Sanctum: Minimal Hardware Extensions for Strong Software Isolation

Victor Costan, Ilia Lebedev, and Srinivas Devadas
victor@costan.us, ilebedev@mit.edu, devadas@mit.edu
MIT CSAIL

ABSTRACT

Sanctum offers the same promise as SGX, namely strong provable isolation of software modules running concurrently and sharing resources, but protects against an important class of additional software attacks that infer private information from a program's memory access patterns. We follow a principled approach to eliminating entire attack surfaces through isolation, rather than plugging attack-specific privacy leaks.

Sanctum demonstrates that strong software isolation is achievable with a surprisingly small set of minimally invasive hardware changes, and a very reasonable overhead. Sanctum does not change any major CPU building block. Instead, we add hardware at the interfaces between building blocks, without impacting cycle time.

Our prototype shows a 2% area increase in a Rocket RISC-V core. Over a set of benchmarks, Sanctum's worst observed overhead for isolated execution is 15.1% over an idealized insecure baseline, and 2.7% average overhead over a representative insecure baseline.

1 INTRODUCTION

Today's systems rely on an operating system kernel, or a hypervisor (such as Linux or Xen, respectively) for software isolation. However **each** of the last three years (2012-2014) witnessed over 100 new security vulnerabilities in Linux [1, 11], and over 40 in Xen [2].

One may hope that formal verification methods can produce a secure kernel or hypervisor. Unfortunately, these codebases are far outside our verification capabilities: Linux and Xen have over *17 million*[6] and 150,000[4] lines of code, respectively. In stark contrast, the seL4 formal verification effort [25] spent *20 man-years* to cover 9,000 lines of code.

Between Linux and Xen's history of vulnerabilities and dire prospects for formal verification, a prudent system designer cannot include either in a TCB (trusted computing base), and must look elsewhere for a software isolation mechanism.

Fortunately, Intel's Software Guard Extensions (SGX) [5, 32] has brought attention to the alternative of providing software isolation primitives in the CPU's hardware. This avenue is appealing because the CPU is an unavoidable TCB component, and processor manufacturers have strong economic incentives to build correct hardware.

However, the myriad of security vulnerabilities [15, 17, 35, 43–47] in SGX's predecessor (Intel TXT [21]) show that securing complex hardware is impossible, even in the presence of strong economic incentives. Furthermore, SGX does not protect against **software attacks** that learn information from a module's memory access pattern [22]. These include cache timing attacks, which have a long history, as well as recently introduced page fault-based attacks [48].

This paper's main contribution is a software isolation scheme that defends against the attacks mentioned above. Sanctum is a co-design that combines **minimal** and **minimally invasive** hardware modifications with a trusted software **security monitor** that is amenable to formal verification. We achieve minimality by reusing and lightly modifying existing, well-understood mechanisms. For example, our per-enclave page tables implementation uses the core's existing page walking circuit, and requires very little extra logic.

Sanctum proves that strong software isolation can be achieved without modifying any major CPU building block. We only add hardware to the interfaces between blocks, and do not modify any block's input or output. Our use of conventional building blocks translates into less effort needed to validate a Sanctum implementation.

We demonstrate that memory access pattern attacks can be foiled without incurring unreasonable overheads. Our hardware changes are small enough to present the added circuits, in their entirety, in Figures 8 and 9. Sanctum cores have the same clock speed as their insecure counterparts, as we do not modify the CPU core critical execution path. Using a straightforward cache partitioning scheme with Sanctum adds a 2.7% execution time overhead, which is orders of magnitude lower than the overheads of the ORAM schemes [20, 39] that are usually employed to conceal memory access patterns.

We also show that most of the software isolation mechanism can be provided by small, trusted software. Most of our security monitor is written in portable C++ which, once formally verified, can be used across different CPU implementations. Furthermore, even the non-portable assembly code can be reused across different implementations of the same architecture. In comparison, SGX's microcode is CPU model-specific, so each microarchitectural revision would require a separate formal verification effort.

2 Related Work

Sanctum's main improvement over SGX is preventing software attacks that analyze an isolated container's memory access patterns and infer private information from it. We are particularly concerned with cache timing attacks [7], because they can be mounted by unprivileged software on the same computer as the victim.

Cache timing attacks are known to retrieve cryptographic keys used by AES [8], RSA [10], Diffie-Hellman [26], and elliptic-curve cryptography [9]. Early attacks required access to the victim's CPU core, but more sophisticated recent attacks [31, 49] target the last-level cache (LLC), which is shared by all cores on the same chip package. Recently, [33] demonstrated a cache timing attack that uses JavaScript code in a page visited by a web browser.

Cache timing attacks observe the victim's memory access patterns at cache line granularity. However, recent work shows that useful information can be gleaned even from the page-level memory access pattern obtained by a malicious OS that simply logs the addresses seen by its page fault handler [48].

XOM [29] 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 is not to be trusted. Aegis [40] relies on a trusted security kernel, handles untrusted memory, 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 (§ 3). Resilience against hardware attacks can by obtained by augmenting a Sanctum processor with the countermeasures described in other secure architectures. Aegis protects a container's data when the DRAM is untrusted, and Ascend [19] uses Oblivious RAM [20] to protect a container's memory access patterns against adversaries that can observe the addresses on the memory bus.

Intel's Trusted Execution Technology (TXT) [21] is widely deployed in today's mainstream computers, due to its approach of trying to add security to an existing CPU. After falling victim to attacks [44, 47] where a malicious OS directed a network card to access data in the protected VM, a TXT revision introduced DRAM controller modifications that selectively block DMA transfers, which are also used by Sanctum.

Intel's SGX [5, 32] adapted the ideas in Aegis and XOM to multi-core processors with a shared, coherent last-level cache. Sanctum draws heavy inspiration from SGX's approach to memory access control, which does not modify the core's critical execution path. We reverse-engineered and adapted SGX's method for verifying an OS-conducted TLB shoot-down. We also adapted SGX's clever scheme for having an authenticated tree whose structure is managed by an untrusted OS, and repurposed it for enclave eviction. At the same time, SGX has many security issues that are solved by Sanctum, which are stated in this paper's introduction.

Iso-X [18] attempts to offer the SGX security guarantees, without the limitation that enclaves may only be allocated in a DRAM area that is carved off exclusively for SGX use, at boot time. Iso-X uses per-enclave page tables, like Sanctum, but its enclave page tables require a dedicated page walker. Iso-X's hardware changes add overhead to the core's cycle time, and do not protect against cache timing attacks.

