Sanctum: Minimal RISC Extensions for Isolated Execution

Victor Costan, Ilia Lebedev, and Srinivas Devadas
MIT CSAIL

June 8, 2015

ABSTRACT

Sanctum is a set of minimal extensions to a standard RISC architecture that offers strong provable isolation of software modules running concurrently and sharing resources. Sanctum is similar to SGX in its API, but protects against an important class of additional software attacks, including cache timing and memory access pattern attacks. It does so via a principled approach to eliminating entire attack surfaces through isolation rather than plugging attack-specific privacy leaks.

Sanctum's hardware changes over a standard RISC architecture do not impact the cycle time, as they do not extend critical execution paths. Sanctum does not change any major CPU building block (e.g., ALU, MMU, cache), and only requires additional hardware at the interfaces between these building blocks corresponding to less than two percent chip area overhead. Over a set of benchmarks, Sanctum's worst observed overhead for isolated execution is 14.6% over an idealized insecure baseline.

1 MOTIVATION

Between the Snowden revelations and the seemingly unending series of high-profile hacks of the past few years, the public's confidence in software systems has decreased considerably. At the same time, key initiatives such as cloud computing and the IoT (Internet of Things) require users to trust the systems providing these services. We must therefore develop capabilities to build software systems with better security, and gain back our users' trust.

1.1 The Case for Hardware-Assisted Isolation

Formal software verification produces provably bug-free software, but carries a prohibitively large cost. For example, the seL4 formal verification effort [1], spent *20 man-years* to cover 9,000 lines of code. For comparison, the Linux 3.10 kernel has over *17 million* lines of code [2], while the Xen hypervisor has around 150,000 lines of code [3].

The amount of code that requires formal verification in a system can be reduced by modularizing the system and applying information flow control techniques that reduce the number of modules that can be impacted by a compromised module. For example, Quark [4] is a secure browser whose security argument relies on a small formally verified kernel, and information flow control between modules. Quark was able to avoid formally verifying its rendering engine (WebKit), which has millions of lines of code.

Fundamentally, all information flow control assumes a trusted method for isolating modules in a system, such as the process abstraction enforced by the OS kernel. This assumption breaks down when an attacker compromises privileged software such as the kernel or hypervisor. Unfortunately, in **each** of the last three years (2012-2014), over 100 security vulnerabilities were discovered in Linux [5, 6], and over 40 in Xen [7]. Given the dire prospects for applying formal verification to software like Linux and Xen, we must design systems with the expectation that the hypervisor and kernel can and will be compromised by motivated attackers.

SecureBlue++ [8] and Intel Software Guard Extensions (SGX) [9, 10] have shown that we can avoid trusting a hypervisor or OS kernel, as long as we're willing to trust the hardware support for isolation built into a CPU.

1.2 A Call for Trustworthy Trusted Hardware

Relying on hardware (such as an SGX-enabled processor) for isolation eliminates the hypervisor and OS kernel from a system's trusted code base (TCB), but includes into the TCB the hardware and microcode that implements the CPU's isolation mechanism. A vulnerability in this complex hardware may trivially undermine the security of the entire system. Thus, a rigorous security analysis of a system must account for both the software and the hardware in the TCB. Unfortunately, a system whose TCB includes black-box hardware with opaque microcode precludes any security analysis.

For example, while SGX provides some isolation from a malicious OS or hypervisor, it does not aim to protect the enclave software's memory access patterns [11]. Cache timing attacks (§ 3.1) are summarily bundled and

dismissed together with other side-channel attacks, such as power consumption analysis, which require physical access to the computer running the victim software.

Alarmingly, *cache timing attacks require only unprivileged software running on the victim's host computer*, and do not rely on any physical access to the machine. This is particularly concerning in a cloud computing scenario, where gaining software access to the victim's computer only requires a credit card [12], whereas physical access is a harder prospect, requiring tresspass, coercion, or social engineering on the cloud provider's employees.

If the SGX successor claimed to protect against cache timing attacks, substantiating such a claim would require an analysis of its hardware and microcode, and ensuring that no implementation detail is vulnerable to cache timing attacks. Barring a highly unlikely shift to open-source hardware from Intel, such analysis will simply never happen.

Fortunately, the architecture research community has also produced open processor designs. For example, the Rocket Chip generator [13] is completely open-source, and implements the RISC V [14, 15] open ISA. Extending a Rocket Chip with hardware support for software module isolation yields a system whose entire TCB is public and can be freely scrutinized by the research community.

2 Overview

This paper presents Sanctum, a minimal set of architectural extensions that can be used to implement isolated execution containers that provide meaningful security guarantees against software attacks. We call our containers enclaves, to reflect the similarity with SGX's isolated execution containers.

Sanctum aims to present the same model as SGX to software developers. Programmers are expected to separate applications into sensitive modules, decrypting a patient's X-ray and executing an image processing algorithm, for example, and non-sensitive modules, such as receiving encrypted X-ray images over the network and storing the encrypted images in a database. An application's sensitive modules must be executed within enclaves to have guarantees of isolation from the rest of the software running on the machine.

An enclave has the same memory access privileges as its host application, and is therefore confined by the isolation barriers set up by the OS kernel. Enclaves cannot make direct system calls; enclave code must be linked against a Sanctum-aware standard library (akin to `libc` for C programs) that relies on the host application to proxy operating system calls such as filesystem and network I/O requests.

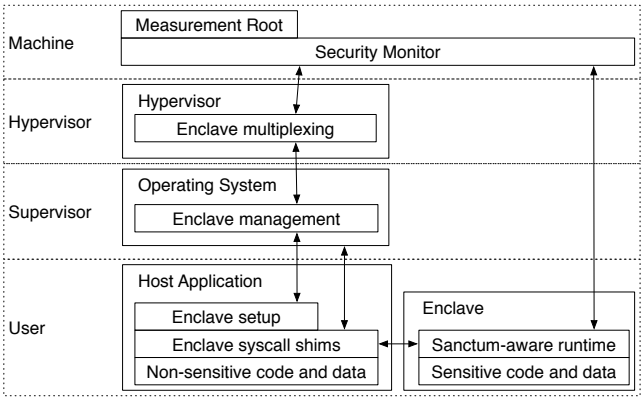

Figure 1: The software stack in a Sanctum environment

Figure 1 shows how enclaves fit in a computer's software stack. Sanctum replaces SGX's undocumented microcode with a trusted software *security monitor* that can be read, and even replaced, by the machine owner. The monitor enforces software isolation in Sanctum, and runs at the highest privilege level, so that it cannot be attacked by a compromised OS or hypervisor.

Our design preserves the operating system's role as the manager of computer resources. The OS makes memory and CPU core allocation decisions, and the security monitor confines enclaves to the resources assigned to them by the OS.

Sanctum is compatible with hypervisors, which are expected to multiplex the enclave-related API calls made by their guest OSes. Furthermore, hypervisors can use Sanctum's cache isolation primitive, the DRAM region, to protect against cross-VM cache timing attacks [16].

For simplicity, the rest of this paper uses the term *operating system* to refer to the system software that manages the computer's resources, whether it is an OS kernel or a hypervisor.

2.1 Protection Boundaries

Sanctum's isolation protects the privacy and integrity of an enclave's software, even in the face of a malicious operating system. We improve upon SGX by isolating the cache sets and page tables used to access an enclave's private memory. The improved isolation defeats attacks that exploit the memory access pattern information leaks that result from cache and page table sharing.

Our isolation is also stronger than SGX's with respect to fault handling. While SGX sanitizes the information that an OS receives during a fault, we achieve full isola-

tion by having the security monitor route the faults that occur inside an enclave to that enclave's fault handler. This removes all information leaks via the fault timing channel.

Strong isolation in Sanctum allows us to give software developers a simple model for reasoning about applications: *all computation that executes inside an enclave, and only accesses data inside the enclave, is protected from any attack mounted by software outside the enclave*. All communication with the outside world, including accesses to non-enclave memory, is subject to attacks.

We assume that the Sanctum-aware standard library linked with an enclave implements the security measures needed to protect the enclave's communication with other software modules. For example, any algorithm's memory access patterns can be protected by ensuring that the algorithm only operates on enclave data. The library can implement this protection simply by copying any input buffer from non-enclave memory into the enclave before computing on it.

