Lecture 21 Notes: Komodo

Butler Lampson
MIT 6.826
November 17, 2020

Komodo: Using verification to disentangle secure-enclave hardware from software
Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, Bryan Parno
SOSP ’17, October 28, 2017, Shanghai, China

Background

Many of the authors were Ironfleet authors.

What is SGX for? Why initially not on server chips?
- DRM?
- Or maybe that Intel starts with desktop chips historically.

SGX issues
- Complex
- Hard to change
- Side channels/controlled channels, esp. via page faults.
  - Cache partitioning— to control side channels, don't share resources

Komodo goal: **minimal** hardware support

Security basics

Security = isolation + sharing

Isolation = secrecy (control data out) + integrity (control actions in)

Sharing = exercising control: **who** can do **what**

Who = *authentication*: who gets data / gives command — principals

People, programs, groups, channels

At runtime need secure channels in/out: wire, host, crypto

To *manage* security, need meaningful principals

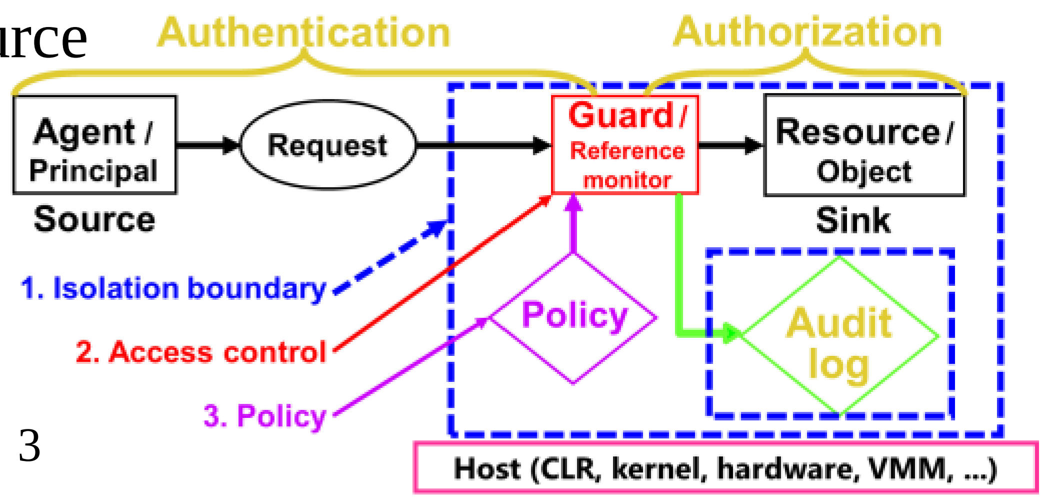
Connect them by the "speaks for" relation: $A \Rightarrow B$

If A says something, B says it too.

Handoff: **if** $A \Rightarrow B \;\; \wedge \;\; A$ **says** $(C \Rightarrow B)$ **then** $C \Rightarrow B$

What = *authorization*: what data / commands

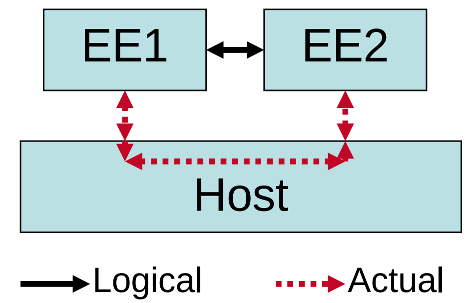
Channel ⇒ user/group ⇒ label/resource



Isolation mechanisms

Host creates n execution environments (EE)

- Separate machines—including co-processors (which usually fail)
- Enclave
- Hypervisor / VMM
- Operating system
- Browser



How does third party know what code is in an EE: **attestation**:

channel ⇒ code hash (measurement) ⇒ code name

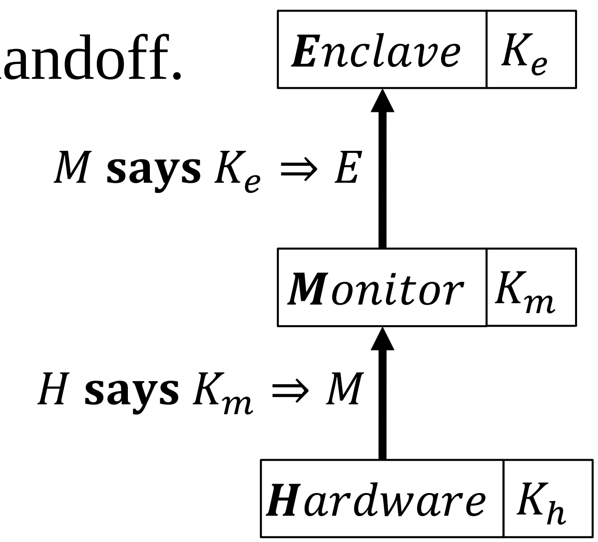
Host says channel (key) ⇒ code hash, policy says code hash ⇒ code name