SecureME [12] also proposes a co-design of hardware modifications and a trusted hypervisor for ensuring software isolation, but adapts the on-chip mechanisms generally used to prevent physical attacks, in order to protect applications from an untrusted OS. Just like SGX, SecureME is vulnerable to memory access pattern attacks.

The research community has brought forward various defenses against cache timing attacks. PLcache [27, 41] and the Random Fill Cache Architecture (RFill, [30]) 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. When used to isolate entire enclaves in the LLC, RFill performs at least 37%-66% worse than Sanctum.

RPcache [27, 41] trusts the OS to assign different hardware process IDs to mutually mistrusting entities, and its mechanism does not directly scale to large LLCs. The non-monopolizable cache [14] uses a well-principled partitioning scheme, but does not completely stop leakage, and relies on the OS to assign hardware process IDs.

Sanctum uses a very simple cache partitioning scheme with reasonable overheads. It is likely that sophisticated schemes like ZCache [36] and Vantage [37] can be combined with Sanctum's framework to yield better performance.

The seL4 [25] formal verification effort sets an upper bound of 9 kloc for trusted system software. Sanctum's measurement root and security monitor are smaller (< 5 kloc), so they are amenable to formal verification.

3 Threat Model

Like SGX, Sanctum isolates the software inside an **enclave** from any other software on the system, including privileged system software. Programmers are expected to move the sensitive code in their applications into enclaves. In general, an enclave receives encrypted sensitive information from outside, decrypts the information and performs some computation on it, and then returns encrypted results to the outside world.

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 any operating system and hypervisor present on the computer executing the enclave, and can launch rogue enclaves. The attacker knows the target computer's architecture and micro-architecture. The attacker can analyze passively collected data, such as page fault addresses, as well as mount active attacks, such as direct memory probing, memory probing via DMA transfers, and cache timing attacks.

Sanctum also defeats attackers who aim to compromise an operating system or hypervisor by running malicious applications and enclaves. This addresses concerns that enclaves provide new attack vectors for malware [13, 34]. We assume that the benefits of meaningful software isolation outweigh the downside of giving malware authors a new avenue for frustrating malware detection and reverse engineering [16].

Lastly, Sanctum protects against a malicious computer owner who attempts to lie about the security monitor running on the computer. Specifically, the attacker aims to obtain an attestation stating that the computer is running an uncompromised security monitor, whereas a different supervisor had been loaded in the boot process. The uncompromised security monitor must not have any known vulnerability that causes it to disclose its cryptographic keys. The attacker is assumed to know the target computer's architecture and micro-architecture, and is allowed to run any combination of malicious security monitor, hypervisor, operating system, applications and enclaves.

We do not prevent timing attacks that exploit bottlenecks in the cache coherence directory bandwidth or in the DRAM bandwidth, deferring these to future work.

Sanctum does not protect against denial-of-service (DoS) attacks carried out by compromised system software, as a malicious OS may deny service by refusing to allocate any resources to an enclave. We *do* protect against malicious enclaves attempting to DoS an uncompromised OS.

We assume correct underlying hardware, so we do not protect against software attacks that exploit hardware bugs, such as rowhammer [24, 38] and other fault-injection attacks.

Sanctum's isolation mechanisms exclusively target software attacks. § 2 mentions related work that can harden a Sanctum system against some physical attacks. Furthermore, we consider software attacks that rely on sensor data to be physical attacks. For example, we do not address information leakage due to power variations, because software would require a temperature or current sensor to carry out such an attack.

4 Programming Model Overview

By design, Sanctum's programming model deviates from SGX as little as possible, while providing stronger security guarantees. We expect that application authors will link against a Sanctum-aware runtime that abstracts away most aspects of Sanctum's programming model. For example, C programs would use a modified implementation of the `libc` standard library. Due to space constraints, we describe the programming model assuming that the reader is familiar with SGX.

The software stack on a Sanctum machine, shown in Figure 1, is very similar to the stack on a SGX system with one notable exception: SGX's microcode is replaced by a trusted software component, the **security monitor**, which runs at the highest privilege level (machine level in RISC-V) and therefore is immune to compromised

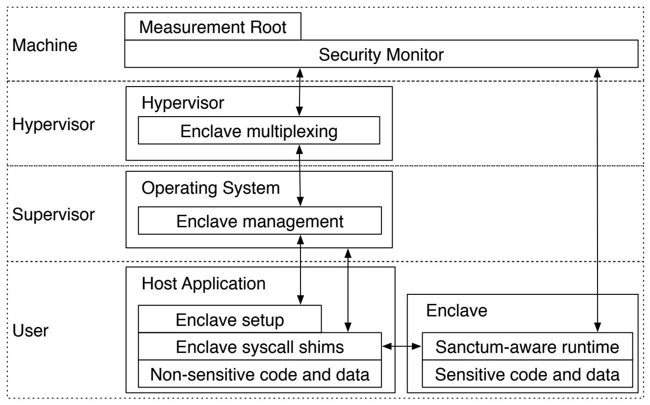


Figure 1: Software stack of a Sanctum machine

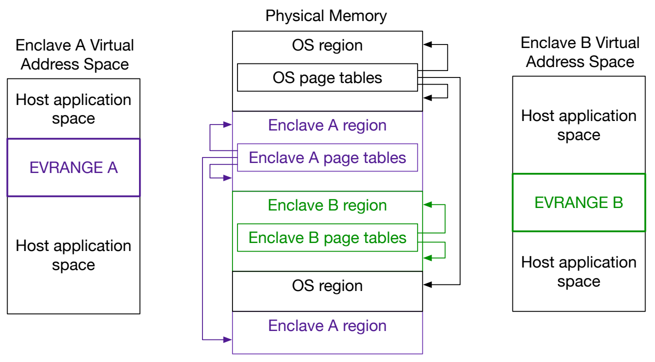


Figure 2: Per-enclave page tables

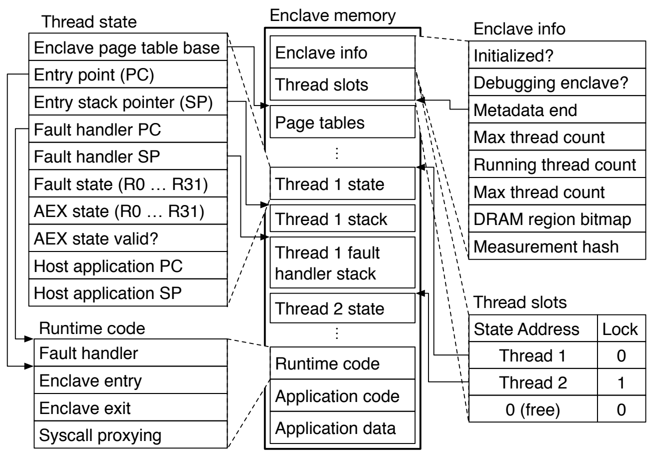


Figure 3: Enclave layout and data structures

system software.

