Sanctum: Minimal Hardware Extensions for Strong Software Isolation

Victor Costan, Ilia Lebedev, and Srinivas Devadas, *MIT CSAIL*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

This paper is included in the Proceedings of the
25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

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 Intel's Software Guard Extensions (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. Sanctum shuns unnecessary complexity, leading to a simpler security analysis. We follow a principled approach to eliminating entire attack surfaces through isolation, rather than plugging attack-specific privacy leaks. Most of Sanctum's logic is implemented in trusted software, which does not perform cryptographic operations using keys, and is easier to analyze than SGX's opaque microcode, which does.

Our prototype targets a Rocket RISC-V core, an open implementation that allows any researcher to reason about its security properties. Sanctum's extensions can be adapted to other processor cores, because we do not change any major CPU building block. Instead, we add hardware at the interfaces between generic building blocks, without impacting cycle time.

Sanctum demonstrates that strong software isolation is achievable with a surprisingly small set of minimally invasive hardware changes, and a very reasonable overhead.

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 [26] spent *20 man-years* to cover 9,000 lines of code.

Given Linux and Xen's history of vulnerabilities and uncertain 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, 36] 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.

Unfortunately, although the SGX design includes a vast array of defenses against a variety of software and physical attacks, it fails to offer meaningful software isolation guarantees. The SGX threat model protects against all direct attacks, but excludes "side-channel attacks", even if they can be performed via software alone.

Furthermore, our analysis [13] of SGX reveals that it is impossible for anyone but Intel to reason about SGX's security properties, because significant implementation details are not covered by the publicly available documentation. This is a concern, as the myriad of security vulnerabilities [16, 18, 39, 50–54] in TXT [22], Intel's previous attempt at securing remote computation, show that securing the machinery underlying Intel's processors is incredibly challenging, even in the presence of strong economic incentives.

Our main contribution is a software isolation scheme that addresses the issues raised above: Sanctum's isolation provably defends against known software side-channel attacks, including cache timing attacks and passive address translation attacks. Sanctum is a co-design that combines **minimal** and **minimally invasive** hardware modifications with a trusted software **security monitor** that is amenable to rigorous analysis and does not perform cryptographic operations using keys.

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 is minimally invasive because it does not require 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 limits the effort needed to validate a Sanctum implementation.

We demonstrate that memory access pattern attacks by malicious software can be foiled without incurring unreasonable overheads. 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 page-coloring-based cache partitioning scheme with Sanctum adds a few percent of overhead in execution time, which is orders of magnitude lower than the overheads of the ORAM schemes [21, 43] that are usually employed to conceal memory access patterns.

All layers of Sanctum's TCB are open-sourced at `https://github.com/pwnall/sanctum` and unencumbered by patents, trade secrets, or other similar intellectual property concerns that would disincentivize security researchers from analyzing it. Our prototype targets the Rocket Chip [29], an open-sourced implementation of the RISC-V [47, 49] instruction set architecture, which is an open standard. Sanctum's software stack bears the MIT license.

To further encourage analysis, most of our security monitor is written in portable C++ which, once rigorously analyzed, can be used across different CPU implementations. Furthermore, even the non-portable assembly code can be reused across different implementations of the same architecture.

2 Related Work

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

Cache timing attacks are known to retrieve cryptographic keys used by AES [8], RSA [10], Diffie-Hellman [27], and elliptic-curve cryptography [9]. While early attacks required access to the victim's CPU core, recent sophisticated attacks [35, 56] target the last-level cache (LLC), which is shared by all cores in a socket. Recently, [37] demonstrated a cache timing attack that uses JavaScript code in a page visited by a web browser.

Cache timing attacks observe a victim's memory access patterns at cache line granularity. However, recent work shows that private 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 [55].

XOM [30] introduced the idea of having sensitive code and data execute in isolated containers, and placed the OS in charge of resource allocation without trusting it. Aegis [44] 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 from software threats.

Sanctum only considers software attacks in its threat model (§ 3). Resilience against physical attacks can be added by augmenting a Sanctum processor with the countermeasures described in other secure architectures, with associated increased performance overheads. Aegis protects a container's data when the DRAM is untrusted through memory encryption and integrity verification; these techniques were adopted and adapted by SGX. Ascend [20] and GhostRider [32] use Oblivious RAM [21] to protect a container's memory access patterns against adversaries that can observe the addresses on the memory bus. An insight in Sanctum is that these overheads are unnecessary in a software-only threat model.

Intel's Trusted Execution Technology (TXT) [22] is widely deployed in today's mainstream computers, due to its approach of trying to add security to a successful CPU product. After falling victim to attacks [51, 54] 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 Sanctum also does.

Intel's SGX [5, 36] 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. At the same time, SGX has many security issues that are solved by Sanctum, which are stated in this paper's introduction.

Iso-X [19] 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 [28, 46] and the Random Fill Cache Architecture (RFill, [34]) 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 [28, 46] 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 [15] uses a well-principled partitioning scheme, but does not completely stop leakage, and relies on the OS to assign hardware process IDs. CATalyst [33] trusts the Xen hypervisor to correctly tame Intel's Cache Allocation Technology into providing cache pinning, which can only secure software whose code and data fits into a fraction of the LLC.

Sanctum uses very simple cache partitioning [31] based on page coloring [24, 45], which has proven to have reasonable overheads. It is likely that sophisticated schemes like ZCache [40] and Vantage [41] can be combined with Sanctum's framework to yield better performance.

3 Threat Model

Sanctum isolates the software inside an **enclave** from other software on the same computer. All outside software, including privileged system software, can only interact with an enclave via a small set of primitives provided by the security monitor. 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.

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 or DMA memory probing, and cache timing attacks.

Sanctum's isolation protects the integrity and privacy of the code and data inside an enclave against any practical **software** attack that relies on observing or interacting with the enclave software via means outside the interface provided by the security monitor. In other words, we do not protect enclaves that leak their own secrets directly (e.g., by writing to untrusted memory) or by timing their operations (e.g., by modulating their completion times). In effect, Sanctum solves the security problems that emerge from sharing a computer among mutually distrusting applications.

This distinction is particularly subtle in the context of cache timing attacks. We do not protect against attacks like [10], where the victim application leaks information via its public API, and the leak occurs even if the victim runs on a dedicated machine. We *do* protect against attacks like Flush+Reload [56], which exploit shared hardware resources to interact with the victim via methods outside its public API.

Sanctum also defeats attackers who aim to compromise an OS or hypervisor by running malicious applications and enclaves. This addresses concerns that enclaves provide new attack vectors for malware [14, 38]. We assume that the benefits of meaningful software isolation outweigh enabling a new avenue for frustrating malware detection and reverse engineering [17].

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 monitor 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, OS, 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 by compromised system software: 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 (fault-injection attacks), such as rowhammer [25, 42].

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.

