Rethinking Isolation Mechanisms for Datacenter Multitenancy

Varun Gandhi and James Mickens

Harvard University

Trusted Execution Environments: Intel SGX

Innovative Instructions and Software Model for Isolated Execution

Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi,
Vedvyas Shanbhogue and Uday Savagaonkar
Intel Corporation

{frank.mckeen, ilya.alexandrovich, alex.berenzon, carlos.v.rozas, his
vedvyas.shanbhogue, uday.savagaonkar }@intel.com

ABSTRACT

For years the PC community has struggled to provide secure solutions on open platforms. Intel has developed innovative new technology to enable SW developers to develop and deploy secure applications on open platforms. The technology enables applications to execute with confidentiality and integrity in the native OS environment. It does this by providing ISA extensions for generating hardware enforceable containers at a granularity determined by the developer. These containers while opaque to the operating system are managed by the OS. This paper analyzes the threats and attacks to applications. It then describes the ISA extension for generating a HW based container. Finally it describes the programming model of this container.

1 INTRODUCTION

Today's computer systems handle increasing amounts of important, sensitive, and valuable information. This information must be protected from tampering and theft. An entire industry is devoted to stealing information such as banking data or corporate intellectual property from systems [1]. There are many applications which must keep a secret on a platform. Some example applications are financial programs, ebanking, and medical records programs. Each secret holder may be mutually distrustful of other secret holders. Each secret must be protected independently of the other secrets. This paper describes Intel® Software Guard Extensions, (Intel® SGX), a set of new instructions and memory access changes added to the Intel® Architecture. These extensions allow an application to instantiate a protected container, referred to as an enclave. An enclave is a protected area in the application's address space, Figure 1, which provides confidentiality and integrity even in the presence of privileged malware. Attempted accesses to the enclave memory area from software not resident in the enclave are prevented even from privileged software such as virtual machine monitors, BIOS, or operating systems.

SGX allows the protected portion of an application to be distributed in the clear. Before the enclave is built the enclave code and data is free for inspection and analysis. The protected portion is loaded into an enclave where its code and data is measured. Once the application's code and data is loaded into an enclave, it is protected against all external software access. An application can prove its identity to a remote party and be securely provisioned with keys and credentials. The application can also request an enclave & platform specific key that it can use to protect keys and data that it wishes to store outside the enclave.

In addition to the security properties, the enclave environment offers scalability and performance associated with

execution on the main CPU of an ope

Supporting SGX involves two
Architecture. First is the change
semantics. The second is protection

This paper is divided into seve
provide an overview of the SGX
describes the SGX instruction set an
describes the hardware components u
an application. Section 5 describes t
Section 6 describes how an applicatio
enclave. Section 7 describes how enc
protected memory to allow for mult
Finally, in section 8, we summarize
this technology contains novel enhan
in open systems.

The SGX architecture also
architecture for remote attestation and
[2]. In addition some important usage
described in [3].

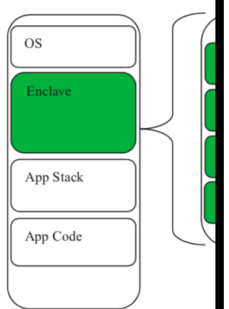


Figure 1: Enclave within Applicati

2 Protection Overview

SGX prevents all other software from
located inside an enclave including
from other enclaves. Attempts to mod
detected and either prevented or exec
of security properties are:

- SGX provides detection o

Intel SGX Explained

Victor Costan and Srinivas Devadas
victor@costan.us, devadas@mit.edu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

ABSTRACT

Intel's Software Guard Extensions (SGX) is a set of extensions to the Intel architecture that aims to provide integrity and confidentiality guarantees to security-sensitive computation performed on a computer where all the privileged software (kernel, hypervisor, etc) is potentially malicious.

This paper analyzes Intel SGX, based on the 3 papers [14, 78, 137] that introduced it, on the Intel Software Developer's Manual [100] (which supersedes the SGX manuals [94, 98]), on an ISCA 2015 tutorial [102], and on two patents [108, 136]. We use the papers, reference manuals, and tutorial as primary data sources, and only draw on the patents to fill in missing information.

This paper's contributions are a summary of the Intel-specific architectural and micro-architectural details needed to understand SGX, a detailed and structured presentation of the publicly available information on SGX, a series of intelligent guesses about some important but undocumented aspects of SGX, and an analysis of SGX's security properties.

1 OVERVIEW

Secure remote computation (Figure 1) is the problem of executing software on a remote computer **owned and maintained by an untrusted party**, with some integrity and confidentiality guarantees. In the general setting, secure remote computation is an unsolved problem. Fully Homomorphic Encryption [61] solves the problem for a limited family of computations, but has an impractical performance overhead [138].

Intel's Software Guard Extensions (SGX) is the latest iteration in a long line of trusted computing (Figure 2) designs, which aim to solve the secure remote computation problem by leveraging trusted hardware in the remote computer. The trusted hardware establishes a secure container, and the remote computation service user uploads the desired computation and data into the secure container. The trusted hardware protects the data's confidentiality and integrity while the computation is being

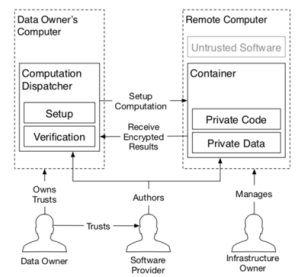


Figure 1: Secure remote computation. A user relies on a remote computer, owned by an untrusted party, to perform some computation on her data. The user has some assurance of the computation's integrity and confidentiality.

performed on it.

SGX relies on *software attestation*, like its predecessors, the TPM [71] and TXT [70]. Attestation (Figure 3) proves to a user that she is communicating with a specific piece of software running in a secure container hosted by the trusted hardware. The proof is a cryptographic signature that certifies the hash of the secure container's contents. It follows that the remote computer's owner can load any software in a secure container, but the remote computation service user will refuse to load her data into a secure container whose contents' hash does not match the expected value.

The remote computation service user verifies the *attestation key* used to produce the signature against an *endorsement certificate* created by the trusted hardware's manufacturer. The certificate states that the attestation key is only known to the trusted hardware, and only used for the purpose of attestation.

SGX stands out from its predecessors by the amount of code covered by the attestation, which is in the Trusted

30% CPU

MELTDOWN

15% CPU

SPECTRE

50% CPU

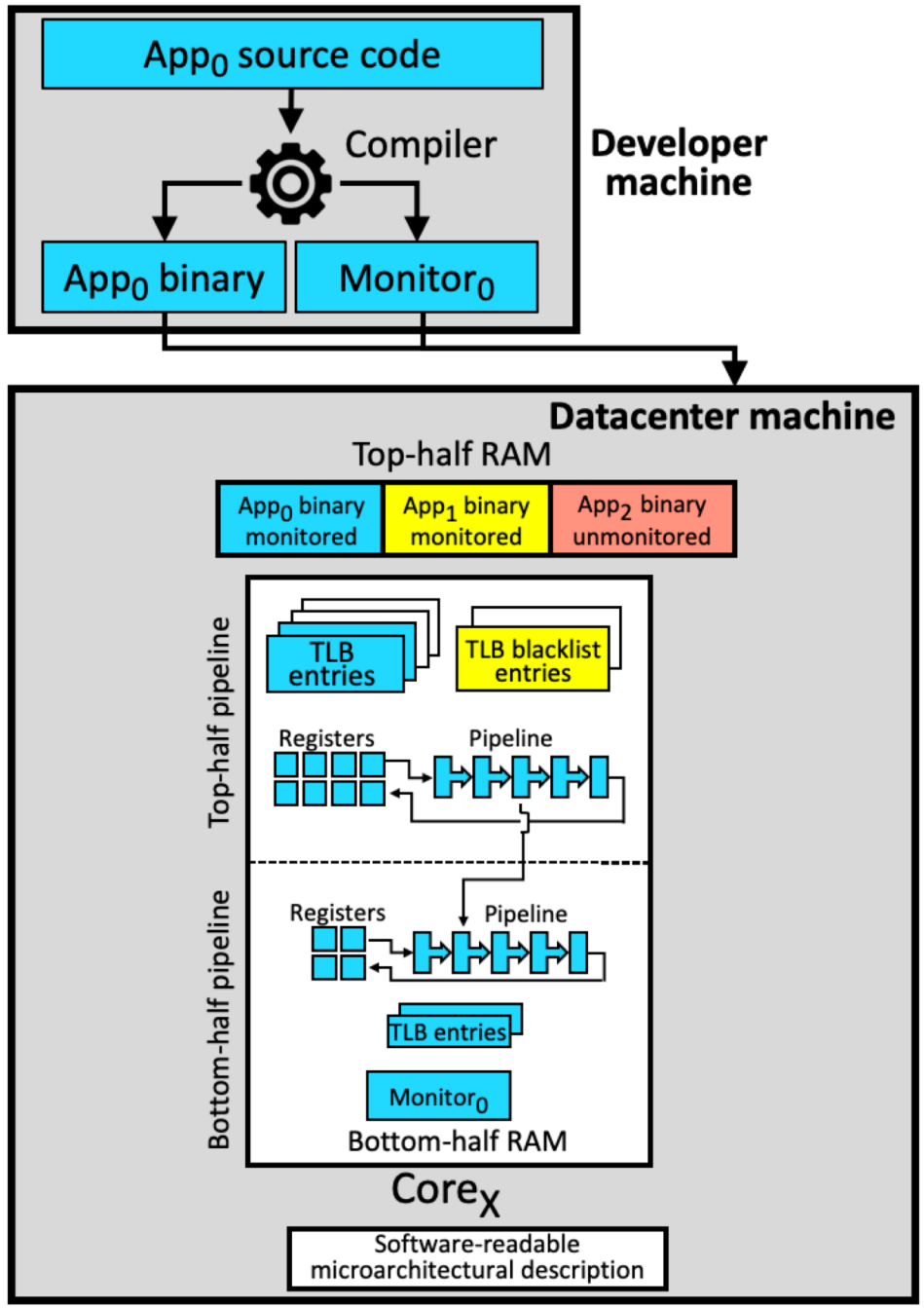
FORESHADOW

Deficiencies of SGX

- Weak Isolation
  - Tenants share micro-architectural state in SGX
- Post-load Integrity
  - SGX incapable of verifying post-load integrity, e.g., CFI
- Opaqueness
  - Poor vulnerability management
  - Microcode patches often incorrect or incomplete

Our Research: Isolated Monitor Execution

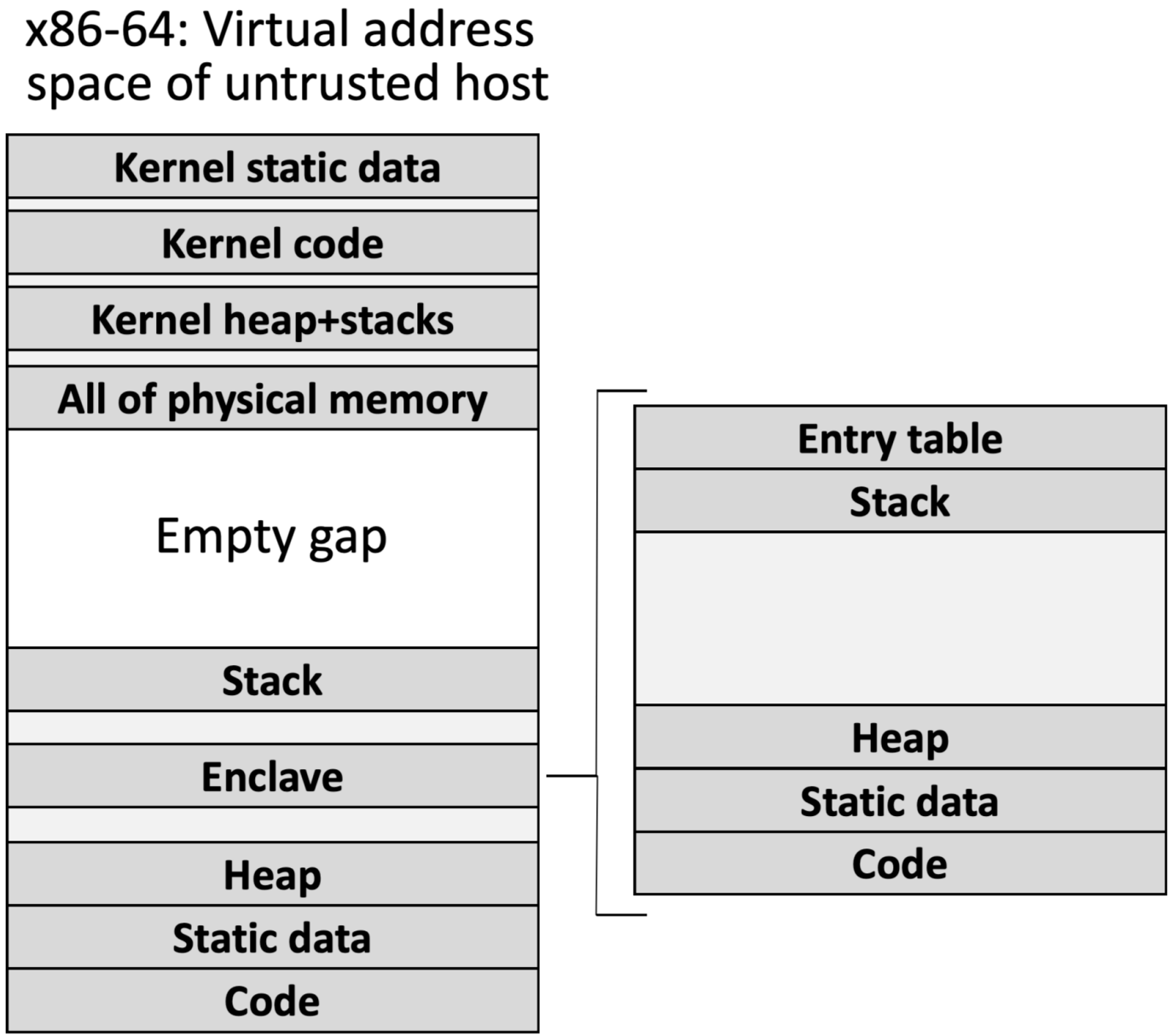
- Enforce dynamic security invariants (e.g., CFI) on the application
- Strong microarchitectural isolation
- Software-readable description of microarchitecture