We follow SGX's approach of relegating the management of computation resources, such as DRAM and execution cores, to untrusted system software. In Sanctum, the security monitor checks the system software's allocation decisions for correctness and commits them into the hardware's configuration. For simplicity, we refer to the software that manages resources as an *OS (operating system)*, even though it may be a combination of a hypervisor and a guest operating system kernel.

Like in SGX, an enclave stores its code and private data in parts of DRAM that have been allocated by the OS exclusively for the enclave's use, which are collectively called **the enclave's memory**. Consequently, we refer to the regions of DRAM that are not allocated to any enclave as **OS memory**. The security monitor tracks DRAM ownership, and ensures that no piece of DRAM is assigned to more than one enclave.

Each Sanctum enclave uses a range of virtual memory addresses (EVRANGE) to access its memory. The enclave's memory is mapped by the enclave's own page tables, which are also stored in the enclave's memory (Figure 2). This deviation from SGX is needed because page table dirty and accessed bits reveal memory access patterns at page granularity.

The enclave's virtual address space outside EVRANGE is used to access its host application's memory, via the page tables set up by the OS. Sanctum's hardware extensions implement dual page table lookup (§ 5.2), and make sure that an enclave's page tables can only point into the enclave's memory, while OS page tables can only point into OS memory (§ 5.3).

Figure 3 shows the contents of an enclave's memory. Like SGX, Sanctum supports multi-threaded enclaves, and enclaves must provision thread state areas for each thread that executes enclave code. Enclave threads, like their SGX cousins, run at the lowest privilege level (user level in RISC-V), so that a malicious enclave cannot compromise the OS. Specifically, enclaves cannot execute privileged instructions, and address translations that use OS page tables will result in page faults when accessing supervisor pages.

Sanctum, like SGX, considers system software to be untrusted, so it regulates transitions into and out of enclave code. An enclave's host application **enters an enclave** via a security monitor call that locks a thread state area, and transfers control to its entry point. After completing its intended task, the enclave code **exits** by asking the monitor to unlock the thread's state area, and transfer control back to the host application.

Like in SGX, enclaves cannot make system calls directly, as we cannot trust the OS to restore an enclave's execution state. Instead, the enclave's runtime must ask the host application to proxy syscalls such as file system and network I/O requests.

Sanctum's security monitor is the first responder for all hardware exceptions, including interrupts and faults.

Like in SGX, an interrupt received during enclave execution causes an *asynchronous enclave exit* (AEX), whereby the monitor saves the core's registers in the current thread's AEX state area, zeroes the registers, exits the enclave, and dispatches the interrupt as if it was received by the code entering the enclave.

Unlike in SGX, resuming enclave execution after an AEX means re-entering the enclave using its normal entry point, and having the enclave's code ask the security monitor to restore the pre-AEX execution state. Sanctum enclaves are aware of asynchronous exits so they can implement security policies. For example, an enclave thread that performs time-sensitive work, such as periodic I/O, may terminate itself if it ever gets preempted by an AEX.

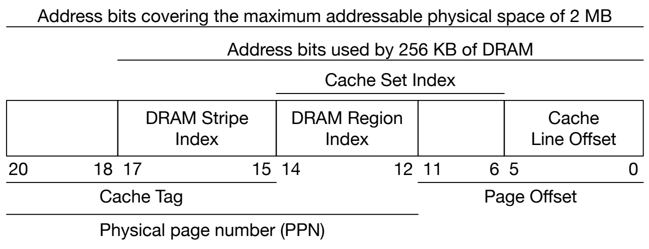
Furthermore, whereas SGX dispatches faults (with sanitized information) to the OS, our security monitor dispatches all faults occurring within an enclave to a designated fault handler inside the enclave, which is expected to be implemented by the enclave's runtime. For example, a `libc` runtime would translate faults into UNIX signals which, by default, would exit the enclave. It is possible, though not advisable for performance reasons (§ 6.3), for a runtime to handle page faults and implement demand paging in a secure manner, without being vulnerable to the attacks described in [48].

Unlike SGX, we isolate each enclave's data throughout the system's cache hierarchy. The security monitor flushes per-core caches, such as the L1 cache and the TLB, whenever a core jumps between enclave and non-enclave code. *Last-level cache* (LLC) isolation is achieved by a simple partitioning scheme supported by Sanctum's hardware extensions (§ 5.1).

Sanctum's strong isolation yields a simple security model for application developers: *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 enclave runtime 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.

The enclave runtime can use Native Client's approach



Address bits covering the maximum addressable physical space of 2 MB
Address bits used by 256 KB of DRAM
Cache Set Index
DRAM Stripe Index
DRAM Region Index
Cache Line Offset
20
18 17
15 14
12 11
6 5
0
Cache Tag
Page Offset
Physical page number (PPN)

Figure 4: Interesting bit fields in a physical address

[50] to make sure that the rest of the enclave software only interacts with the host application via the runtime, and mitigate any potential security vulnerabilities in enclave software.

5 HARDWARE EXTENSIONS

Sanctum uses an LLC partitioning mechanism that is readily available thanks to the interaction between page tables and direct-mapped or set-associative caches. By manipulating the input to the cache set indexing computation, as described in § 5.1, the computer's DRAM is divided into equally sized contiguous **DRAM regions** that use disjoint LLC sets. The OS allocates the cache dynamically to its processes and to enclaves, by allocating DRAM regions. Each DRAM region can either be entirely owned by an enclave, or entirely owned by the OS and used by non-enclave processes.

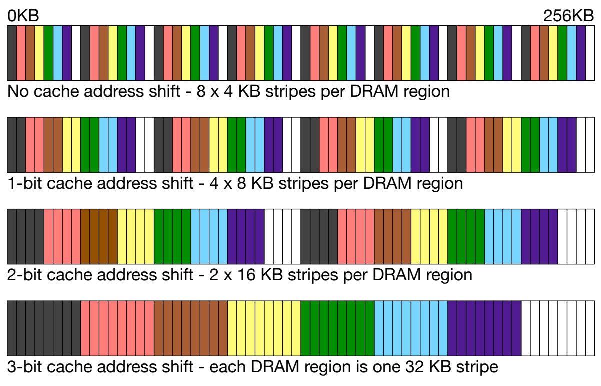
We modify the input to the *page walker* that translates addresses on TLB misses, so it can either use the OS page tables or the current enclave's page tables, depending on the translated virtual address (§ 5.2). We also add logic between the page walker and the L1 cache, to ensure that the OS page table entries can only point into DRAM regions owned by the OS, and the current enclave's page table entries can only point into DRAM regions owned by the enclave (§ 5.3).