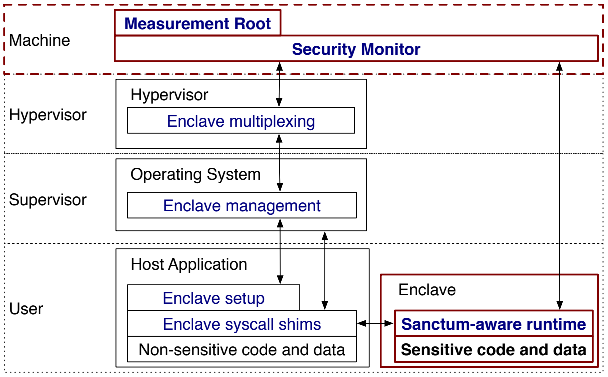


Figure 1: Software stack on a Sanctum machine; The blue text represents additions required by Sanctum. The bolded elements are in the software TCB.

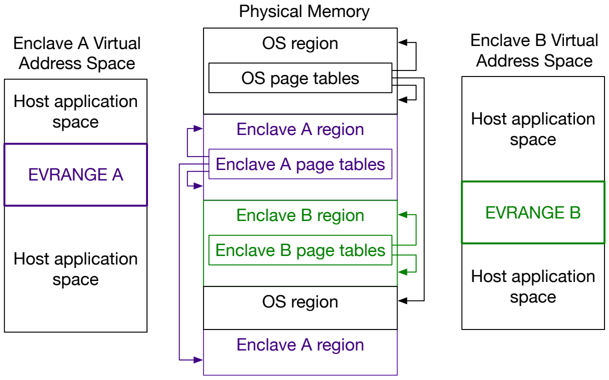


Figure 2: Per-enclave page tables

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 as described in [13].

The software stack on a Sanctum machine, shown in Figure 1, resembles the SGX stack with one notable exception: SGX's microcode is replaced by a trusted software component, the **security monitor**, which is protected from compromised system software, as it runs at the highest privilege level (machine level in RISC-V).

We relegate the management of computation resources, such as DRAM and execution cores, to untrusted system software (as does SGX). In Sanctum, the security monitor checks the system software's allocation decisions for correctness and commits them into the hardware's configuration registers. 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 OS kernel.

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 (as does SGX), 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 stored in the enclave's memory (Figure 2). This makes private the page table dirty and accessed bits, which can reveal memory access patterns at page granularity. Exposing an enclave's page tables to the untrusted OS leaves the enclave vulnerable to attacks such as [55].

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).

Sanctum supports multi-threaded enclaves, and enclaves must appropriately provision for thread state data structures. Enclave threads, like their SGX cousins, run at the lowest privilege level (user level in RISC-V), meaning a malicious enclave cannot compromise the OS. Specifically, enclaves may not execute privileged instructions; address translations that use OS page tables generate page faults when accessing supervisor pages.

The per-enclave metadata used by the security monitor is stored in dedicated DRAM regions (**metadata regions**), each managed at the page level by the OS, and each includes a page map that is used by the security monitor to verify the OS' decisions (much like the EPC and EPCM in SGX, respectively). Unlike SGX's EPC, the metadata region pages only store enclave and thread metadata. Figure 3 shows how these structures are weaved together.

Sanctum considers system software to be untrusted, and governs 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.

Enclaves cannot make system calls directly: we cannot trust the OS to restore an enclave's execution state, so the

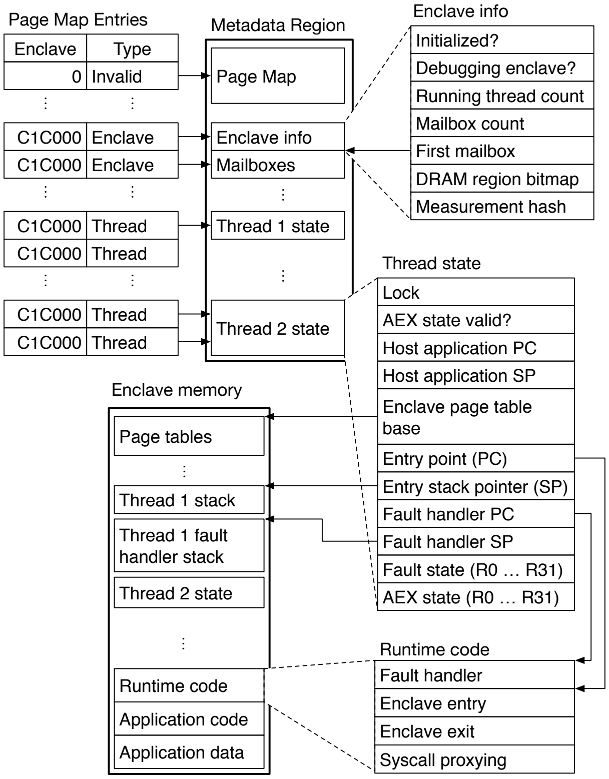


Figure 3: Enclave layout and data structures

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 interrupts: 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 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.

The security monitor configures the CPU to dispatch all faults occurring within an enclave directly to the enclave's designated fault handler, which is expected to be implemented by the enclave's runtime (SGX sanitizes and dispatches faults to the OS). 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 securely, and robust against the attacks described in [55].

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 runtime 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 [57] to ensure that the rest of the enclave software only interacts with the host application via the runtime to mitigate potential security vulnerabilities in enclave software.

The lifecycle of a Sanctum enclave closely resembles the lifecycle of its SGX equivalent. An enclave is created when its host application performs a system call asking the OS to create an enclave from a dynamically loadable module (`.so` or `.dll` file). The OS invokes the security monitor to assign DRAM resources to the enclave, and to load the initial code and data pages into the enclave. Once all the pages are loaded, the enclave is marked as initialized via another security monitor call.

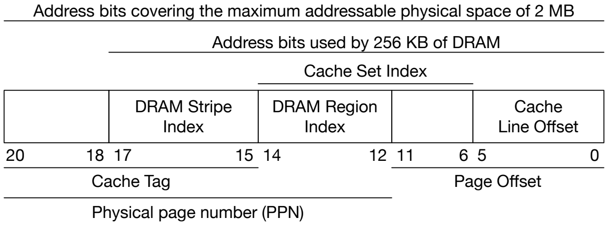
Our software attestation scheme is a simplified version of SGX's scheme, and reuses a subset of its concepts. The data used to initialize an enclave is cryptographically hashed, yielding the enclave's *measurement*. An enclave can invoke a secure inter-enclave messaging service to send a message to a privileged *attestation enclave* that can access the security monitor's attestation key, and produces the attestation signature.