Outline

- SGX Overview
- IME Design
- Related Work
- Open Challenges

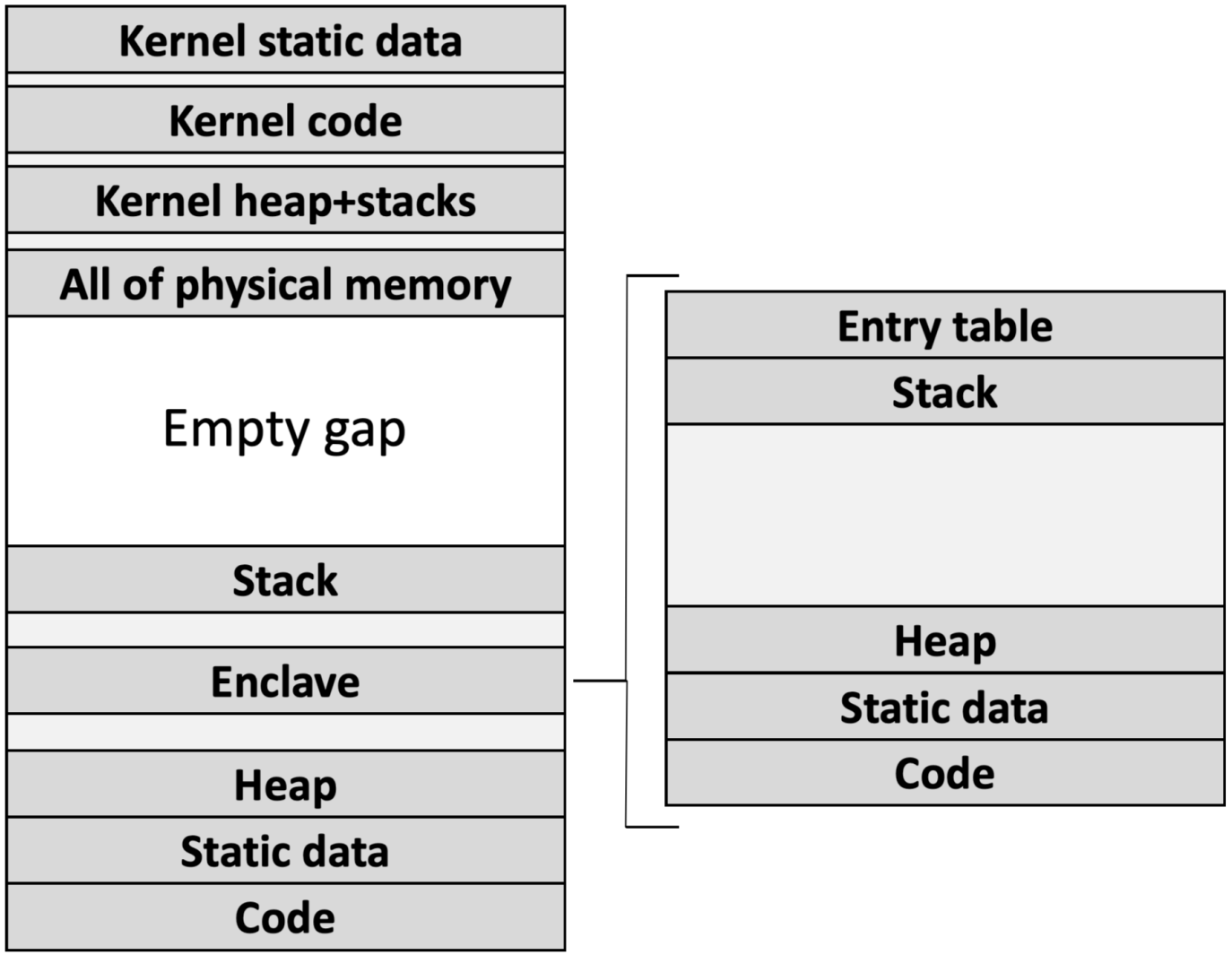
Overview of SGX



- Untrusted host process embeds an enclave
  - Cannot access enclave pages: reads return -1, writes are ignored
  - Jumps to enclave code using EENTER
- Enclave code runs at Ring 3 (i.e., the least-privileged level)
  - However, can access the entire address space of host
  - Cannot issue syscall or int instructions: relies on host for IO!
    - Data for a write IO must be placed in memory accessible to untrusted host
    - Data from a read IO must be pulled from memory accessible to untrusted host
  - Returns to untrusted host using EEXIT instruction

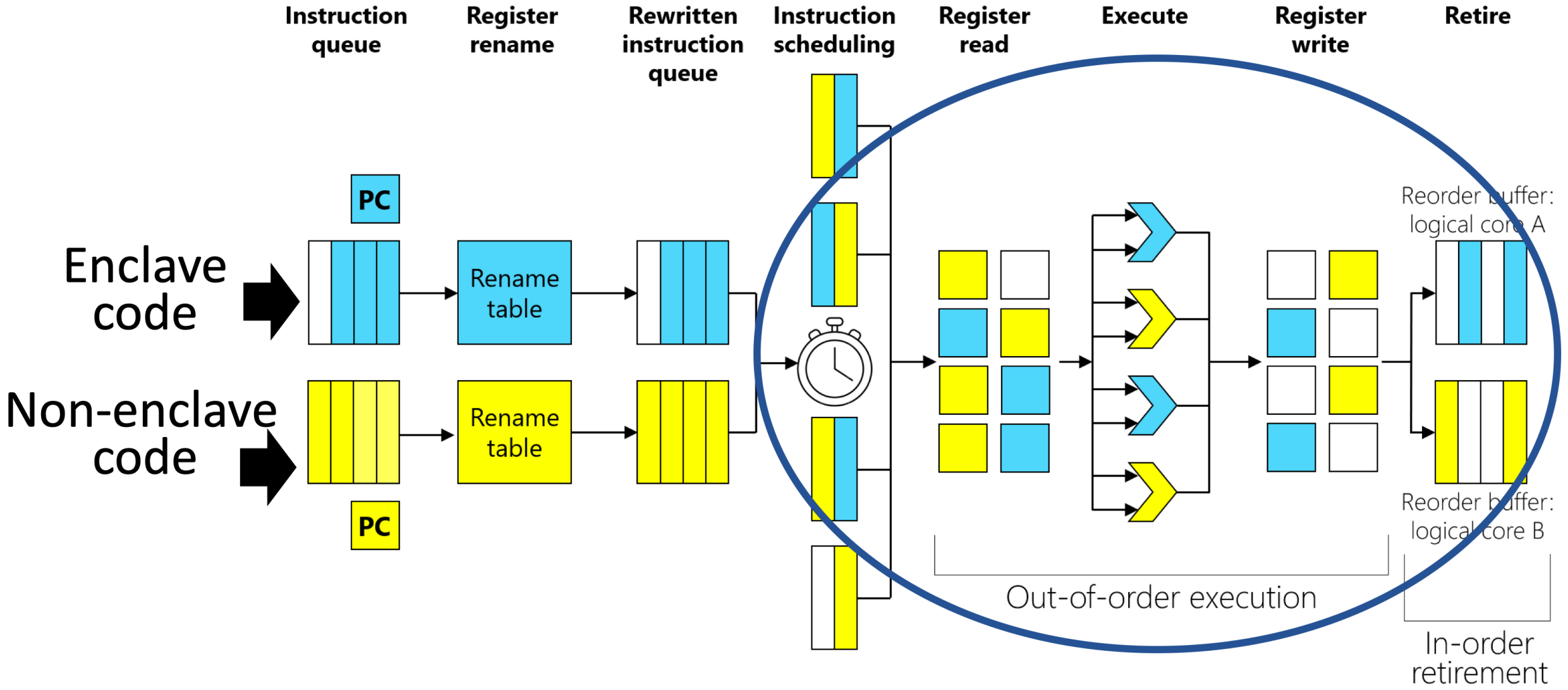
Overview of SGX

x86-64: Virtual address space of untrusted host



- EPCM is an SGX structure
  - Contains one metadata entry for each page in the EPC
  - Can only be modified by SGX instructions
  - Consulted during memory accesses to prevent EPC pages from unauthorized access
- Memory Encryption Engine
  - Adds counters, MACs, and encryption to outgoing memory writes to EPC
  - For incoming reads of EPC, decrypts data and then checks it for integrity and freshness (i.e., no rollbacks)

SGX is Vulnerable to Hyperthreading Side Channels!



SGX is Vulnerable to Cache Side Channels!

- Meltdown showed that, due to out-of-order execution, there are race conditions between:
  - A memory access bringing a value into the cache, and that value being computed on
  - The permission checks on that memory access
- An untrusted host embeds an enclave . . . so the enclave and the untrusted host share the same L1 cache!
  - Untrusted host can:
    - Load enclave
    - Wait for enclave to bring a cleartext secret byte into the L1 cache
    - Leverage Meltdown-style probe array to read the value of the byte
  - Secret to leak must be in L1 cache because the race condition is only winnable if the secret is in L1, not L2 or L3 → L1 terminal fault [Foreshadow Paper]

Outline

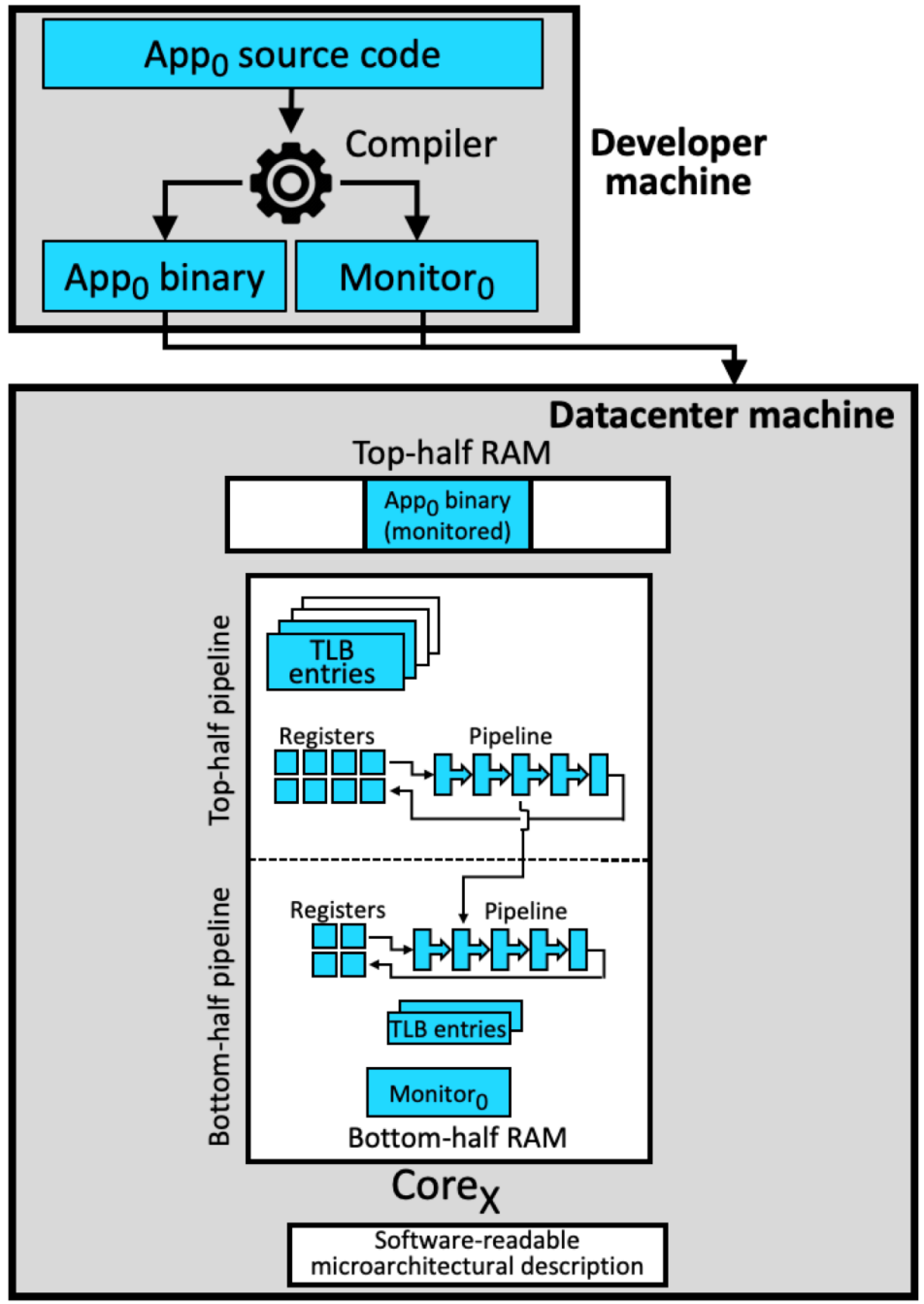
- SGX Overview
- IME Design
- Security Monitor Policies
- Related Work
- Open Challenges