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 (§ 5.4).

5.1 LLC Address Input Transformation

Figure 4 depicts a physical address in a toy computer with 32-bit virtual addresses and 21-bit physical addresses, 4,096-byte pages, a set-associative LLC with 512 sets and 64-byte lines, and 256 KB of DRAM.

The location where a byte of data is cached in the LLC



0KB
256KB
No cache address shift - 8 x 4 KB stripes per DRAM region
1-bit cache address shift - 4 x 8 KB stripes per DRAM region
2-bit cache address shift - 2 x 16 KB stripes per DRAM region
3-bit cache address shift - each DRAM region is one 32 KB stripe

⬛ Region 0	🟥 Region 1	🟫 Region 2	🟨 Region 3
🟩 Region 4	🟦 Region 5	🟪 Region 6	⬜ Region 7

Figure 5: Cache address shifting makes DRAM regions contiguous

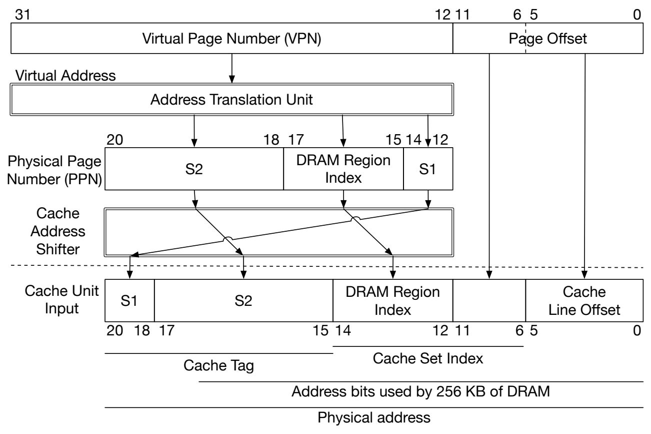


Figure 6: Cache address shifter that shifts the PPN by 3 bits

depends on the low-order bits in the byte's physical address. The *set index* determines which of the LLC lines can cache the line containing the byte, and the *line offset* locates the byte in its cache line. A virtual address's low-order bits make up its *page offset*, while the other bits are its *virtual page number* (VPN). Address translation leaves the page offset unchanged, and translates the VPN into a *physical page number* (PPN), based on the mapping specified by the page tables.

We define the **DRAM region index** in a physical address as the intersection between the PPN bits and the cache index bits. This is the maximal set of bits that impact cache placement *and* are determined by privileged software via page tables. We define a **DRAM region** to be the subset of DRAM with addresses having the same DRAM region index. In Figure 4, for example, address bits $[14\ldots12]$ are the DRAM region index, dividing the physical address space into 8 DRAM regions.

DRAM regions are the basis of our cache partitioning because *addresses in a DRAM region do not collide in the LLC with addresses from any other DRAM region*. If programs Alice and Eve use disjoint DRAM regions, they cannot interfere in the LLC, so Eve cannot mount LLC timing attacks on Alice. Furthermore, the OS can place applications in different DRAM regions by manipulating page tables, without having to modify application code.

In a typical system without Sanctum's hardware extensions, DRAM regions are made up of multiple continuous **DRAM stripes**, where each stripe is exactly one page long. The top of Figure 5 drives this point home, by showing the partitioning of our toy computer's 256 KB of DRAM into DRAM regions. The fragmentation of DRAM regions makes it difficult for the OS to allocate contiguous DRAM buffers, which are essential to the efficient DMA transfers used by high performance devices. In our example, if the OS only owns 4 DRAM regions, the largest contiguous DRAM buffer it can allocate is 16 KB.

We observed that, up to a certain point, circularly shifting (rotating) the PPN of a physical address to the right by one bit, before it enters the LLC, doubles the size of each DRAM stripe and halves the number of stripes in a DRAM region, as illustrated in Figure 5.

Sanctum takes advantage of this effect by adding a **cache address shifter** that circularly shifts the PPN to the right by a certain amount of bits, as shown in Figures 6 and 7. In our example, configuring the cache address shifter to rotate the PPN by 3 yields contiguous DRAM regions, so an OS that owns 4 DRAM regions could hypothetically allocate a contiguous DRAM buffer covering half of the machine's DRAM.

The cache address shifter's configuration depends on the amount of DRAM present in the system. If our example computer could have 128 KB - 1 MB of DRAM, its cache address shifter must support shift amounts from 2 to 5. Such a shifter can be implemented via a 3-position variable shifter circuit (series of 8-input MUXes), and a fixed shift by 2 (no logic). Alternatively, in systems with known DRAM configuration (embedded, SoC, etc.), the shift amount can be fixed, and implemented with no logic.

5.2 Page Walker Input

Sanctum's per-enclave page tables require an enclave page table base register `eptbr` that stores the physical address of the currently running enclave's page tables, and has similar semantics to the page table base register

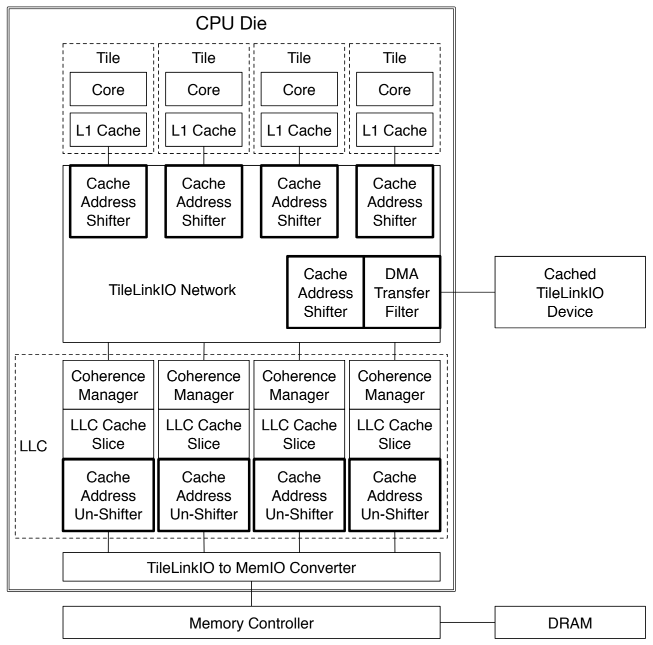


Figure 7: Sanctum's cache address shifter and DMA transfer filter logic in the context of a RISC V-Rocket uncore