5 Hardware Modifications

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



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

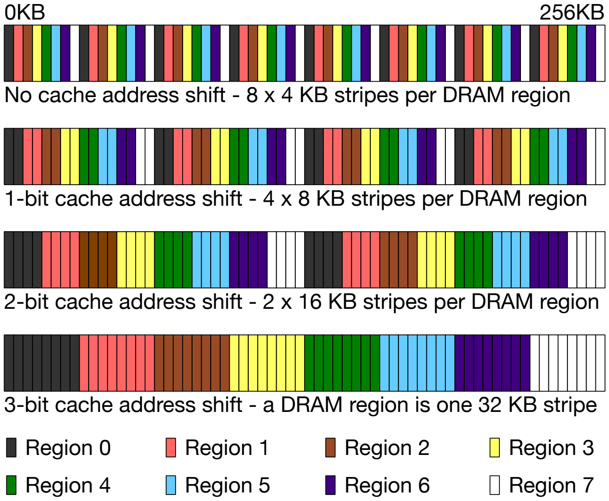


Figure 5: Address shift for contiguous DRAM regions

LLC 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.

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

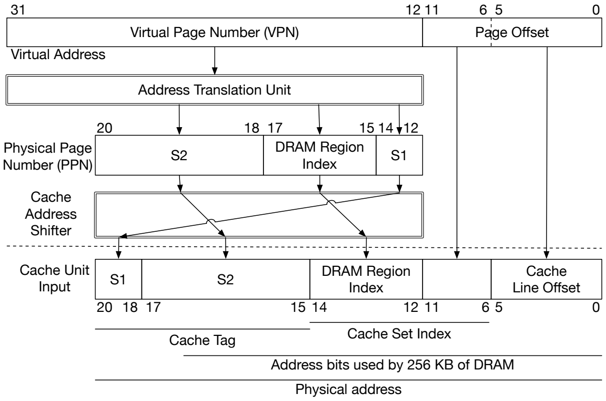


Figure 6: Cache address shifter, 3 bit PPN rotation

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 `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

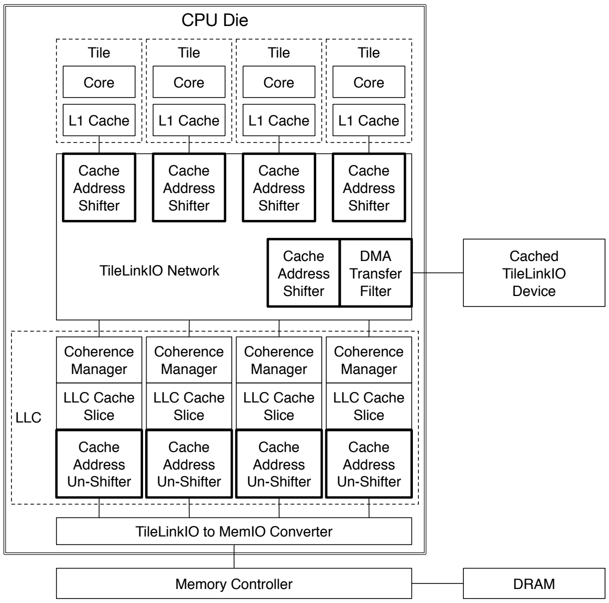


Figure 7: Sanctum's cache address shifter and DMA transfer filter logic in the context of a Rocket uncore

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 EVRANGE, 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 configures the circuit in Figure 9 to ensure that the page tables only point into allowable memory.

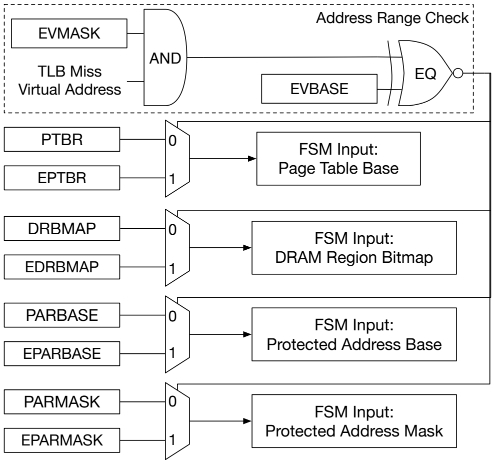

Address Range Check
EVMASK
AND
TLB Miss Virtual Address
EVBASE
EQ
PTBR
EPTBR
0
1
FSM Input: Page Table Base
DRBMAP
EDRBMAP
0
1
FSM Input: DRAM Region Bitmap
PARBASE
EPARBASE
0
1
FSM Input: Protected Address Base
PARMASK
EPARMASK
0
1
FSM Input: Protected Address Mask

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

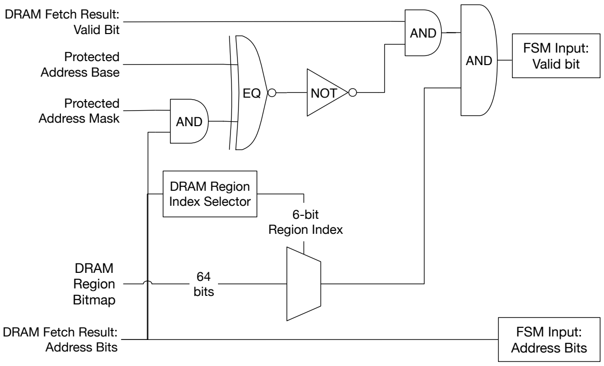


Figure 9: Hardware support for per-enclave page tables: check page table entries fetched by the page walker.

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 and its attestation key are stored in DRAM regions allocated to the OS. For security reasons, the OS must not be able to modify the monitor's code, or to read the attestation key. Sanctum extends the page table entry transformation described above to implement a Protected Address Range (PAR) for each set of page tables.

Each PAR is specified using a base register (parbase) register and a mask register (parmask) with the same semantics as the variable Memory Type Range registers (MTRRs) in the x86 architecture. The page table entry sanitization logic in Sanctum's hardware extensions checks if each page table entry points into the PAR by ANDing the entry's address with the PAR mask and comparing the result with the PAR base. If a page table entry is seen with a protected address, its valid bit is cleared, forcing a page fault.

The above transformation allows the security monitor to set up a memory range that cannot be accessed by other software, and which can be used to securely store the monitor's code and data. 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 above are replicated, 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.

6 Software Design

Sanctum's chain of trust, discussed in § 6.1, diverges significantly from SGX. We replace SGX's microcode with a software security monitor that runs at a higher privilege level than the hypervisor and the OS. On RISC-V, the security monitor runs at machine level. Our design only uses one privileged enclave, the signing enclave, which behaves similarly to SGX's Quoting Enclave.

6.1 Attestation Chain of Trust

Sanctum has three pieces of trusted software: the measurement root, which is burned in on-chip ROM, the security monitor (§ 6.2), which must be stored alongside the computer's firmware (usually in flash memory), and the signing enclave, which can be stored in any untrusted storage that the OS can access.