IME Design: High-level Idea

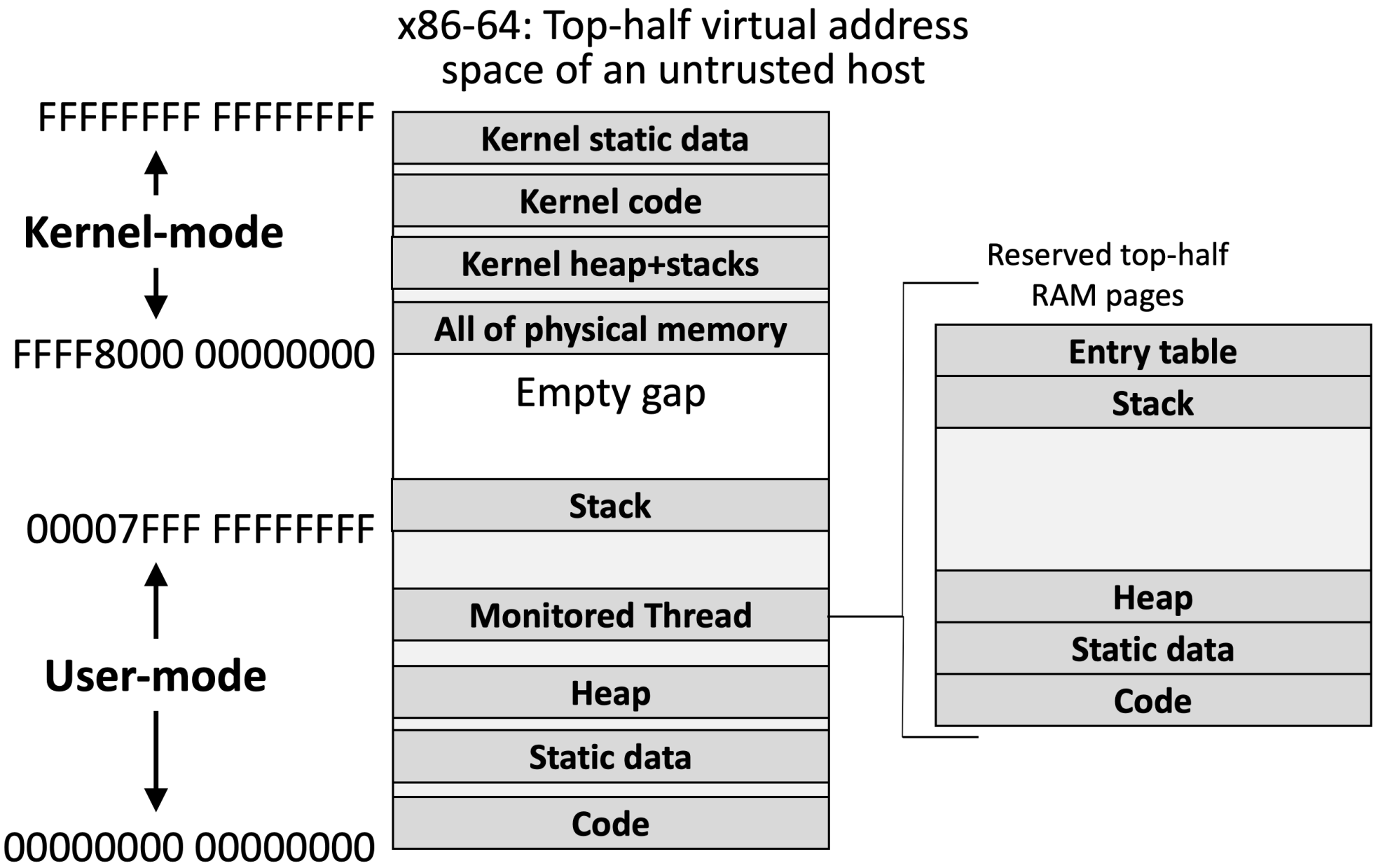
- Dyad
  - Monitored application thread
  - Monitor: A dynamic integrity checker

- Monitor consists of software-defined security checks that run in the bottom-half pipeline

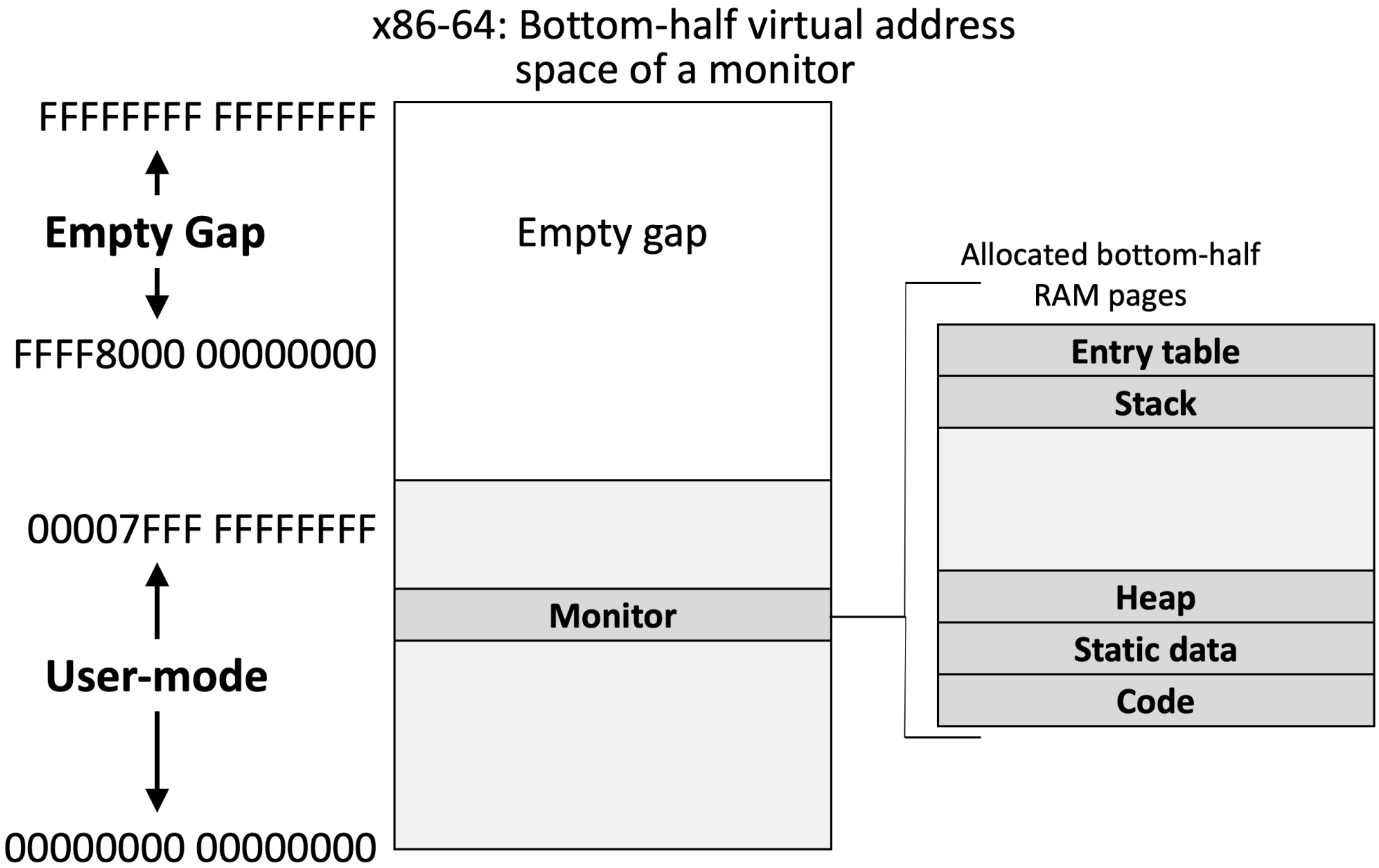
- Bottom-half CPU can
  - Read top-half instruction sequence and register state
  - Stall/resume and raise exception in the top-half pipeline