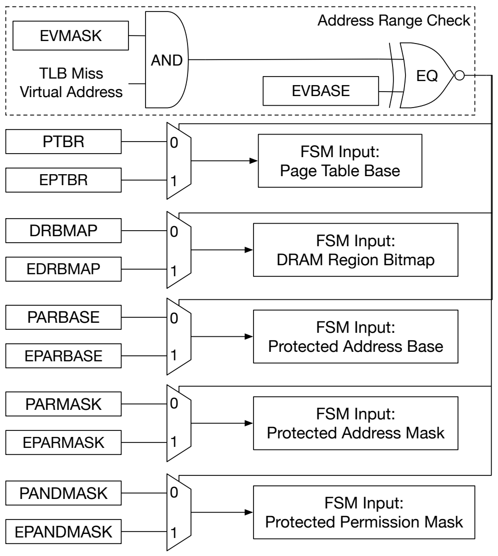

Address Range Check
EVMASK
TLB Miss Virtual Address
AND
EVBASE
EQ
PTBR
EPTBR
FSM Input: Page Table Base
DRBMAP
EDRBMAP
FSM Input: DRAM Region Bitmap
PARBASE
EPARBASE
FSM Input: Protected Address Base
PARMASK
EPARMASK
FSM Input: Protected Address Mask
PANDMASK
EPANDMASK
FSM Input: Protected Permission Mask

Figure 8: Page walker input for per-enclave page tables

`ptbr` pointing to the operating system-managed page tables. These registers may only be accessed by the Sanctum security monitor, which provides an API call for the OS to modify `ptbr`, and ensures that `eptbr` always points to the current enclave's page tables.

The circuitry handling TLB misses switches between `ptbr` and `eptbr` based on two registers that indicate the current enclave's ERANGE, namely `evbase` (enclave virtual address space base) and `evmask` (enclave virtual address space mask). When a TLB miss occurs, the circuit in Figure 8 selects the appropriate page table base by ANDing the faulting virtual address with the mask register and comparing the output against the base register. Depending on the comparison result, either `eptbr` or `ptbr` is forwarded to the page walker as the page table base address.

5.3 Page Walker Memory Accesses

In modern high-speed CPUs, address translation is performed by a hardware **page walker** that traverses the page tables when a TLB miss occurs. The page walker's latency greatly impacts the CPU's performance, so it is implemented as a finite-state machine (FSM) that reads page table entries by issuing DRAM read requests using physical addresses, over a dedicated bus to the L1 cache.

Unsurprisingly, page walker modifications require a lot of engineering effort. At the same time, Sanctum's security model demands that the page walker only references enclave memory when traversing the enclave page tables, and only references OS memory when translating the OS page tables. Fortunately, we can satisfy these requirements without modifying the FSM. Instead, the security monitor works in concert with the circuit in Figure 9 to ensure that the page tables only point into allowable memory.

Sanctum's security monitor must guarantee that `ptbr` points into an OS DRAM region, and `eptbr` points into a DRAM region owned by the enclave. This secures the page walker's initial DRAM read. The circuit in Figure 9 receives each page table entry fetched by the FSM, and sanitizes it before it reaches the page walker FSM.

The security monitor configures the set of DRAM regions that page tables may reference by writing to a DRAM region bitmap (`drbmap`) register. The sanitization circuitry extracts the DRAM region index from the address in the page table entry, and looks it up in the DRAM region bitmap. If the address does to belong to an allowable DRAM region, the sanitization logic forces the page table entry's valid bit to zero, which will cause the page walker FSM to abort the address translation and signal a page fault.

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. Sanctum extends 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.

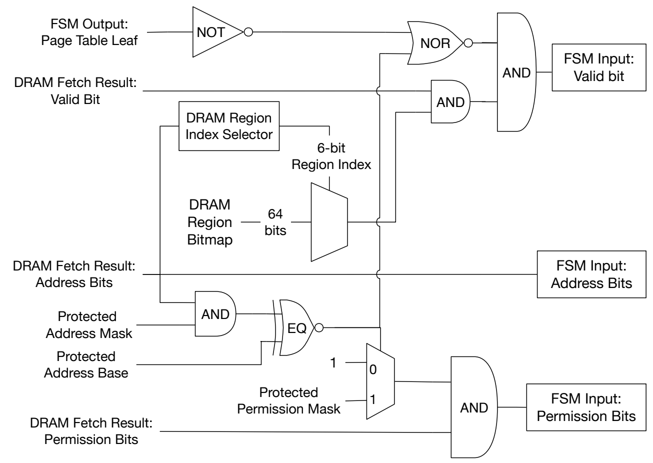

FSM Output: Page Table Leaf
NOT
NOR
AND
FSM Input: Valid bit
DRAM Fetch Result: Valid Bit
AND
DRAM Region Index Selector
6-bit Region Index
DRAM Region Bitmap
64 bits
DRAM Fetch Result: Address Bits
FSM Input: Address Bits
AND
Protected Address Mask
EQ
Protected Address Base
1
0
1
Protected Permission Mask
AND
FSM Input: Permission Bits
DRAM Fetch Result: Permission Bits

Figure 9: Hardware support for per-enclave page tables: transforming the page table entries fetched by the page walker.

The page table entry sanitization logic in Sanctum's hardware extensions checks if each page table entry points into the protected address range by ANDing the entry's address with the protected range mask and comparing the result with the protected range base.

If a leaf page table entry is seen with a protected address, its permission bits are masked with a protected permissions mask (parpmask) register. Upon discovering a protected address in an intermediate page table entry, its valid bit is cleared, 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 modifying the protected region by setting accessed and dirty bits.

All registers mentioned above come in pairs, as Sanctum maintains 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 10 shows Sanctum's logic inserted between the page walker and the cache unit that fetches page table entries.

5.4 DMA Transfer Filtering

We whitelist a DMA-safe DRAM region instead of following SGX's blacklist approach. Specifically, Sanctum adds two registers (a base, dmarbase and an AND mask, dmarmask) to the DMA arbiter (memory controller). The range check circuit shown in Figure 8 compares each DMA transfer's start and end addresses against the allowed DRAM range, and the DMA arbiter drops transfers that fail the check.