2.2 Threat Model

Sanctum protects the integrity and privacy of the code and data inside an enclave against an adversary that can carry out any practical software attack. We assume that an attacker can compromise the operating system and hypervisor (if present) on the computer executing the enclave, and can launch rogue enclaves. We assume that the attacker has access to all the architectural and micro-architectural implementation details of the target computer. Our attacker can use this knowledge to both analyze passively collected data, such as the information provided when a fault occurs, and mount active attacks, such direct memory probing, memory probing via DMA transfers, and cache timing attacks.

Sanctum also protects the operating system against an attacker who can compromise an application and cause it to ask the OS to execute malicious code inside an enclave. This should alleviate current concerns that malware will become unstoppable once it finds it way inside an enclave [17, 18].

Lastly, Sanctum protects against a malicious infrastructure owner who modifies a security monitor, loads the modified version into a computer, and then attempts to either obtain the attestation key for the original security monitor, or attempts to convince a third party that the computer runs the unmodified monitor via the attestation process. The infrastructure owner is allowed to run any combination of hypervisor, operating system, applications and malicious enclaves.

We do not prevent timing attacks that exploit limited cache coherence directory bandwidth or limited DRAM bandwidth. We defer protection against these attacks to future work.

Sanctum does not protect against denial-of-service (DoS) attacks carried out by compromised system software, as malicious system software may deny service by refusing to allocate any resources to an enclave. We *do* protect against DoS attacks carried out by malicious enclaves against an uncompromised OS, as the operating system is always able to take away all CPU cores from an enclave, and then delete the enclave.

We assume a correct implementation of the underlying hardware, so we do not protect against software attacks that exploit hardware bugs, such as rowhammer [19, 20] and other fault-injection attacks.

Sanctum's isolation mechanisms exclusively target software attacks. In Section 3, we note related work that can harden a Sanctum implementation against some types of physical attacks. Furthermore, we consider software attacks that rely on access to sensor readings as physical attacks. For example, we are not concerned with information leakage due to power consumption variations, because software would require a temperature or current sensor to carry out such an attack.

2.3 Security Primitives

Sanctum uses strong isolation to defeat information leaks. Enclaves that execute concurrently on different cores are isolated in the last-level cache (LLC) using a simple partitioning scheme (§ 4.1). Page table sharing is removed by having each enclave map its physical memory with its own page tables (§ 4.2). We achieve isolation in private caches, such as TLBs and the L1 caches, by having the security monitor flush these caches on each context switch that involves an enclave.

Our hardware modifications target the DMA master (§ 4.4) and the interfaces to the last-level cache (LLC) (shown in Figure 2), as well as the interfaces between the memory management unit's (MMU) page walker, the translation lookaside buffers (TLBs), and the L1 data cache (shown in Figure 3). The changes are so small that we present the corresponding circuits, in their entirety, in Figures 7 and 8 .

We interpose on the interface between the LLC and the core-private caches to tweak the mapping between physical addresses and LLC sets, so that the computer's DRAM is split into many equal-sized regions, and the

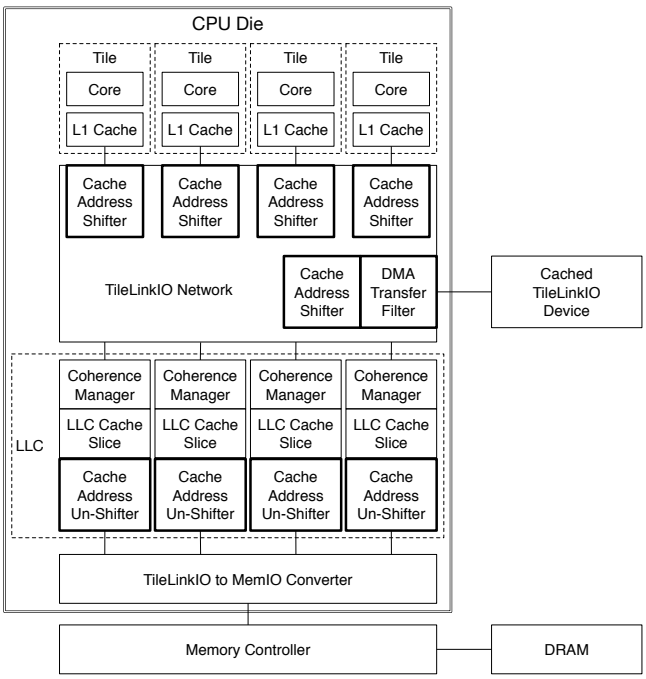

Figure 2: Sanctum's cache address shifter and DMA transfer filter logic in the context of a RISC V Rocket uncore

addresses in each DRAM region use distinct LLC sets (§ 4.1). We augment the interface between the TLBs and the MMU page walker with registers supporting per-enclave page tables (§ 4.2), and we add some logic to the interface between the page walker and the L1 cache to provide a method for constraining a page table to a set of DRAM regions (§ 4.3). We modify the DMA master to reject DMA transfers that fall outside a safe range of memory addresses set by the security monitor (§ 4.4).

We authenticate enclaves using the same principles as earlier secure processors such as Aegis [21] and SGX. Each Sanctum processor has an asymmetric key pair, and a certificate from the manufacturer for its public key. After an enclave is started, it can obtain an attestation intended to convince a remote party that it is communicating to that specific enclave running in a trusted environment. The attestation is a signature chain that starts at the manufacturer's trusted root key, and ends with a signature that covers the remote party's challenge nonce, the enclave's measurement (a cryptographic hash of the enclave's initial state), and a value produced by the enclave, which is generally used to start a key exchange protocol such as Diffie-Hellman [22].

Sanctum's attestation chain starts with the asymmetric key pair built into the processor. The next link in the chain is the *measurement root* (§ 5.1), a piece of trusted software that is burned into the processor's ROM. The

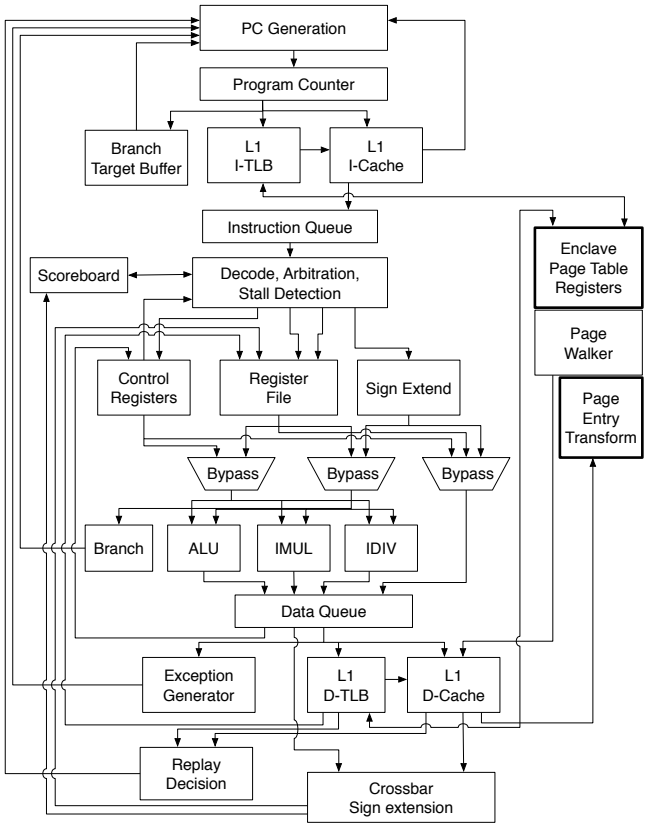

Figure 3: Sanctum's page entry transformation logic in the context of a RISC V Rocket core

measurement root contains the first instructions executed by a processor after it is powered on or reset, and its main job is to compute a measurement of the security monitor (a cryptographic hash) and add it to the attestation chain. The monitor produces enclave attestations.

3 Related Work

3.1 Cache Timing Attacks

We are particularly concerned with a powerful class of software attacks, called cache timing attacks [23], which can be mounted entirely by unprivileged software that measures the latency of its memory accesses and determines whether the accesses caused misses in a cache that is shared with a victim program. The attacker carefully plans memory accesses to reveal the victim's memory access pattern.

