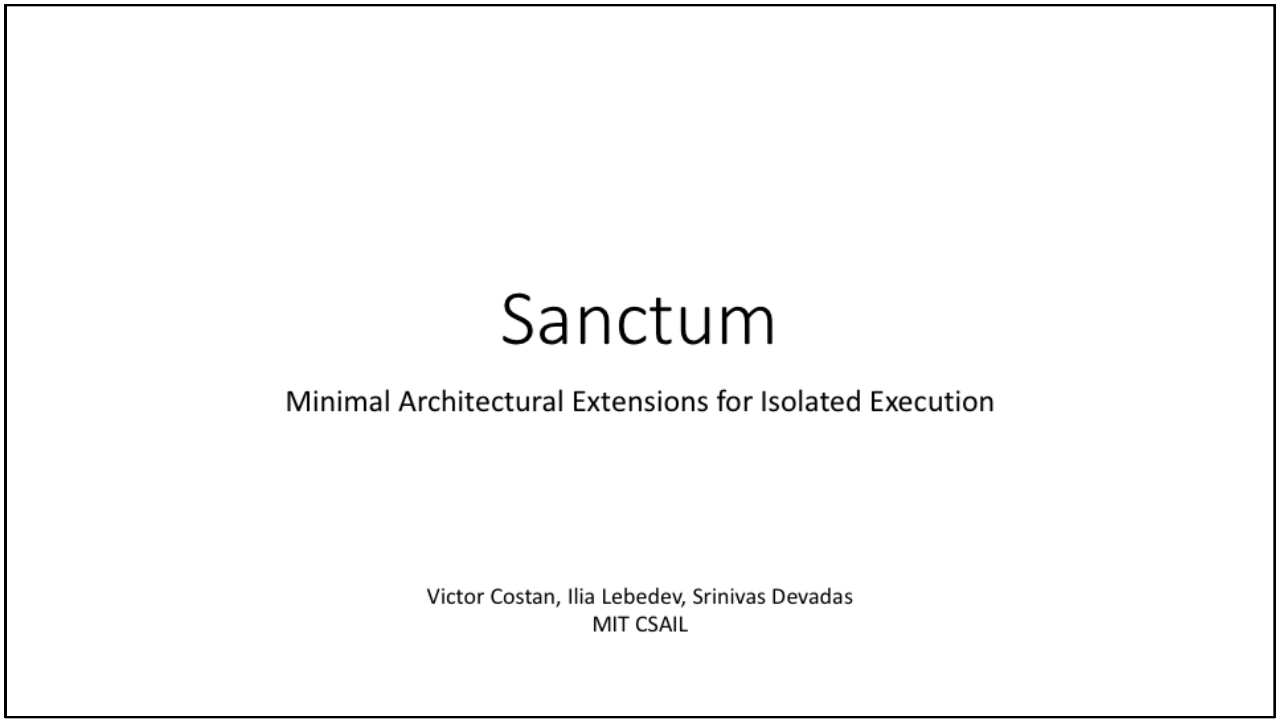


Sanctum

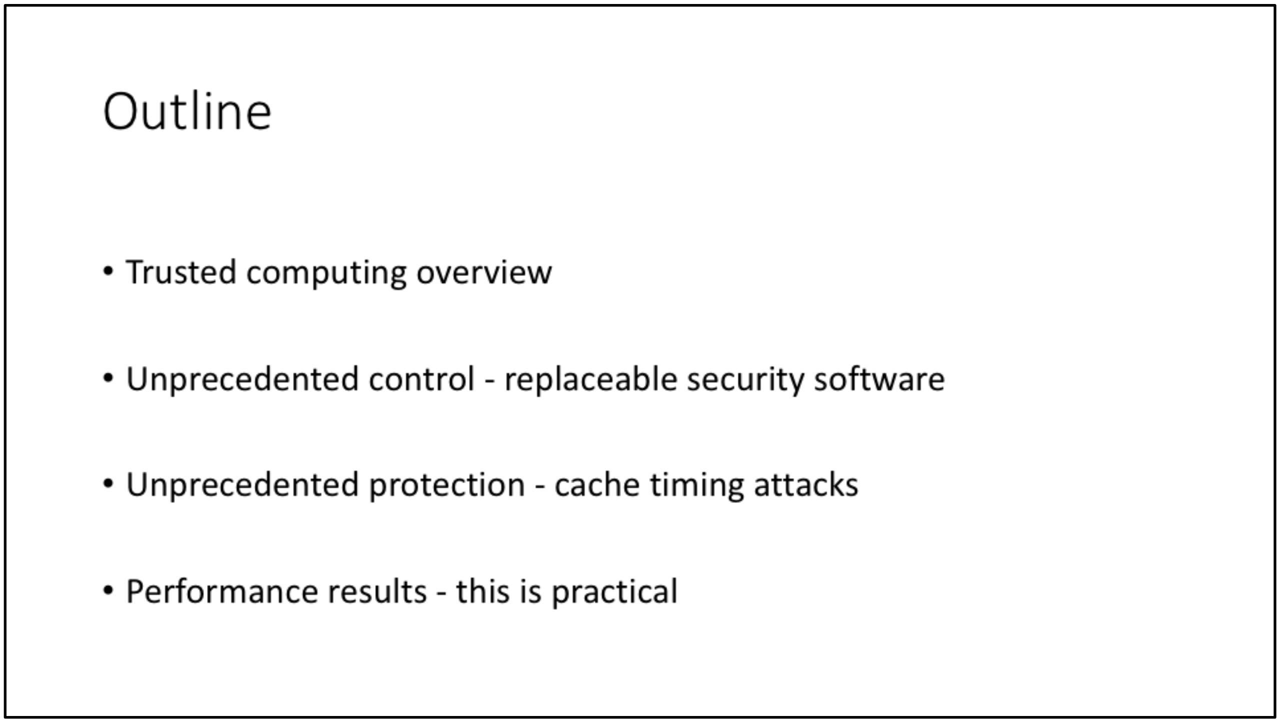
Minimal Architectural Extensions for Isolated Execution

Victor Costan, Ilia Lebedev, Srinivas Devadas
MIT CSAIL

Hi, everyone! Thank you very much for coming!

My name is Victor Costan, and I am here to talk about Sanctum.

This work was done at MIT’s Computer Science and AI Laboratory, in collaboration with Ilia Lebedev and Prof. Srini Devadas.



Outline

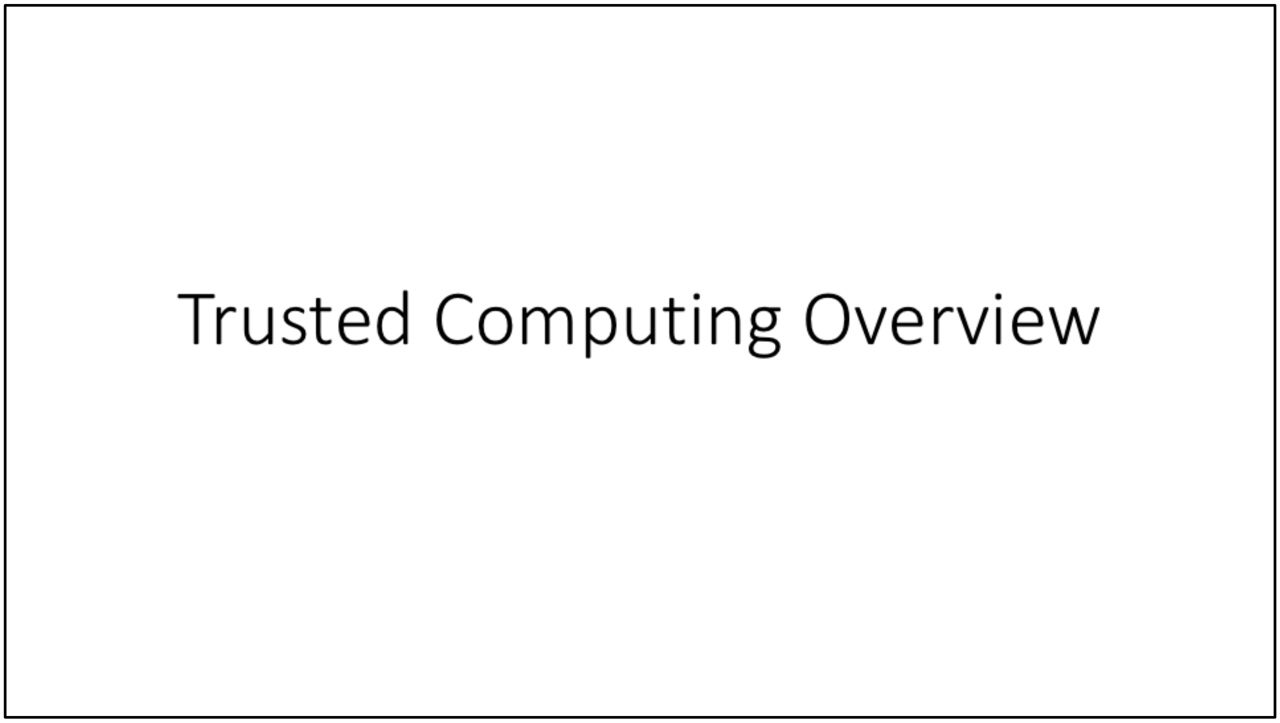
- Trusted computing overview
- Unprecedented control - replaceable security software
- Unprecedented protection - cache timing attacks
- Performance results - this is practical

Sanctum’s goal is trusted computing. In this respect, our work belongs to the same family as TPM, TXT, SGX, and TrustZone. However, Sanctum gives you unprecedented protection and unprecedented control over your computer.

Most of Sanctum’s security logic is in software, not in hardware or microcode. Our software can be inspected by the computer’s owner, and is amenable to formal verification. Most of this software can even be replaced by the computer’s owner. This is an unprecedented level of control.

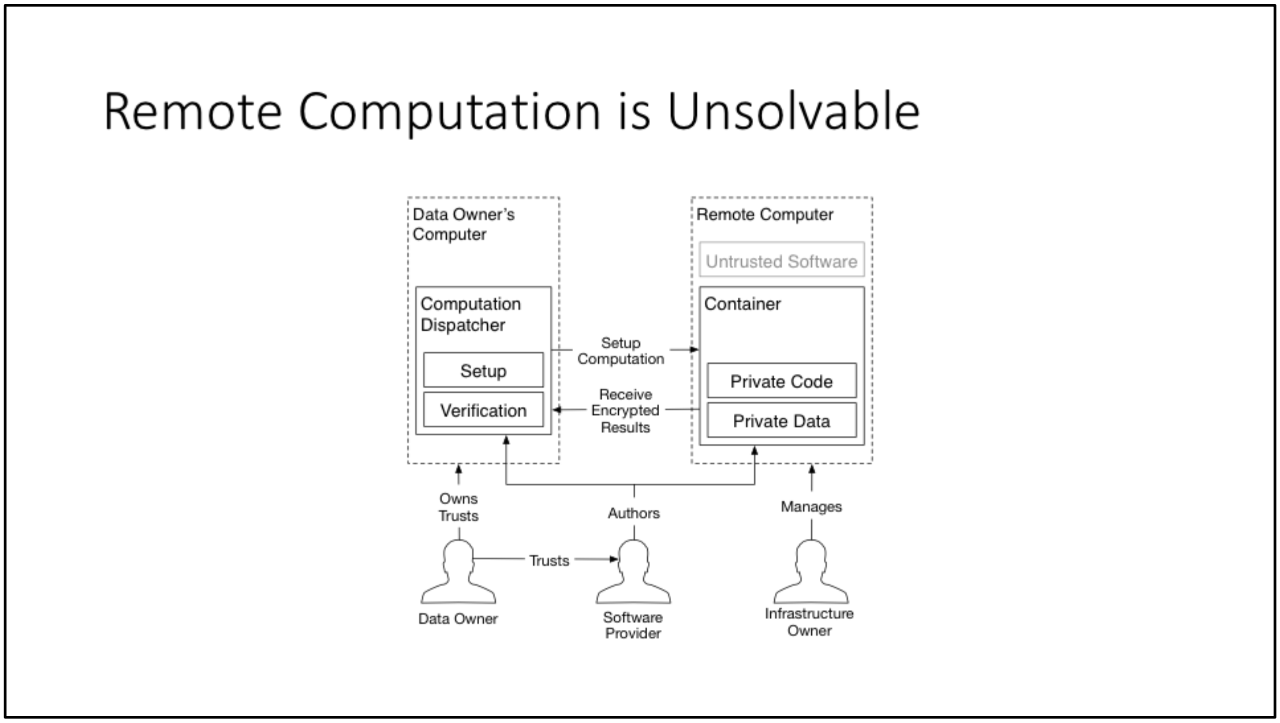
Sanctum’s hardware extensions can be applied to any reasonably modern RISC processor core. In combination with Sanctum’s software, the hardware extensions protect against any practical software attacks, including cache timing attacks. This is an unprecedented level of protection.

Our prototype targets the RISC-V architecture, and shows that Sanctum incurs acceptable performance overheads.



Trusted Computing Overview

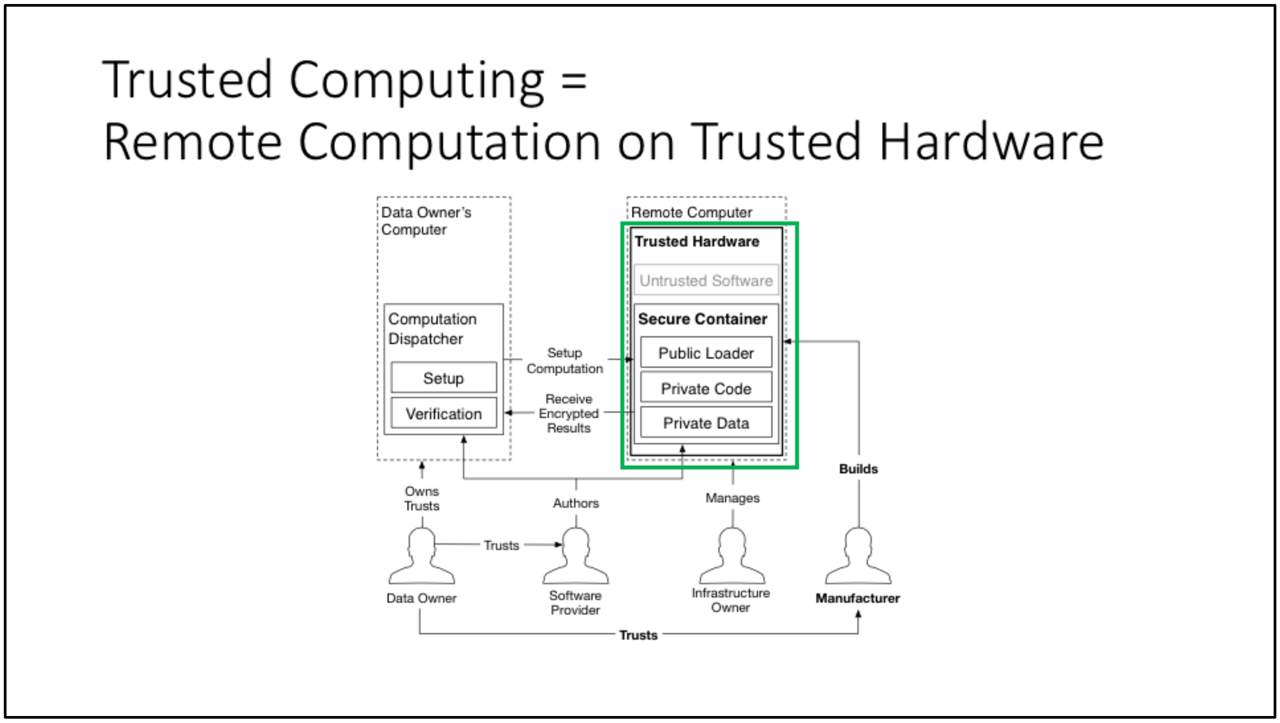
To get things started, let me give you a quick overview of trusted computing, which is the problem that we’re trying to solve.



Remote Computation is Unsolvable

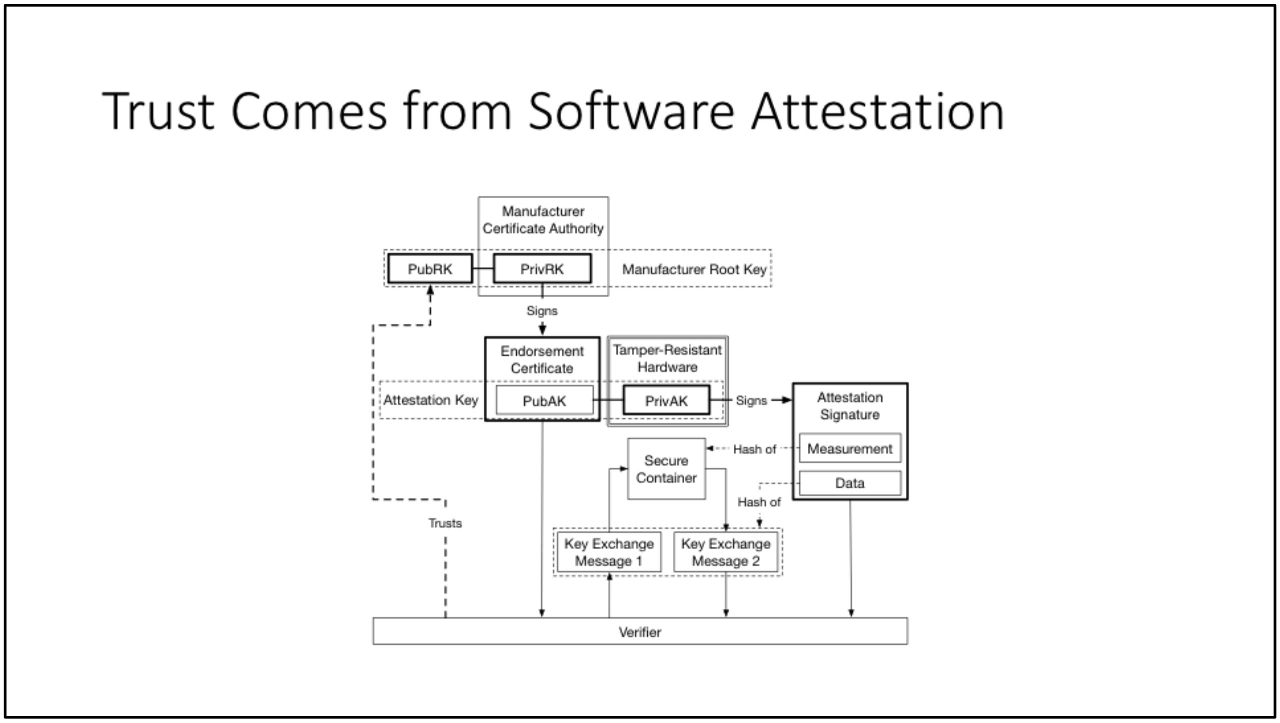
So there's this dream, called secure remote computation. In this dream, you can package your code and data into a bundle, send that bundle over to a remote computer, have the computer run your software, and get the results back. The thing is, you don't own that remote computer, so you rely on magic to protect your code and data from the computer's owner, and from the other software that might be running on that computer.

Unfortunately, this is just a dream. The closest thing we've got is fully homomorphic encryption, and that's nowhere nearly practical.



Trusted Computing = Remote Computation on Trusted Hardware

Fortunately, we can approximate that dream, if we're willing to trust a piece of hardware on the remote computer. The trusted hardware establishes a secure container on the remote computer, which protects our computation and data from untrusted third parties. So, our computation is safe, as long as we only send it to trusted hardware.