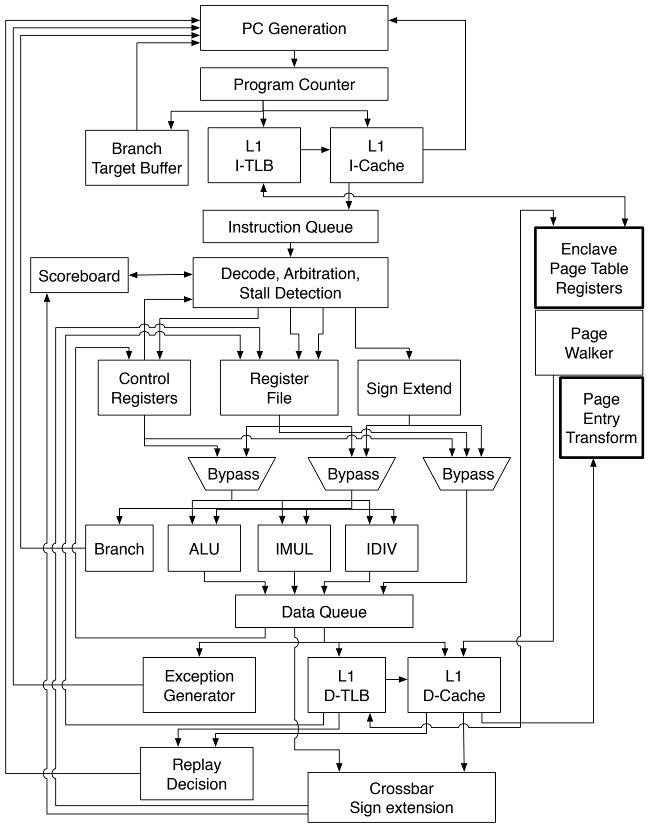


Figure 10: Sanctum's page entry transformation logic in the context of a RISC V Rocket core

6 SOFTWARE DESIGN

Sanctum has two pieces of trusted software: the measurement root (§ 6.1), which is burned in on-chip ROM, and the security monitor (§ 6.2), which is stored alongside the computer's firmware (usually flash memory). We expect both to be amenable to formal verification: our implementation of a security monitor for Sanctum has fewer than 5 kloc of C++, including a subset of the standard library and the cryptography used for enclave attestation.

6.1 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 (shown in Figure 12) is expected to be stored in non-volatile memory (such as an 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.

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.

6.2 Security Monitor

The security monitor receives control after `mroot` finishes setting up the attestation measurement chain.

The monitor provides API calls to the operating system and enclaves for **DRAM region allocation** and **enclave management**. The monitor guards sensitive registers, such as the page table base register (`ptbr`) and the allowed DMA range (`dmarbase` and `dmarmask`). The OS can set these registers via monitor calls that ensure the register values are consistent with the current DRAM region allocation.

Figure 11 shows the DRAM region allocation state transition diagram. After the system boots up, all DRAM regions are allocated to the OS, which can free up DRAM regions so it can re-assign them to enclaves or to itself. A DRAM region can only become free after it is blocked by its owner, which can be the OS or an enclave. While a DRAM region is blocked, any address translations mapping to it cause page faults, so no new TLB entries will be created for that region. Before the OS frees the

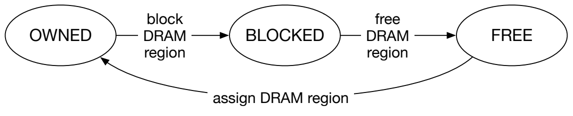


Figure 11: DRAM region allocation states and API calls

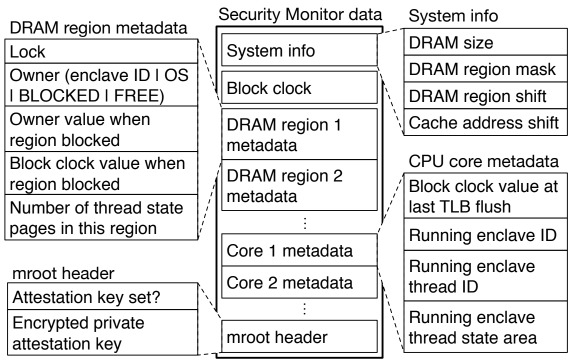


Figure 12: Security monitor data structures

blocked region, it must flush all the cores' TLBs, to remove any stale entries for the region.

The monitor ensures that the OS performs TLB shootdowns, using 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 Figure 3). When a core's TLB is flushed, that core's flush time is set to the current block clock value. When the OS asks the monitor to free a blocked DRAM region, the monitor verifies that no core's flush time is lower than the block clock value stored in the region's metadata. As an optimization, freeing a region owned by an enclave only requires TLB flushes on the cores running that enclave's threads. No other core can have TLB entries for the enclave's memory.

Figure 13 shows the enclave management state diagram. The OS creates an enclave by issuing a monitor call to allocate a free DRAM region to the enclave and initialize the enclave metadata fields (shown in Figure 3) 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 monitor calls that 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 updates the enclave's measurement hash. The OS then issues a monitor 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

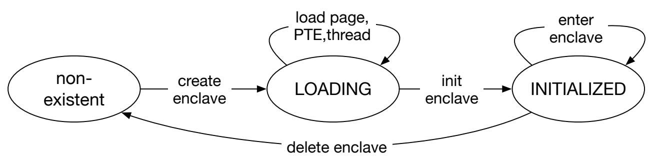


Figure 13: Enclave states and enclave management API calls

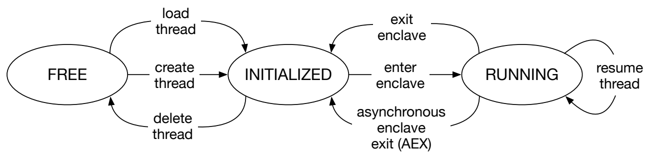


Figure 14: Enclave thread slot states and thread-related API calls

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. Therefore, the metadata area can safely 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 owned by the enclave. Once an enclave is initialized, it can inspect its own page tables and abort if the OS created undesirable mappings. Simple enclaves do not require specific mappings. Complex enclaves are expected to communicate their desired mappings to the OS via out-of-band metadata not covered by this work.

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

Each enclave thread state area is identified by a thread slot that contains its address, which is stored in the enclave metadata. Figure 14 shows the thread slot state diagram. At enclave creation, the OS specifies the number of thread slots in the enclave's metadata. Initially, all thread slots are free. During enclave loading, the OS can initialize a free slot via a *load thread* monitor call, which designates the thread's state area and stores it in the thread slot. Running enclaves may initialize additional slots using a similar monitor call.