IME Design: Address space layout

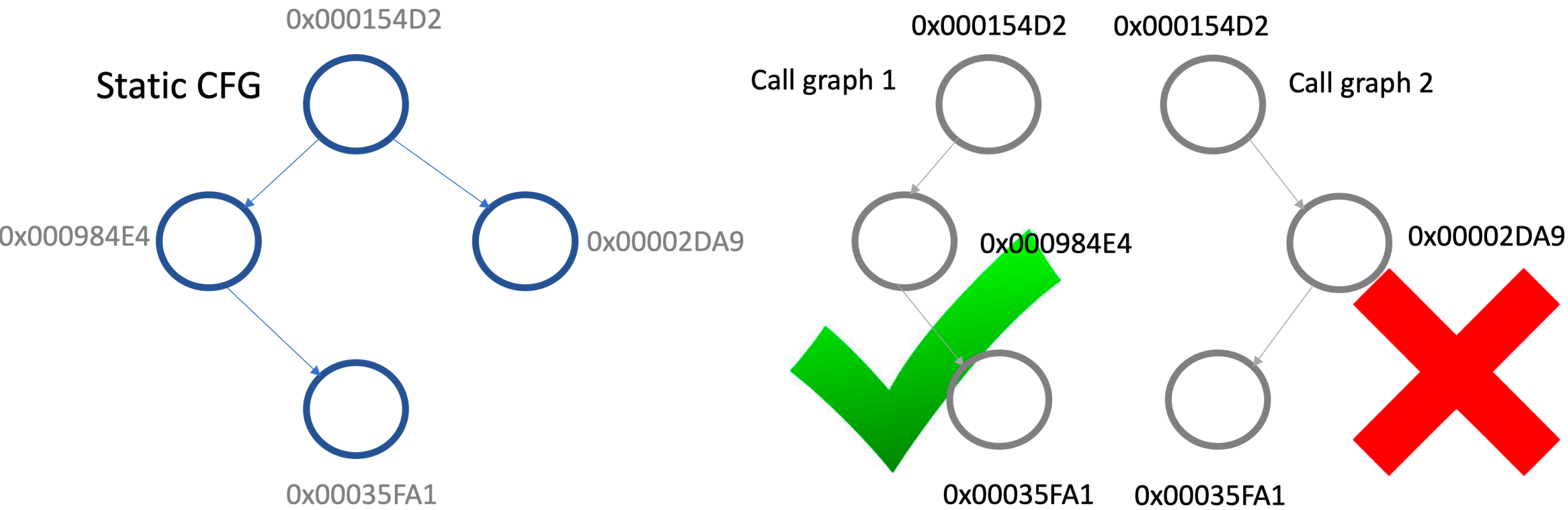


IME Design: Address space layout



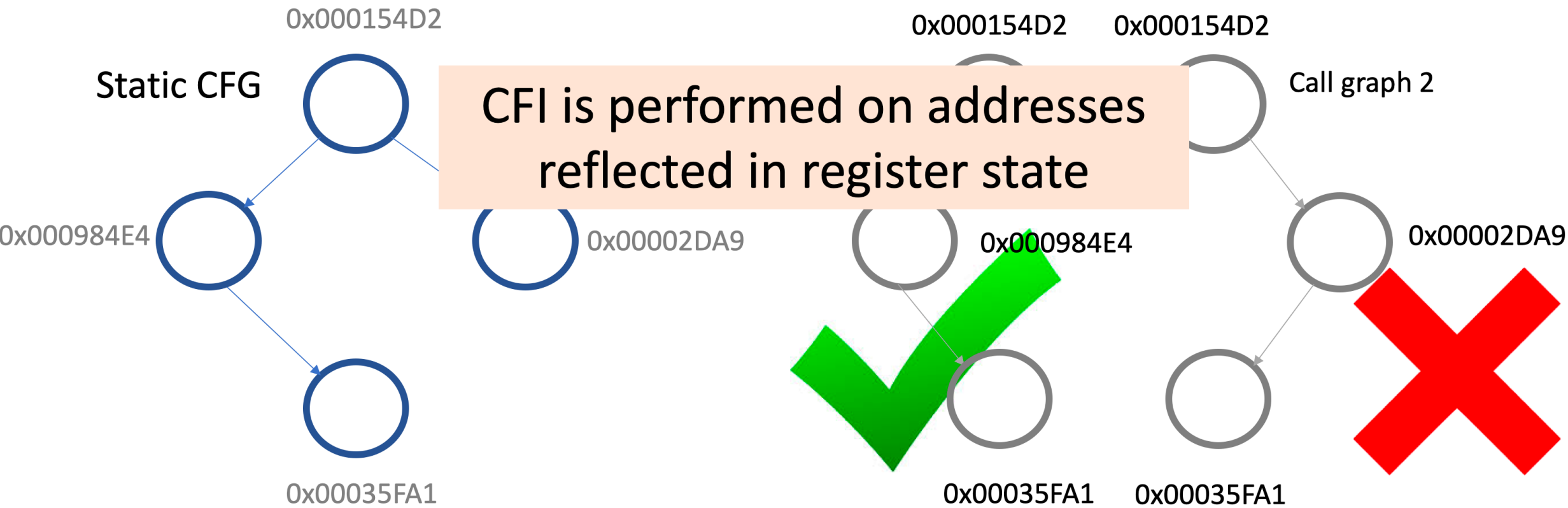
IME: Control Flow Integrity

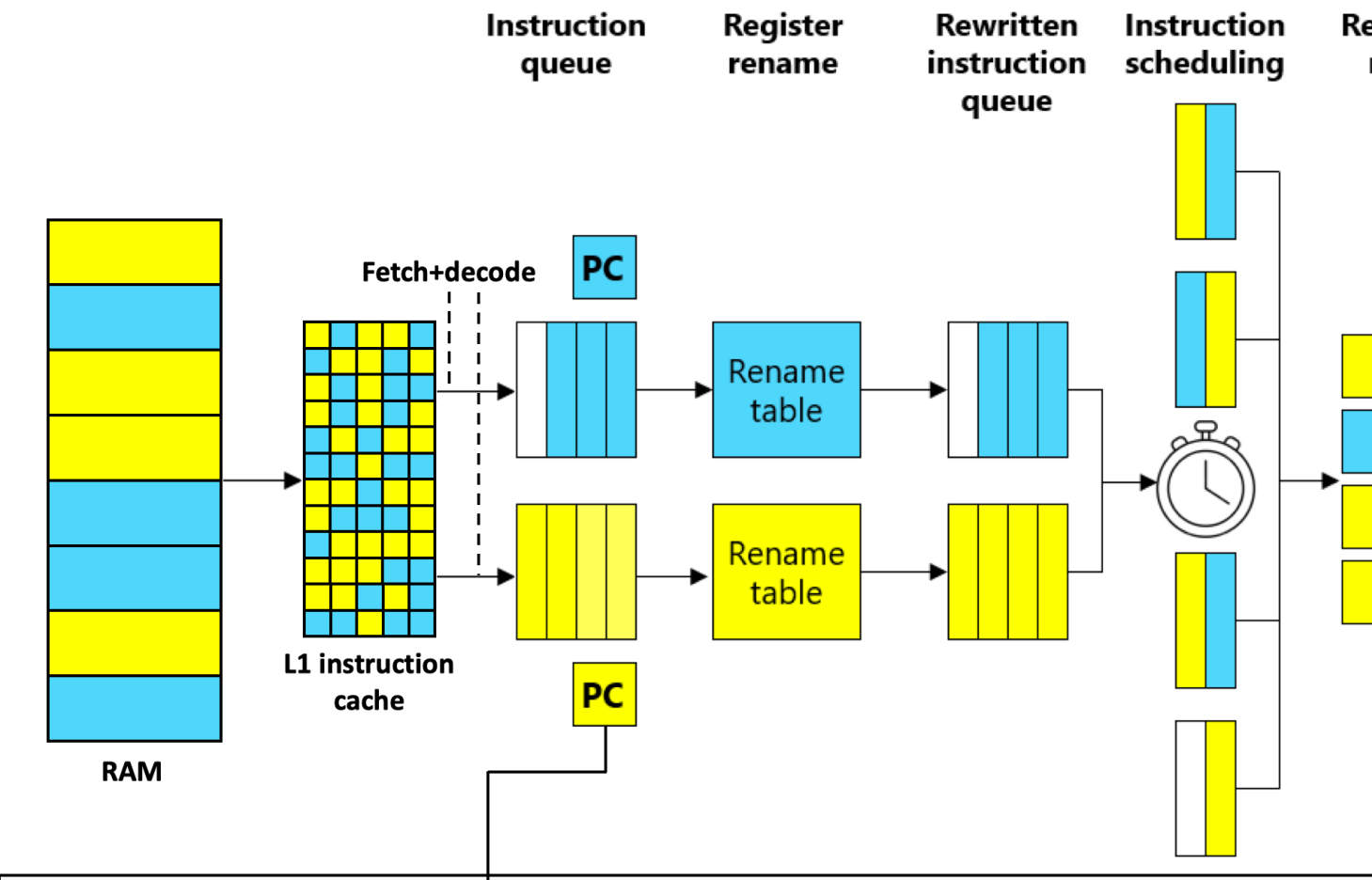
Control flow integrity dictates that software execution follow a path determined by a control flow graph (CFG)

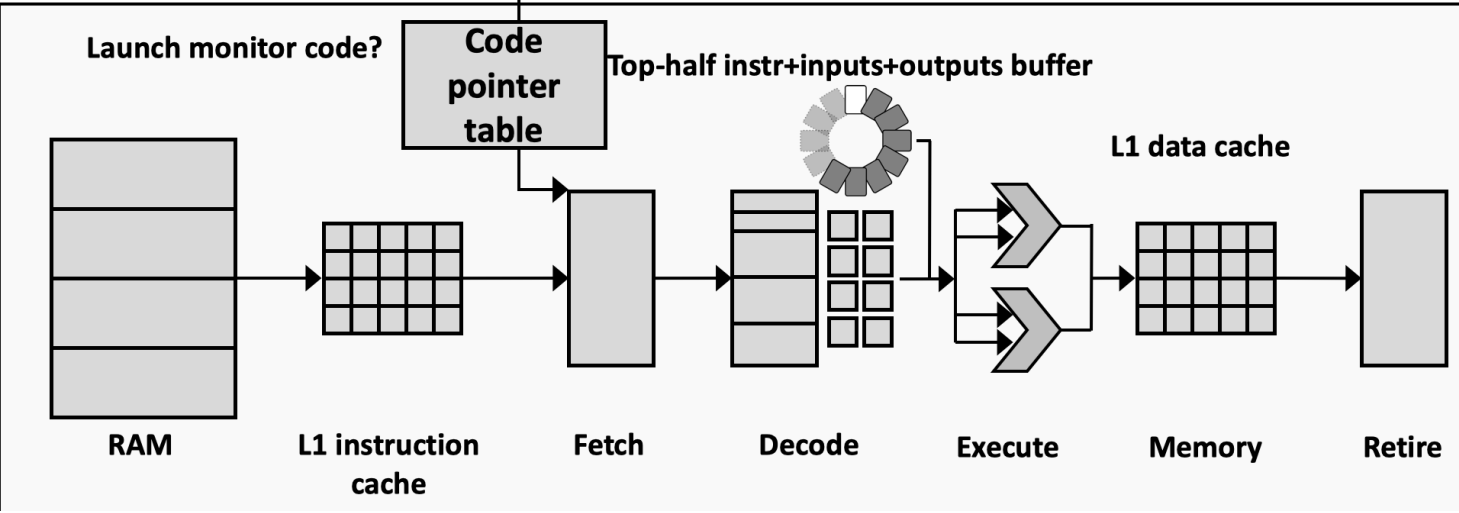


IME: Control Flow Integrity

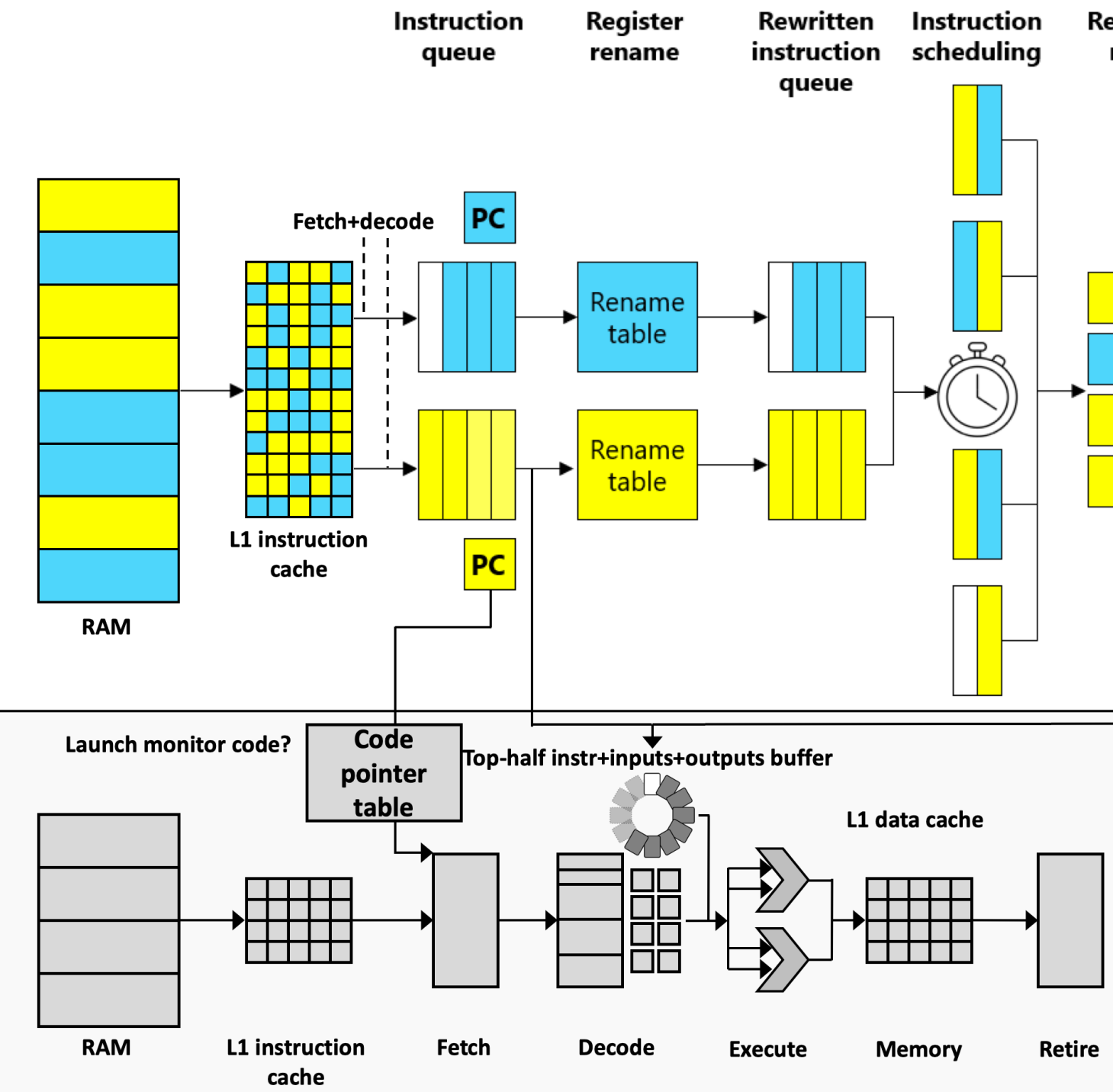
Control flow integrity dictates that software execution follow a path determined by a control flow graph (CFG)