Can do this **recursively**: A attests to B attests to C

HW **says** $K_m \Rightarrow M$, M **says** $K_e \Rightarrow$ enclave hash.

$Policy$ says $M \Rightarrow$ any enclave hash, so M can handoff.

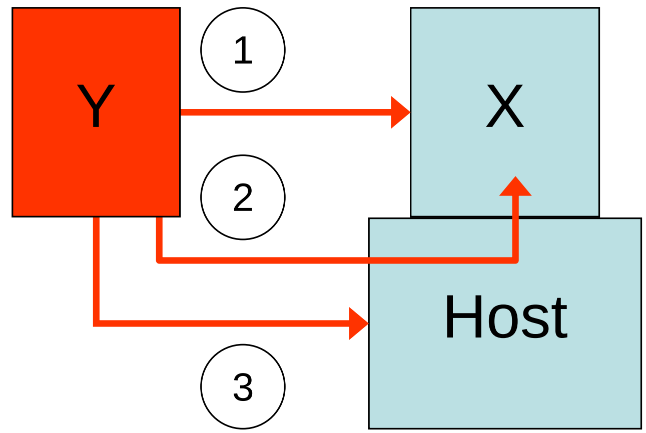
Also for different versions of M and E.



Vulnerabilities

How does the bad guy *Y* make it go wrong?

1. Send *X* some bad input, either directly or indirectly.
2. Use an unsafe function provided by *H* like a debugging interface.
3. Make *X*'s host *H* go bad.



Enclave

A program as principal needs isolation.
At machine level, host could be OS (very complex) or hypervisor (complex)

Idea: Replace hypervisor with hardware—less to trust
  Enclave should be “small”— small TCB
    but of course people push the boundary
  OS is the enemy
  Crypto for external services: storage, networking
  No **resource allocation**, including scheduling!

Uses for enclaves

Factor the application, secure the critical bits. Examples:

DRM,
secure signing
protect crypto keys
perhaps confidential computing.

Run the whole application, as if on a separate machine.

Competition: hypervisor, separate hardware
Much more demanding for the enclave host

Threat model

Assume all *software* outside the enclave is hostile
- In particular, the OS, as well as other enclaves
  - Cache sharing.
  - Power metering.
  - *Induced* faults:
    - Plundervolt: hack frequency/voltage)
    - Rowhammer: hack weak DRAM cells

Physical threats:
- Passive: snoop on busses, sense power, radiation, …
- Active (induce faults): power, temperature, light, alpha particles, …

SGX (Intel Software Guard eXtension)

Enclave implemented by

hardware for memory protection, exceptions, root key K_h, randomness;
microcode for enclave creation, entry/exit, attestation

Attacks:

Side channels as above (don’t share resources)
Bugs—microcode is complicated
Controlled channels—OS can see page faults

Memory data is encrypted, MACed in a Merkle tree
Addresses are visible to snooping

Komodo idea

Minimal hardware + monitor, instead of SGX's microcode.
  Disentangle essential hardware from software.
  Monitor does transfers, checking—no resource allocation
"Monitor" = baby hypervisor (but no multiplexing or I/O drivers)

Komodo does entering, leaving of enclaves.
OS builds enclaves, giving secure or insecure pages to monitor

Rely on hardware only for:
- **Secure memory** region for monitor and enclaves.
  - For SW threats, just protect some physical memory from OS and I/O.
  - For HW threats, SGX has memory encryption / Merkle tree.
- **Protected execution** for the monitor (in SGX, microcode) and enclave.
  - Secure control transfer in and out of monitor.
  - Isolated monitor and enclave state (memory, registers).
- A root of trust for **attestation**.
- **Randomness**.