When performing an AEX, the security monitor atomically tests-and-sets the *AEX state valid* flag in the current thread's state. If the flag is clear, the monitor stores the core's execution state in the thread state's AEX area. Otherwise, the enclave thread was resuming from an AEX, so the monitor does not change the AEX area. When the host application re-enters the enclave, it will resume from the previous AEX. This reasoning avoids the complexity of SGX's state stack.

The security monitor is highly concurrent, with fine-grained locks. 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. An enclave is guarded by the lock of the DRAM region holding its metadata. Each thread slot has a lock guarding it and the thread state that it points to. The thread slot lock is also acquired while the slot's thread state area is used by a core running enclave code. Thus, the *enter enclave* call acquires a slot lock, which is released by an *enclave exit* call or by an AEX.

We avoid deadlocks by using a form of optimistic locking. Each monitor call 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.

6.3 Enclave Eviction

General-purpose software can be enclaved without source code changes, provided that it is linked against a runtime (e.g., *libc*) modified to work with Sanctum. Any such runtime would be included in the TCB.

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, giving the enclave an opportunity to rearrange data in its RAM regions. DRAM region management can be transparent to the programmer if handled by the enclave's runtime.

The security monitor does not allow the OS to forcibly reclaim a single DRAM region from an enclave, as doing so would leak memory access patterns. Instead, the OS can delete an enclave, after stopping its threads, and reclaim its DRAM regions. Thus, a small or short-running enclave may well refuse DRAM region management requests from the OS, and expect the OS to delete and re-run it under memory pressure.

To avoid wasted work, large long-running enclaves may elect to use demand paging to overcommit their DRAM, albeit with the understanding that demand paging leaks page-level access patterns to the OS. Securing this mechanism requires the enclave to obfuscate its page faults via periodic I/O using oblivious RAM techniques, as in the Ascend processor [19], applied at page rather than cache line granularity. This carries a high overhead: even with a small chance of paging, an enclave must generate periodic page faults, and access a large set of pages at each period. Using an analytic model, we estimate the overhead to be upwards of 12ms per page per period for a high-end 10K RPM drive, and 27ms for a value hard drive. Given the number of pages accessed every period grows with an enclave's data size, the costs are easily prohibitive: an enclave accessing pages every second may struggle to make forward progress. While SSDs may alleviate some of this prohibitive overhead, and will be investigated in future work, Sanctum focuses on securing enclaves without demand paging.

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 to obfuscate their access patterns from the operating system. These techniques are independent of application business logic, and can be provided by libraries such as database access drivers.

Lastly, the presented design requires each enclave to always occupy at least one DRAM region, which contains enclave data structures and the memory management code described above. Evicting all of a live enclave's memory requires an entirely different scheme to be described in future work.

Briefly, the OS can ask the security monitor to *freeze* an enclave, encrypting the enclave's DRAM regions in place, and creating a leaf node in a hash tree. When the monitor *thaws* a frozen enclave, it uses the hash tree leaf to ensure freshness, decrypts the DRAM regions, and *relocates* the enclave, updating its page tables to reflect new owners of relevant DRAM regions. The hash tree is managed by the OS using an approach similar to SGX's version array page eviction.

7 Security Analysis

Sanctum protects each enclave in a system from the (potentially compromised) operating system and from other, potentially malicious, enclaves. The security monitor (§ 6.2) keeps track of the operating system's assignment of DRAM regions to enclaves, and enforces the invariant that *every DRAM region is 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 § 5.2 and § 5.3 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 § 5.1 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 § 5.2 and § 5.3 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 OS 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 § 5.2 and § 5.3 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.

8 Performance Evaluation

While we propose a high-level set of hardware and software to implement Sanctum, we focus our evaluation on the concrete example of a 4-core RISC-V system generated by Rocket Chip [28]. As Sanctum conveniently isolates concurrent workloads from each other, we can examine its overhead by examining individual applications executing on a single core, discounting the effect of other running software.

8.1 Experiment Design

We use a Rocket-Chip generator modified to model Sanctum's hardware modifications (§ 5). We generate a 4-core 64-bit RISC-V CPU with private 16KB 4-way set associative instruction and data L1 caches. Using a cycle-accurate simulator for this machine, we produce an LLC access trace and post-process it with a cache emulator for a shared 4MB LLC with and without Sanctum's DRAM region isolation. We accurately compute the program completion time, in cycles, for each benchmark because Rocket cores have in-order single issue pipelines, and cannot make any progress on a TLB or cache miss. We use a simple model for DRAM accesses and assume unlimited 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.

Our cache size choices are informed by Intel's Sandy Bridge [23] 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 hierarchical caches are not implemented by Rocket. We note, however, that a private L2 would greatly reduce each core's LLC traffic, which is Sanctum's main 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 an additional 100 cycles.

Using the hardware model above, we benchmark the subset of SPECINT 2006 [3] 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 and compute-bound long-running workloads with diverse access locality.

We choose not to simulate a complete Linux kernel, and instead use the RISC-V proto kernel [42] that provides the few services used by our benchmarks. We schedule each benchmark on Core 0, and run it to completion, while the other cores are idling.

8.2 Cost of Added Hardware

Sanctum's hardware changes add relatively few gates to the Rocket chip, but do increase its area and power consumption. Like SGX, we avoid modifying the core's critical path: while our addition to the page walker (as analyzed in the next section) may increase the latency of TLB misses, it does not increase the Rocket core's clock cycle, which is competitive with an ARM Cortex-A5 [28].

As illustrated at the gate level in Figures 8 and 9, 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, and do not include it in the percentages above (the LLC generally accounts for half of chip area).

8.3 Added Page Walker Latency

Sanctum's page table entry transformation logic is described in § 5.3, and we expect it can be combined with the page walker FSM logic within a single clock cycle.

Nevertheless, in the worst case, the transformation logic would add a pipeline stage between the L1 data cache and the page walker. The transformation logic is small and combinational, 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.

The overheads due to an additional cycle of TLB miss latency are negligible, as quantified by the completion time of SPECINT benchmarks. All overheads are well below 0.01%, relative to the completion time without added TLB latency. This overhead is insignificant relative to the overheads of cache isolation: TLB misses are infrequent and relatively expensive, and a single additional cycle makes little difference.

8.4 Security Monitor Overhead

Invoking Sanctum's security monitor to load code into an enclave adds a one-time setup cost to each isolated process, when compared against running code without Sanctum's isolation container. This overhead is amortized by the duration of the computation, so we discount it for long-running workloads.