Trust Comes from Software Attestation

Manufacturer Certificate Authority

PubRK

PrivRK

Manufacturer Root Key

Signs

Endorsement Certificate

Tamper-Resistant Hardware

Attestation Key

PubAK

PrivAK

Signs

Attestation Signature

Measurement

Data

Hash of

Secure Container

Hash of

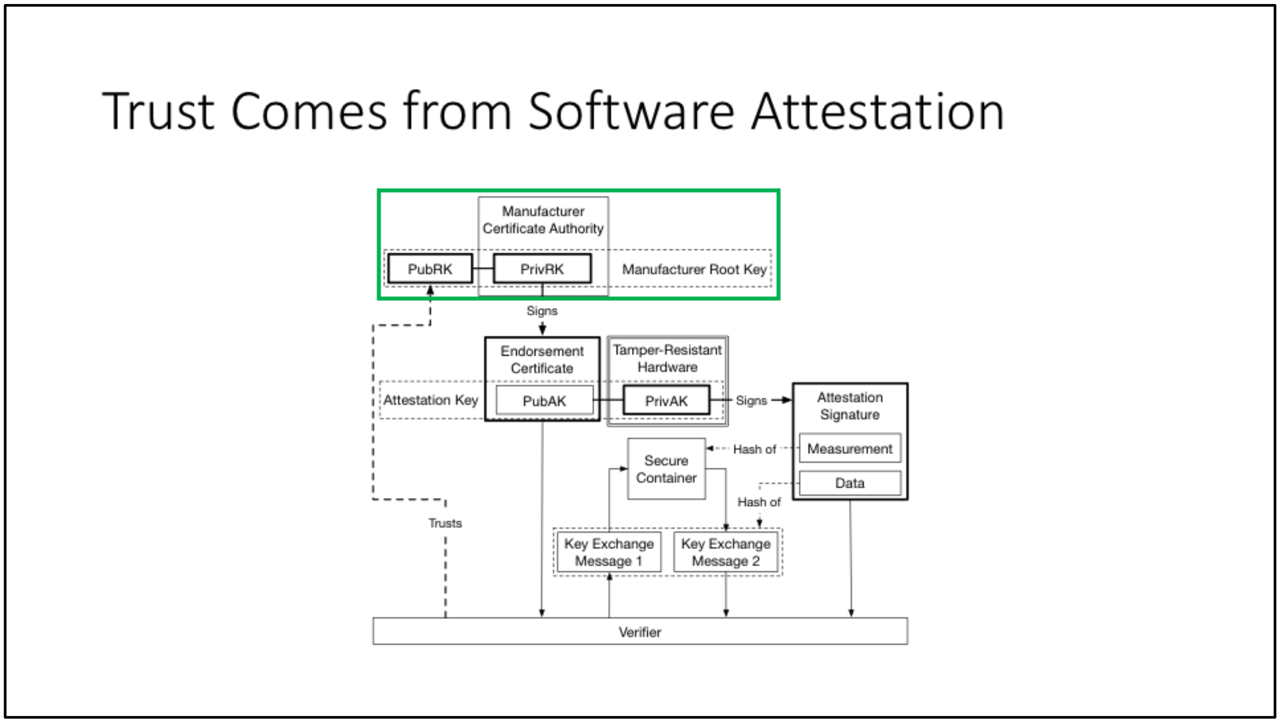
Trusts

Key Exchange Message 1

Key Exchange Message 2

Verifier

So, we have to make sure that we only talk to trusted hardware. We can use software attestation for this.



Trust Comes from Software Attestation

Manufacturer Certificate Authority

PubRK

PrivRK

Manufacturer Root Key

Signs

Endorsement Certificate

Tamper-Resistant Hardware

Attestation Key

PubAK

PrivAK

Signs

Attestation Signature

Measurement

Hash of

Secure Container

Data

Hash of

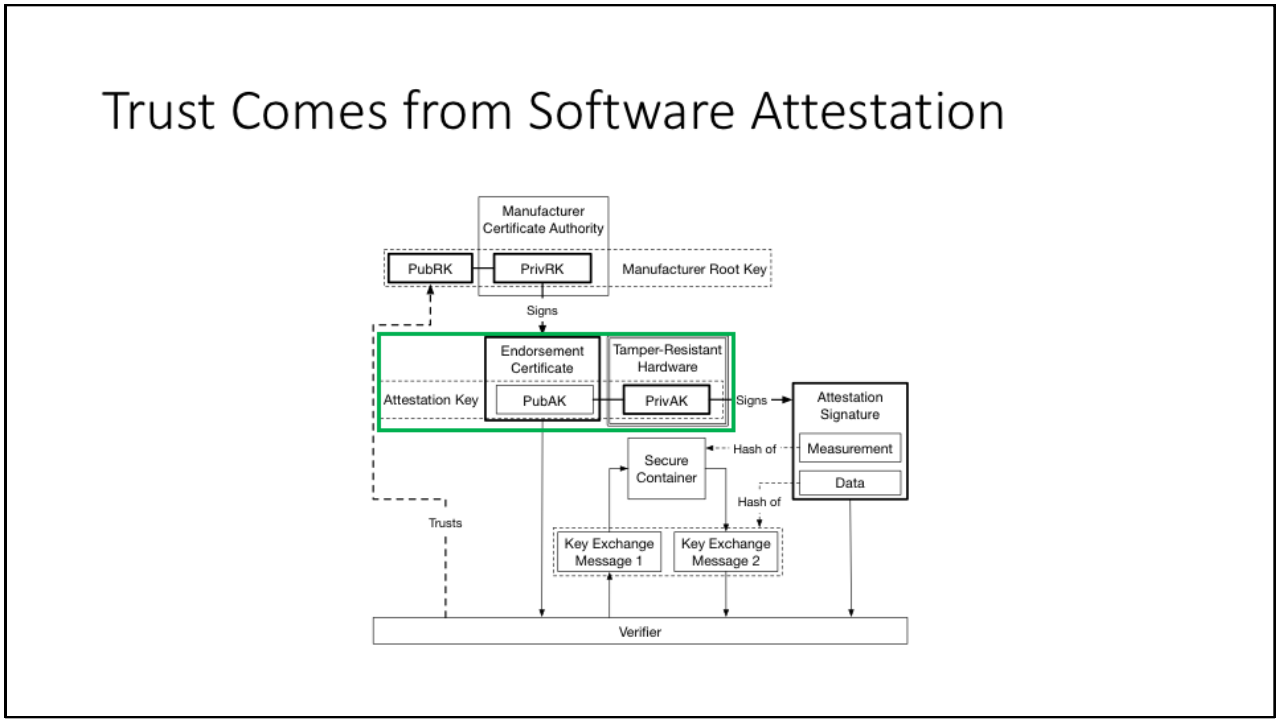
Trusts

Key Exchange Message 1

Key Exchange Message 2

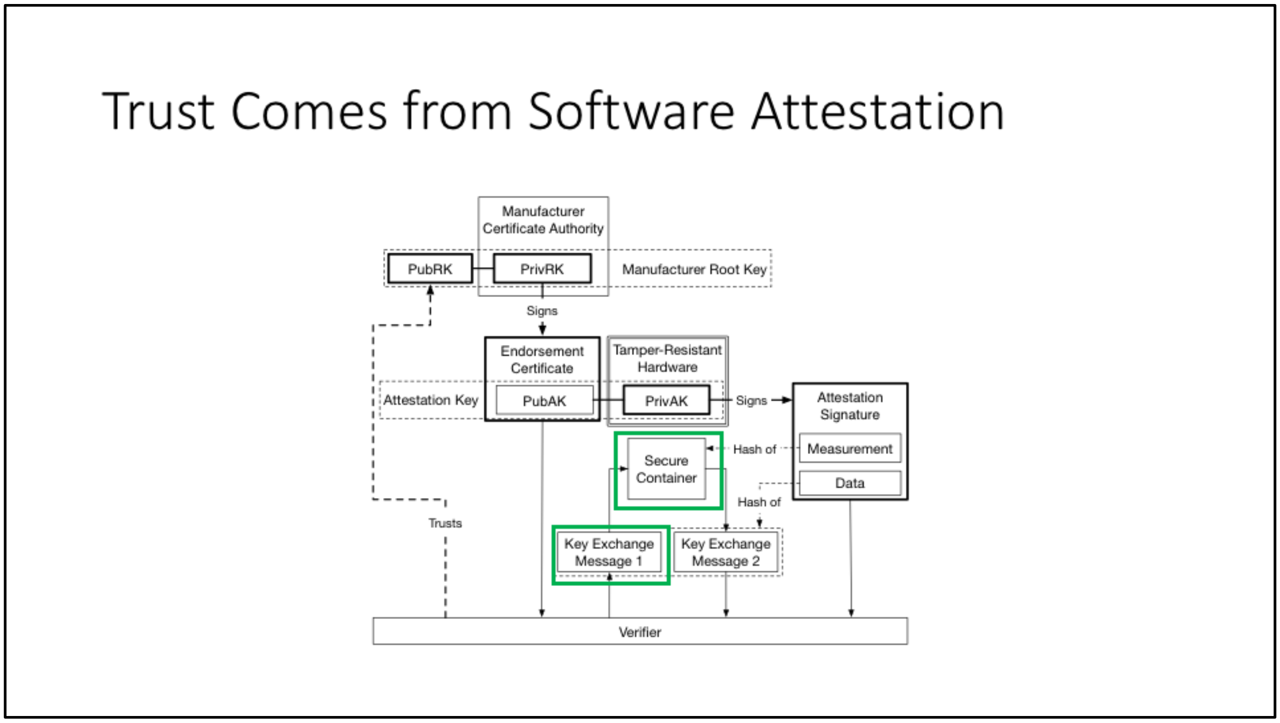
Verifier

Very quickly, software attestation means that there is a hardware manufacturer that we trust, and the manufacturer acts as a certificate authority in a PKI.



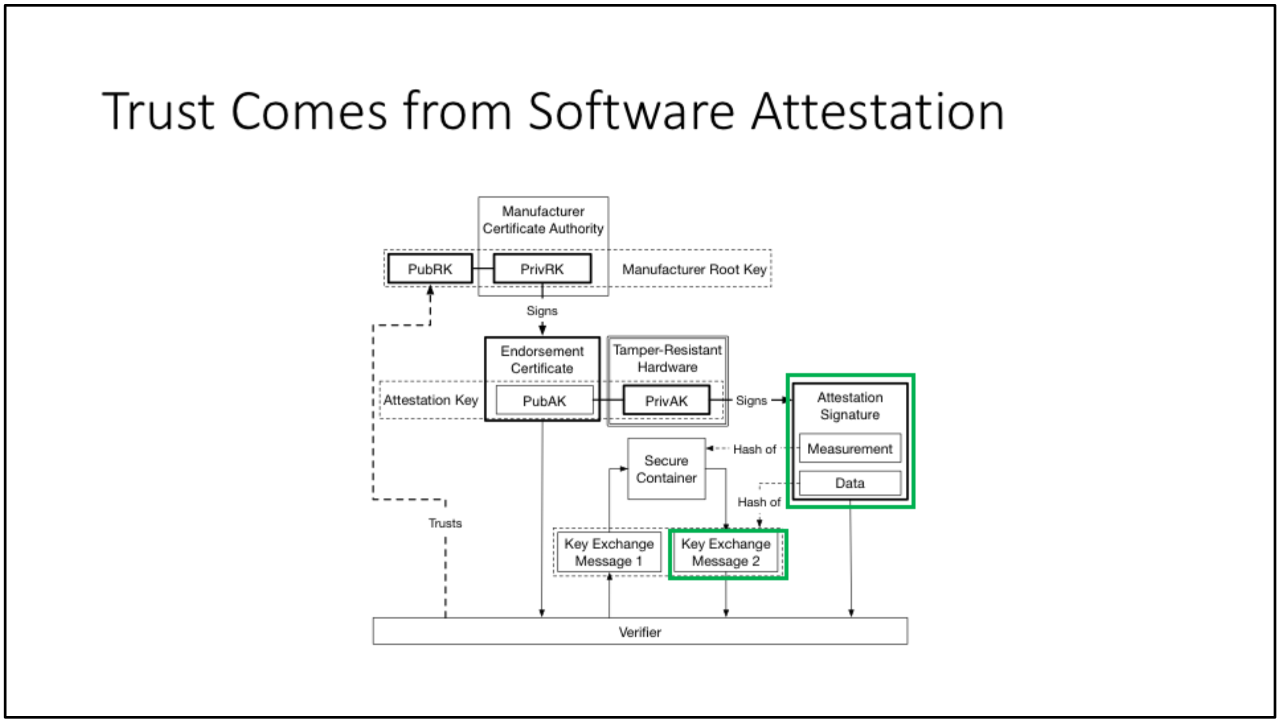
Trust Comes from Software Attestation

Each piece of trusted hardware has an attestation key, and the manufacturer issues an endorsement certificate for the public attestation key. When we see that certificate, we know that we can trust signatures issued by the device's private attestation key, which is stored in secure hardware.



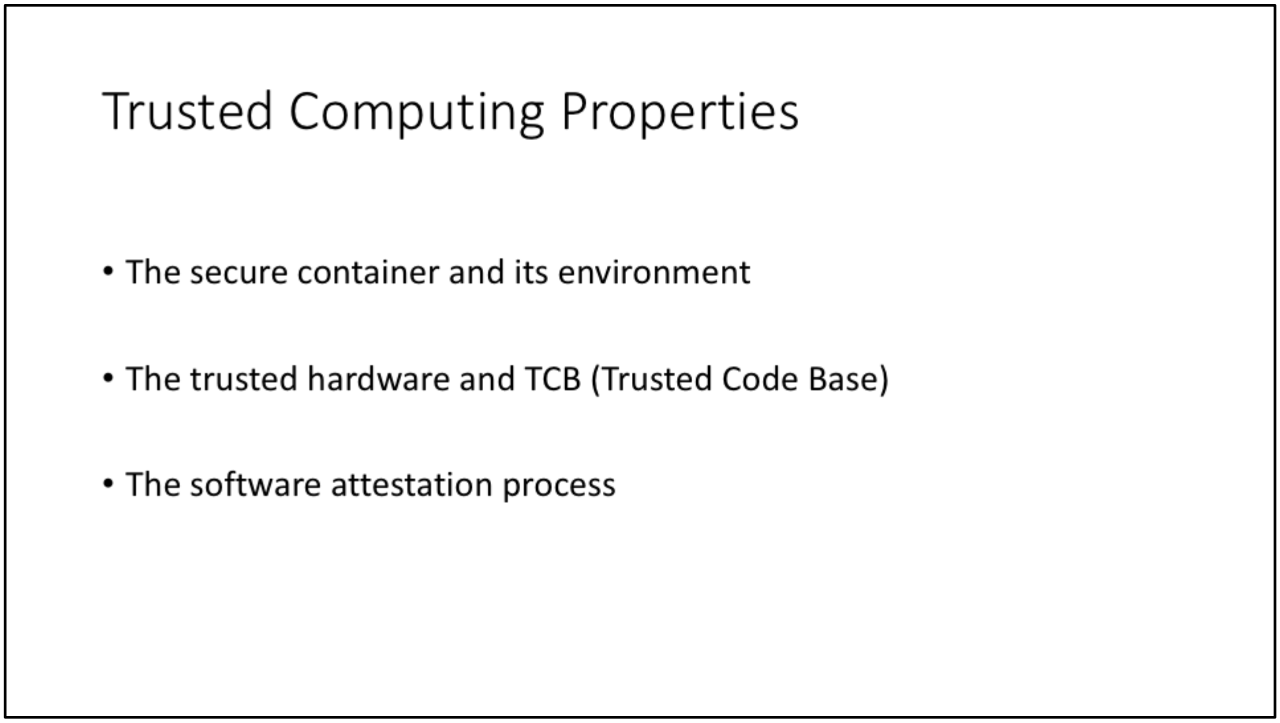
Trust Comes from Software Attestation

Now, when we want to send our code over to the remote computer, we will first ask the trusted hardware to create a secure container for us, and we will send a challenge message to that secure container.



Trust Comes from Software Attestation

Our loader code in the container will receive the challenge message and compute a response message, and then it will ask the trusted hardware to produce an attestation signature. The attestation signature covers the message produced by our container, as well as a measurement of the secure container's initial state. When we validate the attestation signature, we are assured that we are talking to a secure container that is initialized according to our instructions, and hosted by hardware we trust.



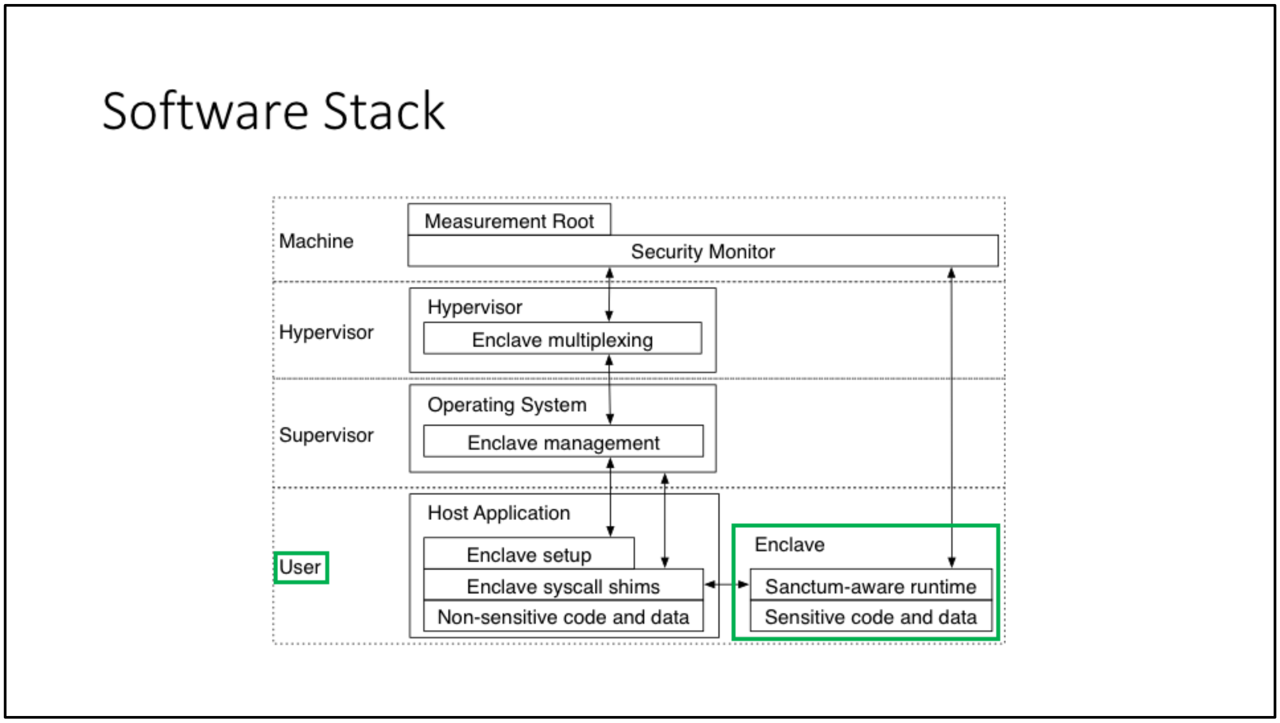
Trusted Computing Properties

- The secure container and its environment
- The trusted hardware and TCB (Trusted Code Base)
- The software attestation process

What I’ve said so far applies to all trusted computing solutions. These solutions differ by the amount of software that goes into a secure container, by the security guarantees offered to the container, by the trusted hardware and software needed to enforce the system’s security, and by the details of their software attestation process.

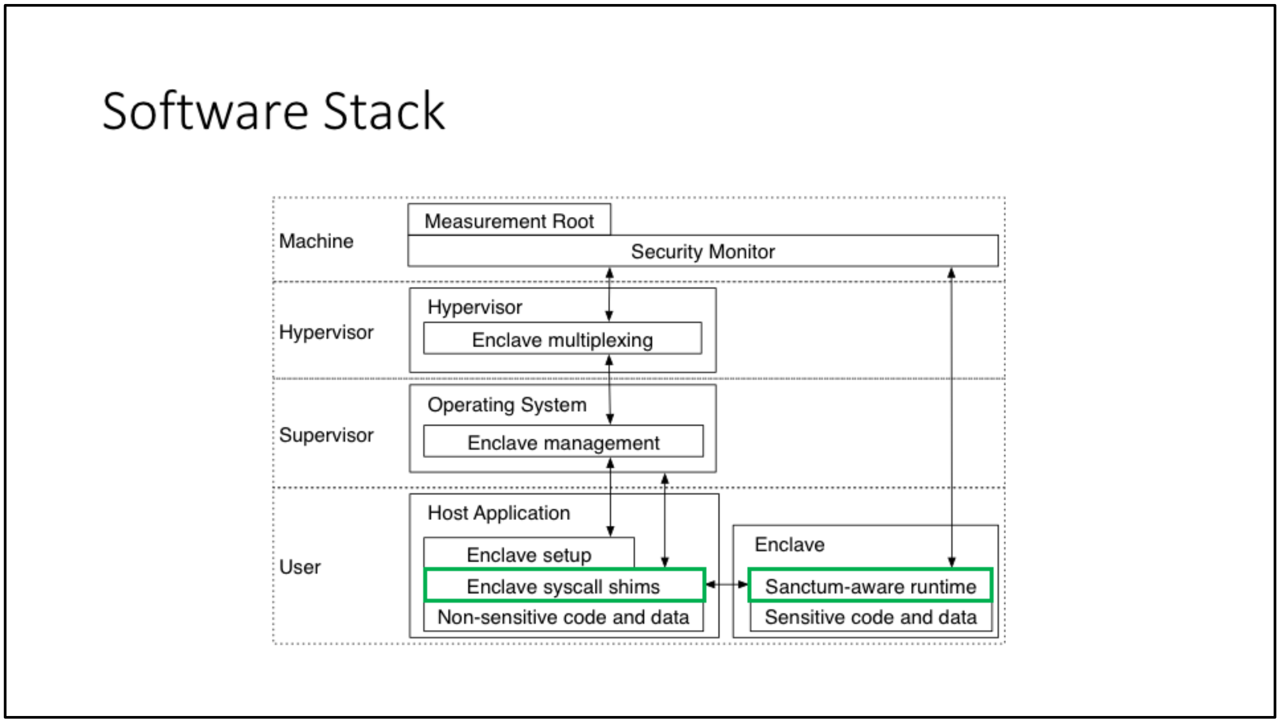
The rest of this talk will hopefully clarify where Sanctum stands on all these aspects.