Local attestation

`Attest(u32 data[8])→u32 mac[8]`

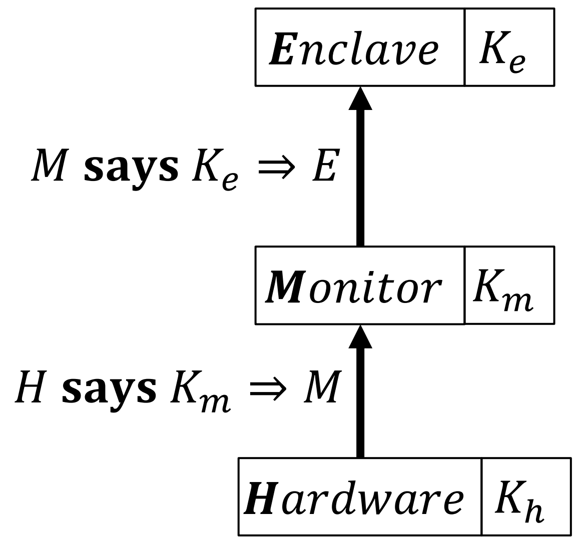
caller enclave **says** `data` ⇒ enclave's measure ms_e

Usually `data` is a signing key K_e, so caller is saying $K_e \Rightarrow ms_e$

This means that if K_e **says** x then ms_e **says** x

`Verify(u32 data[8], u32 measure[8], u32 mac[8])→ bool ok`

monitor **says** `measure` **says** `data` ⇒ `measure`



To convince an external third party

Monitor gets a root-of-trust key K_m from hardware

Hardware makes K_h **says** $K_m \Rightarrow ms_m$

Monitor makes K_m **says** $K_e \Rightarrow ms_e$

or delegates this task to a trusted enclave

that learns$K_e \Rightarrow ms_e$ from `verify`

Third party policy trusts K_h for $K_m \Rightarrow ms_m$ Why?

so it knows ms_m **says** $K_e \Rightarrow ms_e$

and needs to trust ms_m for ms_e

Komodo implementation

Prototype runs on ARM TrustZone
Must trust the hardware (and toolchain)
Formal verification for monitor software:

- spec ⇒ "client is isolated from other software"
  - Only enclave can modify its code or data
  - No bits in an enclave leak outside unless enclave reveals them
- code ⇒ spec

Non-interference

- Confidentiality: all public outputs are determined by public inputs
- Integrity: all trusted outputs are determined by trusted inputs

Komodo doesn't constrain what the enclave does.

"Local" attestation: monitor tells you the MAC of all the enclave code/data

TrustZone

A TrustZone processor runs in one of two worlds:

normal (where a regular OS and applications run), and

secure.

Control registers are banked (including MMU config and page table base).

(Physical memory protection is platform-specific.)

Monitor abstract state

$PageDB$ abstracts memory and threads

$PageNo$ (for secure memory) → (owning enclave, type, page contents).

Type is spare, data, page table, address space, thread (these are puns)

OS can populate a PT

Nothing modeled or proved about enclave behavior—specifically, can read/write unsecured memory.

You might want taint tracking and sanitizing, at least.

TCB

ARM model
Monitor spec (with consistency invariants)

- 12 monitor calls from outside OS
- 7 SVC calls from enclave

Verification tools (Dafny and Z3)
Assembler, linker, and bootloader

Verification

Monitor code uses ARM machine model as state machine.