Cache timing attacks break the software module isolation assumption in information flow control systems. The attacks do not access the victim's memory directly, so they are *not* prevented by the software isolation mechanisms implemented in today's kernels and hypervisors. Therefore, protecting against these attacks requires a stronger notion of isolation than virtual memory.

Cache timing attacks are known to retrieve cryptographic keys used by AES [24], RSA [25], Diffie-Hellman [26], and elliptic-curve cryptography [27]. Early attacks required access to the victim's CPU core, but more sophisticated recent attacks [28, 29] target the last-level cache (LLC), which is shared by all cores on the same chip package. Recently, [30] demonstrated a cache timing attack that uses JavaScript code in a page visited by a Web browser.

Given this pattern of vulnerabilities, ignoring cache timing attacks is dangerously akin to ignoring the string of demonstrated attacks which led to the deprecation of SHA-1 [31–33]. We must also recognize the imminent threat, and remove the root cause of resource sharing between mutually distrustful processes.

3.2 Secure Processors

We draw inspiration from a long line of secure architectures. XOM [34] introduced the idea of having sensitive code and data execute in isolated containers, and suggested that the operating system should be in charge of resource allocation, but cannot be trusted. Aegis [21] relies on a trusted security kernel, and identifies the software in a container by computing a cryptographic hash over the initial contents of the container. Aegis also computes a hash of the security kernel at boot time and uses it, together with the container's hash, to attest a container's identity to a third party, and to derive container keys. Unlike XOM and Aegis, Sanctum protects the memory access patterns of the software executing inside the isolation containers.

Sanctum only considers software attacks in its threat model. If resilience against hardware attacks is desirable, a Sanctum processor can be augmented with the countermeasures described in other secure architectures. Aegis protects a container's data when the DRAM is untrusted, and Ascend [35] uses Oblivious RAM [36] to protect a container's memory access patterns against adversaries that can observe the addresses on the memory bus.

3.3 Attempts to Secure Existing Designs

We also learned from the pitfalls experienced by various industry attempts to add security features to x86 processors.

The Trusted Platform Module (TPM) [37] proved that software attestation with CPU hardware changes is impractical, as the attestation hash covers too much software.

Intel's Trusted Execution Technology (TXT) [38] proved the impact of leaving trusted software out of the measurement hash, as it navigated the patching of security vulnerabilities [39] in the software used to reset the computer to a known state before entering a protected VM. TXT demonstrated the need for a trusted mechanism for blocking DMA transfers from/to isolated containers, as it fell to attacks [40, 41] where a malicious OS directed a network card to access data in the protected VM. TXT's System Management Mode (SMM) vulnerabilities [42–46] showed the dangers of incomplete isolation, and the difficulty of implementing meaningful isolation in a complex system.

SGX avoids the fallacies described above, and tackles many of the issues brought by multi-core processors with a shared, coherent last-level cache. SGX introduces a method for verifying an OS-conducted TLB shoot-down, and a clever scheme for having an authenticated tree whose structure is managed by an untrusted OS. SGX avoids changes on critical execution paths by gating the TLBs and only performing access controls at address translation time. Unfortunately, the SGX papers do not describe any of these innovations in great detail, and we had to re-construct them from various hints.

SGX's memory management scheme exposes page-level memory access patterns to the OS. Furthermore, SGX enclaves are vulnerable to cache timing attacks that can be performed by unprivileged software running on the same chip. This places a huge burden on software developers, as they have to limit their code to data-independent memory accesses. For example, the EPID [47] signature scheme used by the SGX attestation process is implemented in a signing enclave. If the EPID implementation makes data-dependent memory accesses, an attacker could potentially compromise SGX by extracting the processor's EPID signing key.

Because SGX was proposed by Intel, it gained a lot of attention from the industry and academia. In order to capitalize on that, Sanctum reuses SGX's terminology and enclave API whenever possible.

3.4 Defenses Against Cache Timing Attacks

The research community has brought forward various defenses against cache timing attacks. PLcache [48, 49] and the Random Fill Cache Architecture [50] were designed and analyzed in the context of a small region of sensitive data, and scaling them to protect a potentially large enclave without compromising performance is not straightforward. RPcache [48, 49] relies on a trusted operating system to assign different hardware process IDs to mutually mistrusting entities, and uses a mech-

anism that does not directly scale to large LLCs. The non-monopolizable cache [51] uses a well-principled partitioning scheme to protect against timing attacks, but is not zero leakage and also trusts the operating system to assign hardware process IDs.

Prior work does not address the general problem of protecting any software placed inside an enclave against timing attacks when the OS is untrusted. We introduce a simple partitioning scheme that isolates the software inside each enclave from all other software on the computer. The rest of the Sanctum design does not hinge on the details of the partitioning scheme. It is likely that sophisticated schemes like ZCache [52] and Vantage [53] can be combined with Sanctum's framework to yield better performance.

3.5 Software Defect Mitigation

Sanctum relies on trusted software, consisting of a measurement root and a security monitor. While non-trivial, the software is smaller (at <5kloc) than seL4 [1], which was formally verified.

We assume that enclave software will not willingly disclose its secrets. This can be impractical, given the inevitable presence of bugs. Fortunately, the approach of Native Client [54] can be used to guarantee that enclave software only interacts with the outside through a small trusted runtime.

4 HARDWARE DESIGN

Sanctum introduces a cache partitioning scheme that splits up the memory into DRAM "regions" that use disjoint last level cache (LLC) sets. While the mapping is static, the operating system allocates the cache dynamically by allocating regions to software modules. Our cache partitioning scheme requires changes to the cache set indexing computation (§ 4.1) and to the circuitry that translates addresses at each TLB miss (commonly referred to as the *page walker*, § 4.3).

Sanctum enclaves have private page tables, as dirty and accessed bits in page tables reveal the enclave's memory access patterns to the operating system (at page granularity). Per-enclave page tables require a small set of modifications to the page walker (§ 4.2).

Lastly, we trust the DMA bus master to reject DMA transfers pointing into DRAM regions allocated to enclaves, to protect against attacks where a malicious OS programs a peripheral to access enclave data. This requires changes similar to the modifications done by SGX and later revisions of TXT to the integrated memory controller on recent Intel chips (§ 4.4). These changes are documented in detail in the following sections.

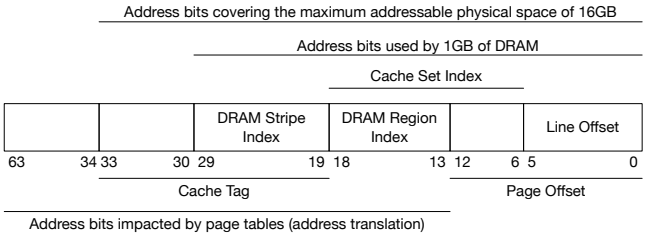

Figure 4: An annotated example of a physical address in a 64-bit computer with a typical set-associative LLC

4.1 Cache Set Indexing

Traditionally, direct-mapped and set-associative caches have used the low-order bits of a physical memory address to compute the possible locations in the cache a line can occupy. The left-most bits make up the *line offset*, denoting a byte in a cache line, preceded by the *set index*. A piece of data can be stored in any way in the set identified by the set index. For example, in a cache with 8,192 sets and 64-byte lines, bits $[5 \ldots 0]$ in a physical address make up the line offset ($64 = 2^6$) and bits $[18 \ldots 6]$ make up the set index ($8192 = 2^{13}$).

Figure 4 illustrates the relevant sections of a physical address. Address translation generally uses a tree-like data structure (page tables), to map virtual addresses to physical memory addresses. Address translation ignores the low-order bits of an address (*page offset*), and addresses differing only in the page offset bits belong to the same *page*. In the 64-bit RISC V architecture, pages are 8KB, so bits $[12 \ldots 0]$ of a virtual address are not translated. Privileged software that controls the page tables can therefore influence the placement of data in the cache, as page tables influence some of the bits that serve as the cache set index. In the example above, address translation sets bits $[18 \ldots 13]$ of a physical address, which double as cache set index bits.

Sanctum defines the notion of **DRAM regions** - subsets of the physical address space corresponding to the set of all pages that share LLC sets with a given page. In the example above, DRAM is divided into 64 regions, and the DRAM region of a physical address is determined by bits $[18 \ldots 13]$ (which we denote the **DRAM region index**). In direct map and set associative LLCs, DRAM regions are not contiguous, so we further define **DRAM stripes** to be the contiguous sections of a DRAM Region (in a typical cache, a stripe is one page long).