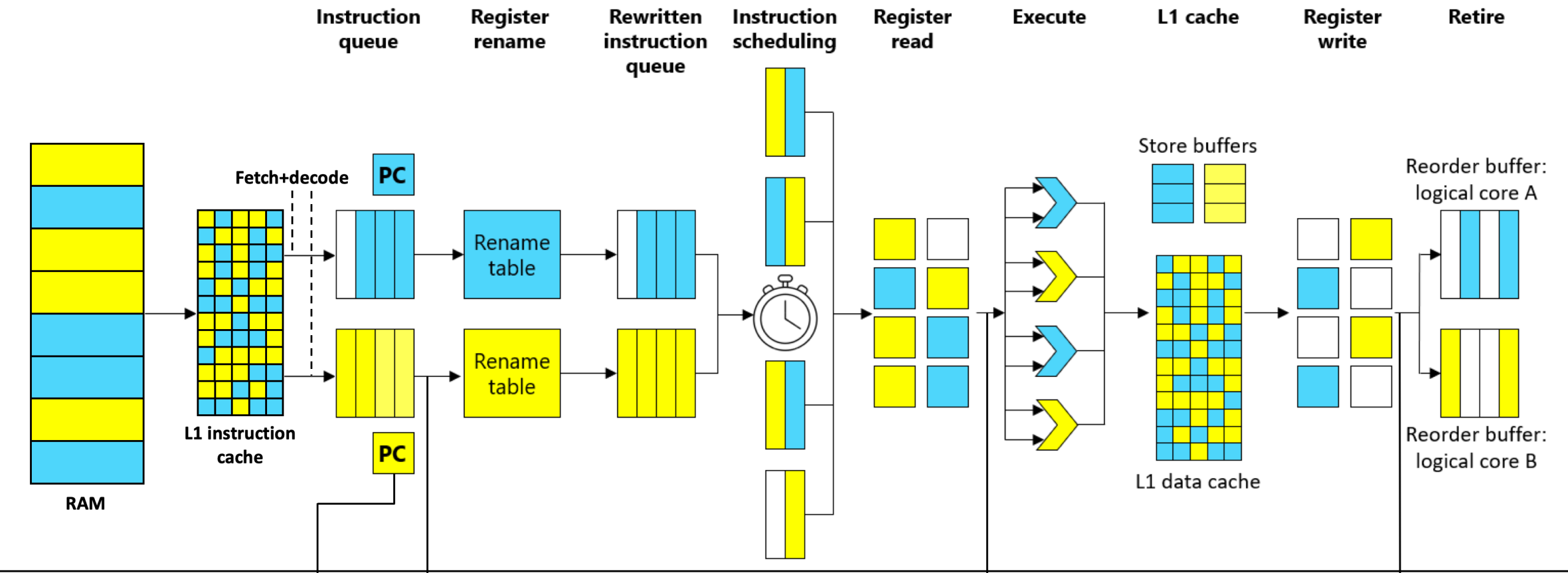


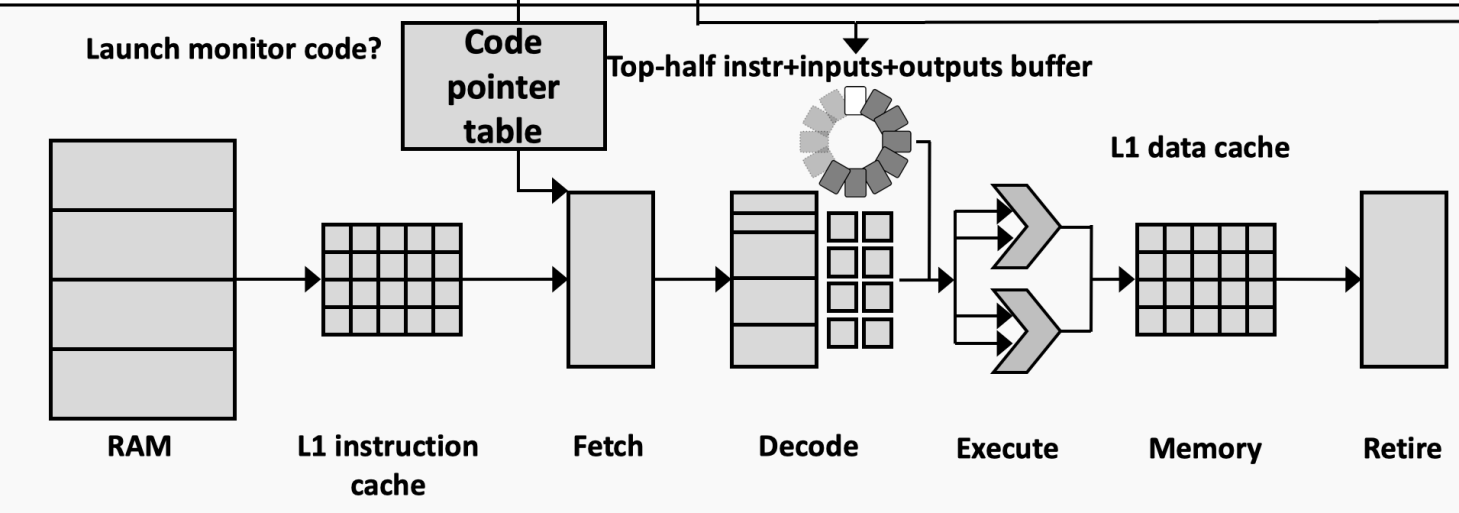


- Developers provide a code pointer table
  - Key: A top-half PC value that should trigger bottom-half monitoring
  - Value: The bottom-half PC of the monitor function
- Hits in the table wake up the bottom-half CPU to execute CFG checks

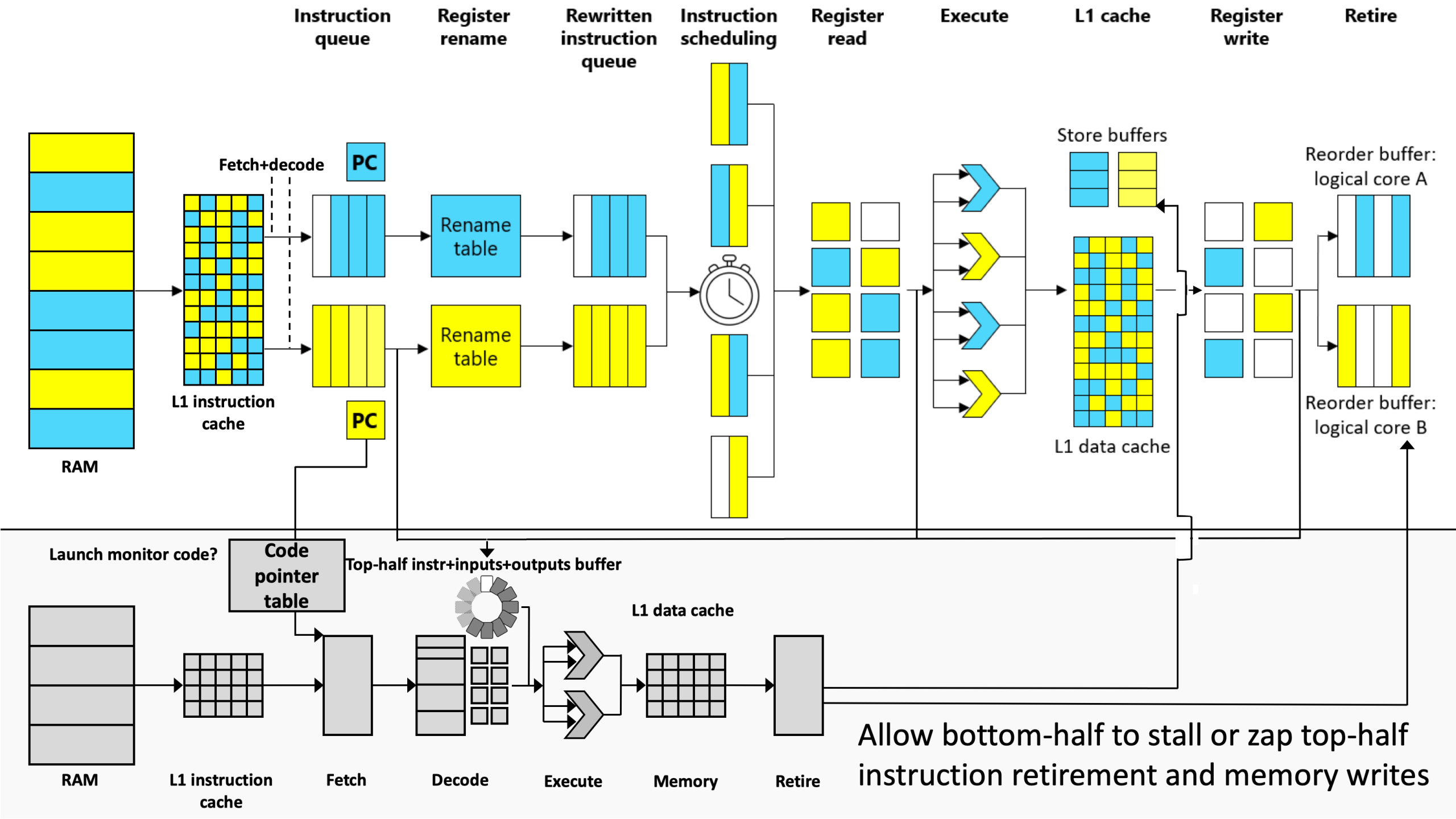


- Bottom-half HW copies the top-half instruction stream into a circular buffer
- Bottom-half ISA defines instructions that can operate on bottom-half state and top-half stream state

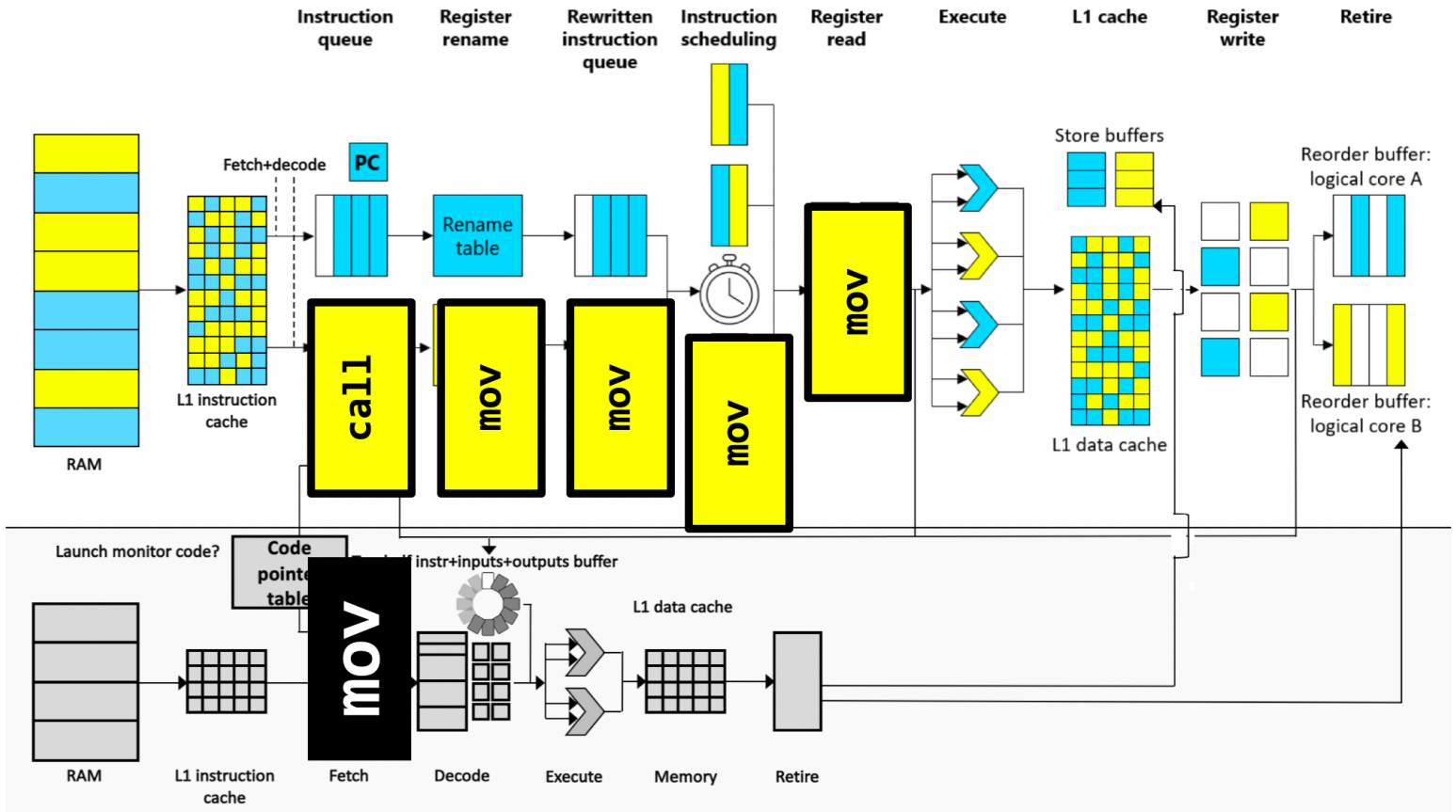




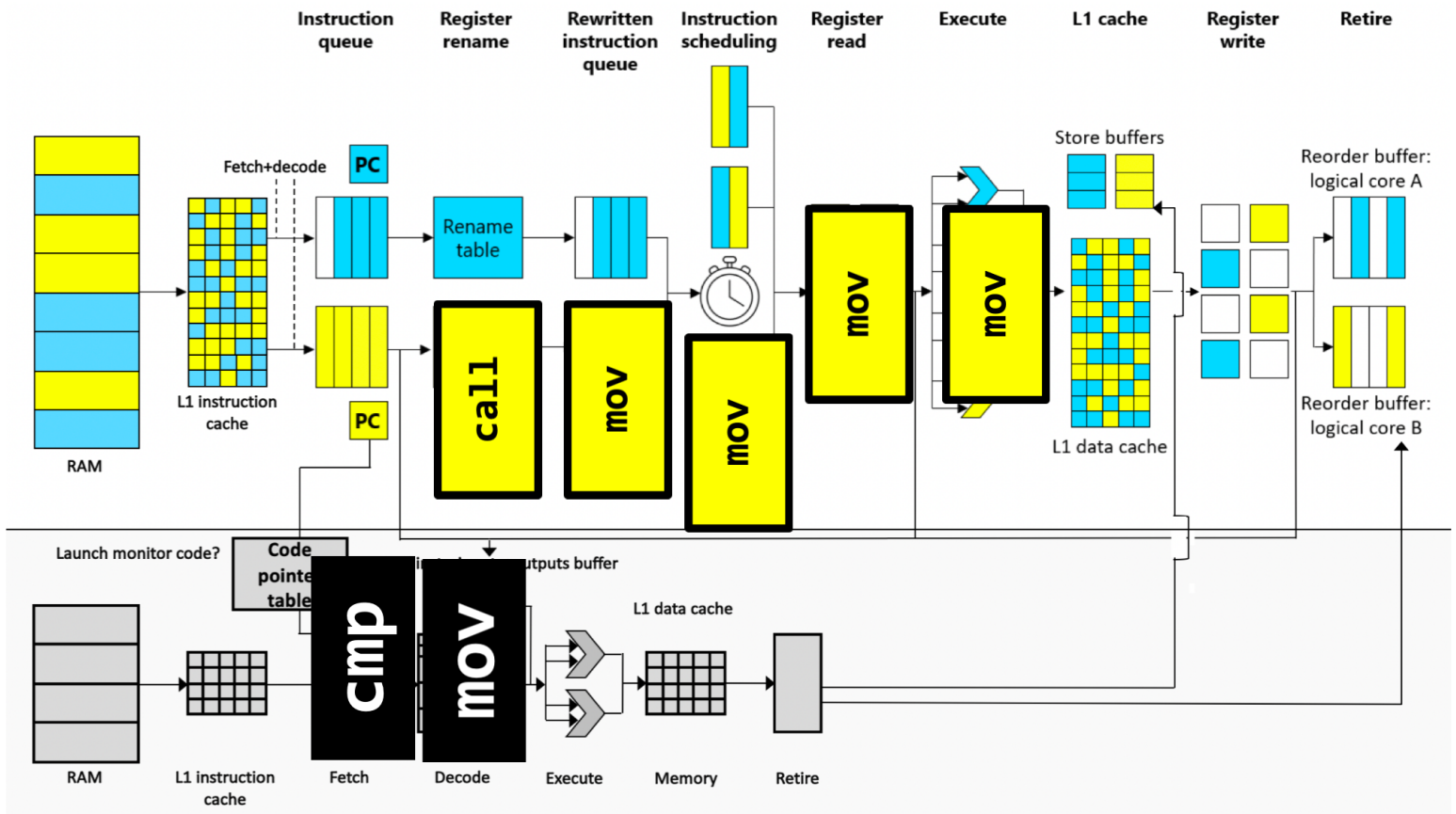

```
cmp %tpcb:%ti0 %b1
//For top-half instruction at
//PC %tpcb, compare the first
//input register to bottom-//half register
%b1
```