Sanctum’s Software Stack



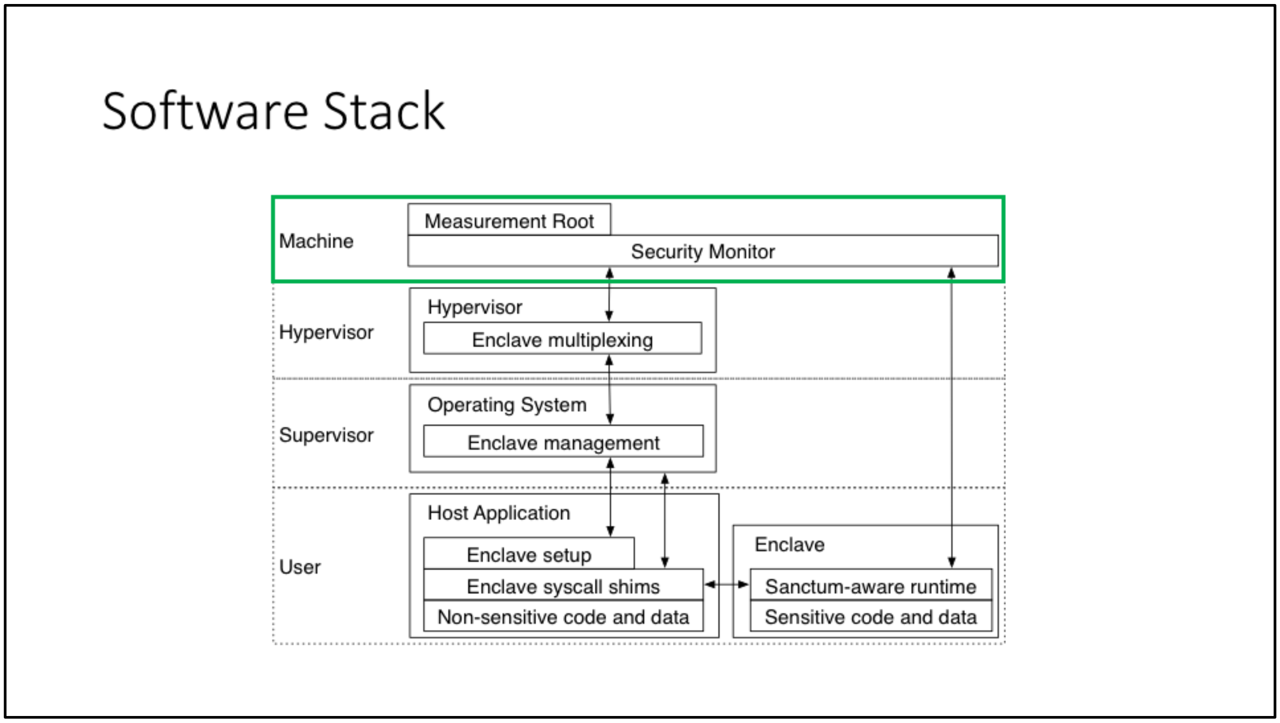
Software Stack

Our secure containers are called enclaves, and they are conceptually extensions of application processes.
Enclaves run at the lowest possible privilege level -- this is known as ring 3 in x86, or user mode anywhere else. This means enclaves cannot compromise the host computer’s OS or hypervisor. So we don’t need to worry about restricting enclaves – the same mechanisms used to police user processes will work for enclaves.



Software Stack

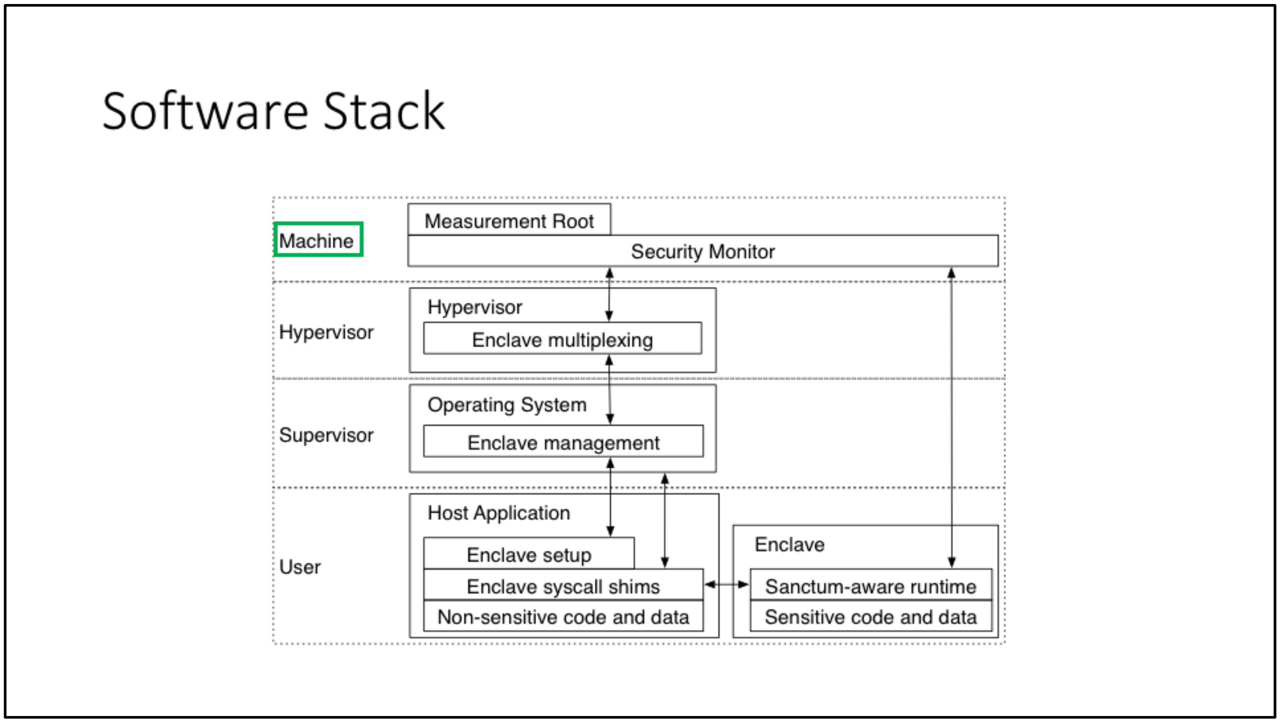
Enclaves can access the memory space of their host applications, but they cannot perform syscalls directly. This is because we don't trust the OS kernel to not damage the enclave's execution context before returning from the syscall. To work around this, the enclave must rely on its host application to proxy syscalls to the OS. The syscall proxying requires code in the enclave, and in the host application. We expect this code to become a part of the runtime library. For example, if your enclave software uses libc, you'd simply use an enclave-aware libc that proxies all the syscalls it makes.



Software Stack

Machine
Measurement Root
Security Monitor
Hypervisor
Hypervisor
Enclave multiplexing
Supervisor
Operating System
Enclave management
User
Host Application
Enclave setup
Enclave syscall shims
Non-sensitive code and data
Enclave
Sanctum-aware runtime
Sensitive code and data

Our enclaves aren't new. SGX's enclaves operate in a very similar way. Something that _is_ new, and that I'm really proud of – the vast majority our security logic is expressed in software. This isn't firmware or microcode. It's normal software that executes using the processor's standard execution facilities. It's software that the computer's owner can inspect and analyze.



Software Stack

Machine

Measurement Root

Security Monitor

Hypervisor

Hypervisor

Enclave multiplexing

Supervisor

Operating System

Enclave management

User

Host Application

Enclave setup

Enclave syscall shims

Non-sensitive code and data

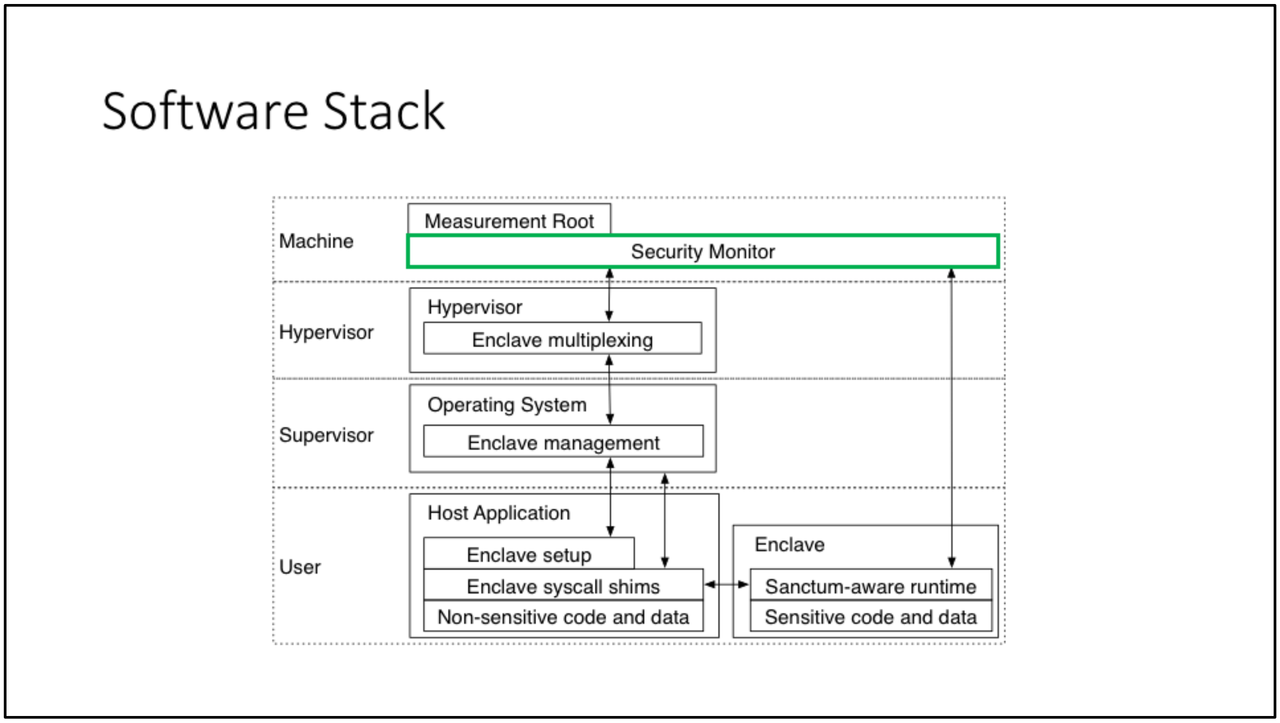
Enclave

Sanctum-aware runtime

Sensitive code and data

The downside of not relying on firmware or microcode is that our software is not automatically isolated from rest of the code running on the machine, and we may be running alongside a malicious hypervisor or operating system. In order to be able to provide any sort of meaningful security guarantees, we assume that our software runs at a higher privilege level than anything else. The RISC-V architecture conveniently includes a machine level, and our prototype takes advantage of the existence of this machine level.

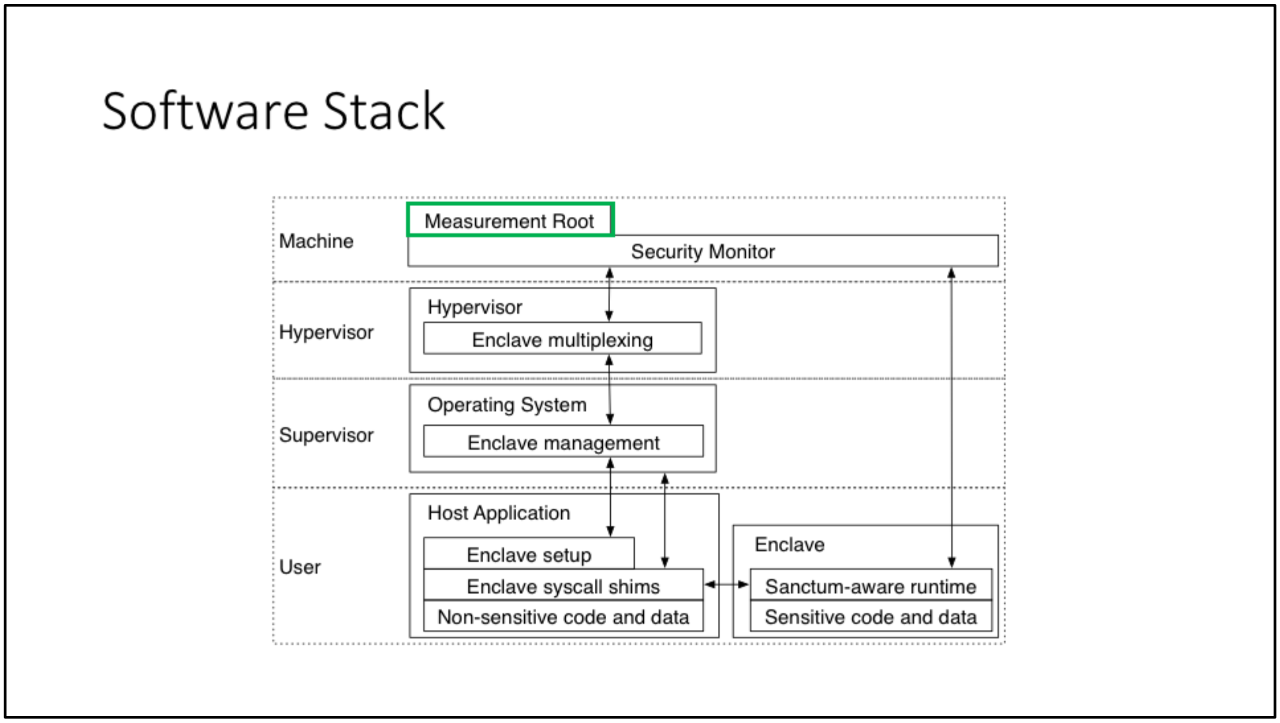
We run two pieces of software at the machine level.



Software Stack

Machine
Measurement Root
Security Monitor
Hypervisor
Hypervisor
Enclave multiplexing
Supervisor
Operating System
Enclave management
User
Host Application
Enclave setup
Enclave syscall shims
Non-sensitive code and data
Enclave
Sanctum-aware runtime
Sensitive code and data

The central piece of software is the security monitor, which enforces Sanctum’s security policies.
Our security monitor is tiny, so it can be amenable to formal verification. The key to keeping it so small is that we don’t make any resource allocation decisions in the monitor. The operating system gets to make all the decisions, just like today.
In Sanctum, the OS must submit its decisions to the security monitor in order to have them applied. This way, we can verify the OS’ decisions in the monitor, and we can reject decisions that would break Sanctum’s security guarantees.



Software Stack

Machine

Measurement Root

Security Monitor

Hypervisor

Hypervisor

Enclave multiplexing

Supervisor

Operating System

Enclave management

User

Host Application

Enclave setup

Enclave syscall shims

Non-sensitive code and data

Enclave

Sanctum-aware runtime

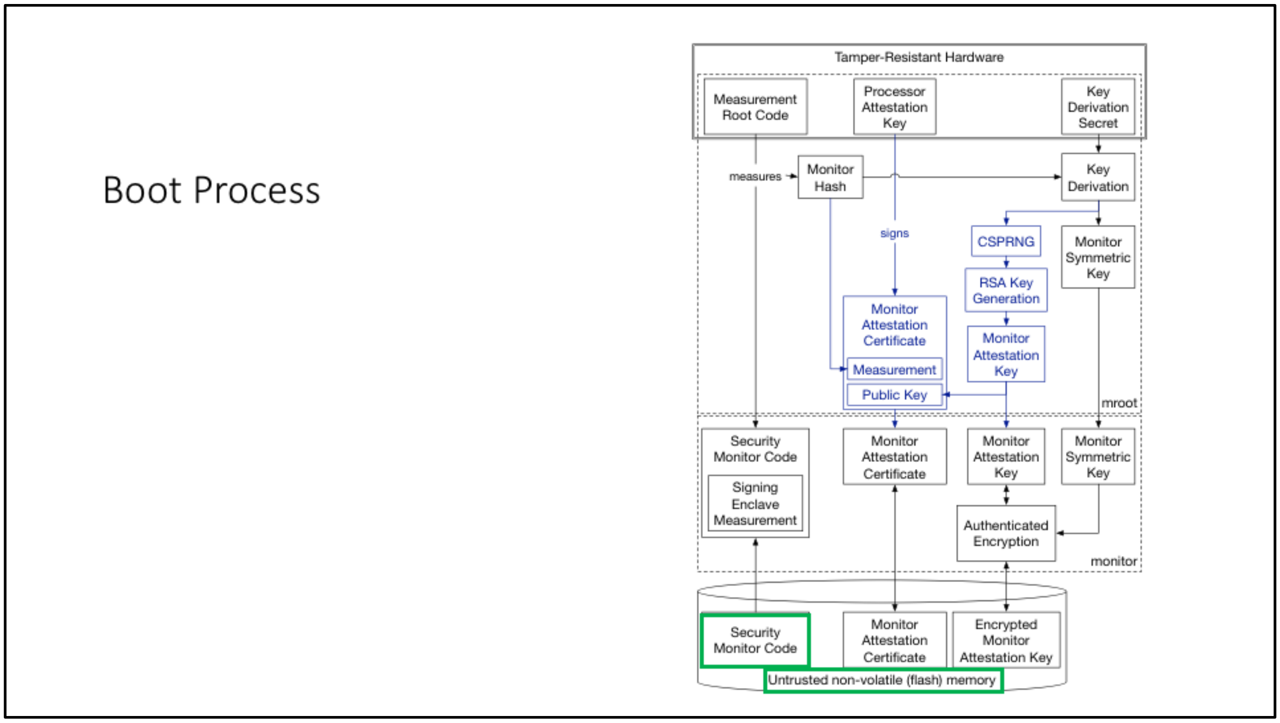
Sensitive code and data

The other piece of software that runs in machine mode is the measurement root. This code runs at boot time, and sets up the software attestation chain. It does not play any part after the boot process completes.

Unprecedented Control

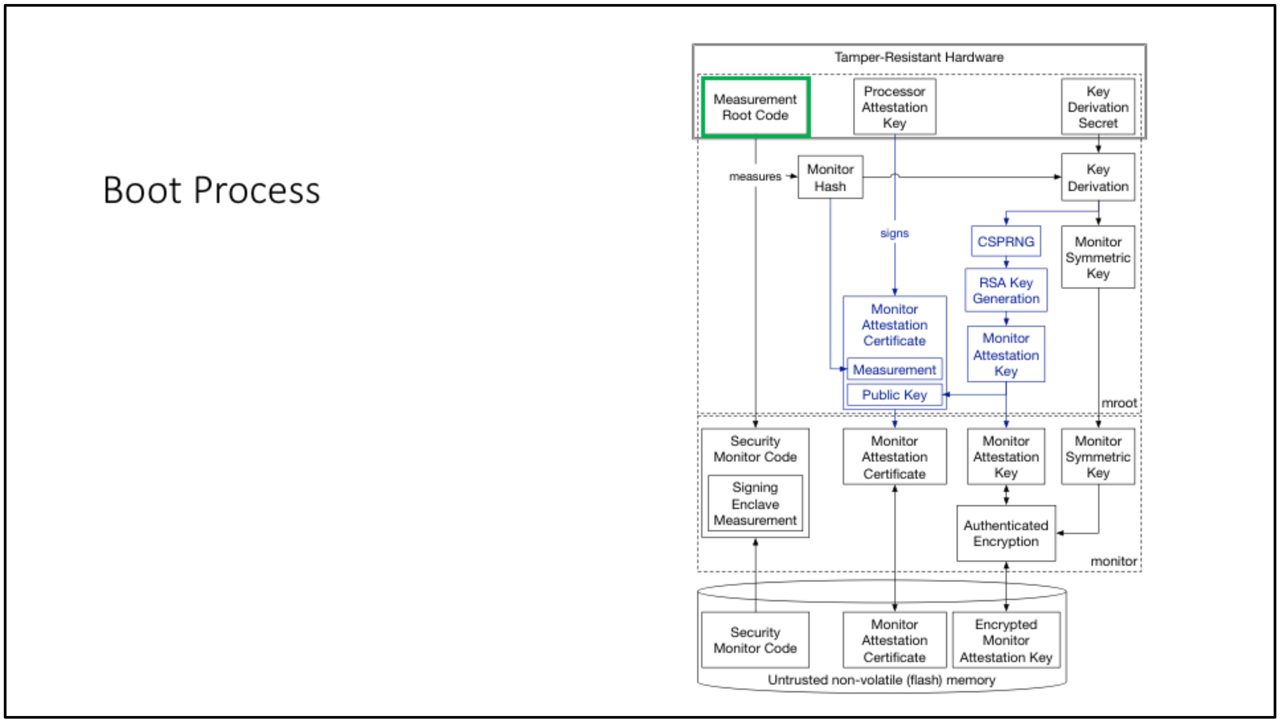
Sanctum's Software Attestation Process

Let's see how Sanctum gives computer owners unprecedented control over their systems.