We are interested in DRAM regions because *addresses in a DRAM region do not collide in the LLC with ad-*

dresses from any other DRAM region. If we were to restrict the code and data of Alice's program to a single DRAM region, over which she has exclusive access, no other program may interfere with the LLC behavior of her program, rendering it immune to cache timing attacks in the LLC.

In Sanctum, we ensure that all DRAM region index bits are defined via address translation, allowing a trusted entity to confine software modules to unique DRAM regions by appropriately populating page table entries. Most importantly, DRAM regions are transparent to application software, which is presented with a contiguous virtual address space.

Unfortunately, DRAM regions are discontinuous in systems with a typical cache set indexing scheme: as shown above, DRAM stripes are one page long in the typical case. This prevents a software module from allocating a multi-page contiguous data structure in physical memory in one DRAM region, which is essential for efficient DMA transfers as used by high performance device drivers To address this shortcoming, we modify the cache index translation and use higher-order bits to specify the cache set index. This effectively increases the length of DRAM stripes: Modifying our earlier example to use, say, bits $[21 \ldots 16]$ as DRAM region results in 8-page long DRAM stripes without affecting DRAM region size. Such a change enables a device driver to allocate a 64KB contiguous buffer in physical memory, amenable to DMA data transfer. A DRAM stripe would optimally equal a DRAM region, obviating region discontinuity, but the corresponding choice of DRAM region index bits depends on the amount of DRAM in the system, which is not always known a priori.

To solve the problem above, we propose adding a shifter between the address translation unit and the cache unit: cache set index is correctly computed using higher order bits of a physical address, resulting in optimally sized DRAM stripes for a variety of DRAM configurations. Figure 5 shows how a shifter rotates the translated bits of a physical address by 3 before the address is provided to the LLC. A computer with 512MB - 4GB of DRAM must implement shift amounts from 4 to 7. Such a shifter can be implemented via a 3-position variable shifter circuit (series of 4-input MUXes), and a fixed shift by 4 (no logic). Alternatively, in systems with known DRAM configuration (embedded, SoC, etc.), the shift amount can be fixed, and implemented with no logic.

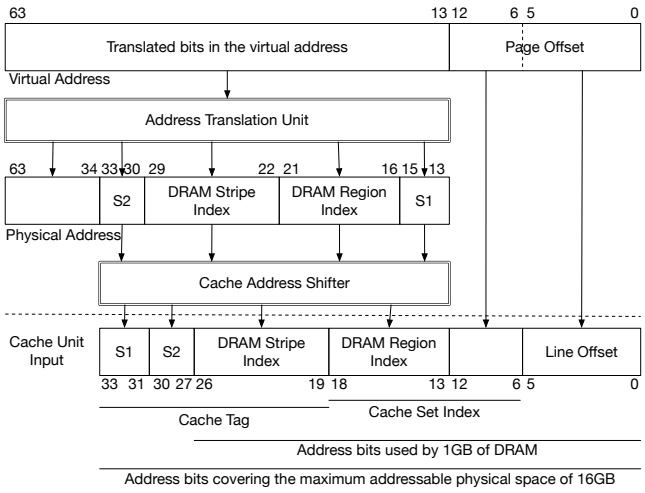

Figure 5: Example addition of a cache address shifter that rotates physical addresses to the right by 3 bits before they reach the LLC

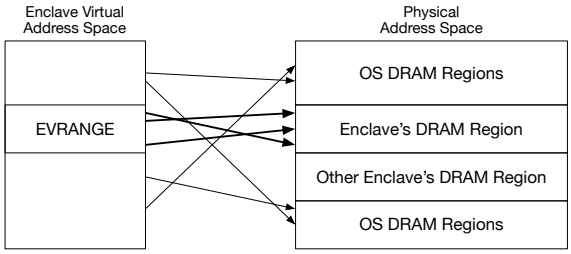

Figure 6: The virtual address space of an enclave: addresses inside EVRANGE are mapped to DRAM regions owned by the enclave, while addresses outside the range are mapped to DRAM regions owned by the OS.

4.2 Page Walker Input

Both Sanctum and SGX enclaves can access unprivileged non-enclave memory, which is used to pass data between an enclave and the application hosting it. Enclaves may not access privileged memory to preserve OS protection against rogue enclaves. As shown in Figure 6, each enclave has a contiguous region of its virtual address space mapped to enclave memory, which is protected from outside access (by a malicious OS, for example). The rest of the address space is mapped to memory that does not belong to any enclave, which we denote *OS memory*.

SGX relies on the operating system to maintain the page tables for enclave software, and uses clever techniques to ensure that the page tables match the enclave author's expectations. The page tables, however, leak information about the enclave's memory access patterns in two ways: page faults directly inform the OS about some enclave memory accesses, while the dirty and accessed bits in the page table entries indirectly reveal page-level access patterns. Page faults are necessary to allow an OS

to over-commit physical memory, presenting an abstraction of a large address space, while dirty and accessed bits enable the OS to evict pages intelligently at page faults. The information leaks above are not viably solved by simply removing the features that cause them.

Sanctum enclaves have private page tables and service their own page faults, obviating the leaks above. The operating system manages system resources at DRAM region granularity by allocating regions to enclaves. Enclaves are free to over-commit their physical memory by implementing their own page fault handlers, and get the information they need to evict pages by reading the dirty and accessed bits in their private page tables.

Per-enclave page tables are implemented by adding an enclave page table base register `eptbr`, which stores the physical address of the currently running enclave's page tables, and has similar semantics to the page table base register `ptbr`, which points to the operating system-managed page tables. The registers can only be accessed by the Sanctum security monitor, which provides an API call for the OS to set up the `ptbr`, and ensures that the `eptbr` always points to the current enclave's page tables.

In order to know when to use the `ptbr` and the `eptbr`, the circuitry handling TLB misses also uses two registers that indicate the range of virtual addresses used by the current enclave, `evbase` (enclave virtual address space base) and `evmask` (enclave virtual address space mask). When a TLB miss occurs, a circuit shown in Figure 7 selects the appropriate base register: a masked faulting address is compared against the base register, and the result determines whether to forward `eptbr` or the `ptbr` to the page walker as the page table base address.

4.3 Page Walker Memory Accesses

In performance-oriented processors, the page walker module is a hardware finite-state machine (FSM). The page walker FSM uses the page table base physical address to issue series of DRAM accesses using physical addresses. These memory accesses are cached, and are suceptible to cache timing attacks without proper countermeasures, which would reveal the enclave's memory access patterns at page granularity. Sanctum obviates this by storing enclave page tables in the enclave's DRAM regions, isolating these sensitive data structures throughout the memory hierarchy.

For complete isolation between enclaved software modules, the page walker must only fetch addresses

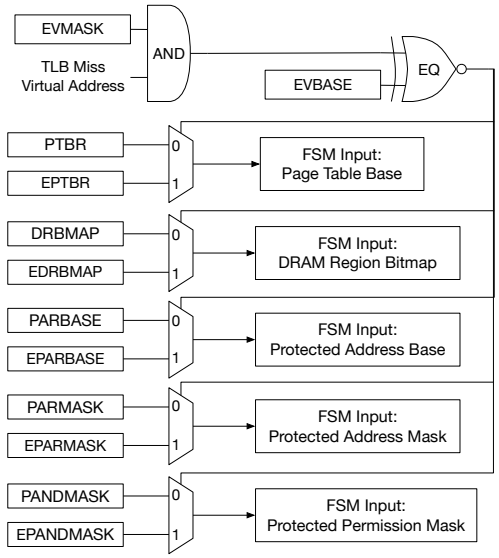

Figure 7: Hardware support for per-enclave page tables: page walker inputs.

within a given entity's (enclave or OS) DRAM regions. Failing this, a malicious OS would build page tables referencing enclave DRAM regions, and observe private accesses performed by the FSM.

Sanctum's trusted security monitor controls the page table base registers (`ptbr` and `eptbr`), so the initial fetch by the FSM is safe. We sanitize the page table entries before the page walker acts on them, preventing fetches into other modules' DRAM regions (by clearing a page table entry's valid bit, causing a page fault). The monitor populates a DRAM region bitmap (`drbmap`) register, where each set bit corresponds to an accessible DRAM region. The leaf page table entry is likewise sanitized, ensuring address translation always yields a physical address in an accessible DRAM region.