Instruction queue
Register rename
Rewritten instruction queue
Instruction scheduling
Register read
Execute
L1 cache
Register write
Retire
Fetch+decode
PC
Rename table
Rename table
PC
L1 instruction cache
RAM
Store buffers
L1 data cache
Reorder buffer: logical core A
Reorder buffer: logical core B
Launch monitor code?
Code pointer table
Top-half instr+inputs+outputs buffer
L1 data cache
RAM
L1 instruction cache
Fetch
Decode
Execute
Memory
Retire
Allow bottom-half to stall or zap top-half instruction retirement and memory writes

```
//TOP-HALF CODE
//foo()
mov 0x1, %rax
mov -0x20(%rbp), %rcx
mov %r12, %rsi
mov 0x8, %rdx
call %rcx
//InstrAddr: 0x0000F126


//MONITOR CODE
//Ensure that foo()
//invokes bar()
mov 0x0000F126, %tpcb
cmp %tpcb:%trcx, %brdx:(0x0000F126)
failnz %tpcb
```

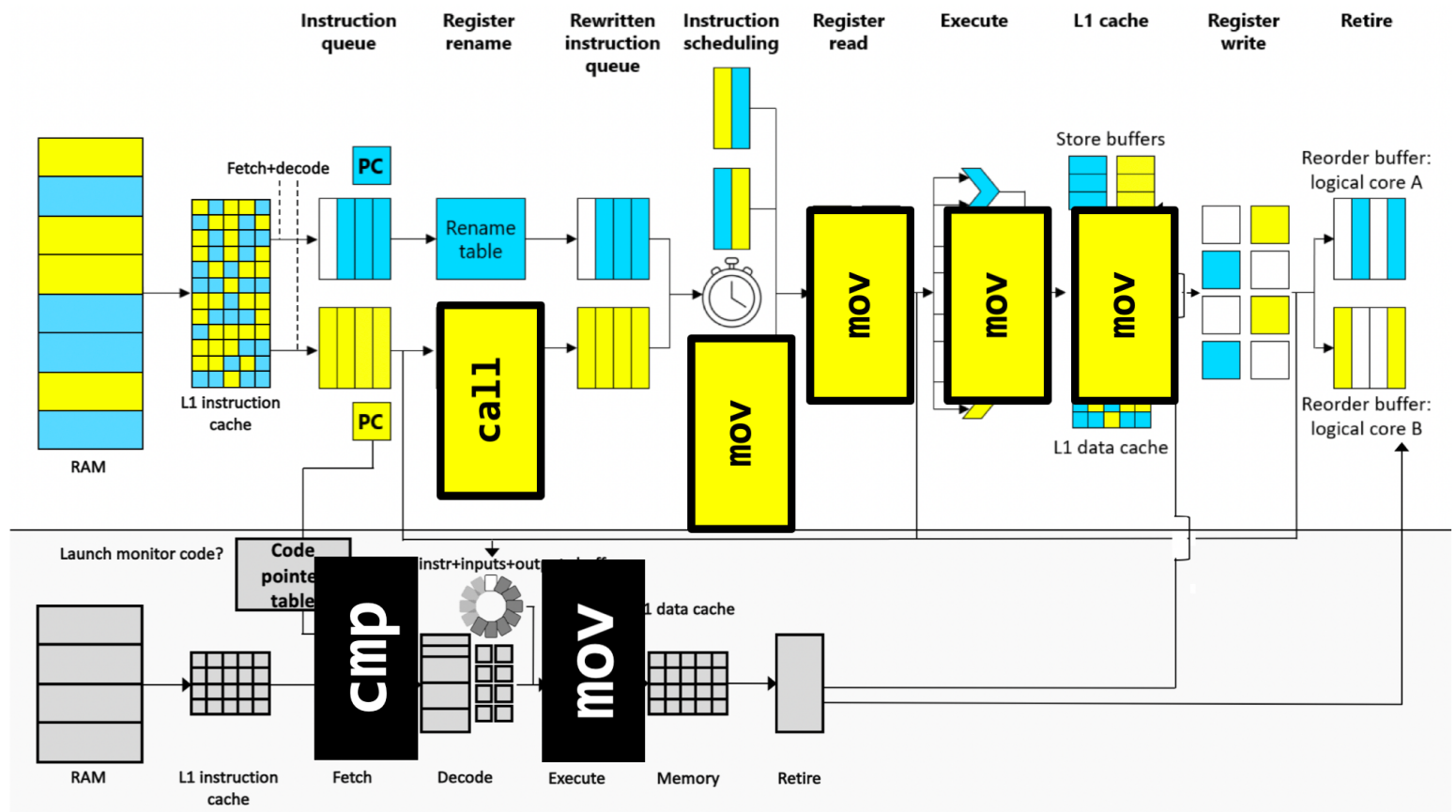


```
//TOP-HALF CODE
//foo()
mov 0x1, %rax
mov -0x20(%rbp), %rcx
mov %r12, %rsi
mov 0x8, %rdx
call %rcx
//InstrAddr: 0x0000F126
```




```
//MONITOR CODE
//Ensure that foo()
//invokes bar()
mov 0x0000F126, %tpcb
cmp %tpcb:%trcx, %brdx:(0x0000F126)
failnz %tpcb
```

```
//TOP-HALF CODE
//foo()
mov 0x1, %rax
mov -0x20(%rbp), %rcx
mov %r12, %rsi
mov 0x8, %rdx
call %rcx
//InstrAddr: 0x0000F126
```

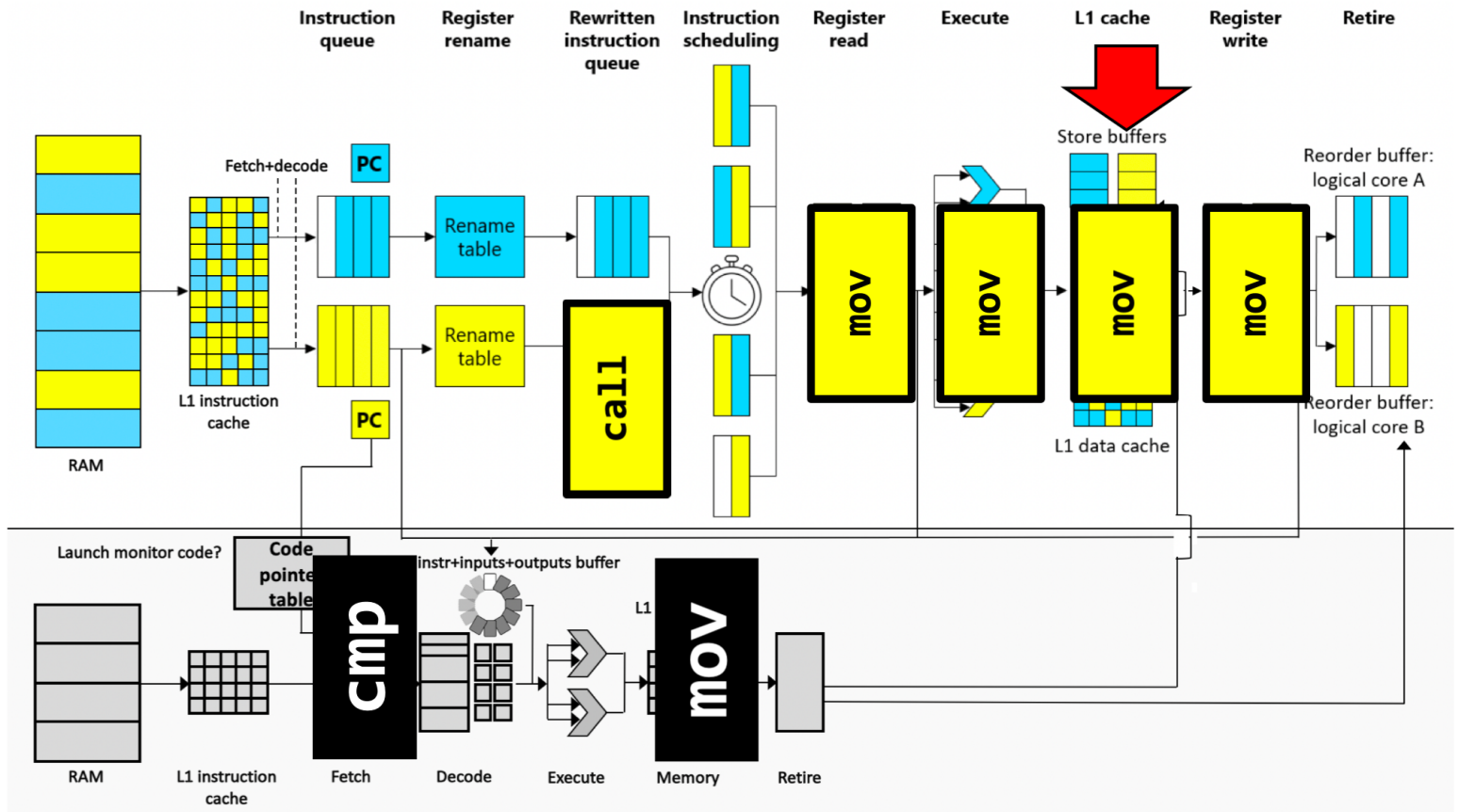



```
//MONITOR CODE
//Ensure that foo()
//invokes bar()
mov 0x0000F126, %tpcb
cmp %tpcb:%trcx, %brdx:(0x0000F126)
failnz %tpcb
```

Stalls because %rcx hasn't been written by top-half yet!

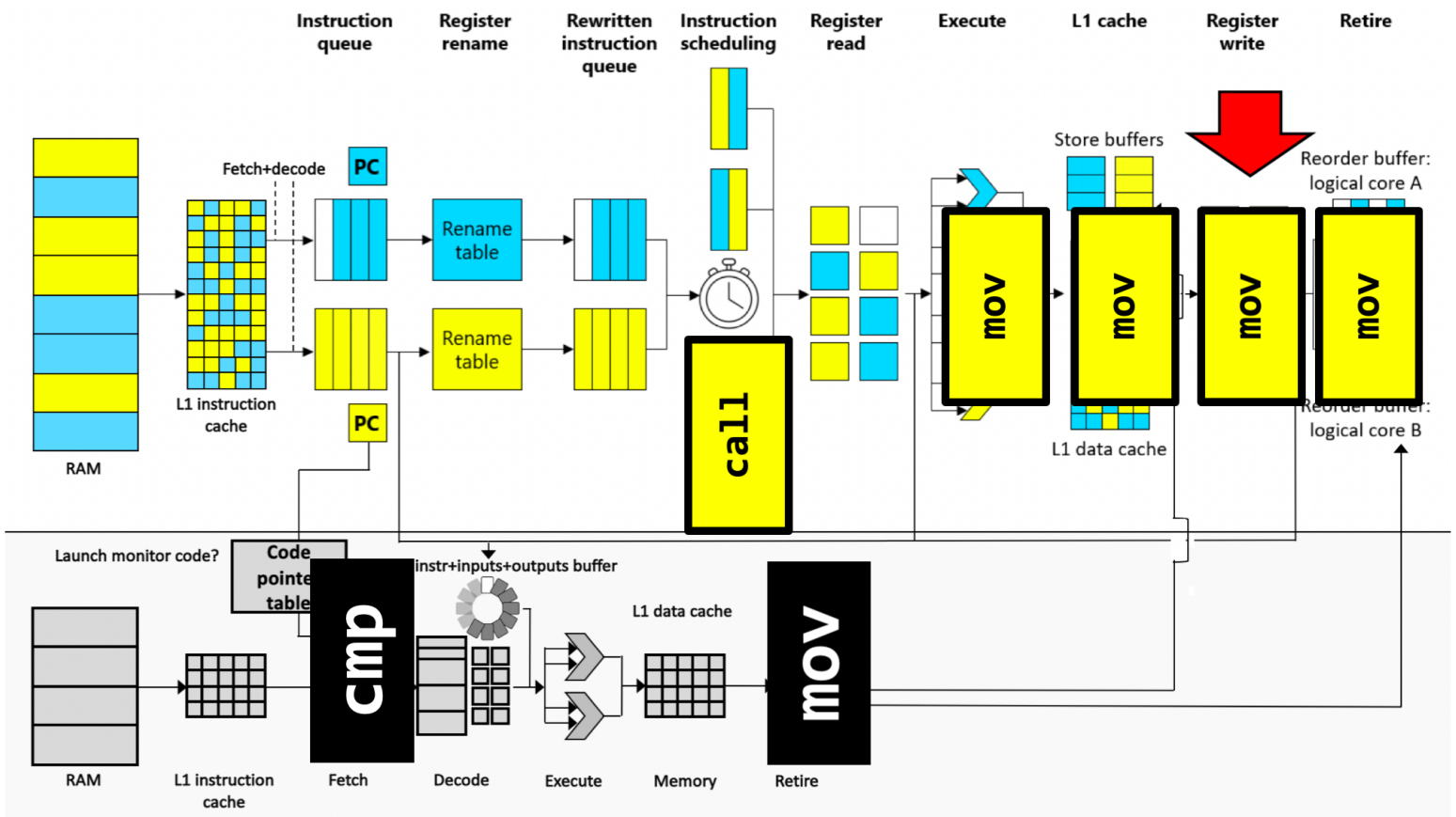
```
//TOP-HALF CODE
//foo()
mov 0x1, %rax
mov -0x20(%rbp), %rcx
mov %r12, %rsi
mov 0x8, %rdx
call %rcx
//InstrAddr: 0x0000F126
```

```
//MONITOR CODE
//Ensure that foo()
//invokes bar()
mov 0x0000F126, %tpcb
cmp %tpcb:%trcx, %brdx:(0x0000F126)
failnz %tpcb
```



Stalls because %rcx hasn't been written by top-half yet!

