw it said was patched in May, November 13, 2019. Engadget. https://www.engadget.com/2019/11/13/intel-fixes-cpu-security-flaw-for-real/.

[22] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proceedings of USENIX Security*, pages 995–972, August 2018.

[23] A. Greenberg. Intel Is Patching the Patch for the Patch for Its "Zombieload" Flaw, January 27, 2020. Wired. https://www.wired.com/story/intel-zombieload-third-patch-speculative-execution/.

[24] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proceedings of MobiSys*, pages 488–501, June 2017.

[25] S. Gueron. Memory Encryption for General-Purpose Processors. *IEEE Security and Privacy Magazine*, 14:54–62, Nov–Dec 2016.

[26] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In *Proceedings of USENIX Security*, pages 541–556, August 2017.

[27] Intel. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology, January 2011. Revision 2.5.

[28] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, June 2016. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[29] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, 2020. Intel Developer Zone. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling.

[30] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of NDSS*, February 2014.

[31] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of CCS*, pages 2373–2387, October 2017.

[32] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2019.

[33] P. Koppe, B. Kollenda, M. Fyrblak, C. Kison, R. Gawlik, C. Paar, and T. Holz. Reverse Engineering x86 Processor Microcode. In *Proceedings of USENIX Security*, pages 1163–1180, August 2017.

[34] P. Kunert. If at first you don't succeed, you're likely Intel: Second Spectre microcode fix emitted, February 21, 2018. The Register. https://www.theregister.co.uk/2018/02/21/intel_spectre_2_microcode_patch/.

[35] B. Lee, C. Song, T. Kim, and W. Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of USENIX Security*, pages 81–96, August 2015.

[36] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proceedings of USENIX Security*, pages 549–564, 2016.

[37] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of USENIX Security*, pages 973–990, August 2018.

[38] K. Lu, C. Song, B. Lee, S.P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of CCS*, pages 280–291, October 2015.

[39] K. Murdock, D. Oswald, F.D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020.

[40] A. Myers and B. Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[41] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-Parallel Finite-State Machines. In *Proceedings of ASPLOS*, pages 529–542, March 2016.

[42] B. Parno, J.M. McCune, and A. Perrig. Bootstrapping Trust in Modern Computers, 2011. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/BootstrappingTrustBook.pdf.

[43] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1), March 2012.

[44] R. Rogowski, M. Morton, F. Li, F. Monrose, K.Z. Snow, and M. Polychronaki. Revisiting Browser Security in the Modern Era: New Data-only Attacks and Defenses. In *Proceedings of the IEEE European Symposium on Security and Privacy*, pages 366–381, April 2017.

[45] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 38–54, May 2015.

[46] M. Schwarz, M. Lipp, D. Moghimi, J. van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of CCS*, pages 753–768, November 2019.

[47] D. Shen. Exploiting Trustzone on Android. In *Black Hat*, August 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf.

[48] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of NDSS*, February 2017.

[49] K.Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 574–588, May 2013.

[50] Trusted Computing Group. TCG Infrastructure Working Group Architecture Part II: Integrity Management, November 17, 2006. Specification Version 1.0, Revision 1.0.

[51] Trusted Computing Group. TCG Attestation PTS Protocol: Binding to TNC IF-M, November 24, 2011. Specification Version 1.0, Revision 28.

[52] S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue In-Flight Data Load. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020.

[53] A. Waterman and K. Asanovic. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, May 7, 2017. Version 1.10. https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

[54] D. Williams-King, G. Gobieski1, K. Williams-King, J.P. Blake, X. Yuan, P. Colp, M. Zheng, V.P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of OSDI*, pages 367–382, November 2016.

[55] B. Yee, D. Sehr, G. Dardyk, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2009.

[56] D. Zhang. From Concurrent State Machines to Reliable Multi-threaded Java Code, April 2018. PhD thesis. Technische Universiteit Eindhoven.

[57] N. Zhang, K. Sun, D. Shands, W. Lou, and Y.T. Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices, 2016. Cryptology ePrint Archive: Report 2016/980. https://eprint.iacr.org/2016/980.pdf.

[58] W. Zheng, A. Dave, J.G. Beekman, R.A. Popa, J.E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of NSDI*, pages 283–298, March 2017.