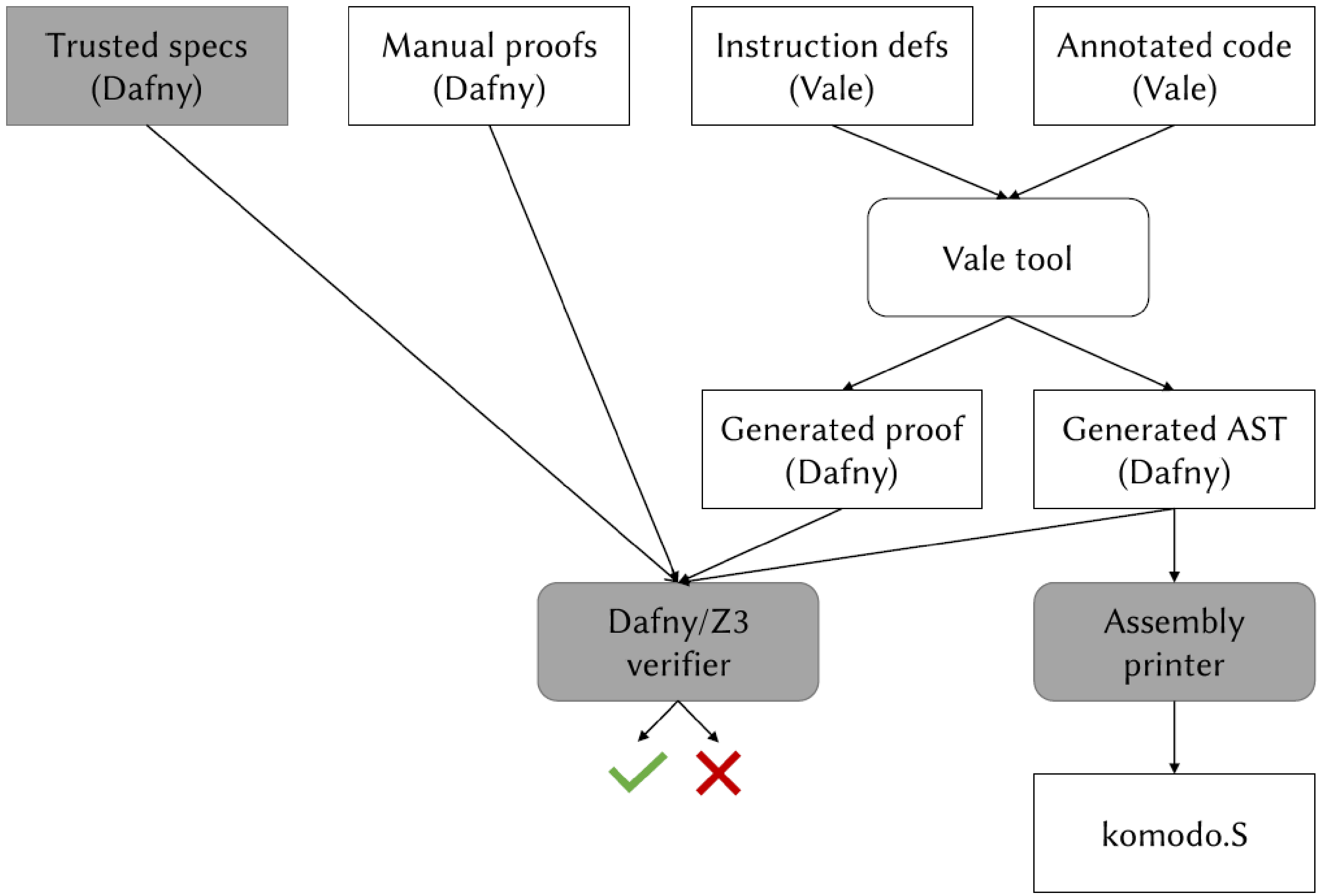
- State = everything visible: memory, registers (including banks)
  - Hack: **if**, **while**, call rather than PC changes
    - except for monitor ↔ enclave
- Exceptions: avoid by preconditions, except interrupts

Enclave code spec: trashes all accessible state, then raises an exception.