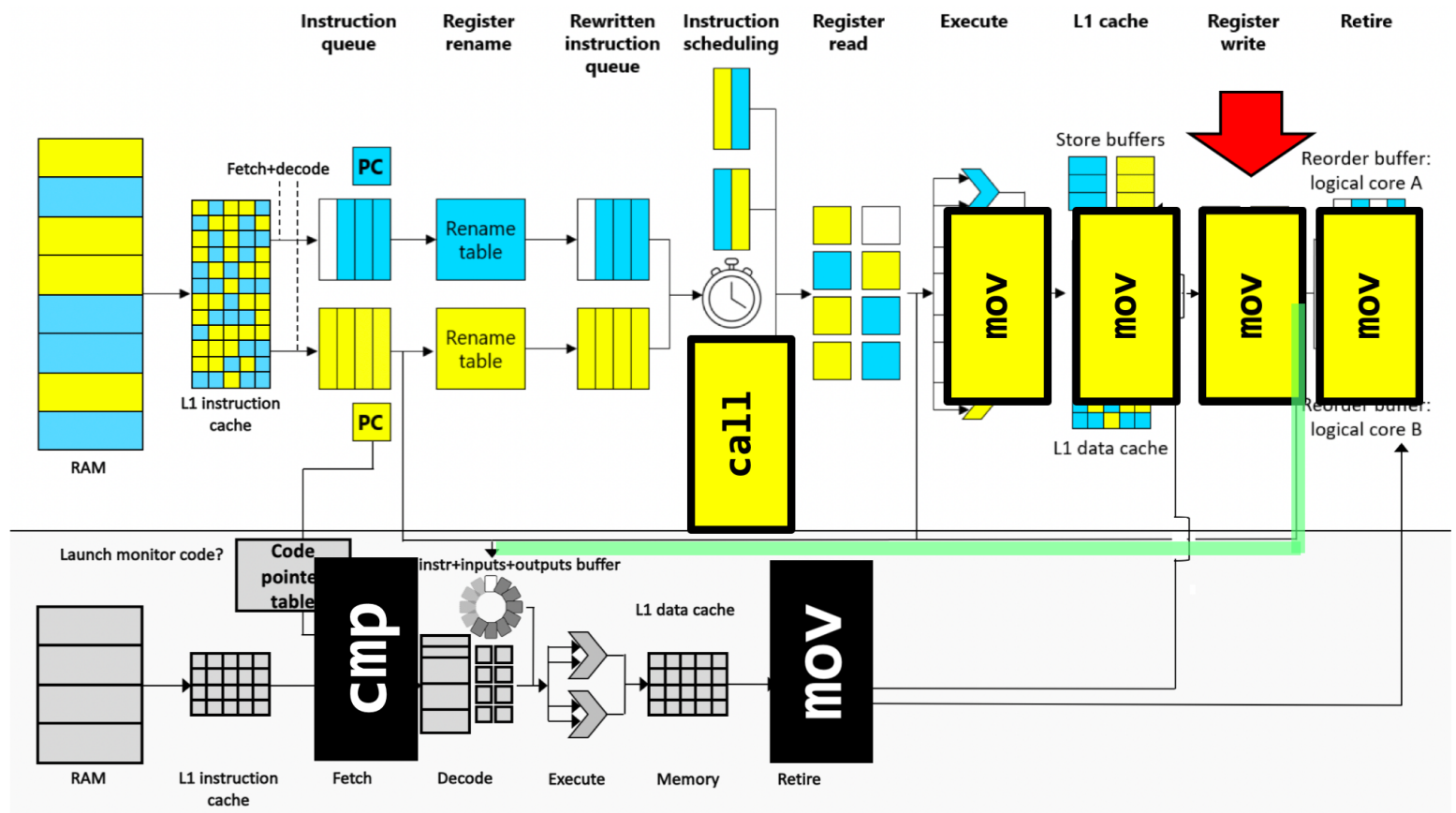
```
//TOP-HALF CODE
//foo()
mov 0x1, %rax
mov -0x20(%rbp), %rcx
mov %r12, %rsi
mov 0x8, %rdx
call %rcx
//InstrAddr: 0x0000F126
```

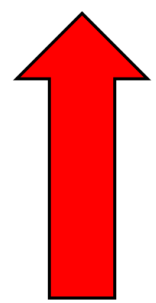



```
//MONITOR CODE
//Ensure that foo()
//invokes bar()
mov 0x0000F126, %tpcb
cmp %tpcb:%trcx, %brdx:(0x0000F126)
failnz %tpcb
```

Stalls because %rcx hasn't been written by top-half yet!

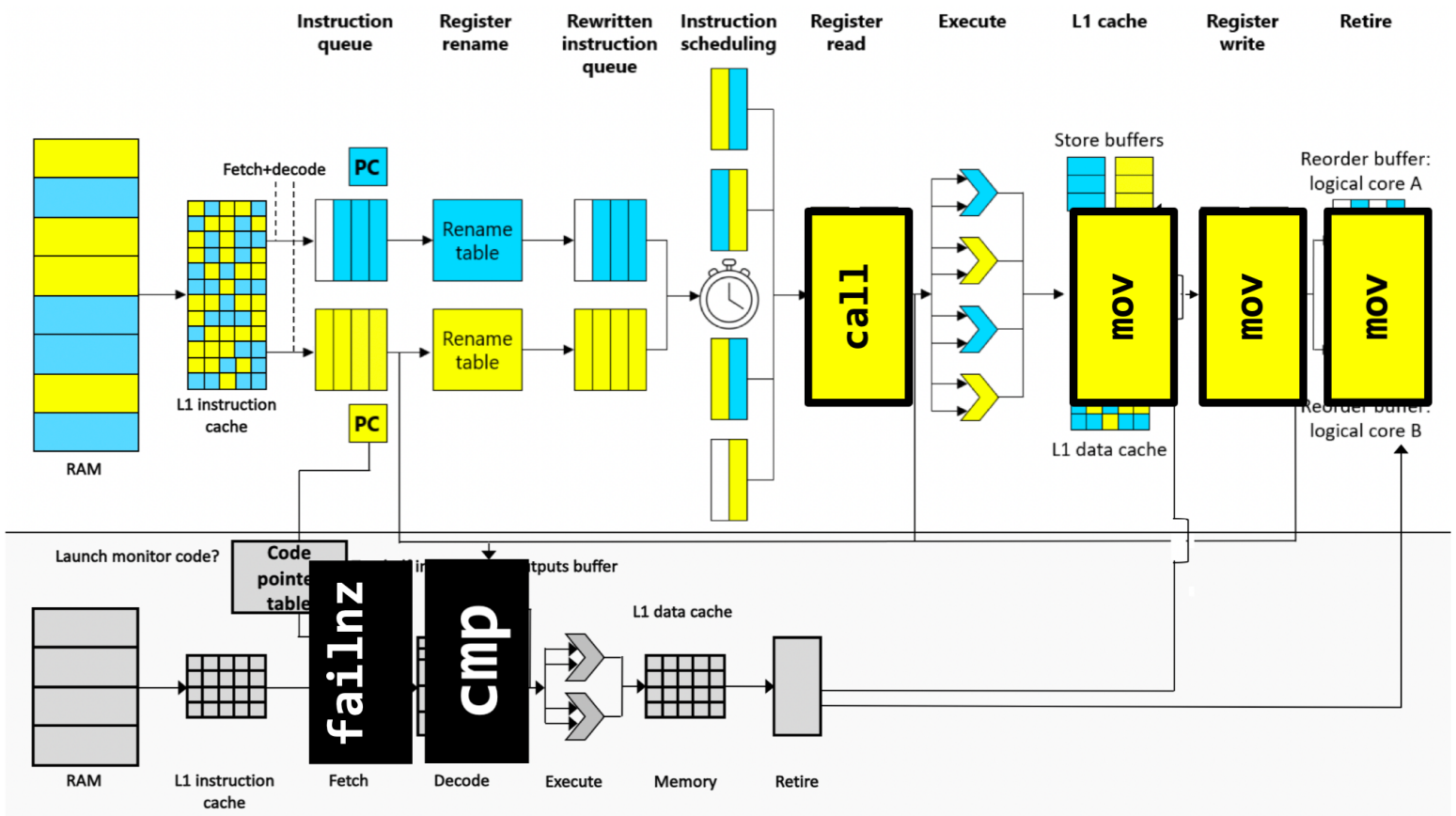
```
//TOP-HALF CODE
//foo()
mov 0x1, %rax
mov -0x20(%rbp), %rcx
mov %r12, %rsi
mov 0x8, %rdx
call %rcx
//InstrAddr: 0x0000F126


//MONITOR CODE
//Ensure that foo()
//invokes bar()
mov 0x0000F126, %tpcb
cmp %tpcb:%trcx, %brdx:(0x0000F126)
failnz %tpcb
```





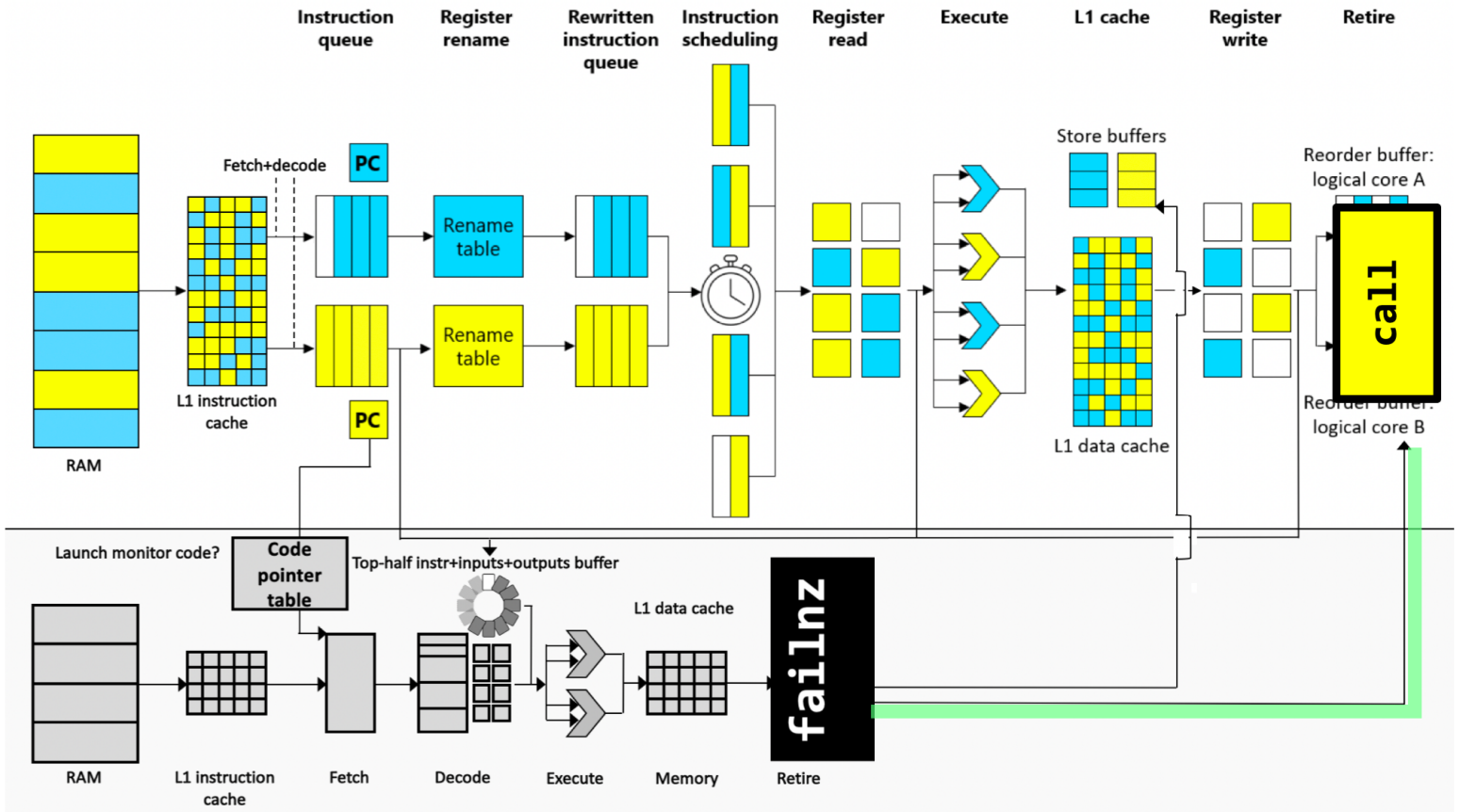
Stalls because %rcx hasn't been written by top-half yet!

```
//TOP-HALF CODE
//foo()
mov 0x1, %rax
mov -0x20(%rbp), %rcx
mov %r12, %rsi
mov 0x8, %rdx
call %rcx
//InstrAddr: 0x0000F126
```




```
//MONITOR CODE
//Ensure that foo()
//invokes bar()
mov 0x0000F126, %tpcb
cmp %tpcb:%trcx, %brdx:(0x0000F126)
failnz %tpcb
```