Sanctum's security monitor must maintain metadata about each enclave, and does so in the enclave's DRAM regions. For security reasons, the metadata must not be writable by the enclave. We extend the page table entry transformation described above to implement per-enclave read-only areas. A protected address range base (`parbase`) register and a protected address range mask (`parmask`) register denote this protected physical address range. We mask (bitwise AND) addresses in the page table entry and compare it with the address base, identifying protected addresses. If we encounter a leaf page table entry with a protected address, we mask its permission bits with a protected permissions mask (`parpmask`) register. If we discover a protected address in an intermediate page table entry, we clear its valid bit,

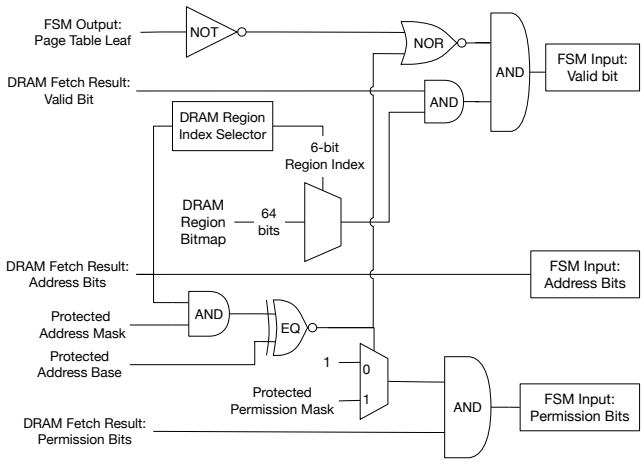

Figure 8: Hardware support for per-enclave page tables: transforming the page table entries fetched by the page walker.

forcing a page fault.

The above transformation allows the security monitor to set up a read-only range by clearing permission bits (write-enable, for example). Entry invalidation ensures no page table entries are fetched from the protected range, which prevents the page walker FSM from moodifying the protected region by setting accessed and dirty bits.

All registers mentioned above come in pairs, as we maintain separate OS and enclave page tables. The security monitor sets up a protected range in the OS page tables to isolate its own code and data structures (most importantly its private attestation key) from a malicious OS.

Figure 8 shows Sanctum's logic inserted between the page walker and the cache unit that fetches page table entries.

4.4 DMA Transfer Filtering

Like SGX, we assume DMA regions targeting enclave memory are filtered, preventing software attacks that rely on a compromised operating system that instructs a device (such as a NIC) to launch DMA transfers on enclave private memory.

Unlike SGX, we whitelist a DMA-safe DRAM region instead of using a blacklist. Specifically, Sanctum implements two registers (a base, `dmarbase` and an AND mask, `dmarmask`) in a DMA arbiter (memory controller) for the security monitor to configure the range of physical memory usable for DMA transfers. The monitor allows the OS to set the range register, and ensures it falls within the OS-owned DRAM regions. For each DMA request, a circuit illustrated in Figure 7 checks the transfer start and end addresses against the range registers (each address is ANDed with `dmarmask` and the result is compared with `dmarbase`). The transfer dropped if either end of the DMA transfer's range falls outside the (contiguous) allowed DMA range.

5 Software Design

Sanctum has two pieces of trusted software: the measurement root (§ 5.1), which is shipped via an on-chip ROM, and the security monitor (§ 5.2), which is stored alongside the computer's firmware (usually flash memory). Together, the trusted software in Sanctum fulfills the same functions as the extended microcode in the implementation of SGX.

5.1 Measurement Root

The measurement root (`mroot`) is stored in a ROM at the top of the physical address space, and covers the reset vector. Its main responsibility is to compute a cryptographic hash of the security monitor, and generate an attestation key pair and certificate based on the monitor's hash. This allows the machine owner to patch or customize the security monitor, while preserving the attestation mechanism needed to convince a third party that it is talking to a specific enclave built in a well-defined environment.

The security monitor is expected to be stored in non-volatile memory (such as a SPI flash chip) that can respond to memory I/O requests from the CPU, perhaps via a special mapping in the computer's chipset. When `mroot` starts executing, it computes a cryptographic hash over the security monitor. `mroot` then reads the processor's key derivation secret, and derives a symmetric key based on the monitor's hash. `mroot` will eventually hand down the key to the monitor.

The security monitor contains a header that includes the location of an attestation key existence flag. If the flag is not set, the measurement root generates an attestation key pair for the monitor, and produces an attestation certificate by signing the monitor's public attestation key with the processor's private key. The certificate includes the monitor's hash.

The security monitor is expected to encrypt its private attestation key with the symmetric key produced earlier, and store the encrypted key in its flash memory. When writing the key, the monitor is expected to set the asymmetric key existence flag, instructing future boot sequences not to re-generate a key. The public attestation key and certificate can be stored unencrypted in any untrusted memory.

Figure 9: DRAM region management state transition diagram. The transitions are API calls issued by enclaves or by the operating system.

Before handing control to the monitor, `mroot` sets a lock that blocks any software from reading the processor's symmetric key derivation seed and private key until a reset occurs.

5.2 Security Monitor

The security monitor receives control after `mroot` finishes setting up the attestation measurement chain. The monitor provides APIs to the operating system and enclaves for **DRAM region management** and **enclave management**. The monitor also guards sensitive registers, such as the page table base register (`ptbr`) and the allowed DMA range (`dmarbase` and `dmarmask`). The monitor allows the OS to set these registers via APIs only, and sanitizes the values against current DRAM region allocation.

Figure 9 shows the DRAM region management state transition diagram. After the system boots up, all DRAM regions are allocated to the operating system, which can assign free DRAM regions to itself or to enclaves. In order for a DRAM region to become free, it must be first blocked by its owner, which can be the OS or an enclave. While a DRAM region is blocked, any address translations mapping to it result in page faults, so no TLB entries can be created for that region, and no entity may access the region via stale TLB entries. Before the OS issues a free DRAM region API call, it must flush all relevant TLB entries, as this is verified by the security monitor.

Monitor checks rely on a global *block clock*. When a region is blocked, the block clock is incremented and the current block clock value is stored in the metadata associated with the DRAM region (shown in Table 1). When a core's TLBs are flushed, the core's flush time (maintained for each core) is updated to the current block clock value. When the OS signals to transition a DRAM region from BLOCKED to FREE, the monitor checks that all the relevant cores' flush times are no less than the block clock value stored in the region's metadata. The set of relevant cores is defined by the DRAM region's owner when it is blocked: OS-owned regions require all TLBs to be flushed, whereas enclave-owned regions mandate flushes only on cores with that enclave's threads.

Field	Description
lock	acquired for all operations on the DRAM region
owner	the ID of the enclave owning the DRAM region; the OS has its own enclave ID; BLOCKED and FREE states are represented as special invalid enclave IDs
previous_owner	the owner at the time when the region was blocked
pinned_pages	regions with a non-zero pinned pages counter cannot be blocked
blocked_at	the block clock value when this enclave was blocked

Table 1: Per-DRAM region metadata

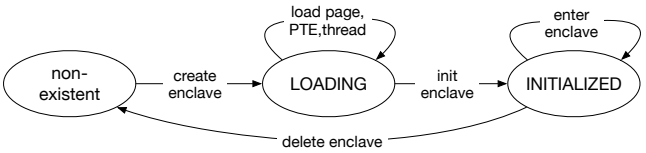

Figure 10: The enclave management state diagram. The transitions are API calls issued by the operating system.

Figure 10 shows the enclave management state diagram. The OS creates an enclave by issuing a monitor API call to allocate a free DRAM region to the enclave and initialize the enclave metadata fields (shown in Table 2) in that DRAM region. When an enclave is created, it enters the LOADING state, where the OS sets up the enclave's initial state via API calls to create hardware threads and page table entries, and copy code and data into the enclave. Every operation performed on an enclave in the LOADING state update the enclave's measurement hash. The OS then issues an API call to transition the enclave to the INITIALIZED state, which finalizes its measurement hash. The application hosting the enclave is now free to run enclave threads.