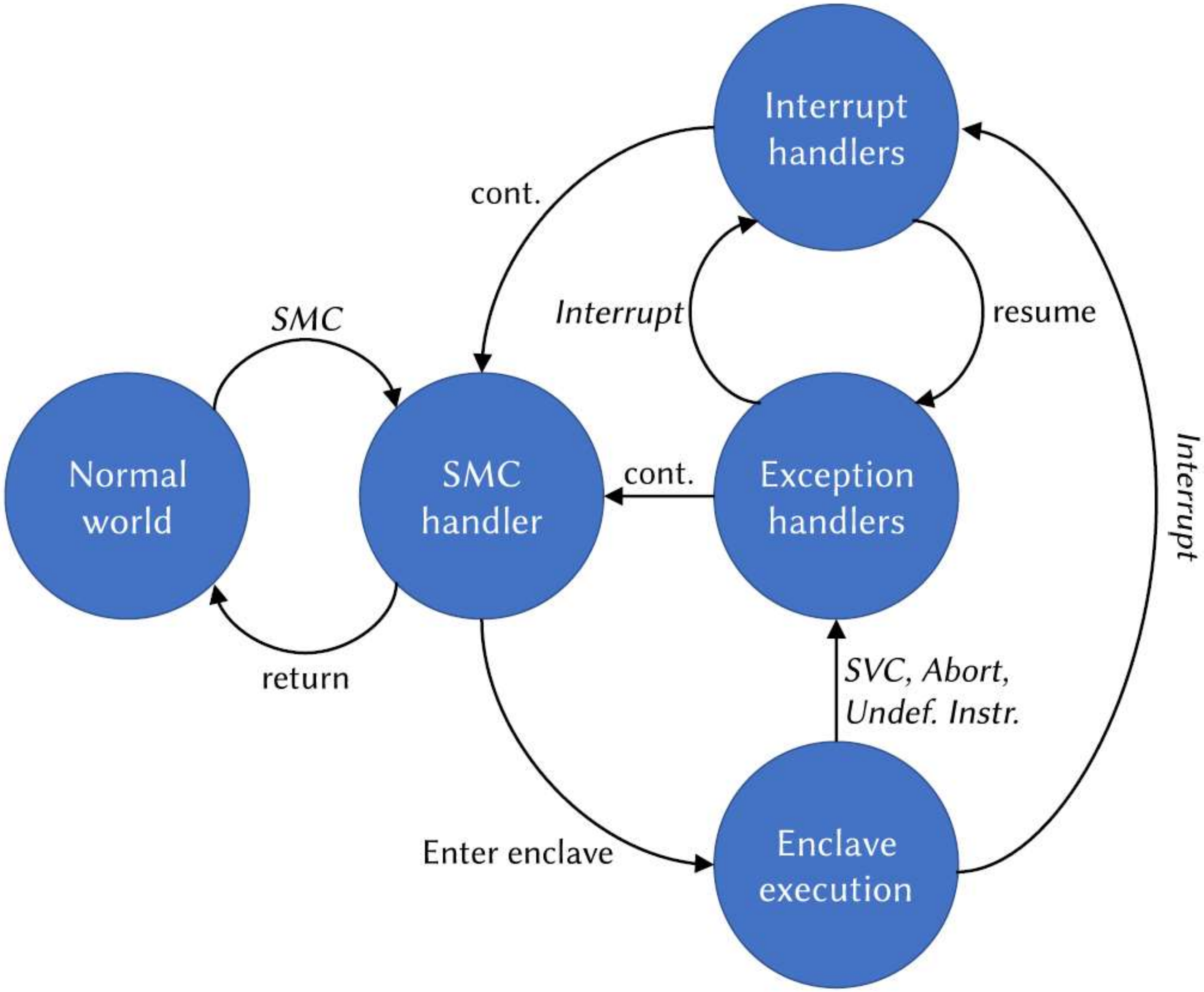
Idea: everything between two world transitions is a single atomic action

- Transitions are between two of enclave, monitor, and normal
- Transitions need not be deterministic
  - Modeled as an unknown (integer) seed

Verification flow



State machine for non-sequential execution



Top level spec

Top level spec: a $Next$ predicate describing the SMC handler.

predicate $smchandler(s: state, d: PageDb, s': state, d': PageDb)$

Relates the concrete machine and abstract $PageDB$ states (s and d)

just after taking an SMC exception from the OS,

to the final states (s' and d') just prior to returning:

Only two SMCs involve enclave execution: `Enter` and `Resume`.

The rest are pure functions $(PageDB, params) \rightarrow (PageDB, \text{OK?})$.

Enter and Resume

`Enter` and `Resume` also relate two states and $PageDB$s.

Spec forces the code to enter from a highly constrained state.

- PT base = enclave PT base.
- PT in memory matches abstract one in $PageDB$.
- TLB is consistent.
- Secure pages and registers have correct content.

Monitor code can do what it likes as long as it makes a correct state.

Non-interference

Secrecy : publicly observable outputs depend only on observable inputs
Integrity: trusted outputs depend only on trusted inputs

How? Define an "observably equivalent" relation: $\approx_{adv}$.
If the initial states of two executions are related, so are the final states.

The nondeterminism of enclave execution is modeled with an oracle,
an unknown integer seed (same idea as step objects in Armada).

Declassification

Violations of non-interference:

- Type of exception from enclave
- Return value from enclave `Exit`
- Which pages are allocated from spare or returned to spare

The axioms that allow this are part of the TCB

Lessons (from the paper)

Need verification: Even a small code base has bugs

“Trusted” code can have bugs. Really means “untrustworthy”.