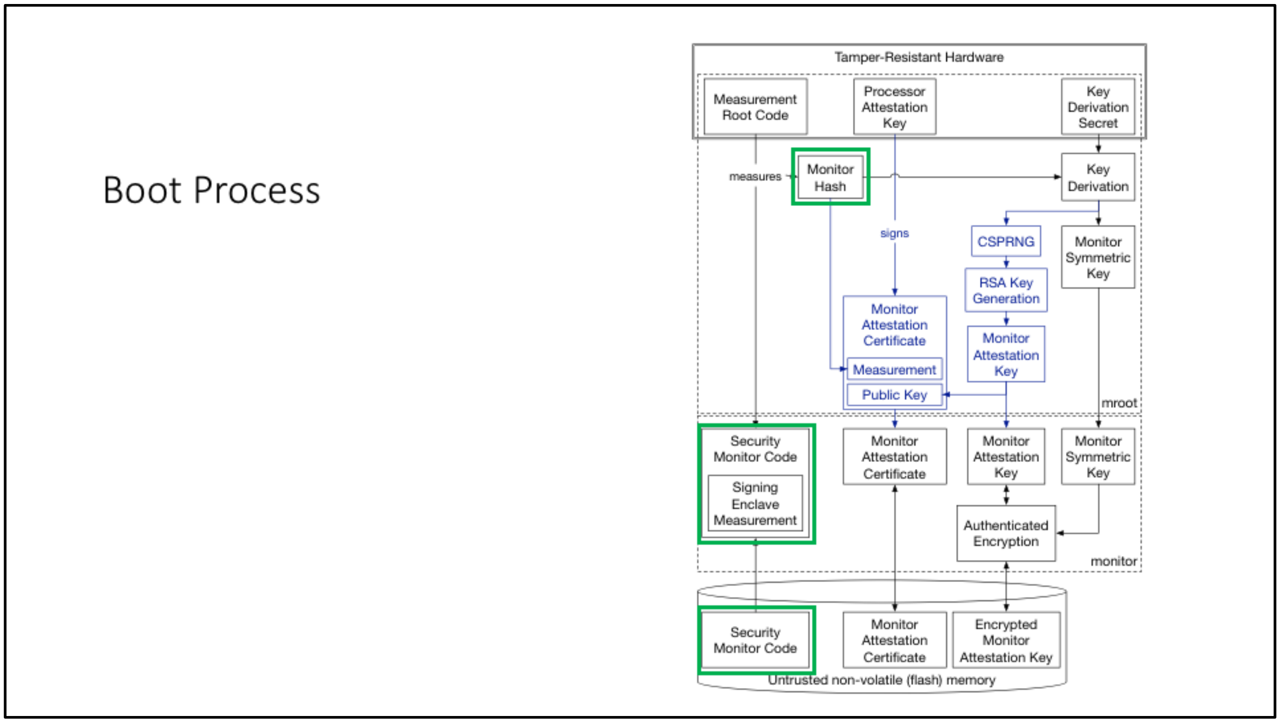
Boot Process

Before we go into attestation, let’s talk a little bit about Sanctum’s boot process. One of the things that make Sanctum special is that the security monitor, which is most of our logic, is stored in flash memory, and can be replaced by the computer’s owner.



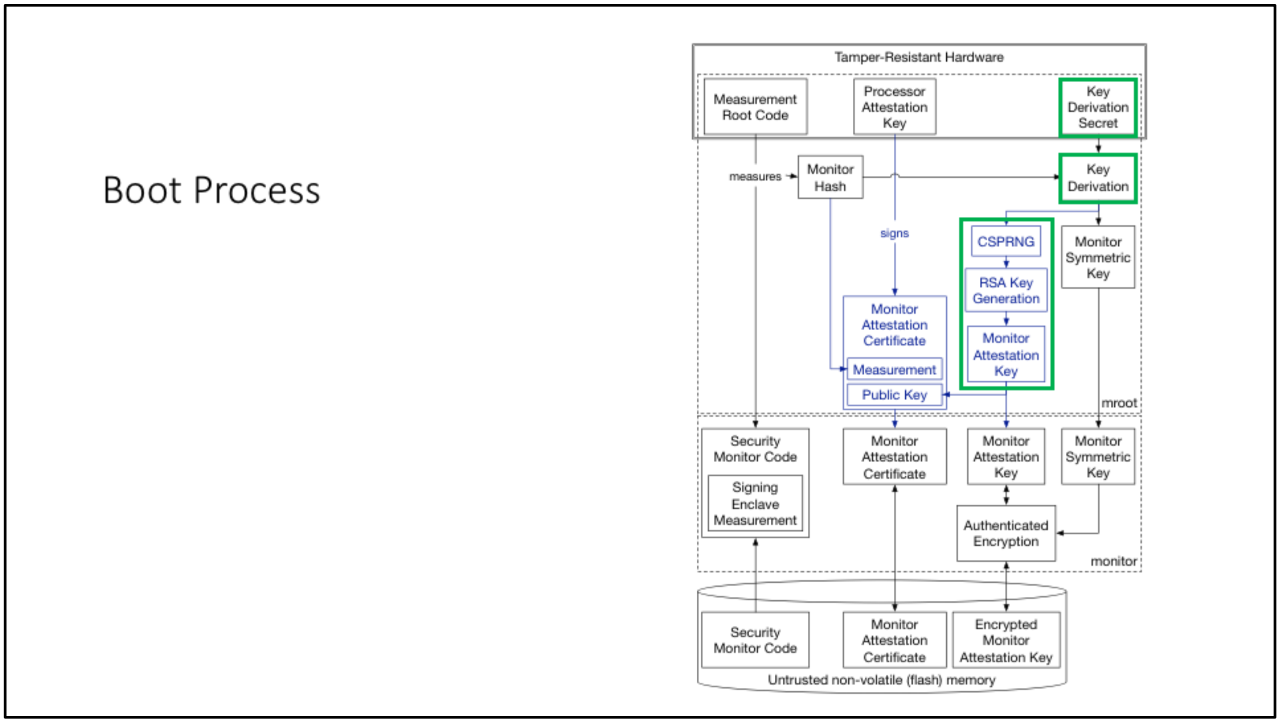
Boot Process

In order to keep things secure, we rely on the measurement root code, which is stored in the processor’s ROM, and cannot be modified.



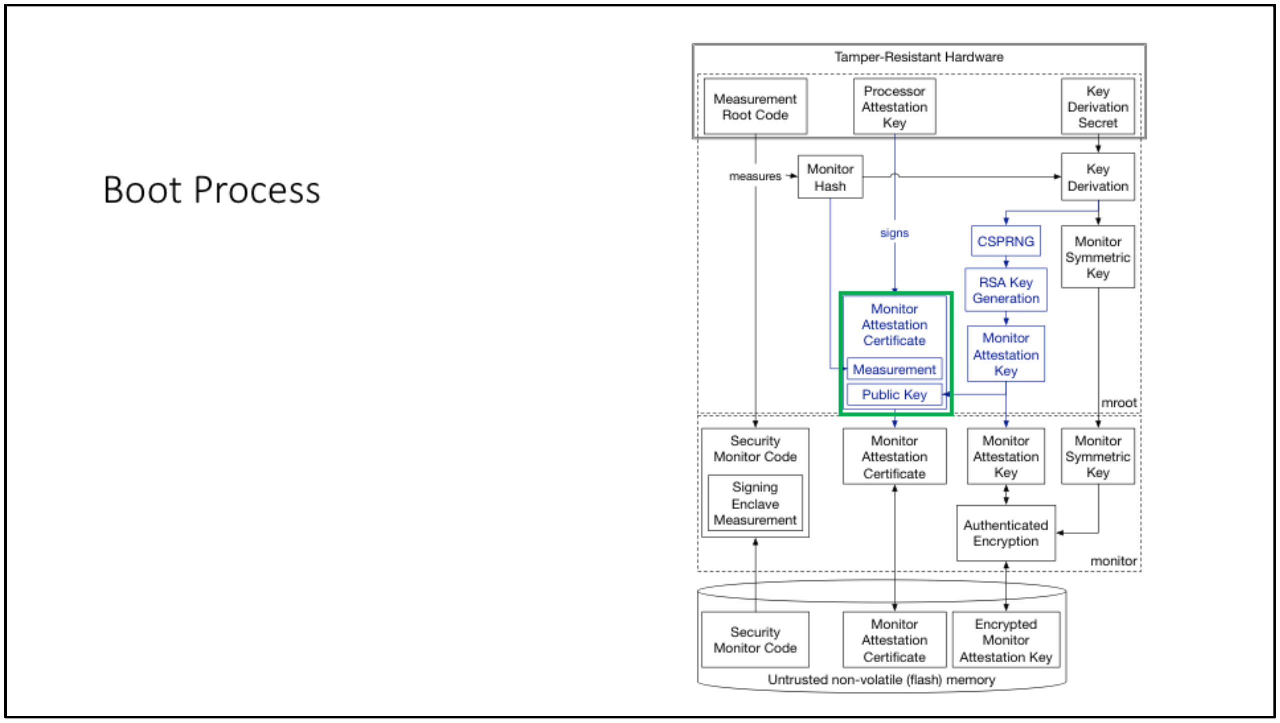
Boot Process

When the computer powers up, the measurement root reads the security monitor and computes its cryptographic hash.



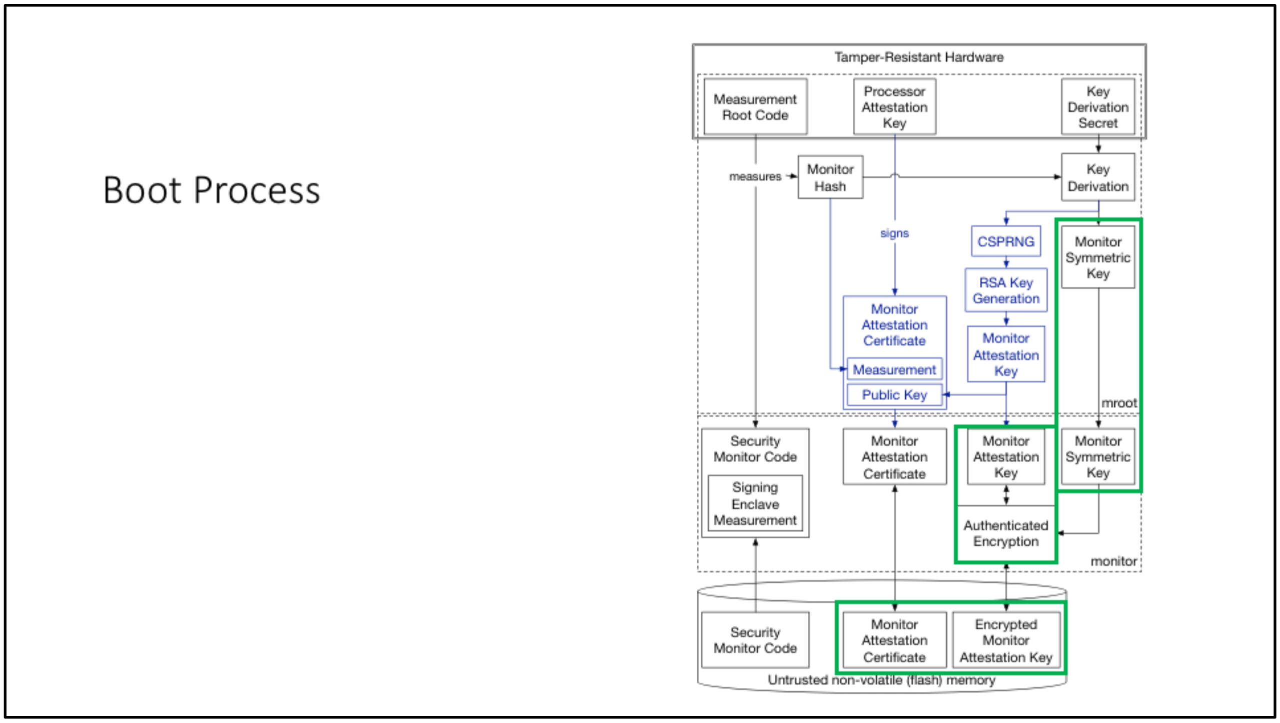
Boot Process

This hash is fed into some cryptographic machinery that eventually generates an RSA key pair, which becomes the monitor's attestation key.



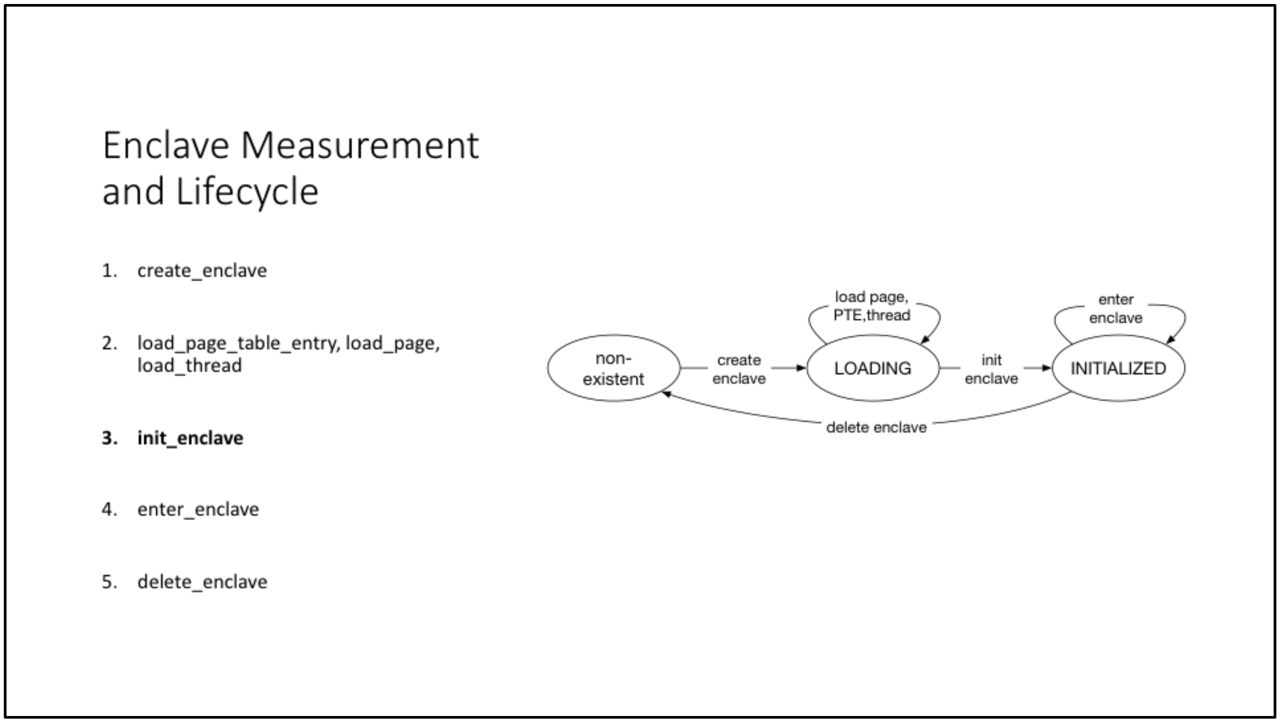
Boot Process

Finally, the measurement root uses the processor's attestation key to issue a certificate that contains the monitor's public attestation key and its measurement hash. After this happens, the measurement root hands control over to the security monitor. The key fact to note here is that the monitor's certificate contains the monitor's measurement. If the computer owner modifies the monitor, the changes will be reflected in the attestation certificate.



Boot Process

If you’re horrified by the prospect of having to generate an RSA key on every boot, rest assured. We have a mechanism to securely cache the monitor’s RSA key in flash. This way, we only need to regenerate the RSA key when the computer owner replaces the security monitor, which shouldn’t happen very often.

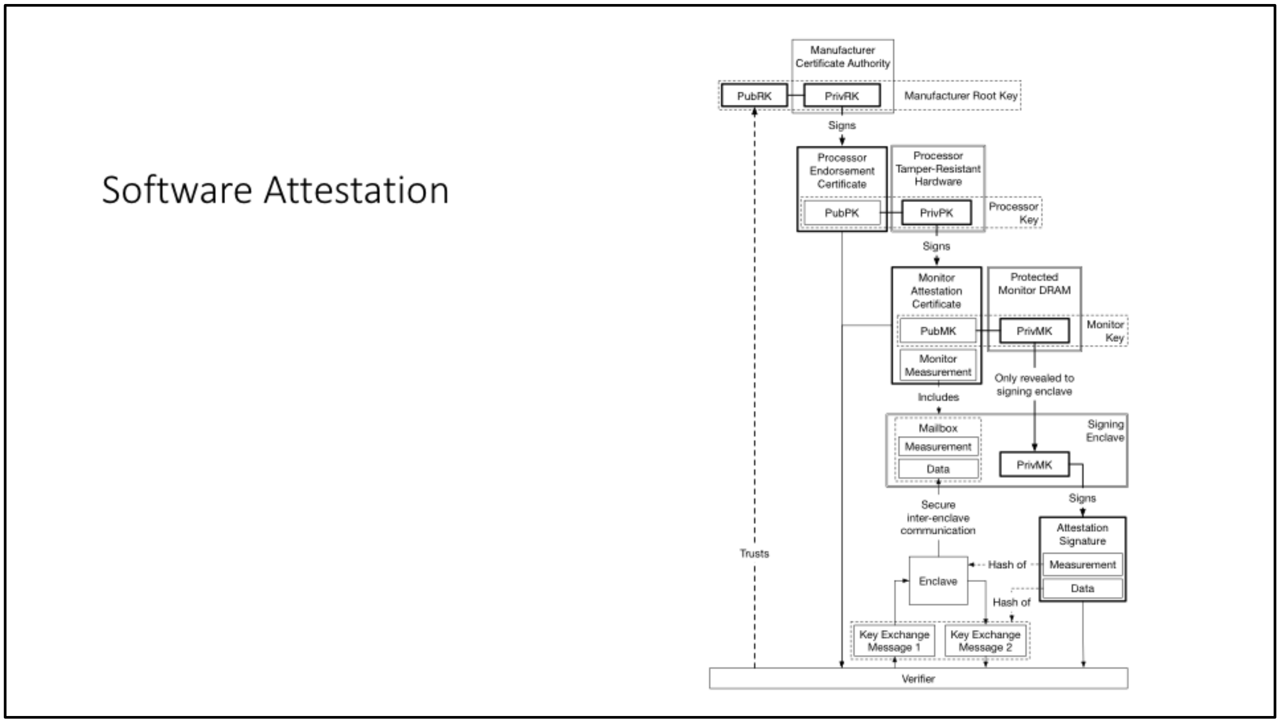


Enclave Measurement and Lifecycle

1. create_enclave
2. load_page_table_entry, load_page, load_thread
3. **init_enclave**
4. enter_enclave
5. delete_enclave

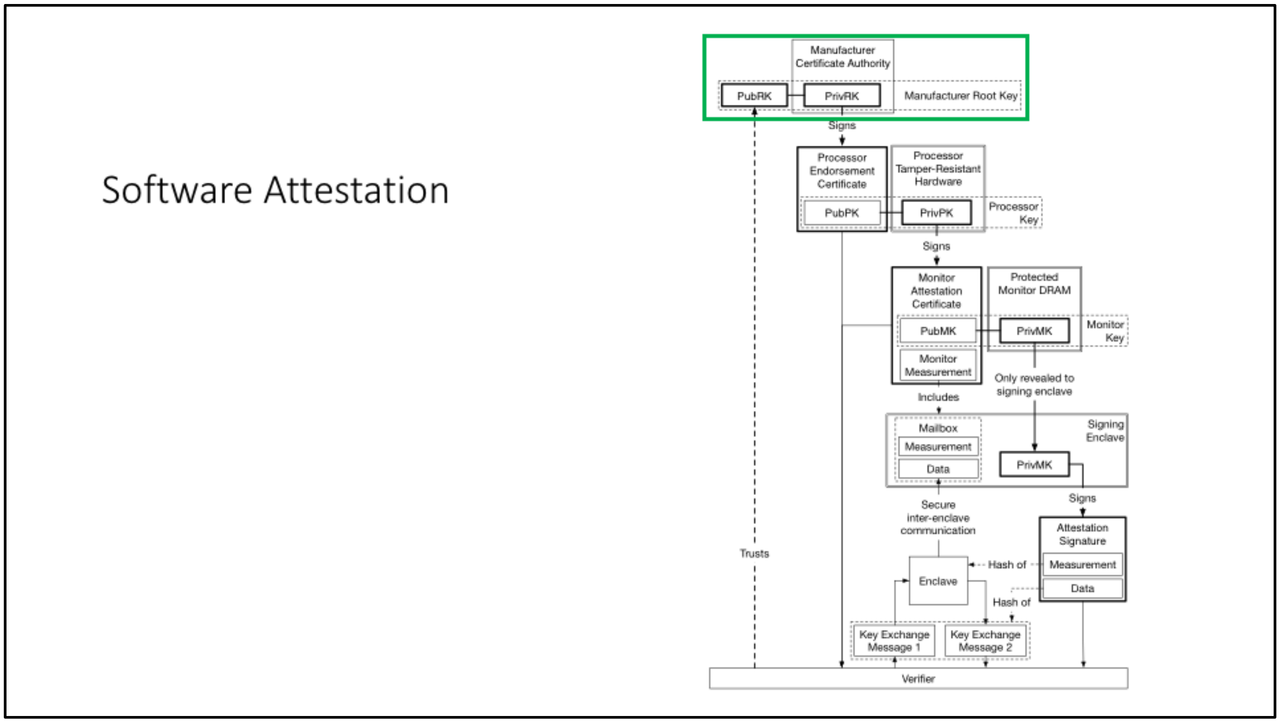
We borrowed SGX’s general approach to enclave measurement. When an enclave is created, it is in a loading state, where it cannot be used. The application that created the enclave works with the OS to allocate resources to the enclave, and to load the initial pages of code and data into the enclave. The parameters of each loading operation are hashed and contribute to the enclave’s measurement.