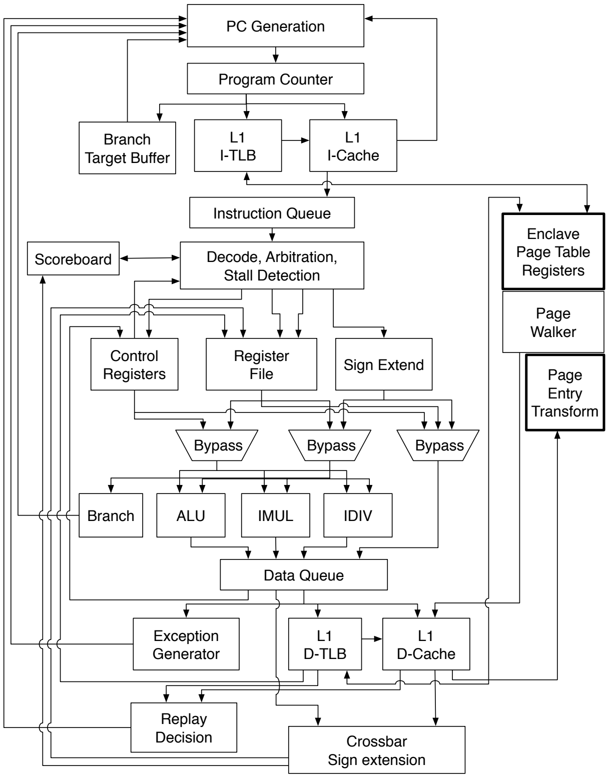


Figure 10: Sanctum's page entry transformation logic in the context of a Rocket core

We expect the trusted software to be amenable to rigorous analysis: our implementation of a security monitor for Sanctum is written with verification in mind, and has fewer than 5 kloc of C++, including a subset of the standard library and the cryptography for enclave attestation.

6.1.1 The 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 a monitor attestation key pair and certificate based on the monitor's hash, as shown in Figure 11.

The security monitor 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

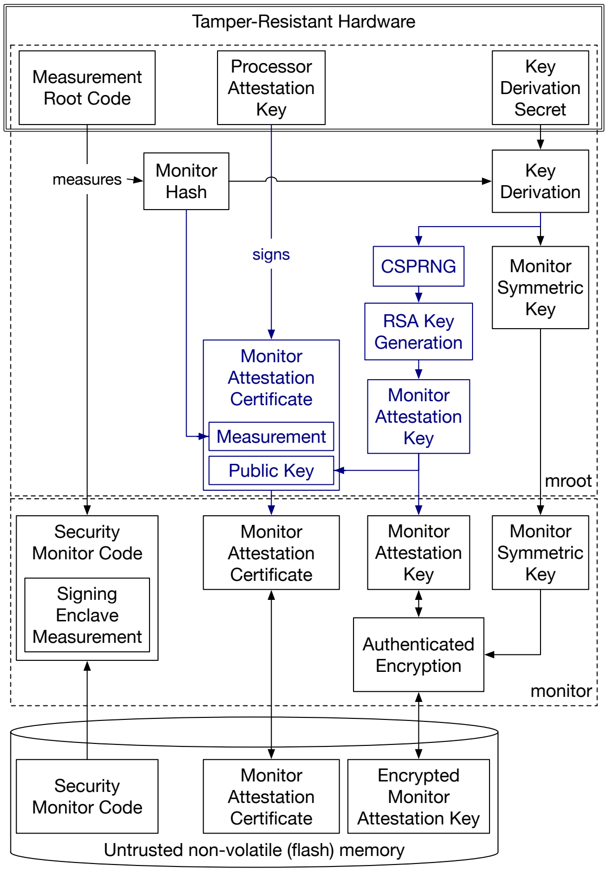


Figure 11: Sanctum's root of trust is a measurement root routine burned into the CPU's ROM. This code reads the security monitor from flash memory and generates an attestation key and certificate based on the monitor's hash. Asymmetric key operations, colored in blue, are only performed the first time a monitor is used on a computer.

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 a monitor attestation key pair, and produces a monitor attestation certificate by signing the monitor's public attestation key with the processor's private attestation key. The monitor attestation certificate includes the monitor's hash.

`mroot` generates a symmetric key for the security monitor so it may encrypt its private attestation key and store it in the computer's SPI flash memory chip. When writing the key, the monitor also sets the monitor attestation 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. This prevents a malicious security monitor from deriving a different monitor's symmetric key, or from generating a monitor attestation certificate that includes a different monitor's measurement hash.

The symmetric key generated for the monitor is similar in concept to the Seal Keys produced by SGX's key derivation process, as it is used to securely store a secret (the monitor's attestation key) in untrusted memory, in order to avoid an expensive process (asymmetric key attestation and signing). Sanctum's key derivation process is based on the monitor's measurement, so a given monitor is guaranteed to get the same key across power cycles. The cryptographic properties of the key derivation process guarantee that a malicious monitor cannot derive the symmetric key given to another monitor.

6.1.2 The Signing Enclave

In order to avoid timing attacks, the security monitor does not compute attestation signatures directly. Instead, the signing algorithm is executed inside a signing enclave, which is a security monitor module that executes in an enclave environment, so it is protected by the same isolation guarantees that any other Sanctum enclave enjoys.

The signing enclave receives the monitor's private attestation key via an API call. When the security monitor receives the call, it compares the calling enclave's measurement with the known measurement of the signing enclave. Upon a successful match, the monitor copies its attestation key into enclave memory using a data-independent sequence of memory accesses, such as `memcpy`. This way, the monitor's memory access pattern does not leak the private attestation key.

Sanctum's signing enclave authenticates another enclave on the computer and securely receives its attestation data using mailboxes (§ 6.2.5), a simplified version of SGX's local attestation (reporting) mechanism. The enclave's measurement and attestation data are wrapped into a software attestation signature that can be examined by a remote verifier. Figure 12 shows the chain of certificates and signatures in an instance of software attestation.

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 OS 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.

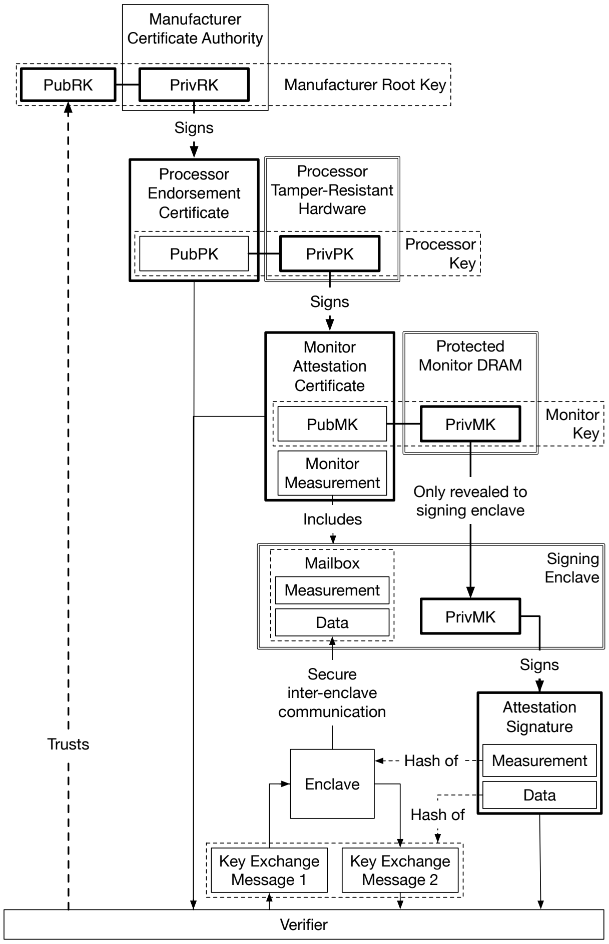


Figure 12: The certificate chain behind Sanctum's software attestation signatures

6.2.1 DRAM Regions

Figure 13 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 blocked region, it must flush all the cores' TLBs, to remove any stale entries for the region.