The security monitor uses Sanctum's MMU extensions to ensure that enclaves cannot modify their own metadata area. Moreover, an enclave's metadata cannot be accessed by the OS or any other enclave, as it is stored in the enclave's DRAM region. This allows us to use the metadata area to store public information with integrity requirements, such as the enclave's measurement hash.

While an OS loads an enclave, it is free to map the enclave's pages, but the monitor maintains its page tables ensuring all entries point to non-overlapping pages in DRAM regions owned by that enclave. Once an enclave

Field	Description
max_threads	number of slots in the enclave's thread slot table
ev_base ev_mask	the enclave's virtual address range
epar_mask	indicates the size of the enclave's metadata to Sanctum's MMU extensions
is_initialized	1 if the enclave is INITIALIZED
is_debug	1 if debugging is enabled
metadata_top	the end of the enclave's metadata
load_eptbr	physical address of the base of the page table used during enclave loading
last_load_addr	physical address of the last enclave page touched by the loading process
running_threads	number of cores executing this enclave's threads
hash	the enclave's measurement hash
hash_block	working area for the enclave measurement process
DRAM region bitmap	one bit per DRAM region; the enclave's regions have their bits set to 1
Thread slots	array of thread slots

Table 2: Enclave metadata fields

Field	Description
thread_info	physical address of the thread's state area; 0 if the slot is free
lock	acquired when changing thread_info or the thread starts executing

Table 3: Thread slot fields

Field	Description
entry_pc	initial program counter
entry_stack	initial stack pointer
fault_pc	fault handler program counter
fault_stack	fault handler stack pointer
eptbr	page table base register value
exit_state	the user register values at the time of the enter enclave call that started the thread
aex_state	the user register values at asynchronous enclave exit (AEX) time
can_resume	1 when this thread can resume from an AEX

Table 4: Thread state area fields

is initialized, it can inspect its own page tables and abort if the OS created undesirable mappings. Simple enclaves do not depend on specific layouts, and we expect that complex enclaves will communicate their desired layouts to the OS via enclave metadata not addressed in this work.

Our monitor makes sure that page tables do not overlap by storing the physical address of the last mapped page in an enclave metadata field. To simplify the monitor, a new mapping is allowed if its physical address is greater than the address of the last mapped page, forcing the OS to map an enclave's pages in monotonically increasing order.

Sanctum supports multi-threaded enclaves: each enclave thread is represented by a thread slot (Table 3), which is stored in the enclave metadata, and a thread state area (Table 4), which is stored in the enclave's DRAM regions.

Figure 11 shows the thread slot management state diagram. At enclave creation, the OS specifies how many thread slots to allocate in the enclave's metadata. Initially, all thread slots are free, but during enclave loading, the OS can initialize a free slot via a *load thread* API call, which designates the thread's state area and stores it in the thread slot. Running enclaves may initialize additional slots using a similar API call.

The application hosting an enclave starts executing

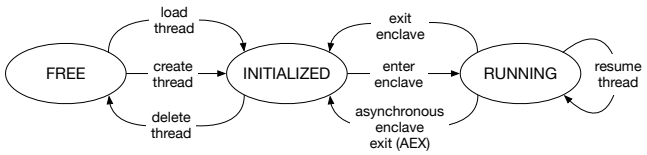

Figure 11: The thread slot management state diagram. The transitions are API calls issued by the enclave owning the thread slots, or by the operating system.

enclave code by issuing an *enclave enter* API call, which must specify an initialized thread slot. The security monitor saves the application's register state in the thread state area, and loads the thread program counter and stack pointer. The enclave's code can return control to the hosting application voluntarily, by issuing an *enclave exit* API call, which restores the application's state from the thread state and sets the API call's return value to `ok`.

If an interrupt occurs while the enclave code is executing, the security monitor's exception handler performs an *asnychronous enclave exit* (AEX), which saves enclave register state in the thread state area, restores the application's registers from the area, sets the API call's return value to `async_exit`, and invokes the standard interrupt handling code. After the OS handles the interrupt, the enclave's host application resumes execution, and re-executes the *enter enclave* API call. The enclave's thread initialization code examines the saved thread state, and seeing that the thread has undergone an AEX, issues a *resume thread* API call. The security monitor restores the enclave's registers from the thread state area, and clears the AEX flag.

Sanctum's security monitor stores a single instance of enclave registers in the thread state area. We avoid SGX's state stack by observing that when an AEX occurs in a thread that hasn't resumed from the previous AEX, the thread was in the process of resuming from a previous AEX, and Sanctum can safely discard the enclave's current register state. The enclave's host application will retry the enter enclave API call, and the enclave thread initialization code will simply restart the process of resuming from the original AEX.

The security monitor is highly concurrent, with fine-grained locking. API calls targeting two different enclaves may be executed in parallel on different cores. Each DRAM region has a lock guarding that region's metadata. Each enclave thread slot has a lock guarding the slot, which is also acquired when the thread starts runs via an *enter enclave* API call, and is released when the thread is stopped via an *enclave exit* or AEX.

We avoid discussing deadlocks by delegating the coordination burden to the OS. Each API call in the security monitor attempts to acquire all the locks it needs via atomic test-and-set operations, and errors with a `concurrent_call` code if any lock is unavailable.

We implemented the Sanctum security monitor in less than 5kloc of C++, which include a subset of the standard library and the cryptography used by enclave attestation.

5.3 Enclave Eviction

General-purpose software can be executed inside an enclave without source code changes, provided that it is linked against a runtime (e.g., *libc*) that was modified to work with Sanctum.

The current Sanctum design allows the operating system to over-commit physical memory allocated to enclaves, by paging out to disk DRAM regions from some enclaves. Sanctum does not give the OS visibility into enclave memory accesses, in order to prevent private information leaks, so the OS must decide the enclave whose DRAM regions will be evicted based on other activity, such as network I/O, or based on a business policy, such as Amazon EC2's spot instances.

Once a victim enclave has been decided, the OS asks the enclave to block a DRAM region, which gives the enclave an opportunity to rearrange data in its RAM regions. DRAM region management can be implemented in the enclave's runtime, and is completely transparent to enclave software writers.

The security monitor does not allow the OS to forcibly reclaim a single DRAM region from an enclave, because that would reveal memory access patterns. Instead, the OS can delete an enclave, after stopping its threads, and reclaim all its DRAM regions. Thus, a small or short-running enclave can include a runtime that refuses DRAM region management requests from the OS, and relies on the OS to delete it under memory pressure and re-start it at a later time.

In order to avoid wasted work, however, large long-running enclaves can use a runtime that implements demand paging. When an enclave uses up most of its physical memory, the memory manager in the runtime finds least recently used pages by examining the enclave's page tables, and adds eviction requests to a global enclave queue. The runtime also implements a page fault handler that adds a page fill request to the global queue, and stalls until the request is completed.

When asked to relinquish a DRAM region, the enclave's runtime follows the same process as above to free up physical pages, frees up all the pages in a DRAM

region by moving the data to free pages outside the region, and finally blocks the DRAM region so the OS can reclaim it.

To be secure, the runtime processes queued page I/O requests in a dedicated enclave thread that obfuscates page fault timing and pattern from the operating system by performing periodic I/O system calls using oblivious RAM techniques, as in the Ascend processor [35], but at page rather than cache line granularity.

Enclaves that perform other data-dependent communication, such as targeted I/O into a large database file, must also use the periodic oblivious I/O techniques described above to obfuscate their access patterns from the operating system. These techniques are not dependent on specific application business logic, and can be implemented in generic library software, such as database access drivers.

Last, the design described above requires that each enclave occupies at least one DRAM region for its entire lifetime, which contains the memory management code described above and its data structures. Evicting an enclave's entire memory while keeping it alive requires an entirely different approach that will be described in future work.

Briefly, the OS can ask the security monitor to *freeze* an enclave, which encrypts all the enclave's DRAM regions in place, and creates a leaf node in a hash tree. When the monitor *thaws* a frozen enclave, it uses the previously created hash tree leaf to ensure freshness, decrypts the data in the enclave's DRAM regions, and *relocates* the enclave, updating its page tables to account for the changes in DRAM region ownership after thawing. The hash tree is managed by the operating system, using a similar approach to SGX's version array page eviction.