When the loading stage is complete, the enclave is initialized. At this point, the enclave’s measurement is finalized, and the security monitor will not allow the use of any loading API on the enclave. So, an enclave’s measurement is an accurate representation of the enclave’s initial memory state.



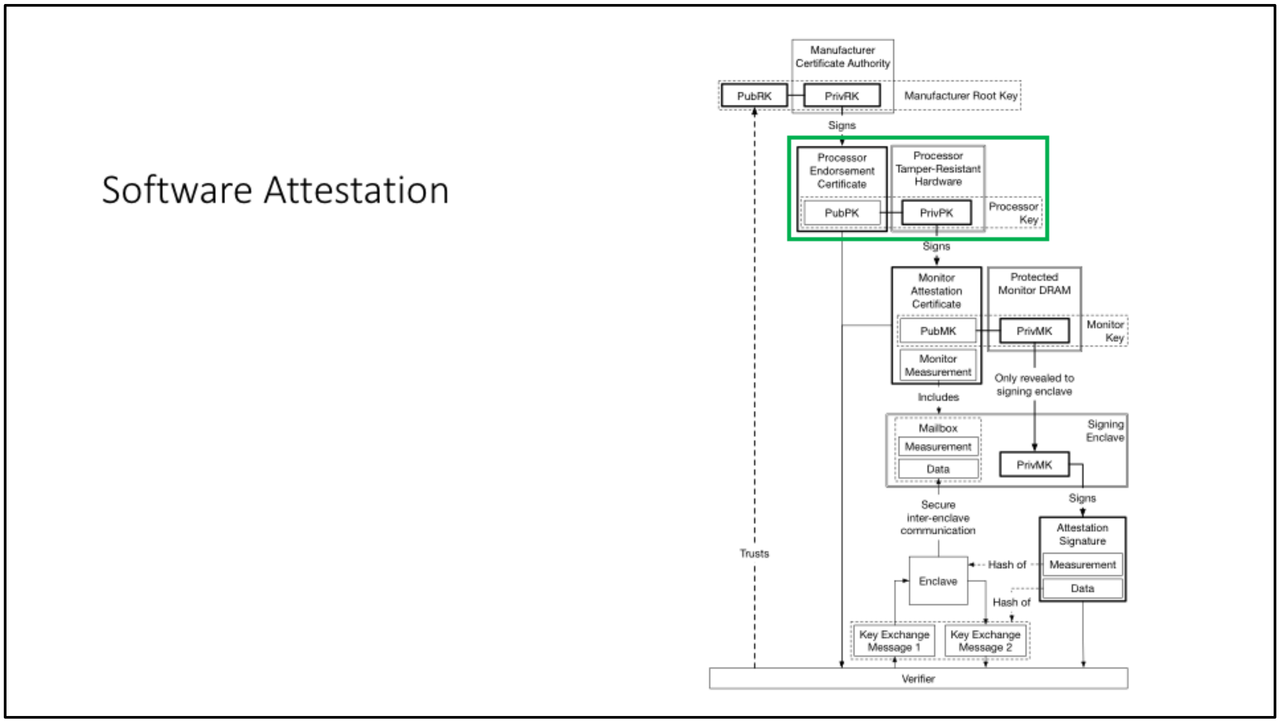
Software Attestation

Now that you know how the boot process works, you'll find the rest of the attestation to be pretty standard. I'll go through it quickly, so you can see the ideas I've outlined come together.



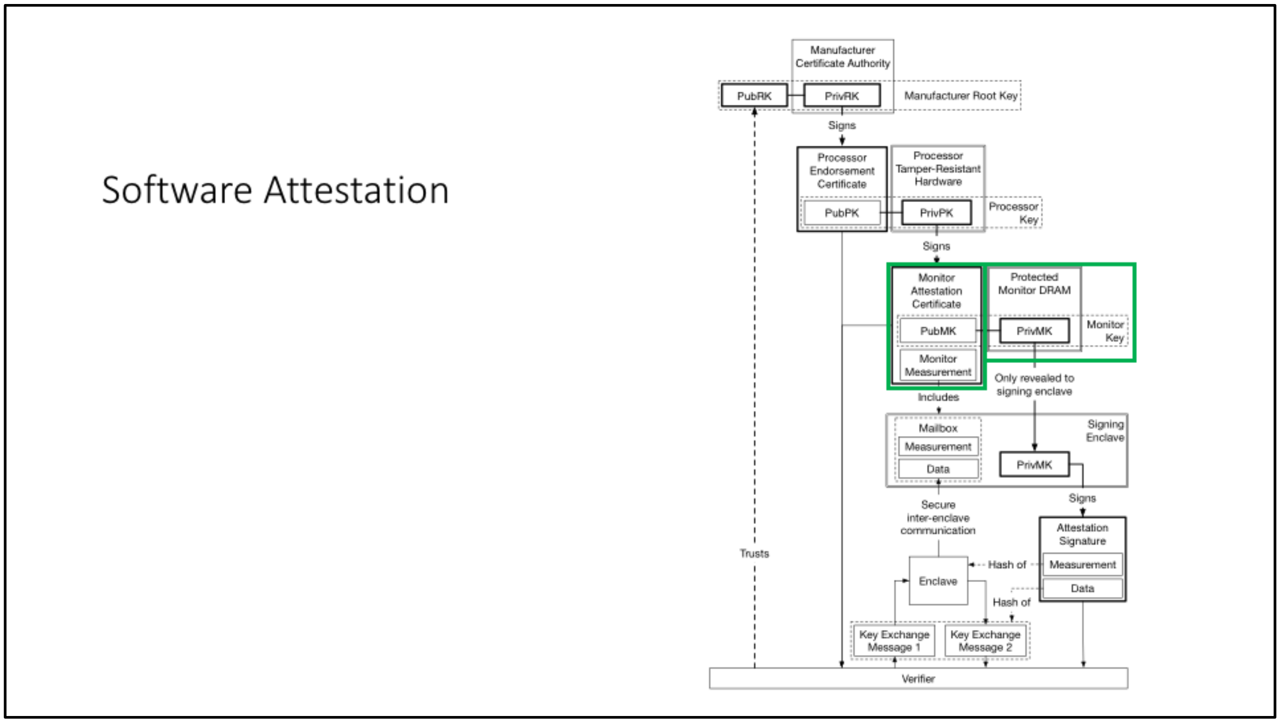
Software Attestation

First, you trust the Sanctum processor manufacturer, which has a root key.



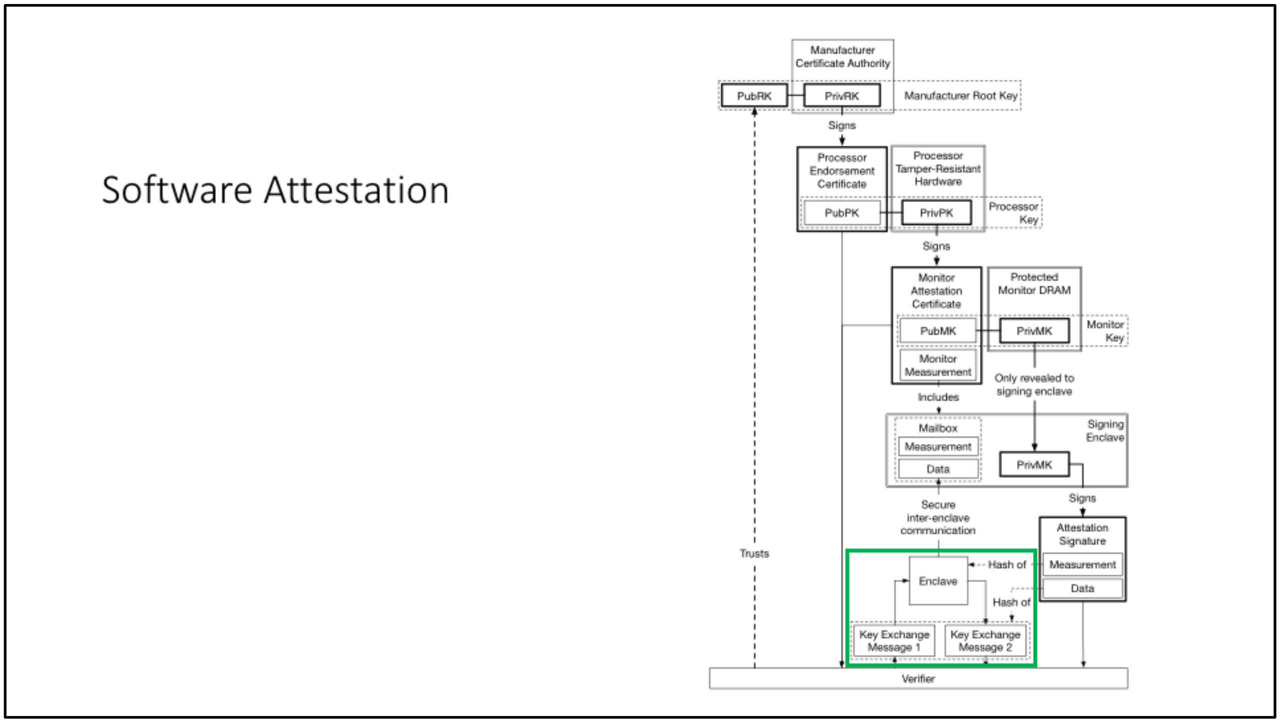
Software Attestation

Each Sanctum processor has a processor attestation key, and an endorsement certificate from the manufacturer.



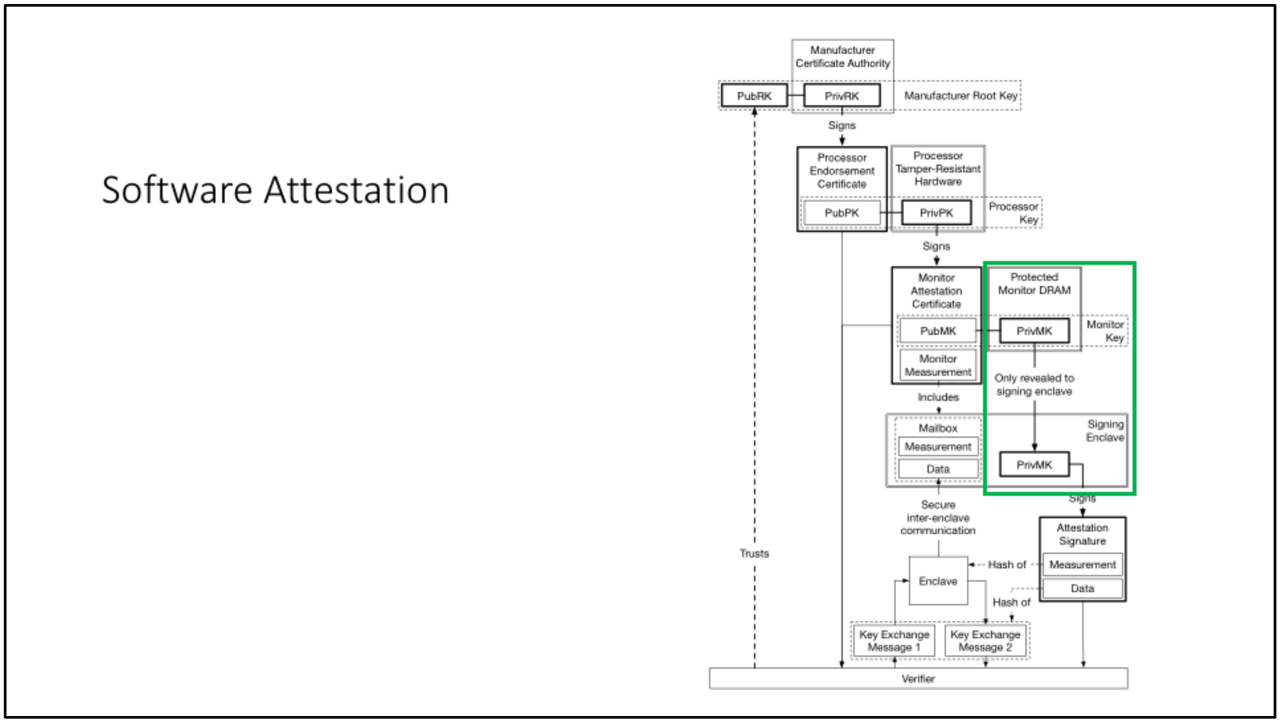
Software Attestation

When the computer powers up, the measurement root creates a monitor attestation key pair and issues an attestation certificate to the monitor.



Software Attestation

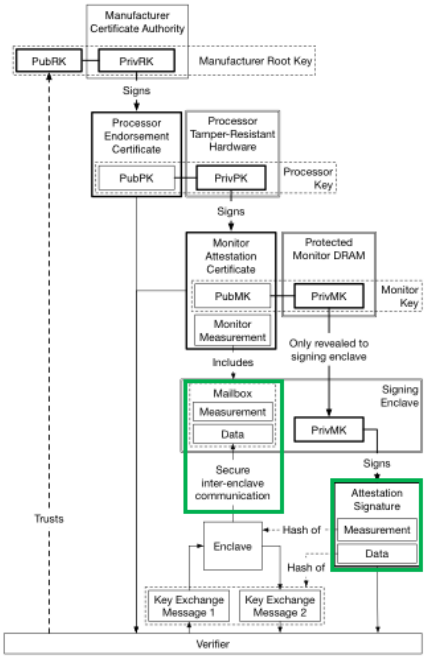
Now, when you want to send your computation over to a Sanctum processor, you send the enclave's initial contents, and a challenge message. The enclave starts executing, reads the challenge message, and computes the response message. The enclave uses Sanctum's secure inter-enclave communication primitives to send a hash of the challenge response to the Signing Enclave, which is a special enclave that the security monitor trusts.



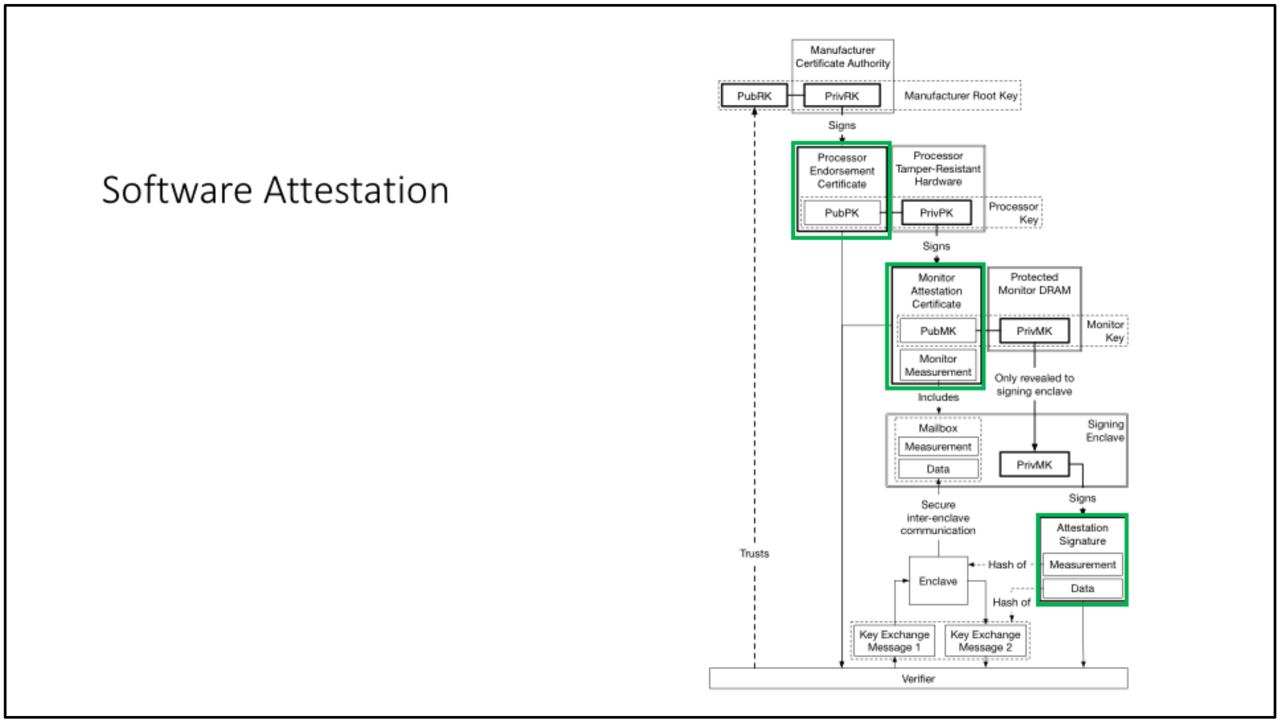
Software Attestation

In Sanctum, the security monitor does not perform any operations using the private attestation key. Instead, there is a special Signing Enclave, whose measurement hash is hard-coded into the security monitor. This is the only enclave that can receive the monitor's private attestation key.