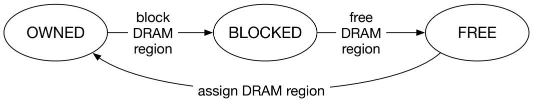


Figure 13: DRAM region allocation states and API calls

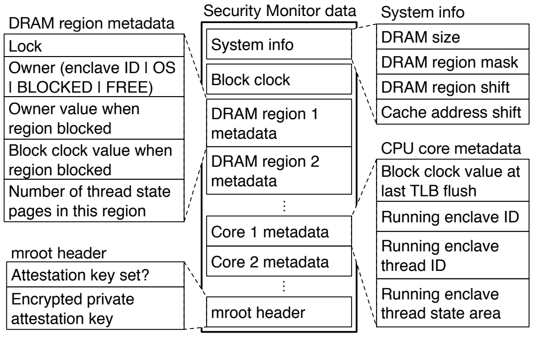


Figure 14: Security monitor data structures

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.

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.

6.2.2 Metadata Regions

Since the security monitor sits between the OS and enclave, and its APIs can be invoked by both sides, it is an easy target for timing attacks. We prevent these attacks with a straightforward policy that states the security monitor is never allowed to access enclave data, and is not allowed to make memory accesses that depend on the attestation key material. The rest of the data handled by the monitor is derived from the OS' actions, so it is already known to the OS.

A rather obvious consequence of the policy above is that after the security monitor boots the OS, it cannot perform any cryptographic operations that use keys. For example, the security monitor cannot compute an attestation signature directly, and defers that operation to a sign-

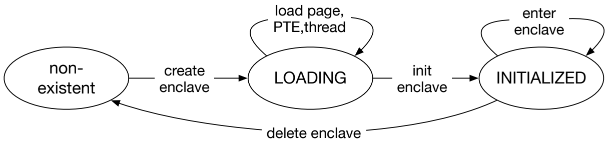


Figure 15: Enclave states and enclave management API calls

ing enclave (§ 6.1.2). While it is possible to implement some cryptographic primitives without performing data-dependent accesses, the security and correctness proofs behind these implementations are non-trivial. For this reason, Sanctum avoids depending on any such implementation.

A more subtle aspect of the access policy outlined above is that the metadata structures that the security monitor uses to operate enclaves cannot be stored in DRAM regions owned by enclaves, because that would give the OS an indirect method of accessing the LLC sets that map to enclave's DRAM regions, which could facilitate a cache timing attack.

For this reason, the security monitor requires the OS to set aside at least one DRAM region for enclave metadata before it can create enclaves. The OS has the ability to free up the metadata DRAM region, and regain the LLC sets associated with it, if it predicts that the computer's workload will not involve enclaves.

Each DRAM region that holds enclave metadata is managed independently from the other regions, at page granularity. The first few pages of each region contain a page map that tracks the enclave that tracks the usage of each metadata page, specifically the enclave that it is assigned to, and the data structure that it holds.

Each metadata region is like an EPC region in SGX, with the exception that our metadata regions only hold special pages, like Sanctum's equivalent of SGX's Secure Enclave Control Structure (SECS) and the Thread Control Structure (TCS). These structures will be described in the following sections.

The data structures used to store Sanctum's metadata can span multiple pages. When the OS allocates such a structure in a metadata region, it must point the monitor to a sequence of free pages that belong to the same DRAM region. All the pages needed to represent the structure are allocated and released in one API call.

6.2.3 Enclave Lifecycle

The lifecycle of a Sanctum enclave is very similar to that of its SGX counterparts, as shown in Figure 15.

The OS creates an enclave by issuing a *create enclave* call that creates the enclave metadata structure, which is Sanctum's equivalent of the SECS. The enclave metadata structure contains an array of mailboxes whose size is established at enclave creation time, so the number of pages required by the structure varies from enclave to enclave. § 6.2.5 describes the contents and use of mailboxes.

The *create enclave* API call initializes the enclave metadata fields shown in Figure 3, and places the enclave in the LOADING state. While the enclave is in this state, the OS sets up the enclave's initial state via monitor calls that assign DRAM regions to the enclave, create hardware threads and page table entries, and copy code and data into the enclave. 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.

Sanctum stores a measurement hash for each enclave in its metadata area, and updates the measurement to account for every operation performed on an enclave in the LOADING state. The policy described in § 6.2.2 does not apply to the secure hash operations used to update enclave's measurement, because all the data used to compute the hash is already known to the OS.

Enclave metadata is stored in a metadata region (§ 6.2.2), so it can only be accessed by the security monitor. 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 ensures 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 that of the last, constraining the OS to map an enclave's DRAM pages in monotonically increasing order.

6.2.4 Enclave Code Execution

Sanctum closely follows the threading model of SGX enclaves. Each CPU core that executes enclave code uses a thread metadata structure, which is our equivalent of SGX's TCS combined with SGX's State Save Area (SSA). Thread metadata structures are stored in a DRAM region dedicated to enclave metadata in order to prevent a malicious OS from mounting timing attacks against an enclave by causing AEXes on its threads. Figure 16 shows the lifecycle of a thread metadata structure.

The OS turns a sequence of free pages in a metadata

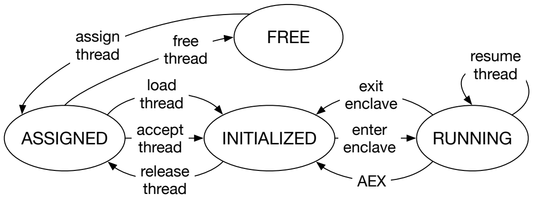


Figure 16: Enclave thread metadata structure states and thread-related API calls

region into an uninitialized thread structure via an *allocate thread* monitor call. During enclave loading, the OS uses a *load thread* monitor call to initialize the thread structure with data that contributes to the enclave's measurement. After an enclave is initialized, it can use an *accept thread* monitor call to initialize its thread structure.

The application hosting an enclave starts executing enclave code by issuing an *enclave enter* API call, which must specify an initialized thread structure. The monitor honors this call by configuring Sanctum's hardware extensions to allow access to the enclave's memory, and then by loading the program counter and stack pointer registers from the thread's metadata structure. The enclave's code can return control to the hosting application voluntarily, by issuing an *enclave exit* API call, which restores the application's PC and SP from the thread state area and sets the API call's return value to `ok`.

When performing an AEX, the security monitor atomically tests-and-sets the *AEX state valid* flag in the current thread's metadata. 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.