6 Security Analysis

Sanctum protects each enclave in a system from the (potentially compromised) operating system, and from the other potentially malicious enclaves. The security monitor (§ 5.2) keeps track of the operating system's assignment of DRAM regions to enclaves, and enforces the invariant that *a DRAM region may only be assigned to exactly one enclave or to the operating system.*

The region blocking mechanism guarantees that when a DRAM region is assigned to an enclave or the OS, no stale TLB mappings associated with the DRAM region exist. The monitor uses the MMU extensions described in § 4.2 and § 4.3 to ensure that once a DRAM region is assigned, no software other than the region's owner may create TLB entries pointing inside the DRAM region. Together, these mechanisms guarantee that the DRAM regions allocated to an enclave cannot be accessed by the operating system or by another enclave.

The LLC modifications in § 4.1 ensure that an enclave confined to a set of DRAM regions is not vulnerable to cache timing attacks from software that cannot access the enclave's DRAM regions. The security monitor enforces the exclusive ownership of each DRAM region.

Sanctum's security monitor lives in a DRAM region assigned to the OS, so no enclave can set up its page tables to point to the monitor's physical memory. The monitor uses the MMU extensions in § 4.2 and § 4.3 to prevent the OS from creating TLB mappings pointing into the monitor's physical memory, meaning the security monitor's integrity cannot be compromised by a malicious enclave or OS.

Sanctum also protects the operating system from (potentially malicious) enclaves: the security monitor executes enclave code with the same privilege as application code, so the barriers erected by the OS kernel to protect itself against malicious applications also prevent malicious enclaves from compromising the OS.

Each enclave has full control over its own page tables, but the security monitor configures the MMU extensions in § 4.2 and § 4.3 to confine an enclave's page tables to the DRAM regions assigned to it by the OS. This means that enclaves cannot compromise the OS memory, and that enclaves may only use the DRAM regions given to them by the OS.

The security monitor preempts an enclave thread (via AEX) when its CPU core receives an interrupt, allowing the OS to preempt a rogue enclave's threads via interprocessor interrupts (IPI), and then destroy the enclave.

7 Performance Evaluation

While we propose a high-level set of hardware and software to implement Sanctum, we focus on the concrete example of a 4-core RISC V system generated by Rocket Chip [13]. As Sanctum isolates concurrent workloads from each other, we can examine its overhead by running a single application on one core, and not worry about having to account for workload interactions.

7.1 Experiment Design

We use a Rocket-Chip generator modified to model the hardware modifications described in § 4. We generate a 4-core 64-bit RISC V CPU with per-core 16KB 4-way

set associative instruction and data L1 caches. We use a cycle-accurate simulator for this machine to produce an LLC access trace. We post-process the trace with a cache emulator for a shared 4MB L2 cache (LLC), with and without Sanctum's DRAM region isolation, and we compute the program completion time, in cycles, for each benchmark. We obtain accurate results because Rocket cores have in-order single pipelines, and cannot make any progress on a TLB or cache miss.

Our cache size choices were inspired by Intel's Sandy Bridge [55] desktop models, which have 8 logical CPUs on a 4-core hyper-threaded system with 32KB 8-way L1s, and an 8MB LLC. We do not model Intel's 256KB per-core L2, because it is not supported by Rocket's implementation. We note, however, that a private L2 would greatly reduce each core's LLC requests, which is Sanctum's main source of overhead.

We simulate a machine with 4GB of memory that is divided into 64 DRAM regions by Sanctum's cache address indexing scheme. In our model, an LLC access adds a 12-cycle latency, and a DRAM access costs 100 cycles.

We do not model the behavior of the DRAM memory controller, nor limited DRAM bandwidth. We also omit an evaluation of the on-chip network and cache coherence overhead, as we do not make any changes that impact any of these subsystems.

Using the hardware model above, we benchmark the subset of SPECINT 2006 [56] that we could compile using the RISC V toolchain without additional infrastructure, specifically `bzip2`, `gcc`, `lbm`, `mcf`, `milc`, `sjeng`, and `998.specrand`. This is a mix of memory-bound and compute-bound long-running workloads with a range access locality.

We avoided the overhead of having to simulate a complete Linux kernel, and instead used the RISC V proto kernel [57] that provides the services used by our benchmarks. We scheduled each benchmark on Core 0, and ran it to completion, while the other cores were idling.

7.2 Cost of Added Hardware

Sanctum's hardware changes add a small amount of gates to the Rocket chip, which increases its area and power consumption. Like SGX, we did not touch the core's critical execution path. Our addition to the page walker, analyzed in the next section, may increase the latency of TLB misses, but is guaranteed not to increase the Rocket core's clock cycle, which was competitive with an ARM Cortex-A5 [13].

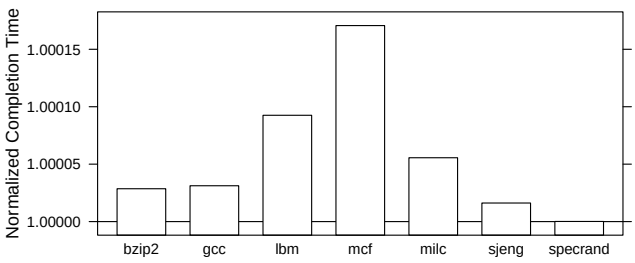

Figure 12: Sanctum's modified page walk has minimal effect on benchmark performance

Based on the gate-level illustrations in Figures 7 and 8, we estimate Sanctum's changes to the Rocket hardware to require 1500 (+0.78%) gates and 700 (+1.9%) flip-flops per core, consisting of 50 gates for the cache index calculation, 1000 gates and 700 flip-flops for the extra address page walker configuration, and 400 gates for the page table entry transformations. DMA filtering requires 600 gates (+0.8%) and 128 flip-flops (1.8%) in the uncore. We do not make any changes to the LLC, which generally accounts for 50% of a chip.

7.3 Page Walker Latency Changes

Sanctum's page table entry transformation logic is described in § 4.3. We expect that the logic can be combined with the page walker FSM logic, without pushing the logic latency over the cycle budget.

In the worst case, the transformation logic would have to become its own pipeline stage on the path between the L1 data cache and the page walker. The transformation logic is guaranteed to fit in 1 cycle on its own, as it is significantly simpler than the ALU in the core's execute stage. In this case, every memory fetch issued by the page walker would experience a 1-cycle latency, which adds 3 cycles of latency to each TLB miss.

Figure 12 shows the completion time of selected benchmarks, normalized to the completion time without the extra TLB miss latency. The overheads are well below 0.01%, which is insignificant compared to the overheads of cache isolation.

7.4 Security Monitor Overhead

Invoking Sanctum's security monitor to load code in an enclave adds a one-time setup cost to each isolated process, when compared to insecurely running the computation without Sanctum's isolation support. This overhead does not scale with the duration of the computation, so we consider it to be negligible for long-running workloads.

Entering and exiting enclaves is more expensive than

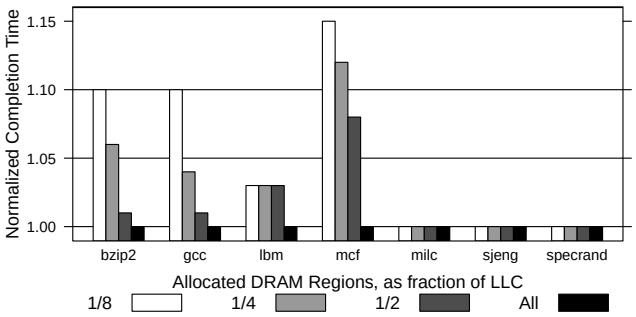

Figure 13: The impact of the number of DRAM regions allocated to an enclave on the benchmark's completion time

hardware context switches, because the security monitor must flush TLBs and L1 caches. However, a sensible OS is expected to minimize the number of context switches by allocating some cores to an enclave and allowing them to execute to completion. Therefore, we also consider this overhead to be negligible for long-running computations.

7.5 Overhead of DRAM Region Isolation

The crux of Sanctum's strong isolation is that each DRAM region is cached in separate LLC sets. Therefore, when the OS assigns DRAM regions to an enclave, it also gives it a share of the LLC, which impacts the enclave's performance. At the same time, Sanctum does not partition the per-core caches, so a thread can utilize its core's entire L1 caches and TLBs.

Figure 13 shows the completion times of the SPECINT workloads, where each number is normalized to the completion time of the same benchmark running on an ideal insecure OS that allocates the entire LLC to the benchmark.