Entering and exiting enclaves is more expensive than hardware context switches: the security monitor must flush TLBs and L1 caches to prevent a privacy leak. However, a sensible OS is expected to minimize the number

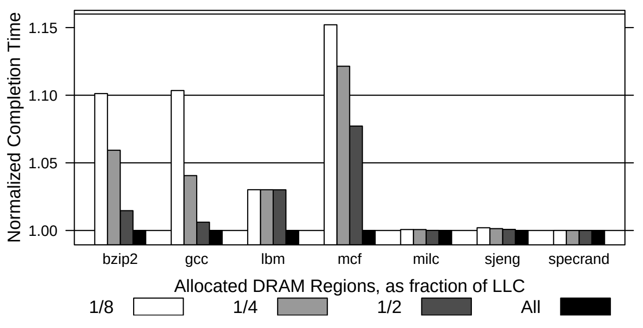


Figure 15: The impact of DRAM region allocation on the completion time of an enclaved benchmark, relative to an idea insecure baseline

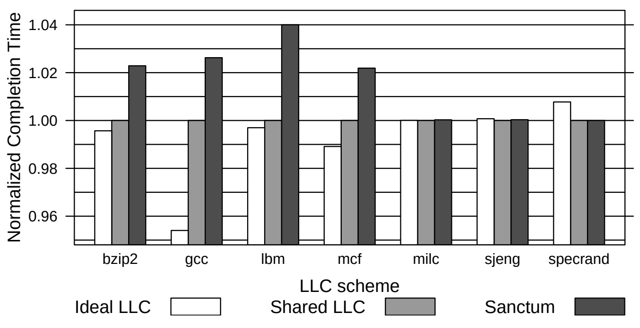


Figure 16: Sanctum's enclave overheads for one core utilizing 1/4 of the LLC compared against an idealized baseline (non-enclaved app using the entire LLC), and against a representative baseline (non-enclaved app sharing the LLC with concurrent instances)

of context switches by allocating some cores to an enclave and allowing them to execute to completion. We therefore also consider this overhead to be negligible for long-running computations.

8.5 Overhead of DRAM Region Isolation

The crux of Sanctum's strong isolation is caching DRAM regions in distinct sets. Therefore, when the OS assigns DRAM regions to an enclave, it also confines it to a share of the LLC. An enclaved thread effectively runs on a machine with a smaller LLC, which impacts the enclave's performance. Note, however, that Sanctum does not partition the per-core caches, so a thread can utilize its core's entire L1 caches and TLBs.

Figure 15 shows the completion times of the SPECINT workloads, each 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 15.1% 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.

We also consider a more representative baseline by considering the performance of a system executing multiple copies of the benchmark concurrently, in different virtual address spaces, effectively normalizing LLC resources available to each instance. Overheads for a reasonable allocation of 1/4th of the LLC shared among 4 instances are shown in Figure 16. With this baseline, `mcf` overheads are reduced to 4.6% and 2.2% for allocations of 1/8th and 1/4th of the LLC, respectively. Over the full set of benchmarks, overheads fall below 4%, averaging 1.9%. Memory-bound benchmarks exhibit a 2.7% average overhead over this insecure baseline.

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 of 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 [23]) 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.

9 Conclusion

We have shown through the design of Sanctum that strong provable isolation of concurrent software modules can be achieved with low overhead. The worst observed overhead across all benchmarks when compared to a representative insecure baseline is 4.6%. 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 secure hardware architecture approaches away from plugging specific security holes to a principled approach to eliminating attack surfaces.

REFERENCES

[1] 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].

[2] 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].

[3] Spec cpu 2006. Technical report, Standard Performance Evaluation Corporation, May 2015.

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

[5] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent 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*, volume 13, 2013.

[6] Sebastian 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].

[7] Sebastian Banescu. Cache timing attacks. 2011. [Online; accessed 26-January-2014].

[8] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 201–215. Springer, 2006.

[9] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*, pages 355–371. Springer, 2011.

[10] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[11] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

[12] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing*, pages 108–119. ACM, 2011.

[13] Shaun Davenport. Sgx: the good, the bad and the downright ugly. *Virus Bulletin*, 2014.

[14] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[15] Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. Using cpu system management mode to circumvent operating system security functions. *CanSecWest/core06*, 2006.

[16] Alan Dunn, Owen Hofmann, Brent Waters, and Emmett Witchel. Cloaking malware with the trusted platform module. In *USENIX Security Symposium*, 2011.

[17] Shawn Embleton, Sherri Sparks, and Cliff C Zou. Smm rootkit: a new breed of os independent malware. *Security and Communication Networks*, 2010.

[18] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 190–202. IEEE, 2014.

[19] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.

[20] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.

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

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

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

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur 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*, pages 361–372. IEEE Press, 2014.

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

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

[27] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures*, pages 25–34. ACM, 2008.

[28] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanovic, and Krste 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*, pages 199–202. IEEE, 2014.

[29] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[30] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 203–215. IEEE, 2014.

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 143–158. IEEE, 2015.

[32] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 13:10, 2013.

[33] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox – practical cache attacks in javascript. *arXiv preprint arXiv:1502.07373*, 2015.

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

[35] Joanna Rutkowska and Rafał Wojtczuk. Preventing and detecting xen hypervisor subversions. *Blackhat Briefings USA*, 2008.

[36] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 187–198. IEEE, 2010.

[37] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 57–68. ACM, 2011.

[38] Mark Seaborn and Thomas 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].

[39] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[40] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

[41] Zhenghong Wang and Ruby 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, pages 494–505, 2007.

[42] Andrew Waterman, Yunsup Lee, and et al. Celio, Christopher. Risc-v proxy kernel and boot loader. Technical report, EECS Department, University of California, Berkeley, May 2015.

[43] Filip Wecherowski. A real smm rootkit: Reversing and hooking bios smi handlers. *Phrack Magazine*, 13(66), 2009.

[44] Rafal Wojtczuk and Joanna Rutkowska. Attacking intel trusted execution technology. *Black Hat DC*, 2009.

[45] Rafal Wojtczuk and Joanna Rutkowska. Attacking smm memory via intel cpu cache poisoning. *Invisible Things Lab*, 2009.

[46] Rafal Wojtczuk and Joanna Rutkowska. Attacking intel txt via sinit code execution hijacking. *Invisible Things Lab*, 2011.

[47] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another way to circumvent intel® trusted execution technology. *Invisible Things Lab*, 2009.

[48] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. IEEE Institute of Electrical and Electronics Engineers, May 2015.

[49] Yuval Yarom and Katrina E Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

[50] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.