If an interrupt occurs while the enclave code is executing, the security monitor's exception handler performs an AEX, which 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.

6.2.5 Mailboxes

Sanctum's software attestation process relies on *mailboxes*, which are a simplified version of SGX's local attestation mechanism. We could not follow SGX's approach because it relies on key derivation and MAC algorithms, and our timing attack avoidance policy (§ 6.2.2) states that the security monitor is not allowed to perform cryptographic operations that use keys.

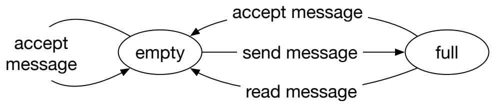


Figure 17: Mailbox states and security monitor API calls related to inter-enclave communication

Each enclave's metadata area contains an array of mailboxes, whose size is specified at enclave creation time, and covered by the enclave's measurement. Each mailbox goes through the lifecycle shown in Figure 17.

An enclave that wishes to receive a message in a mailbox, such as the signing enclave, declares its intent by performing an *accept message* monitor call. The API call is used to specify the mailbox that will receive the message, and the identity of the enclave that is expected to send the message.

The sending enclave, which is usually the enclave wishing to be authenticated, performs a *send message* call that specifies the identity of the receiving enclave, and a mailbox within that enclave. The monitor only delivers messages to mailboxes that expect them. At enclave initialization, the expected sender for all mailboxes is an invalid value (all zeros), so the enclave will not receive messages until it calls *accept message*.

When the receiving enclave is notified via an out-of-band mechanism that it has received a message, it issues a *read message* call to the monitor, which moves the message from the mailbox into the enclave's memory. If the API call succeeds, the receiving enclave is assured that the message was sent by the enclave whose identity was specified in the *accept message* call.

Enclave mailboxes are stored in metadata regions (§ 6.2.2), which cannot be accessed by any software other than the security monitor. This guarantees the privacy, integrity, and freshness of the messages sent via the mailbox system.

Our mailbox design has the downside that both the sending and receiving enclave need to be alive in DRAM in order to communicate. By comparison, SGX's local attestation can be done asynchronously. In return, mailboxes do not require any cryptographic operations, and have a much simpler correctness argument.

6.2.6 Multi-Core Concurrency

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 metadata structure also has a lock guarding it, which is acquired when the structure is accessed, but also while the metadata structure 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 Sanctum design allows the operating system to over-commit physical memory allocated to enclaves, by collaborating with an enclave to page some of its DRAM regions to disk. 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 (cf. Figure 13), 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 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 that is deferred to future work.

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 all 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 restart 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 [20], applied at page rather than cache line granularity, and with integrity verification. 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. While SSDs may alleviate some of this prohibitive overhead, and will be investigated in future work, currently 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.

7 Security Argument

Our security argument rests on two pillars: the enclave isolation enforced by the security monitor, and the guarantees behind the software attestation signature. This section outlines correctness arguments for each of these pillars.

Sanctum's isolation primitives protect enclaves from outside software that attempts to observe or interact with the enclave software via means outside the interface provided by the security monitor. We prevent direct attacks by ensuring that the memory owned by an enclave can only be accessed by that enclave's software. More subtle attacks are foiled by also isolating the structures used to access the enclave's memory, such as the enclave's page tables and the caches that hold enclave data.

7.1 Protection Against Direct Attacks

The correctness proof for Sanctum's DRAM isolation can be divided into two sub-proofs that cover the hardware and software sides of the system. First, we need to prove that the page walker modifications described in § 5.2 and § 5.3 behave according to their descriptions. Thanks to the small sizes of the circuits involved, this sub-proof can be accomplished by simulating the circuits for all logically distinct input cases. Second, we must prove that the security monitor configures Sanctum's extended page walker registers in a way that prevents direct attacks on enclaves. This part of the proof is significantly more complex, but it follows the same outline as the proof for SGX's memory access protection presented in [13].

The proof revolves around a main invariant stating that all TLB entries in every core are consistent with the programming model described in § 4. The invariant breaks down into three cases that match [13], after substituting DRAM regions for pages.

7.2 Protection Against Subtle Attacks

Sanctum also protects enclaves from software attacks that attempt to exploit side channels to obtain information indirectly. We focus on proving that Sanctum protects against the attacks mentioned in § 2, which target the page fault address and cache timing side-channels.

The proof that Sanctum foils page fault attacks is centered around the claims that each enclave's page fault handler and page tables and page fault handler are isolated from all other software entities on the computer. First, all the page faults inside an enclave's EVRANGE are reported to the enclave's fault handler, so the OS cannot observe the virtual addresses associated with the faults. Second, page table isolation implies that the OS cannot access an enclave's page tables and read the access and dirty bits to learn memory access patterns.

Page table isolation is a direct consequence of the claim that Sanctum correctly protects enclaves against direct attacks, which was covered above. Each enclave's page tables are stored in DRAM regions allocated to the enclave, so no software outside the enclave can access these page tables.

The proof behind Sanctum's cache isolation is straightforward but tedious, as there are many aspects involved. We start by peeling off the easier cases, and tackle the most difficult step of the proof at the end of the section. Our design assumes the presence of both per-core caches and a shared LLC, and each cache type requires a separate correctness argument. Per-core cache isolation is achieved simply by flushing per-core caches at every transition between enclave and non-enclave mode. To prove the correctness of LLC isolation, we first show that enclaves do not share LLC lines with outside software, and then we show that the OS cannot indirectly reach into an enclave's LLC lines via the security monitor.

Showing that enclaves do not share LLC lines with outside software can be accomplished by proving a stronger invariant that states at all times, any LLC line that can potentially cache a location in an enclave's memory cannot cache any location outside that enclave's memory. In steady state, this follows directly from the LLC isolation scheme in § 5.1, because the security monitor guarantees that each DRAM region is assigned to exactly one enclave or to the OS.

Last, we focus on the security monitor, because it is the only piece of software outside an enclave that can access the enclave's DRAM regions. In order to claim that an enclave's LLC lines are isolated from outside software, we must prove that the OS cannot use the security monitor's API to indirectly modify the state of the enclave's LLC lines. This proof is accomplished by considering each function exposed by the monitor API, as well as the monitor's hardware fault handler. The latter is considered to be under OS control because in a worst case scenario, a malicious OS could program peripherals to cause interrupts as needed to mount a cache timing attack.

7.3 Operating System Protection

Sanctum protects the operating system from direct attacks against malicious enclaves, but does not protect it against subtle attacks that take advantage of side-channels. Our design assumes that software developers will transition all sensitive software into enclaves, which are protected even if the OS is compromised. At the same time, a honest OS can potentially take advantage of Sanctum's DRAM regions to isolate mutually mistrusting processes.