Sanctum excels at isolating compute-bound workloads operating on sensitive data. Thus, SPECINT's large, multi-phase workloads heavily exercise the entire memory hierarchy, and therefore paint an accurate picture of a worst case for our system. `mcf`, in particular, is very sensitive to the available LLC size, so it incurs noticeable overheads when being confined to a small subset of the LLC.

We consider `mcf`'s 15% decrease in performance when limited to 1/8th of the LLC to be a very pessimistic view of our system's performance, as it explores the case where the enclave receives 1/4th of the CPU power (a core), but 1/8th of the LLC. For a reasonable allocation of 1/4 of DRAM regions (in a 4-core system), DRAM regions add a 3-6% overhead to most memory-bound benchmarks (with the exception of `mcf`), and do not impact compute-bound workloads.

In the LLC, our region-aware cache index translation forces consecutive physical pages in DRAM to map to the same cache sets within a DRAM region, creating interference. We expect the OS memory management implementation to be aware DRAM regions, and map data structures to pages spanning all available DRAM regions.

The locality of DRAM accesses is also affected: an enclaved process has exclusive access to its DRAM region(s), each a contiguous range of physical addresses. DRAM regions therefore cluster process accesses to physical memory, decreasing the efficiency of bank-level interleaving in a system with multiple DRAM channels. Row or cache line-level interleaving (employed by some Intel processors [55]) of DRAM channels better parallelizes accesses within a DRAM region, but introduces a trade-off in the efficiency of individual DRAM channels. Considering the low miss rate in a modern cache hierarchy, and multiple concurrent threads, we expect this overhead is small compared to the cost of cache partitioning. We leave a thorough evaluation of DRAM overhead in a multi-channel system for future work.

8 Conclusion

We have shown through the design of Sanctum that strong provable isolation of concurrent software modules can be achieved with low overhead. The average overhead observed across all benchmarks with a reasonable allocation of DRAM regions was 3.5%, with memory-heavy benchmarks averaging 6.25%. This approach provides strong security guarantees against an insidious threat model including cache timing and memory access pattern attacks. With this work, we hope to enable a shift in discourse in the secure hardware architecture approaches away from plugging specific security holes to a principled approach to eliminating attack surfaces.

References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, ACM, 2009.

[2] S. Anthony, "Who actually develops linux? the answer might surprise you." `http://www.extremetech.com/computing/175919-`

who-actually-develops-linux-the-answer-might-surprise-you, 2014. [Online; accessed 27-April-2015].

[3] "Xen project software overview." http://wiki.xen.org/wiki/Xen_Project_Software_Overview, 2015. [Online; accessed 27-April-2015].

[4] D. Jang, Z. Tatlock, and S. Lerner, "Establishing browser security guarantees through formal shim verification," in *Proceedings of the 21st USENIX conference on Security symposium*, pp. 8–8, USENIX Association, 2012.

[5] "Linux kernel: Cve security vulnerabilities, versions and detailed reports." http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, 2014. [Online; accessed 27-April-2015].

[6] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, p. 5, ACM, 2011.

[7] "Xen: Cve security vulnerabilities, versions and detailed reports." http://www.cvedetails.com/product/23463/XEN-XEN.html?vendor_id=6276, 2014. [Online; accessed 27-April-2015].

[8] R. Boivie and P. Williams, "Secureblue++: Cpu support for secure executables," tech. rep., IBM Research Division, Apr 2013.

[9] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," *HASP*, vol. 13, p. 10, 2013.

[10] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, vol. 13, 2013.

[11] Intel Corporation, *Software Guard Extensions Programming Reference*, 2013. Reference no. 329298-002US.

[12] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 199–212, ACM, 2009.

[13] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic, "A 45nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators," in *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*, pp. 199–202, IEEE, 2014.

[14] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," Tech. Rep. UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[15] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual volume ii: Privileged architecture version 1.7," Tech. Rep. UCB/EECS-2015-49, EECS Department, University of California, Berkeley, May 2015.

[16] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, "Fine grain cross-vm attacks on xen and vmware are possible!." Cryptology ePrint Archive, Report 2014/248, 2014. http://eprint.iacr.org/.

[17] J. Rutkowska, "Thoughts on intel's upcoming software guard extensions (part 2)," *Invisible Things Lab*, 2013.

[18] S. Davenport, "Sgx: the good, the bad and the downright ugly," *Virus Bulletin*, 2014.

[19] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st annual International Symposium on Computer Architecuture*, pp. 361–372, IEEE Press, 2014.

[20] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges." http://googleprojectzero.blogspot.com/2015/03/exploiting-

dram-rowhammer-bug-to-gain.html, 3 2015. [Online; accessed 9-March-2015].

[21] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th annual international conference on Supercomputing*, pp. 160–171, ACM, 2003.

[22] W. Diffie and M. E. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, 1976.

[23] S. Banescu, "Cache timing attacks," 2011. [Online; accessed 26-January-2014].

[24] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *Cryptographic Hardware and Embedded Systems-CHES 2006*, pp. 201–215, Springer, 2006.

[25] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[26] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology – CRYPTO*, pp. 104–113, Springer, 1996.

[27] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Computer Security–ESORICS 2011*, pp. 355–371, Springer, 2011.

[28] Y. Yarom and K. E. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack.," *IACR Cryptology ePrint Archive*, vol. 2013, p. 448, 2013.

[29] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 143–158, IEEE, 2015.

[30] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox – practical cache attacks in javascript," *arXiv preprint arXiv:1502.07373*, 2015.

[31] "Nist's policy on hash functions." http://csrc.nist.gov/groups/ST/hash/policy.html, 2014. [Online; accessed 4-May-2015].

[32] "Gradually sunsetting sha-1." http://googleonlinesecurity.blogspot.com/2014/09/gradually-sunsetting-sha-1.html, 2014. [Online; accessed 4-May-2015].

[33] "Sha1 deprecation policy." http://blogs.technet.com/b/pki/archive/2013/11/12/sha1-deprecation-policy.aspx, 2013. [Online; accessed 4-May-2015].

[34] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[35] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pp. 3–8, ACM, 2012.

[36] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pp. 182–194, ACM, 1987.

[37] T. C. Group, "Tpm main specification." http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2003.

[38] D. Grawrock, *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.

[39] R. Wojtczuk and J. Rutkowska, "Attacking intel txt via sinit code execution hijacking," *Invisible Things Lab*, 2011.

[40] R. Wojtczuk and J. Rutkowska, "Attacking intel trusted execution technology," *Black Hat DC*, 2009.

[41] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Another way to circumvent intel® trusted execution technology," *Invisible Things Lab*, 2009.

[42] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," *CanSecWest/core06*, 2006.

[43] J. Rutkowska and R. Wojtczuk, "Preventing and detecting xen hypervisor subversions," *Blackhat Briefings USA*, 2008.

[44] R. Wojtczuk and J. Rutkowska, "Attacking smm memory via intel cpu cache poisoning," *Invisible Things Lab*, 2009.

[45] F. Wecherowski, "A real smm rootkit: Reversing and hooking bios smi handlers," *Phrack Magazine*, vol. 13, no. 66, 2009.

[46] S. Embleton, S. Sparks, and C. C. Zou, "Smm rootkit: a new breed of os independent malware," *Security and Communication Networks*, 2010.

[47] E. Brickell and J. Li, "Enhanced privacy id from bilinear pairing," *IACR Cryptology ePrint Archive*, vol. 2009, p. 95, 2009.

[48] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pp. 494–505, 2007.

[49] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 2nd ACM workshop on Computer security architectures*, pp. 25–34, ACM, 2008.

[50] F. Liu and R. B. Lee, "Random fill cache architecture," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 203–215, IEEE, 2014.

[51] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 35, 2012.

[52] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pp. 187–198, IEEE, 2010.

[53] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 57–68, ACM, 2011.

[54] 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 *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 79–93, IEEE, 2009.

[55] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Sep 2014. Reference no. 248966-030.

[56] "Spec cpu 2006," tech. rep., Standard Performance Evaluation Corporation, May 2015.

[57] A. Waterman, Y. Lee, and e. a. Celio, Christopher, "Risc-v proxy kernel and boot loader," tech. rep., EECS Department, University of California, Berkeley, May 2015.