Tools can get better.

- Failed verifications are hard to debug

Opaque functions are good, to guide the prover.

From https://medium.com/corda/intro-to-sgx-from-http-to-enclaves-1bf38a3bf595

How can we verify that such a signature over a report comes from a genuine Intel chip? We can't. But Intel can, and this is what their Intel Attestation Service is for. They have a REST API to send such signed reports to, and if the report is valid and signed by a genuine Intel CPU then the IAS will reply with an OK, signed with Intel's root key.

Note: this is a simplification, the real protocol is more complex and includes an additional "EPID provisioning" step, the CPU key isn't used directly.

From the paper

Decouple the core hardware mechanisms such as memory encryption, address-space isolation and [minimal] attestation from the management thereof, which Komodo delegates to a privileged software monitor. We show that the approach is practical and performant with a concrete implementation of a prototype in verified assembly code on ARM TrustZone.
What distinguishes SGX is memory encryption, independence from a large untrusted OS, and the folklore intuition that hardware is more secure than software. Komodo replaces folklore with formal verification. Komodo is implemented as a software reference monitor in verified assembly code.

The SGX implementation consists of three components:
(i) encryption and integrity protection for a static region of physical memory by an encryption engine in the memory controller,
(ii) a set of instructions to mess with enclaves, and
(iii) changes to the processor's TLB miss and exception handling procedures that enforce enclave protections on access to the encrypted memory region.

Although it has no direct access to encrypted pages, the OS allocates and maps them to enclaves, and although it cannot directly manipulate an enclave's register state, the OS chooses when, and on which CPUs, to execute enclave threads.
There are SGX instructions that manipulate the enclave page cache map (EPCM) which stores metadata for every encrypted page, including its allocation state, type, owning enclave, permissions, and virtual address. Effectively a reverse map of encrypted pages, the EPCM is also consulted on a TLB miss to enforce enclave protections on memory—every page table mapping must be consistent with the EPCM.

Threat model

Like SGX, we seek to protect the confidentiality and integrity of user-mode code in an enclave from an attacker who has full control over a platform's privileged software (OS and hypervisor). Two variants: physical attacks on memory in scope or not. If so, the attacker may access any RAM external to the CPU package. This includes bus snooping and cold-boot [36] attacks.

Primitives

We rely on five hardware primitives:

- Isolated memory for monitor code/data and enclave pages, protected by crypto against physical attacks (or on-chip for small enclaves). Else just IOMMU.
- Protected execution for the monitor. In SGX, microcode. Could be DEC Alpha PALcode, RISC-V machine mode, secure monitor mode of ARM TrustZone.
  - Secure control transfer between monitor code and normal execution
  - Protection against unprogrammed control transfers in monitor code or access to its registers. **Not** another (costly) layer of memory translation.
- Protected execution for enclaves. A typical user mode is OK, if protected from the OS.
  - A TrustZone processor runs in one of two worlds: normal for a regular OS and applications run, and secure. Control registers are banked.

- A root of trust for attestation. Either hardware or an early bootloader attests to a secure hash of the monitor. The monitor in turn implements enclave attestation.
- A source of randomness.

Strategy: build enclave bit by bit, to minimize monitor complexity. Some Hyperkernel strategies. Then finalize before running.

High-level invariants on spec: it maintains consistency invariants on page state (described in §5.2) and that it guarantees enclave confidentiality and integrity (§6).

Have a formal model of ARM: core registers R0–R12, stack pointer (SP), link register (LR), portions of the current and saved program status registers (CPSR and SPSRs), privilege modes, control flow, interrupts, exceptions, and semantics of 25 instructions.
No PC, instead if, while, call. But do model monitor entry and exit.
Model VM explicitly as part of load/store.

User mode execution is modeled as `havoc`; don't prove anything about user code, just that it can't mess up the monitor.
TLB consistency: …

Dynamic allocation: *Spare* pages to or from the OS, and enclave can map them.

A Komodo attestation is a message authentication code (MAC) using a secret key generated at boot from a cryptographically secure source of randomness. The MAC is computed over (i) the attesting enclave's measurement, and (ii) enclave-provided data, which may be used to bind a public key-pair to the enclave and hence bootstrap encrypted communication with code outside the enclave [56].