Proving that a malicious enclave cannot attack the host computer's operating system is accomplished by first proving that the security monitor's APIs that start executing enclave code always place the core in unprivileged mode, and then proving that the enclave can only access OS memory using the OS-provided page tables. The first claim can be proven by inspecting the security monitor's code. The second claim follows from the correctness proof of the circuits in § 5.2 and § 5.3. Specifically, each enclave can only access memory either via its own page tables or the OS page tables, and the enclave's page tables cannot point into the DRAM regions owned by the OS.

These two claims effectively show that Sanctum enclaves run with the privileges of their host application. This parallels SGX, so all arguments about OS security in [13] apply to Sanctum as well. Specifically, malicious enclaves cannot DoS the OS, and can be contained using the mechanisms that currently guard against malicious user software.

7.4 Security Monitor Protection

The security monitor is in Sanctum's TCB, so the system's security depends on the monitor's ability to preserve its integrity and protect its secrets from attackers. The monitor does not use address translation, so it is not exposed to any attacks via page tables. The monitor also does not protect itself from cache timing attacks, and instead avoids making any memory accesses that would reveal sensitive information.

Proving that the monitor is protected from direct attacks from a malicious OS or enclave can be accomplished in a few steps. First, we invoke the proof that the circuits in § 5.2 and § 5.3, are correct. Second, we must prove that the security monitor configures Sanctum's extended page walker registers correctly. Third, we must prove that the DRAM regions that contain monitor code or data are always allocated to the OS.

Since the monitor is exposed to cache timing attacks from the OS, Sanctum's security guarantees rely on proofs that the attacks would not yield any information that the OS does not already have. Fortunately, most of the security monitor implementation consists of acknowledging and verifying the OS' resource allocation decisions. The main piece of private information held by the security monitor is the attestation key. We can be assured that the monitor does not leak this key, as long as we can prove that the monitor implementation only accesses the key when it is provided to the signing enclave (§ 6.1.2), that the key is provided via a data-independent memory copy operation, such as `memcpy`, and that the attestation key is only disclosed to the signing enclave.

7.5 The Security of Software Attestation

The security of Sanctum's software attestation scheme depends on the correctness of the measurement root and the security monitor. `mroot`'s sole purpose is to set up the attestation chain, so the attestation's security requires the correctness of the entire `mroot` code. The monitor's enclave measurement code also plays an essential role in the attestation process, because it establishes the identity of the attested enclaves, and is also used to distinguish between the signing enclave and other enclaves. Sanctum's attestation also relies on mailboxes, which are used to securely transmit attestation data from the attested enclave to the signing 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 [29]. Sanctum conveniently isolates concurrent workloads from one another, so we can examine its overhead via individual applications 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 additional hardware (§ 5) to generate a 4-core 64-bit RISC-V CPU. Using a cycle-accurate simulator for this machine, coupled with a custom Sanctum cache hierarchy simulator, we compute the program completion time for each benchmark, in cycles, for a variety of DRAM region allocations. The Rocket chip has an in-order single issue pipeline, and does not make forward progress on a TLB or cache miss, which allows us to accurately model a variety of DRAM region allocations efficiently.

We use a vanilla Rocket-Chip as an insecure baseline, against which we compute Sanctum's overheads. To produce the analysis in this section, we simulated over 250 billion instructions against the insecure baseline, and over 275 billion instructions against the Sanctum simulator. We compute the completion time for various enclave configurations from the simulator's detailed event log.

Our cache hierarchy follows Intel's Skylake [23] server models, with 32KB 8-way set associative L1 data and instruction caches, 256KB 8-way L2, and an 8MB 16-way LLC partitioned into core-local slices. Our cache hit and miss latencies follow the Skylake caches. We use a simple model for DRAM accesses and assume unlimited DRAM bandwidth, and a fixed cycle latency for each DRAM access. 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 integer subset of SPECINT 2006 [3] benchmarks (unmodified), specifically `perlbench`, `bzip2`, `gcc`, `mcf`, `gobmk`, `hmmer`, `sjeng`, `libquantum`, `h264ref`, `omnetpp`, and `astar_base`. This is a mix of memory and compute-bound long-running workloads with diverse locality.

We simulate a machine with 4GB of memory that is divided into 64 DRAM regions by Sanctum's cache address indexing scheme. Scheduling each benchmark on Core 0, we run it to completion, while the other cores are idling. While we do model its overheads, we choose not to simulate a complete Linux kernel, as doing so would invite a large space of parameters of additional complexity. To this end, we modify the RISC-V proto kernel [48] to provide the few services used by our benchmarks (such as filesystem io), while accounting for the expected overhead of each system call.

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 [29].

As illustrated at the gate level in Figures 8 and 9, we estimate Sanctum to add to Rocket hardware 500 (+0.78%) gates and 700 (+1.9%) flip-flops per core. Precisely, 50 gates for 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 exclude it from 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

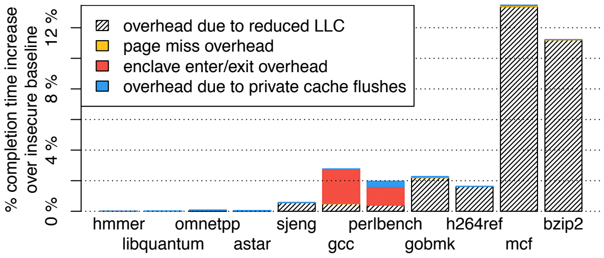


Figure 18: Detail of enclave overhead with a DRAM region allocation of 1/4 of LLC sets.

cache and the page walker. This logic is small and combinational, and 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 additional cycles of TLB miss latency are negligible, as quantified in Figure 18 for SPECINT benchmarks. All TLB-related overheads contribute less than 0.01% slowdown relative to completion time of the insecure baseline. This overhead is insignificant relative to the overheads of cache isolation: TLB misses are infrequent and relatively expensive, several additional cycles 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, relative to running code without Sanctum's enclave. 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 avoid leaking private information. Given an estimated cycle cost of each system call in a Sanctum enclave, and in an insecure baseline, we show the modest overheads due to enclave context switches in Figure 18. Moreover, 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. 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. When the OS assigns DRAM regions to an enclave, it confines it to a part of the LLC. An enclaved thread effectively runs on a machine with fewer LLC sets, impacting its performance. Note, however, that Sanctum does not partition private caches, so a thread can utilize its core's entire L1/L2 caches and TLB.

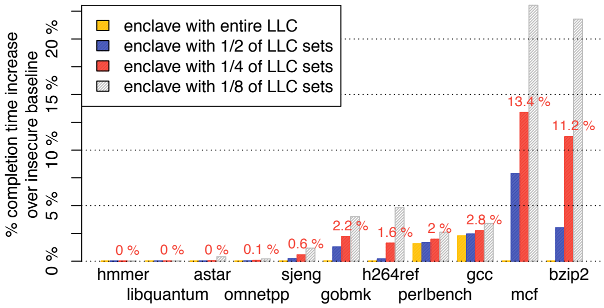


Figure 19: Overhead of enclaves of various size relative to an ideal insecure baseline.

Figure 19 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. 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. Figure 18 further underlines that the majority of Sanctum's enclave overheads stem from a reduction in available LLC sets.