Software Attestation

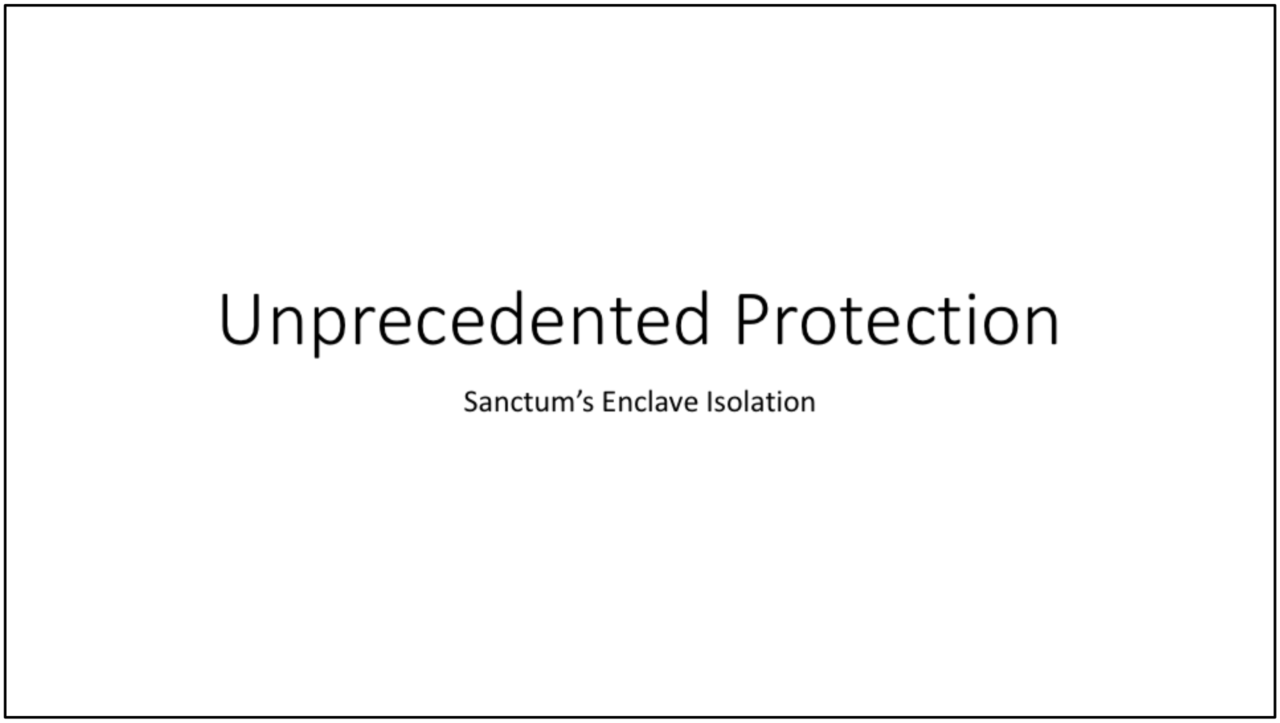


So, our enclave must use Sanctum’s secure inter-enclave communication primitives to send a hash of the response message to the Signing Enclave. The Signing Enclave creates an attestation signature that covers our enclave’s measurement and the response message.



Software Attestation

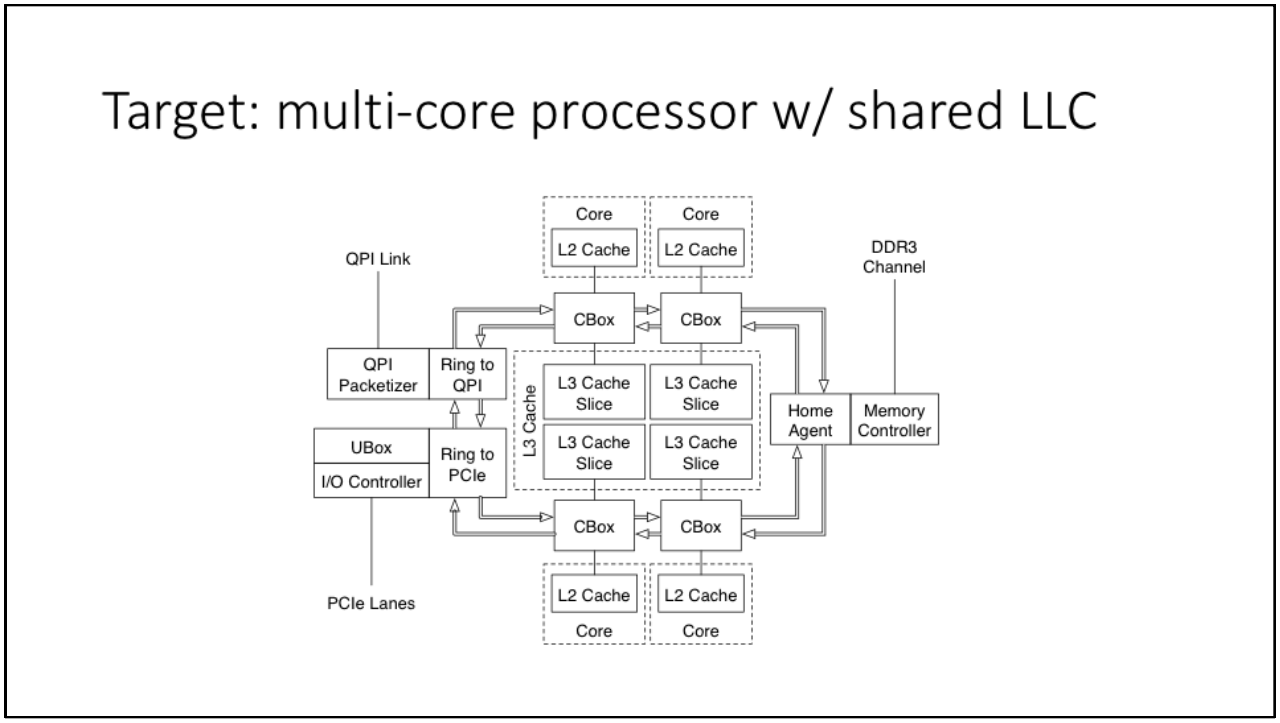
At this point, we can examine the attestation signature, and the monitor and processor's attestation certificates, to convince ourselves that we are communicating with an enclave built according to our instructions, hosted on a Sanctum processor running a security monitor that we trust.



Unprecedented Protection

Sanctum's Enclave Isolation

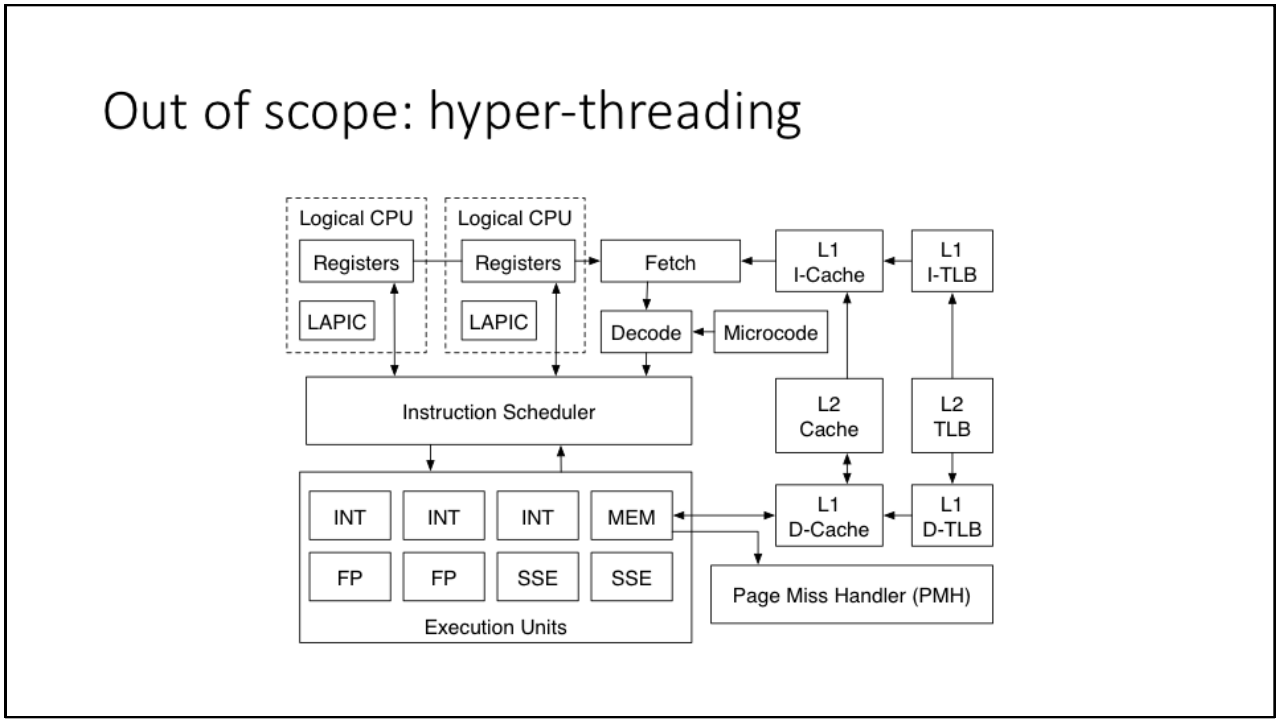
Now, let's talk a bit about protection.



Target: multi-core processor w/ shared LLC

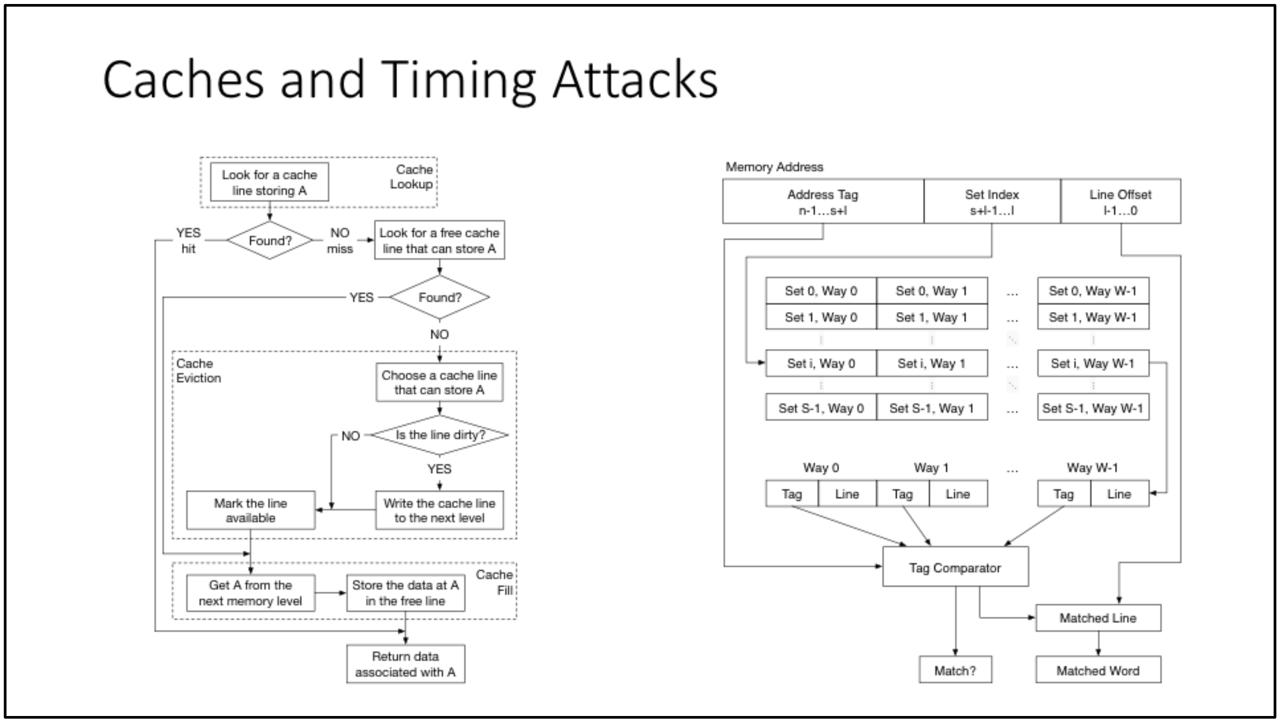
Sanctum was designed for multi-core processors with a shared last-level cache. This matters because per-core caches can be protected pretty easily – whenever you enter or exit an enclave, flush the core's caches. This is a simple strategy, and we like simple, so this is exactly what we do to protect per-core caches.

The shared LLC is more interesting, because an attacker thread could be attempting a timing attack at any time. Sanctum uses a very simple scheme, called page coloring, which I will outline soon.



Out of scope: hyper-threading

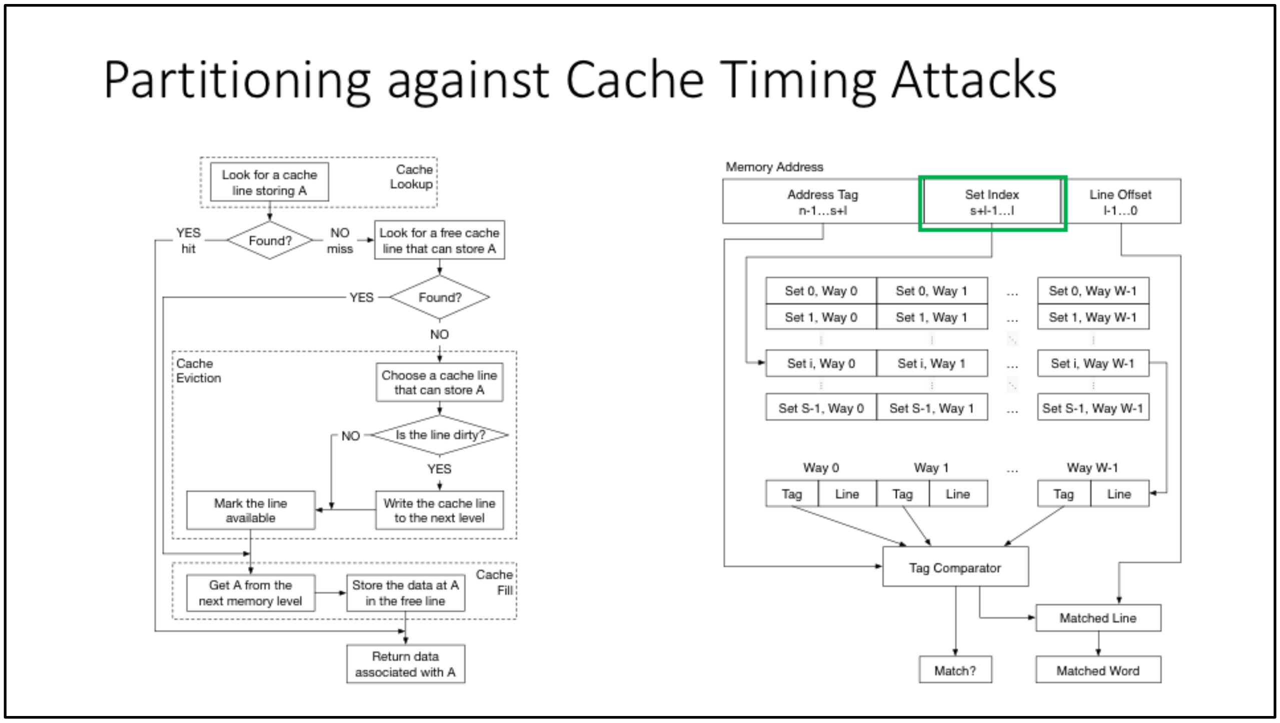
Sanctum *almost* scales to a full-fledged Intel desktop system. The one aspect we don’t support is hyper-threading. This is because with hyper-threading, there’s just simply too much micro-architectural resource sharing.
We couldn’t come up with any sane way to protect an enclave thread from being attacked by a thread running on the same core. So Sanctum doesn’t support hyper-threading.



Caches and Timing Attacks

Now, let’s turn onto cache timing attacks. At a very high level, these attacks take advantage of the fact that a cache hit is much faster than a cache miss. Virtually all cache timing attacks require the attacker to access the same cache as the victim, so the memory access timing can be observed.

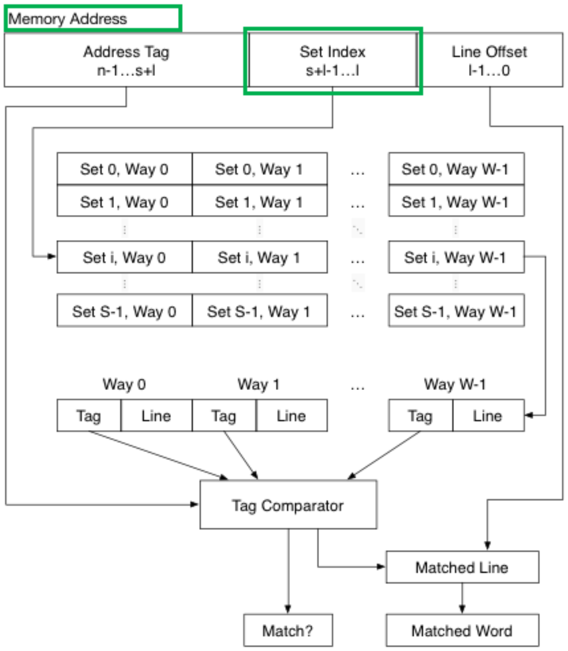
Sanctum targets set-associative caches, which are the most commonly used caches. For the purpose of this talk, all that matters is that a set-associative cache is made up of multiple sets, and a memory location can only be cached in exactly one set, depending on its address. Conceptually, we can pretend that each set in a set-associative cache is a separate cache memory.



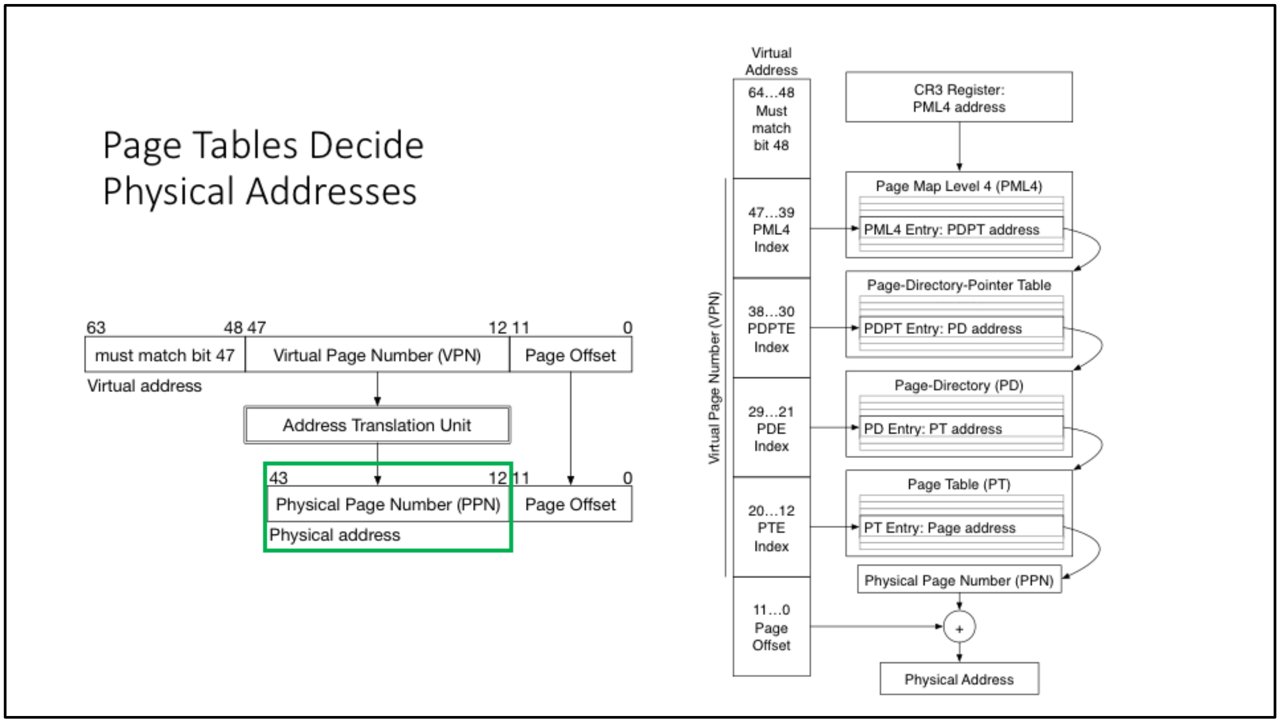
Partitioning against Cache Timing Attacks

This is the insight behind cache partitioning. If we can ensure that memory owned by an enclave will never end up in the same cache set as memory owned by an attacker, we can consider that the enclave and the attacker are using different caches. This defeats cache timing attacks, because they require a shared cache.