We consider `mcf`'s 23% 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 uses a quarter of 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), enclaves add under 3% overhead to most memory-bound benchmarks (with the exception of `mcf` and `bzip`, which rely on a very large LLC), and do not encumber compute-bound workloads.

9 Conclusion

Sanctum shows that strong provable isolation of concurrent software modules can be achieved with low overhead. This approach provides strong security guarantees against an insidious software 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 away from plugging specific security holes to a principled approach to eliminating attack surfaces.

Acknowledgements: Funding for this research was partially provided by the National Science Foundation under contract number CNS-1413920.

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. Tech. rep., 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] ANATI, I., GUERON, S., JOHNSON, S. P., AND SCARLATA, V. R. Innovative technology for CPU based attestation and sealing. In *HASP* (2013).

[6] ANTHONY, S. Who actually develops linux? the answer might surprise you. http://www.extremetech.com/computing/175919-who-actually-develops-linux, 2014. [Online; accessed 27-April-2015].

[7] BANESCU, S. Cache timing attacks. [Online; accessed 26-January-2014].

[8] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems-CHES 2006*. Springer, 2006, pp. 201–215.

[9] BRUMLEY, B. B., AND TUVERI, N. Remote timing attacks are still practical. In *Computer Security–ESORICS*. Springer, 2011.

[10] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. *Computer Networks* (2005).

[11] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Asia-Pacific Workshop on Systems* (2011), ACM.

[12] CHHABRA, S., ROGERS, B., SOLIHIN, Y., AND PRVULOVIC, M. SecureME: a hardware-software approach to full system security. In *international conference on Supercomputing (ICS)* (2011), ACM.

[13] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, Feb 2016.

[14] DAVENPORT, S. SGX: the good, the bad and the downright ugly. *Virus Bulletin* (2014).

[15] DOMNITSER, L., JALEEL, A., LOEW, J., ABU-GHAZALEH, N., AND PONOMAREV, D. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *Transactions on Architecture and Code Optimization (TACO)* (2012).

[16] DUFLOT, L., ETIEMBLE, D., AND GRUMELARD, O. Using CPU system management mode to circumvent operating system security functions. *CanSecWest/core06* (2006).

[17] DUNN, A., HOFMANN, O., WATERS, B., AND WITCHEL, E. Cloaking malware with the trusted platform module. In *USENIX Security Symposium* (2011).

[18] EMBLETON, S., SPARKS, S., AND ZOU, C. C. SMM rootkit: a new breed of os independent malware. *Security and Communication Networks* (2010).

[19] EVTYUSHKIN, D., ELWELL, J., OZSOY, M., PONOMAREV, D., ABU GHAZALEH, N., AND RILEY, R. Iso-X: A flexible architecture for hardware-managed isolated execution. In *Microarchitecture (MICRO)* (2014), IEEE.

[20] FLETCHER, C. W., DIJK, M. V., AND DEVADAS, S. A secure processor architecture for encrypted computation on untrusted programs. In *Workshop on Scalable Trusted Computing* (2012), ACM.

[21] GOLDREICH, O. Towards a theory of software protection and simulation by oblivious RAMs. In *Theory of Computing* (1987), ACM.

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

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

[24] KESSLER, R. E., AND HILL, M. D. Page placement algorithms for large real-indexed caches. *Transactions on Computer Systems (TOCS)* (1992).

[25] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA* (2014), IEEE Press.

[26] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal verification of an OS kernel. In *SIGOPS symposium on Operating systems principles* (2009), ACM.

[27] KOCHER, P. C. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Advances in Cryptology (CRYPTO)* (1996), Springer.

[28] KONG, J., ACIICMEZ, O., SEIFERT, J.-P., AND ZHOU, H. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *workshop on Computer security architectures* (2008), ACM.

[29] LEE, Y., WATERMAN, A., AVIZIENIS, R., COOK, H., SUN, C., STOJANOVIC, V., AND ASANOVIC, K. A 45nm 1.3 ghz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *European Solid State Circuits Conference (ESSCIRC)* (2014), IEEE.

[30] LIE, D., THEKKATH, C., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. Architectural support for copy and tamper resistant software. *SIGPLAN Notices* (2000).

[31] LIN, J., LU, Q., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA* (2008), IEEE.

[32] LIU, C., HARRIS, A., MAAS, M., HICKS, M., TIWARI, M., AND SHI, E. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS* (2015).

[33] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA* (Mar 2016).

[34] LIU, F., AND LEE, R. B. Random fill cache architecture. In *Microarchitecture (MICRO)* (2014), IEEE.

[35] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Security and Privacy* (2015), IEEE.

[36] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. *HASP* (2013).

[37] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox – practical cache attacks in javascript. *arXiv preprint arXiv:1502.07373* (2015).

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

[39] RUTKOWSKA, J., AND WOJTCZUK, R. Preventing and detecting xen hypervisor subversions. *Blackhat Briefings USA* (2008).

[40] SANCHEZ, D., AND KOZYRAKIS, C. The ZCache: Decoupling ways and associativity. In *Microarchitecture (MICRO)* (2010), IEEE.

[41] SANCHEZ, D., AND KOZYRAKIS, C. Vantage: scalable and efficient fine-grain cache partitioning. In *SIGARCH Computer Architecture News* (2011), ACM.

[42] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. `http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, Mar 2015. [Online; accessed 9-March-2015].

[43] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: An extremely simple oblivious ram protocol. In *SIGSAC Computer & communications security* (2013), ACM.

[44] SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *international conference on Supercomputing (ICS)* (2003), ACM.

[45] TAYLOR, G., DAVIES, P., AND FARMWALD, M. The TLB slice - a low-cost high-speed address translation mechanism. *SIGARCH Computer Architecture News* (1990).

[46] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *International Symposium on Computer Architecture (ISCA)* (2007).

[47] WATERMAN, A., LEE, Y., AVIZIENIS, R., PATTERSON, D. A., AND ASANOVIC, K. 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.

[48] WATERMAN, A., LEE, Y., AND CELIO, CHRISTOPHER, E. A. RISC-V proxy kernel and boot loader. Tech. rep., EECS Department, University of California, Berkeley, May 2015.

[49] WATERMAN, A., LEE, Y., PATTERSON, D. A., AND ASANOVIC, K. 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.

[50] WECHEROWSKI, F. A real SMM rootkit: Reversing and hooking BIOS SMI handlers. *Phrack Magazine* (2009).

[51] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking intel trusted execution technology. *Black Hat DC* (2009).

[52] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking SMM memory via intel CPU cache poisoning. *Invisible Things Lab* (2009).

[53] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking intel TXT via SINIT code execution hijacking, 2011.

[54] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Another way to circumvent intel® trusted execution technology. *Invisible Things Lab* (2009).

[55] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Oakland* (May 2015), IEEE.

[56] YAROM, Y., AND FALKNER, K. E. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive* (2013).

[57] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy* (2009), IEEE.