Caches Use Physical Addresses

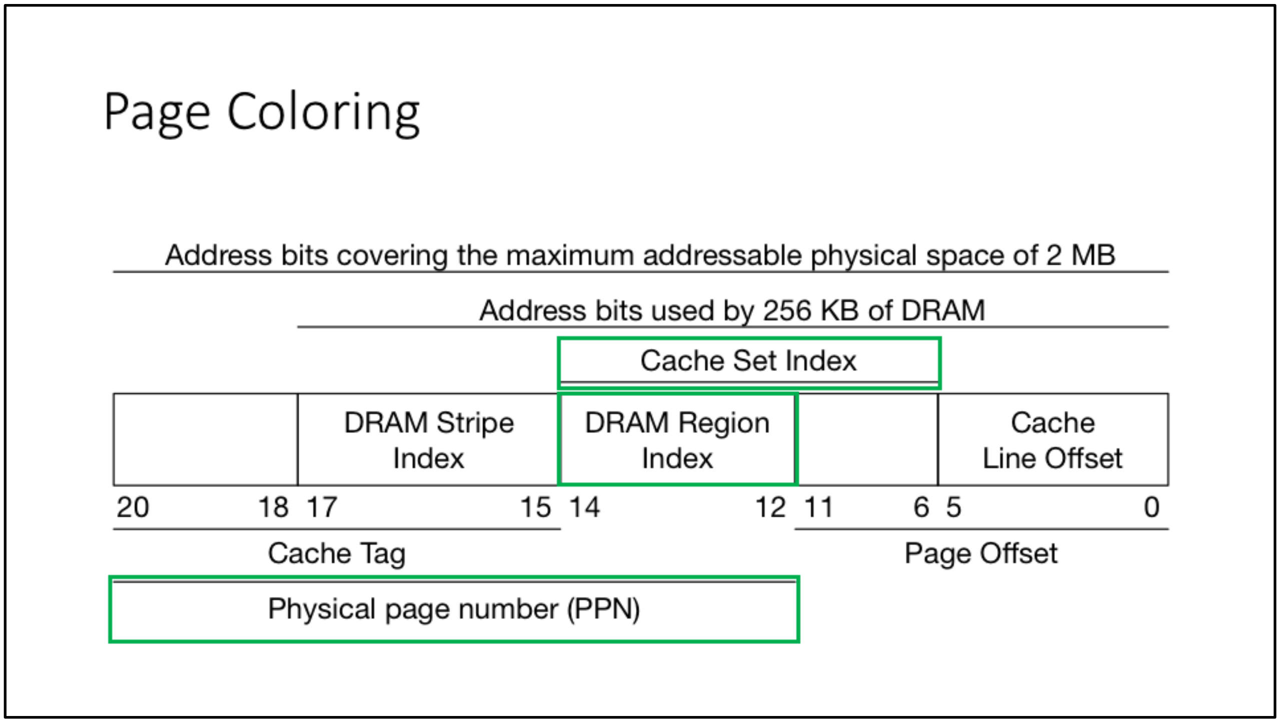


Set-associative caches use physical memory addresses for set index computation.



Page Tables Decide Physical Addresses

This is convenient for us, because software uses address translation, which abstracts away physical addresses almost completely. We can ask the operating system to structure its page tables in any way that helps us achieve our security goal.



Page Coloring

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

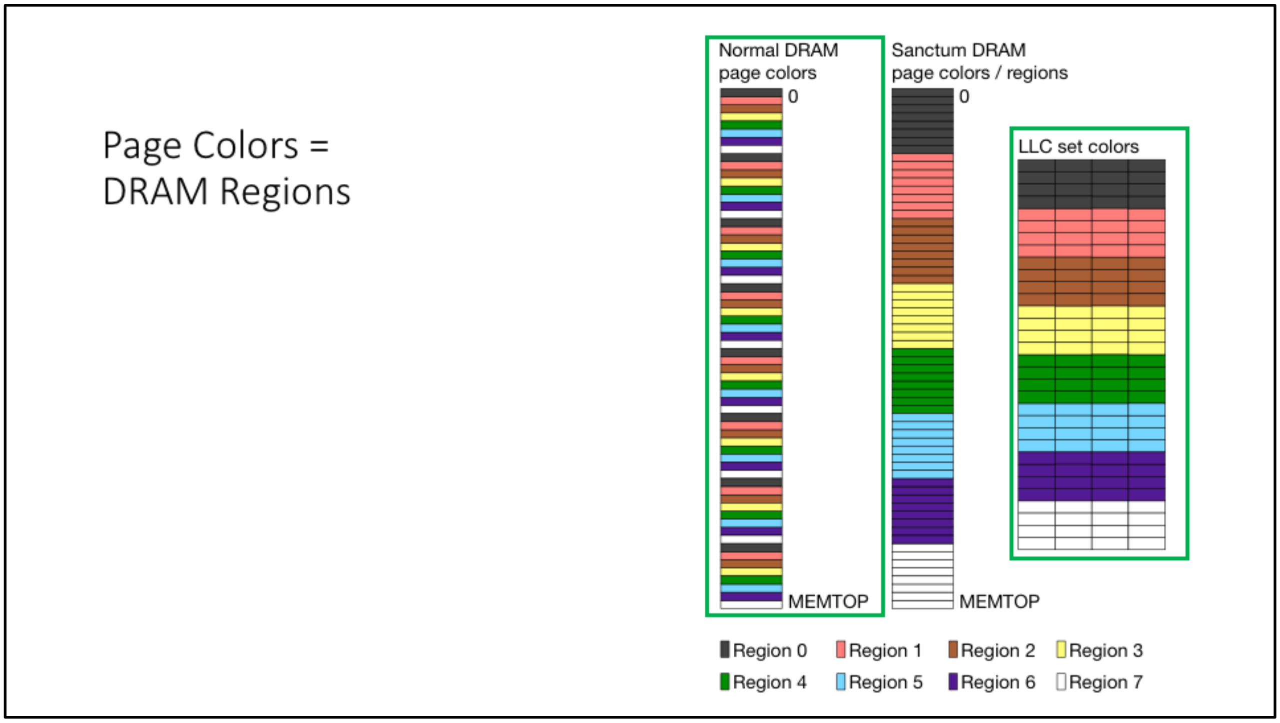
20 18 17 15 14 12 11 6 5 0

Cache Tag

Page Offset

Physical page number (PPN)

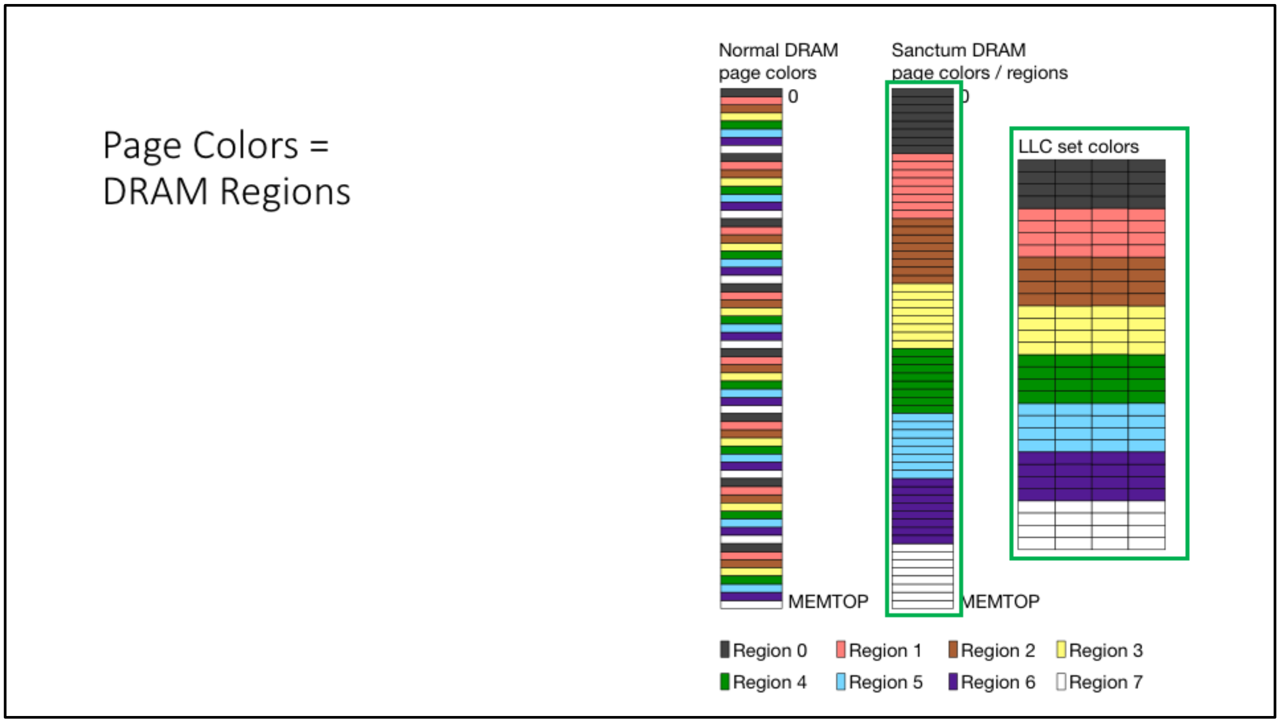
Page coloring essentially comes down to this observation that I just made. There are some physical address bits that belong to the cache set index, and are set by the page tables. These bits can be used to control cache placement.



Page Colors = DRAM Regions

The problem is, by default, the page color bits are at the bottom of an address' physical page number. If you draw your DRAM and color each page, it's going to look like the stack on the left.

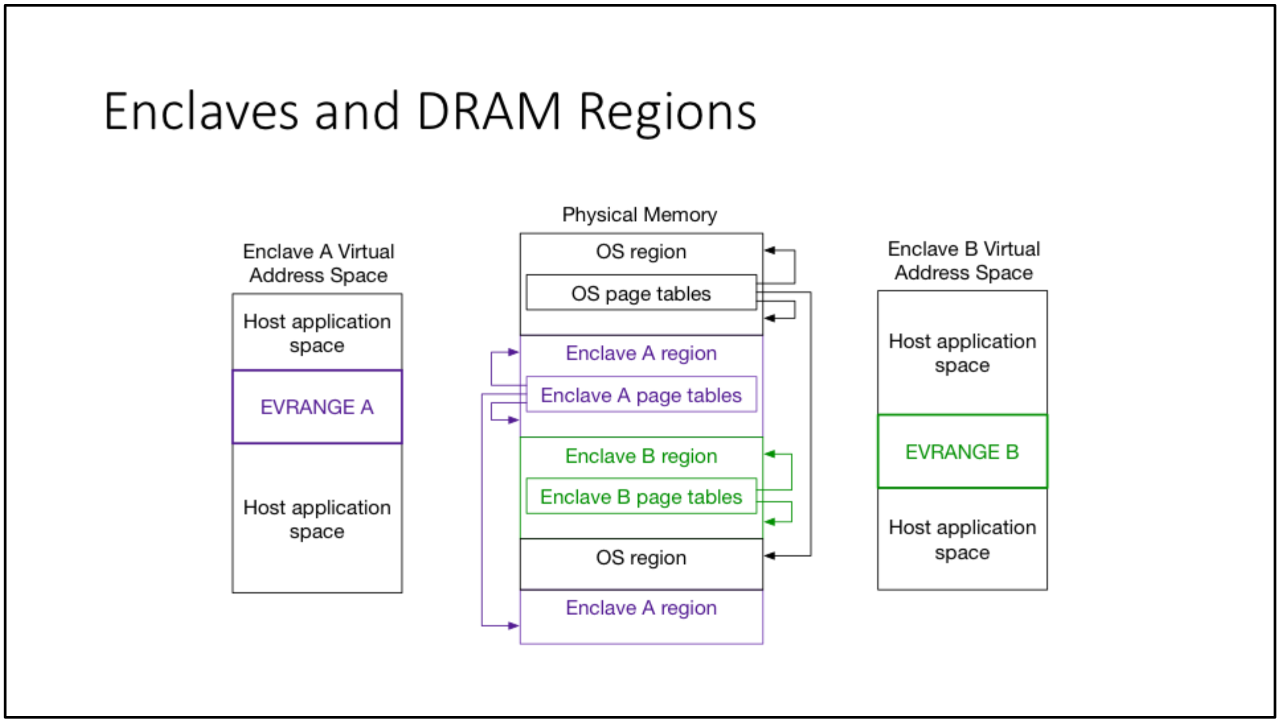
This is a problem because, in Sanctum, we'd like to assign some colors exclusively to enclaves. At the same time, the operating system probably needs some large continuous chunks of DRAM for DMA. For example, if you have a graphics card or a high-performance network card, these things tend to like large DMA buffers.



Page Colors =
DRAM Regions

We use a bit of hardware to munge the physical addresses as they enter the cache unit, and obtain the color map on the right.

In Sanctum, the DRAM is effectively split into equally-sized DRAM regions, and each region maps to disjoint LLC sets.



Enclaves and DRAM Regions

Enclave A Virtual Address Space

Host application space

EVRANGE A

Host application space

Physical Memory

OS region

OS page tables

Enclave A region

Enclave A page tables

Enclave B region

Enclave B page tables

OS region

Enclave A region

Enclave B Virtual Address Space

Host application space

EVRANGE B

Host application space

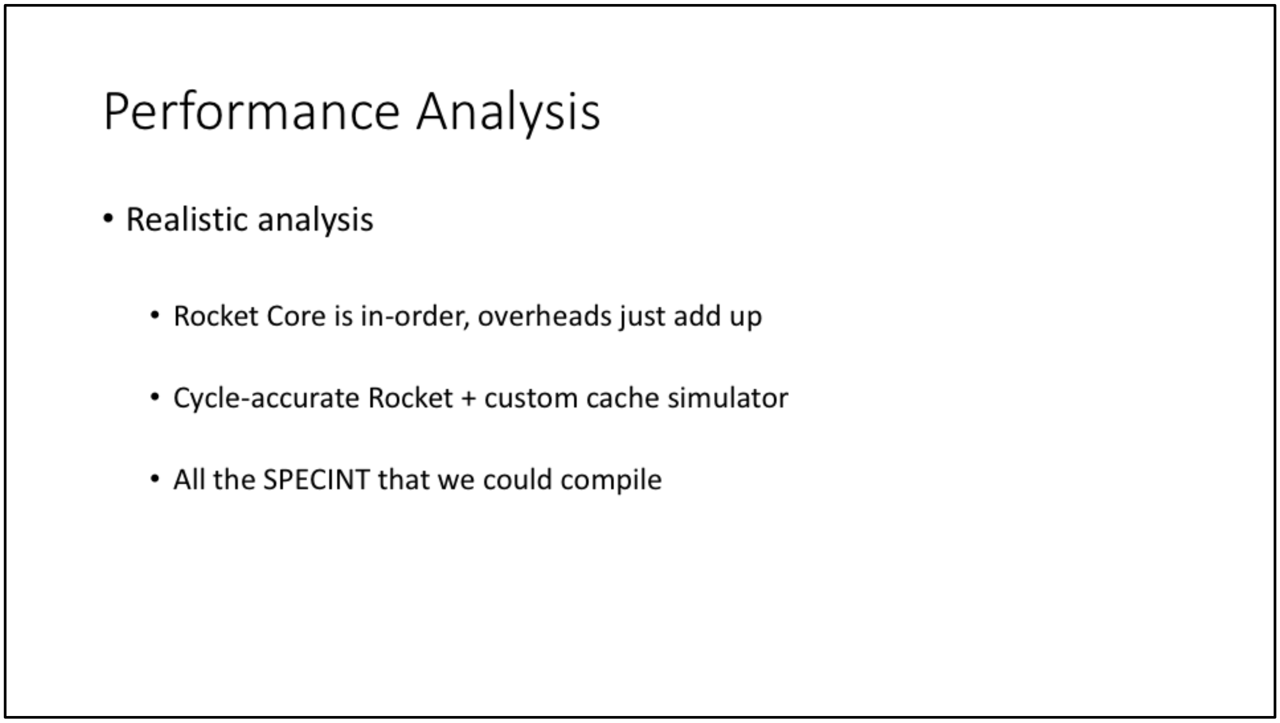
Each DRAM region can be assigned to exactly one enclave, or to the operating system and its untrusted processes. Most of each enclave’s virtual address space is mapped into the host application’s memory, using the page tables prepared by the OS for the host application. However, one continuous range is used to map the enclave’s own DRAM regions, using a separate set of page tables that is managed by the enclave.

This gives us isolation at all levels. The data in an enclave’s DRAM regions is not accessible to any other software. The page tables used to map the enclave’s private memory are also isolated, so they don’t leak memory access patterns.

… and this wraps up my high-level overview. The exciting details are in the paper!

This Is Practical

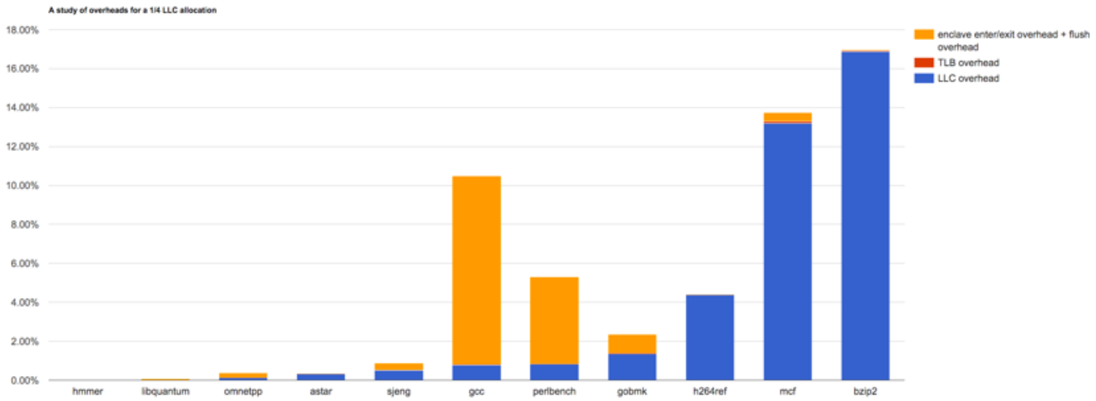
Sanctum Performance Evaluation

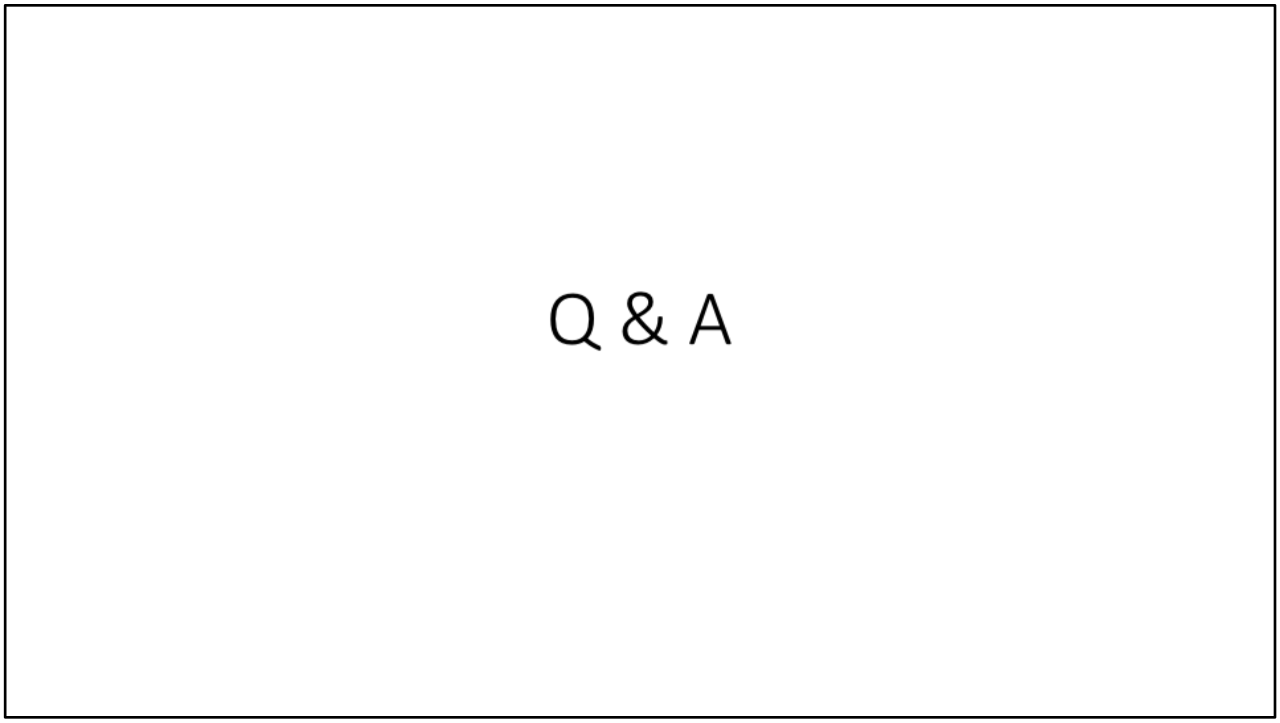


Performance Analysis

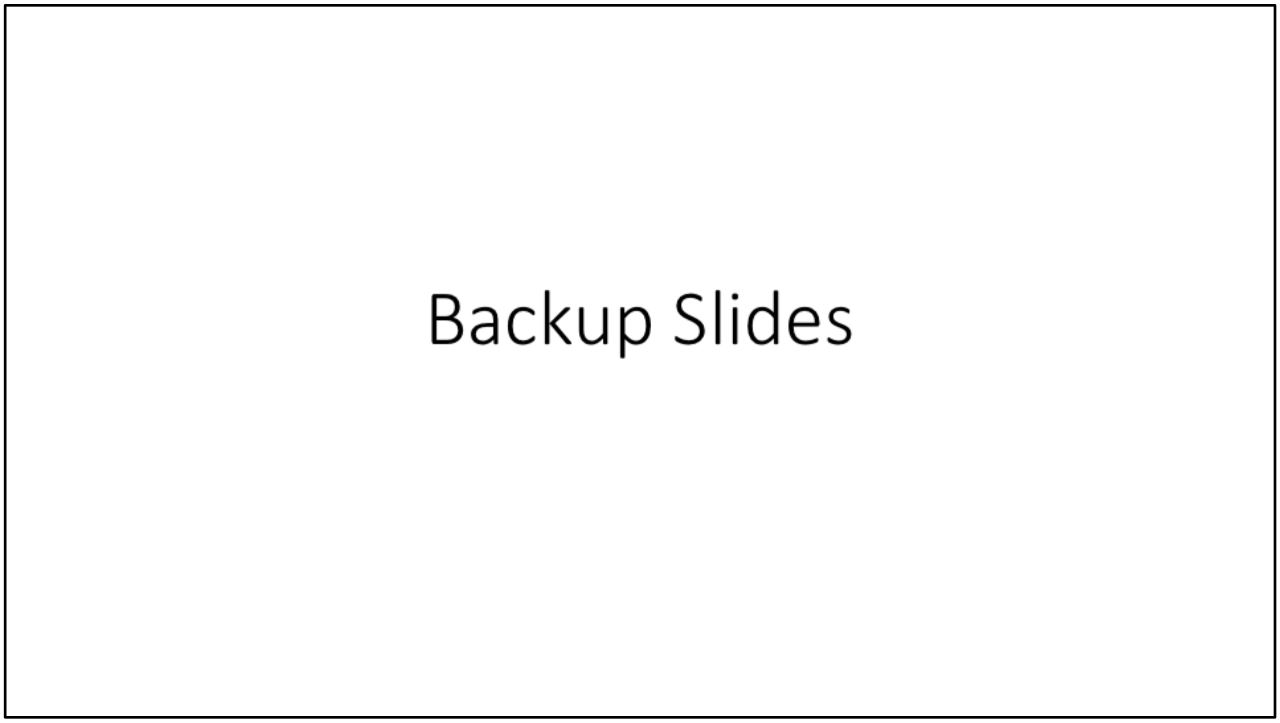
- Realistic analysis
  - Rocket Core is in-order, overheads just add up
  - Cycle-accurate Rocket + custom cache simulator
  - All the SPECINT that we could compile

Sanctum Performance Overheads



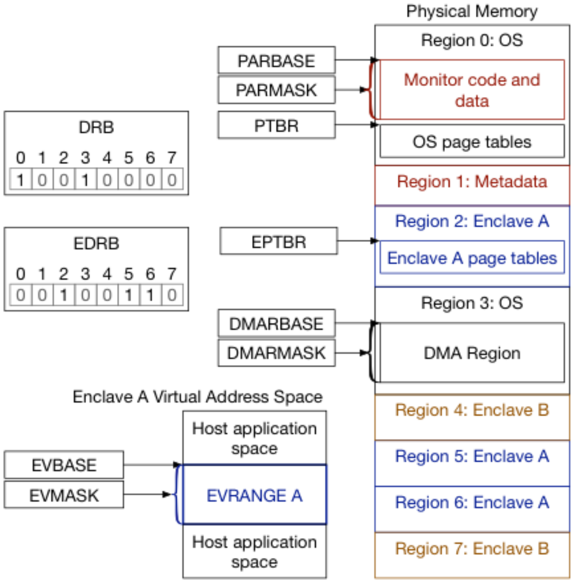


Q & A



Backup Slides

Sanctum PMH Register Configuration



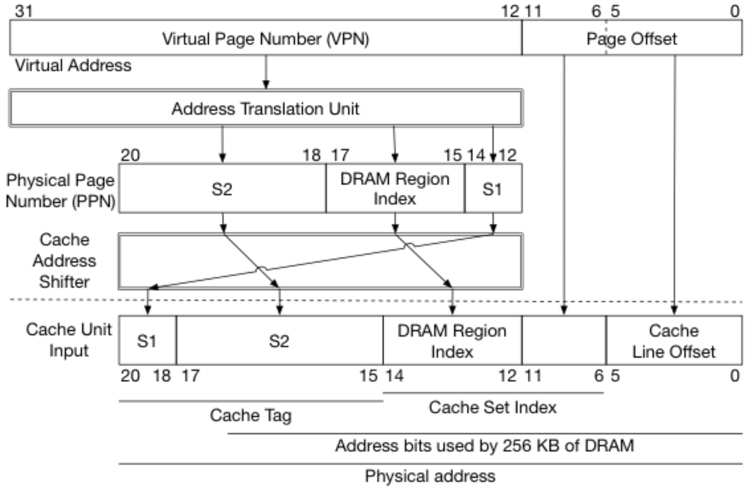
DRB

0	1	2	3	4	5	6	7
1	0	0	1	0	0	0	0

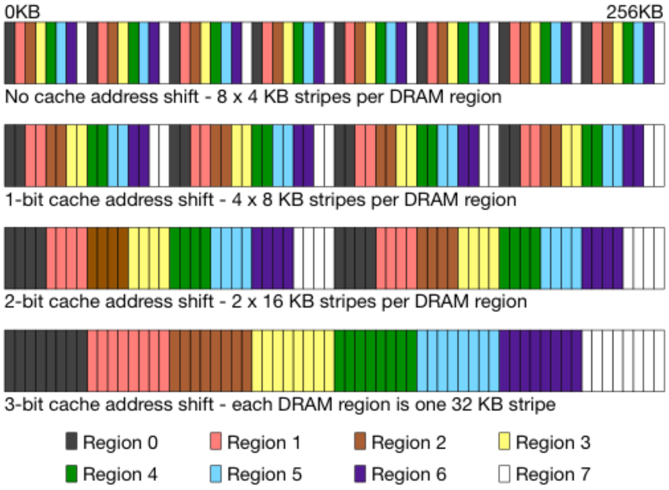
EDRB

0	1	2	3	4	5	6	7
0	0	1	0	0	1	1	0

DMA-Friendly Page Coloring with Minimal Hardware



DMA-Friendly Page Coloring



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

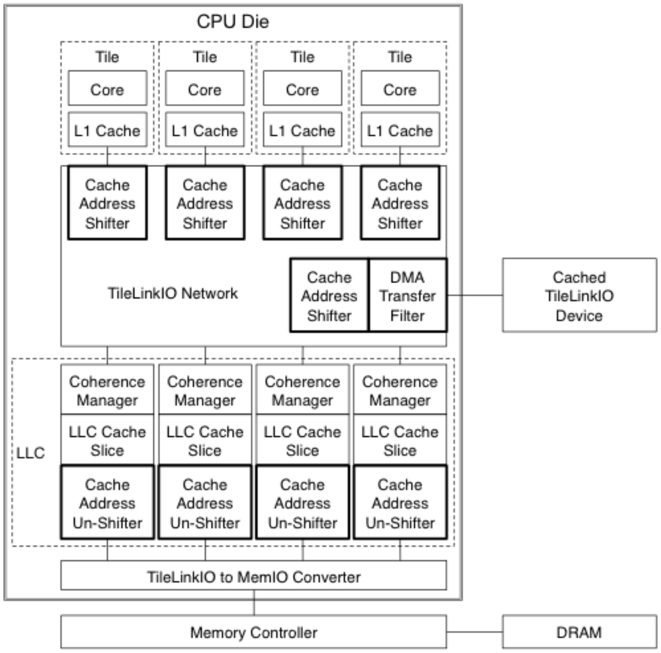
2-bit cache address shift - 2 x 16 KB stripes per DRAM region

3-bit cache address shift - each DRAM region is one 32 KB stripe

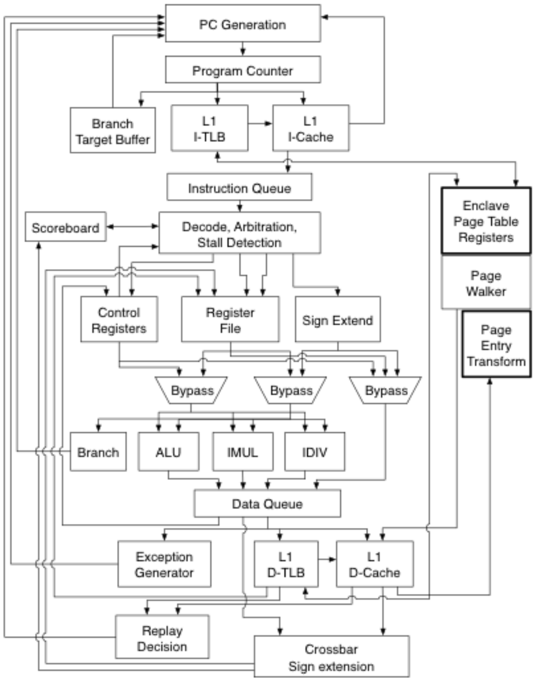
■ Region 0　■ Region 1　■ Region 2　■ Region 3

■ Region 4　■ Region 5　■ Region 6　□ Region 7

DMA-Friendly Page Coloring with Minimal Hardware

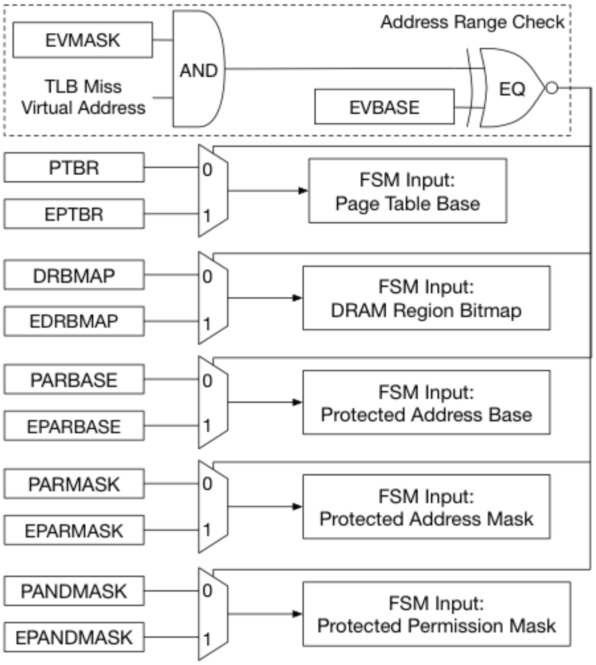


Per-Enclave Page Tables with Minimal Hardware

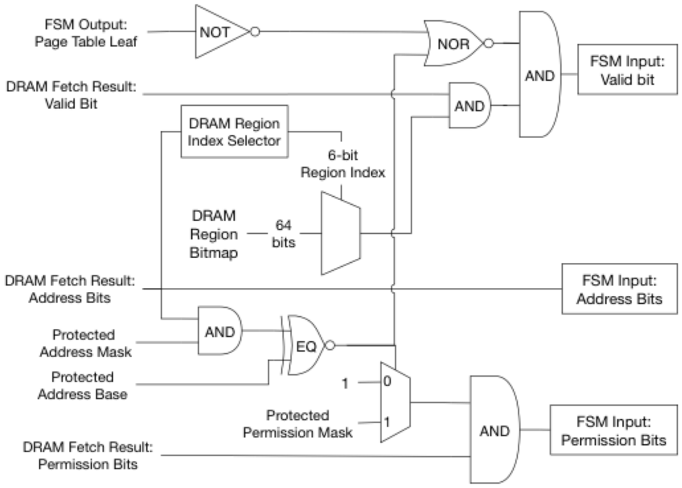


Per-Enclave Page Tables with Minimal Hardware

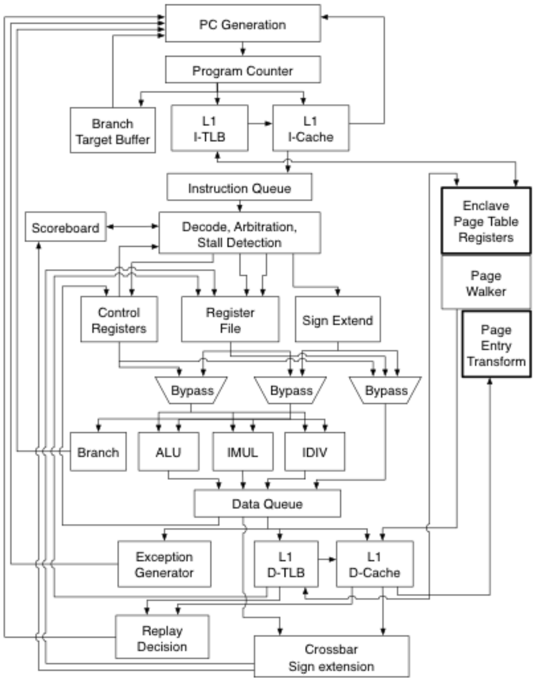
Enclave Page Table Registers



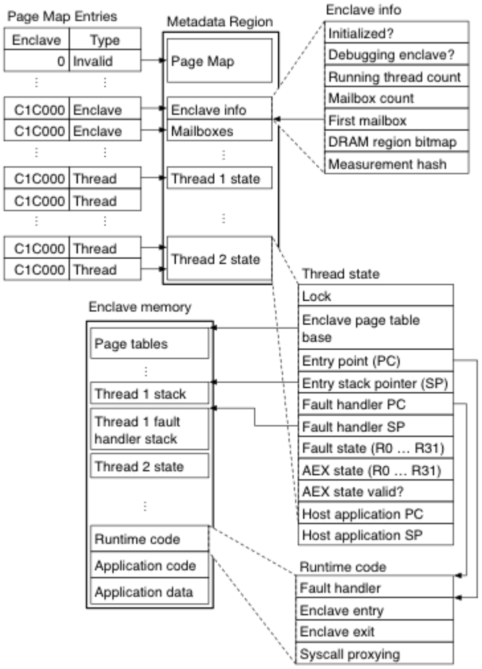
Per-Enclave Page Tables with Minimal Hardware: Page Entry Transform



Per-Enclave Page Tables with Minimal Hardware



Sanctum Enclaves: Memory Layout



Page Map Entries

Enclave	Type
0	Invalid
⋮	⋮
C1C000	Enclave
C1C000	Enclave
⋮	⋮
C1C000	Thread
C1C000	Thread
⋮	⋮
C1C000	Thread
C1C000	Thread

Metadata Region

- Page Map
- Enclave info
- Mailboxes
- ⋮
- Thread 1 state
- ⋮
- Thread 2 state

Enclave info

- Initialized?
- Debugging enclave?
- Running thread count
- Mailbox count
- First mailbox
- DRAM region bitmap
- Measurement hash

Thread state

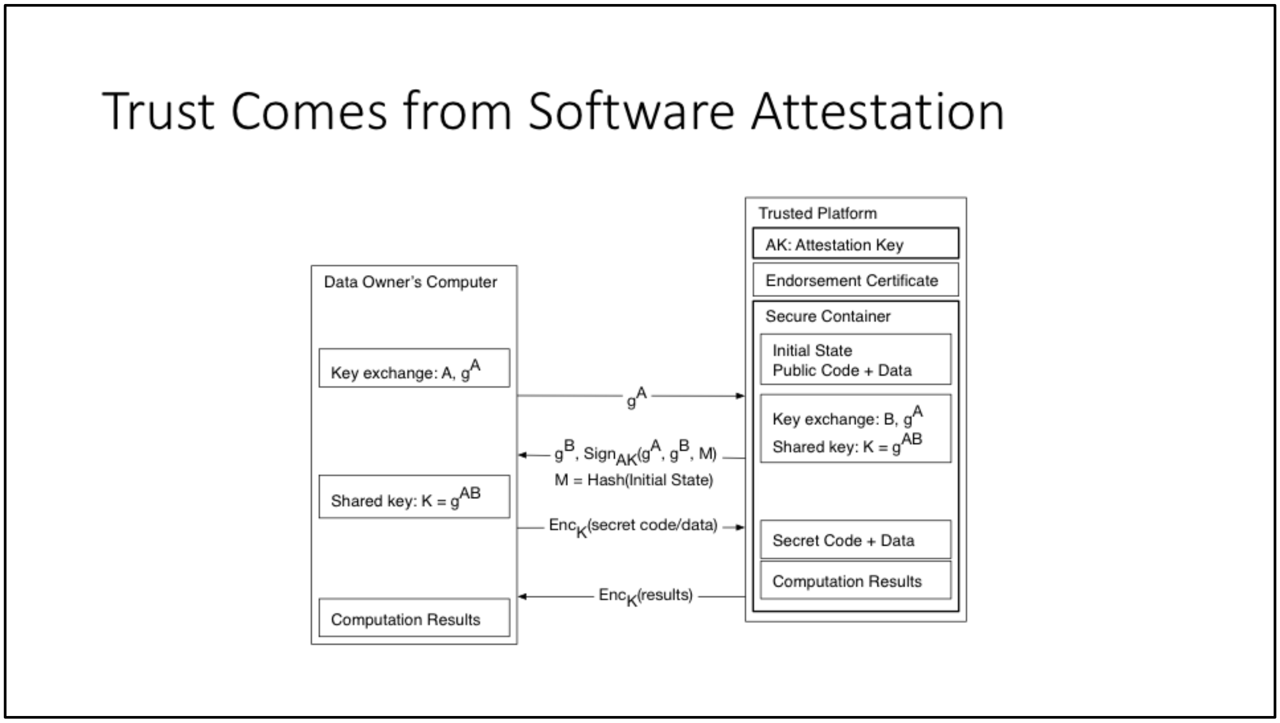
- Lock
- Enclave page table base
- Entry point (PC)
- Entry stack pointer (SP)
- Fault handler PC
- Fault handler SP
- Fault state (R0 … R31)
- AEX state (R0 … R31)
- AEX state valid?
- Host application PC
- Host application SP

Enclave memory

- Page tables
- ⋮
- Thread 1 stack
- Thread 1 fault handler stack
- Thread 2 state
- ⋮
- Runtime code
- Application code
- Application data

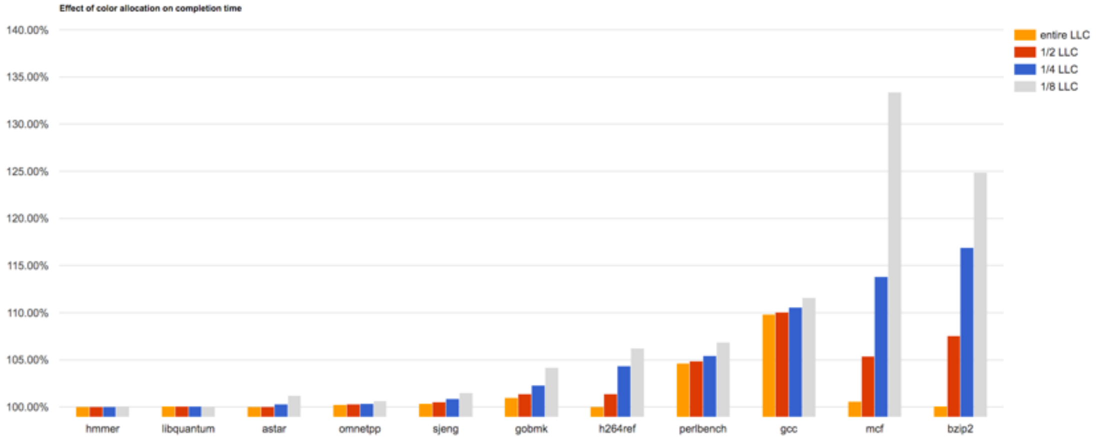
Runtime code

- Fault handler
- Enclave entry
- Enclave exit
- Syscall proxying



Trust Comes from Software Attestation

Sanctum Performance
Cache Partitioning Overhead


Effect of color allocation on completion time
hmmer
libquantum
astar
omnetpp
sjeng
gobmk
h264ref
perlbench
gcc
mcf
bzip2
entire LLC
1/2 LLC
1/4 LLC
1/8 LLC