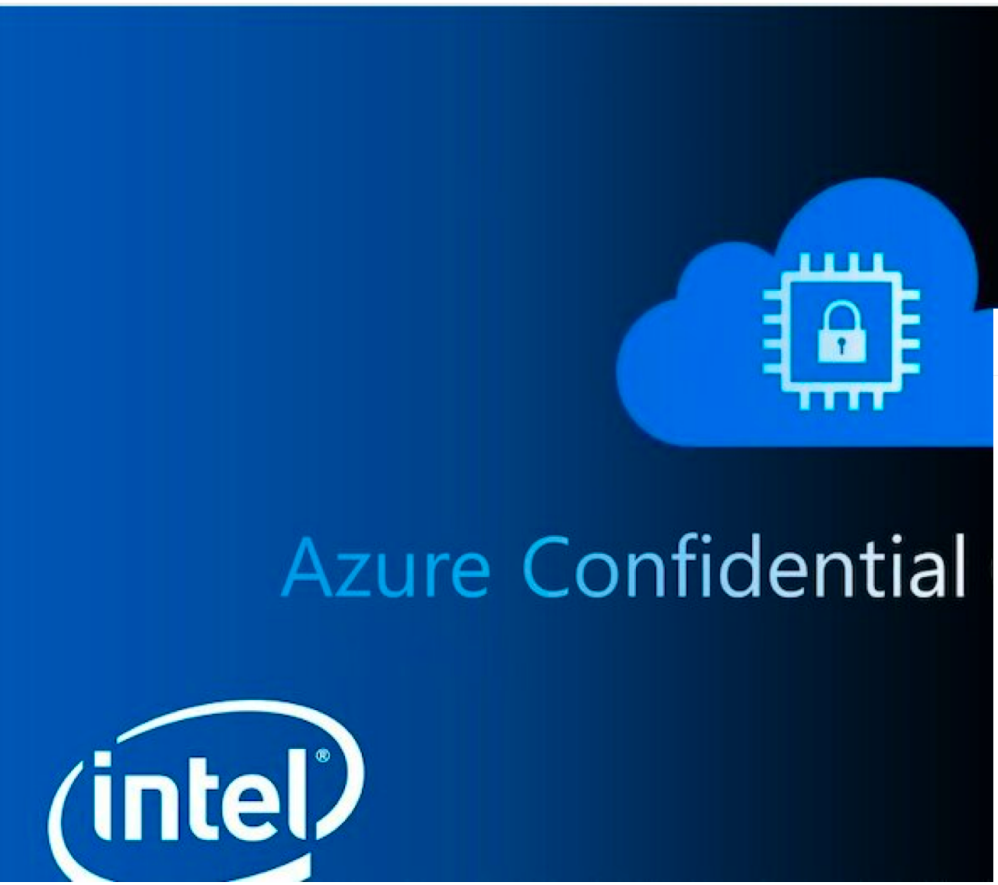
```
//TOP-HALF CODE
//foo()
mov 0x1, %rax
mov -0x20(%rbp), %rcx
mov %r12, %rsi
mov 0x8, %rdx
call %rcx
//InstrAddr: 0x0000F126


//MONITOR CODE
//Ensure that foo()
//invokes bar()
mov 0x0000F126, %tpcb
cmp %tpcb:%trcx, %brdx:(0x0000F126)
failnz %tpcb
```



Outline

- Trusted Computing Overview
- IME Design
- Security Monitor Policies
- Implementation
- Related Work
- Open Challenges

Related Work: TEEs in real world



Signal

Get Signal Support Blog Developers Careers Donate

SGX contact discovery

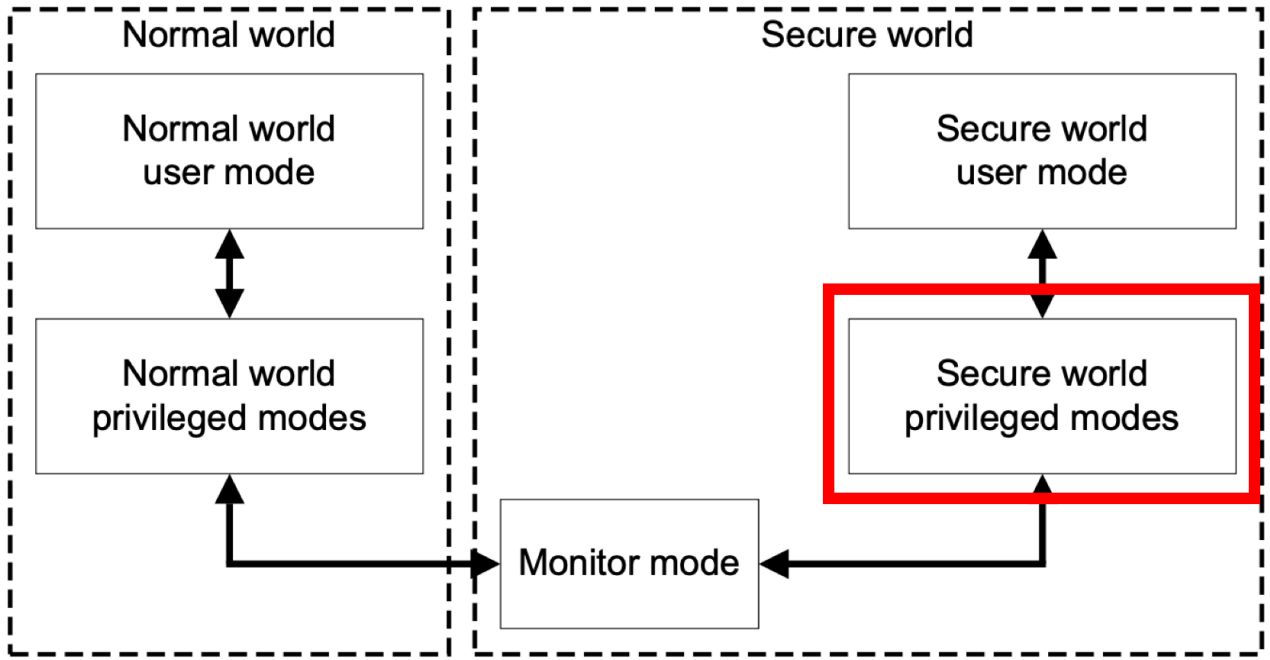
Private contact discovery using SGX is fairly simple at a high level:

1. Run a contact discovery service in a secure SGX enclave.
2. Clients that wish to perform contact discovery negotiate a secure connection over the network all the way through the remote OS to the enclave.
3. Clients perform remote attestation to ensure that the code which is running in the enclave is the same as the expected published open source code.
4. Clients transmit the encrypted identifiers from their address book to the enclave.
5. The enclave looks up a client's contacts in the set of all registered users and encrypts the results back to the client.

Since the enclave attests to the software that's running remotely, and since the remote server and OS have no visibility into the enclave, the service learns nothing about the contents of the client request. It's almost as if the client is executing the query locally on the client device.

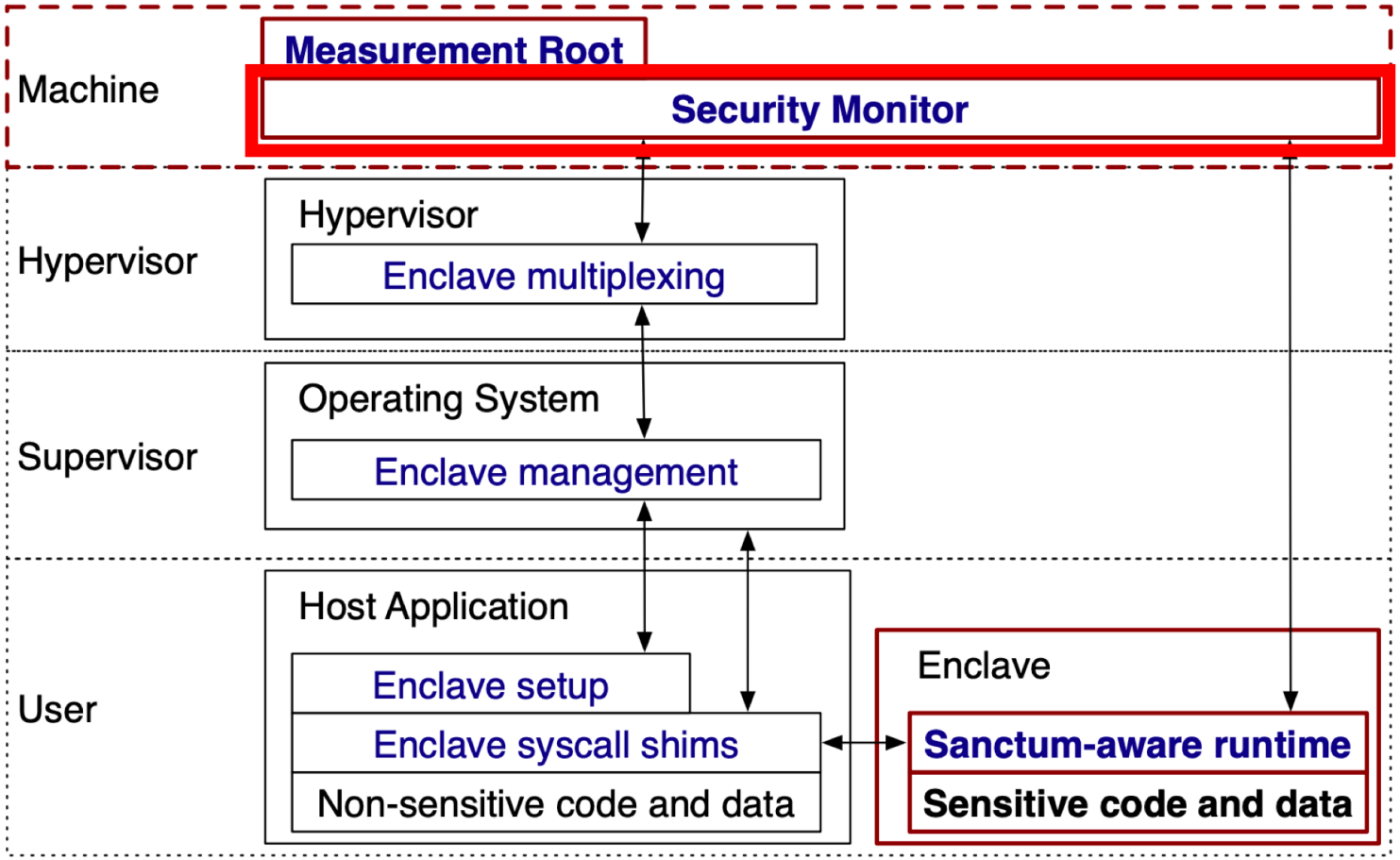
Unfortunately, doing private computation in an SGX enclave is more difficult than it may initially seem.

Related Work: ARM TrustZone



: **Modes in an ARM core implementing the Security Extensions**

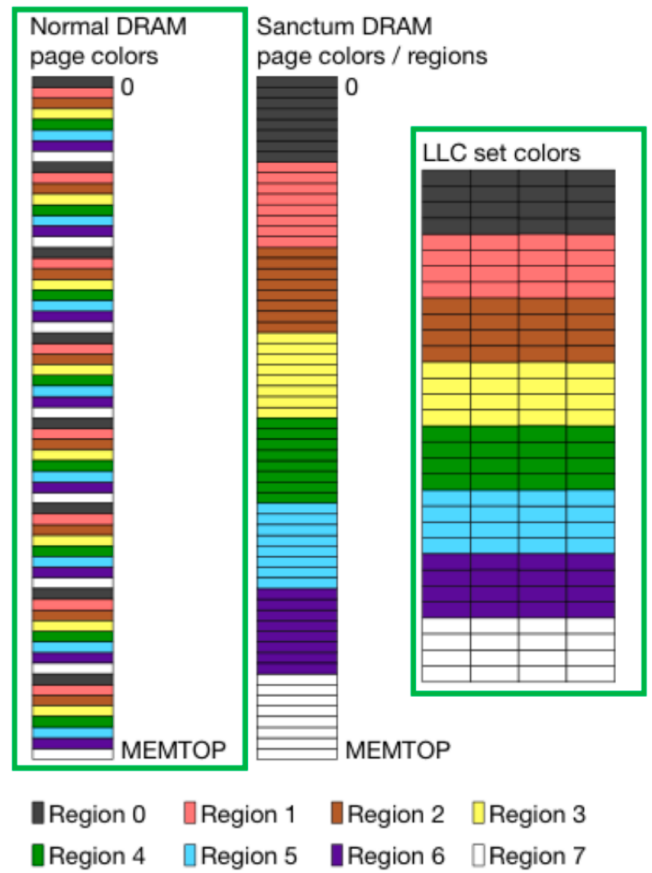
Related Work: Sanctum



Software Stack

Sanctum: Minimal Hardware Extensions for Strong Software Isolation, Costan, *et al.*

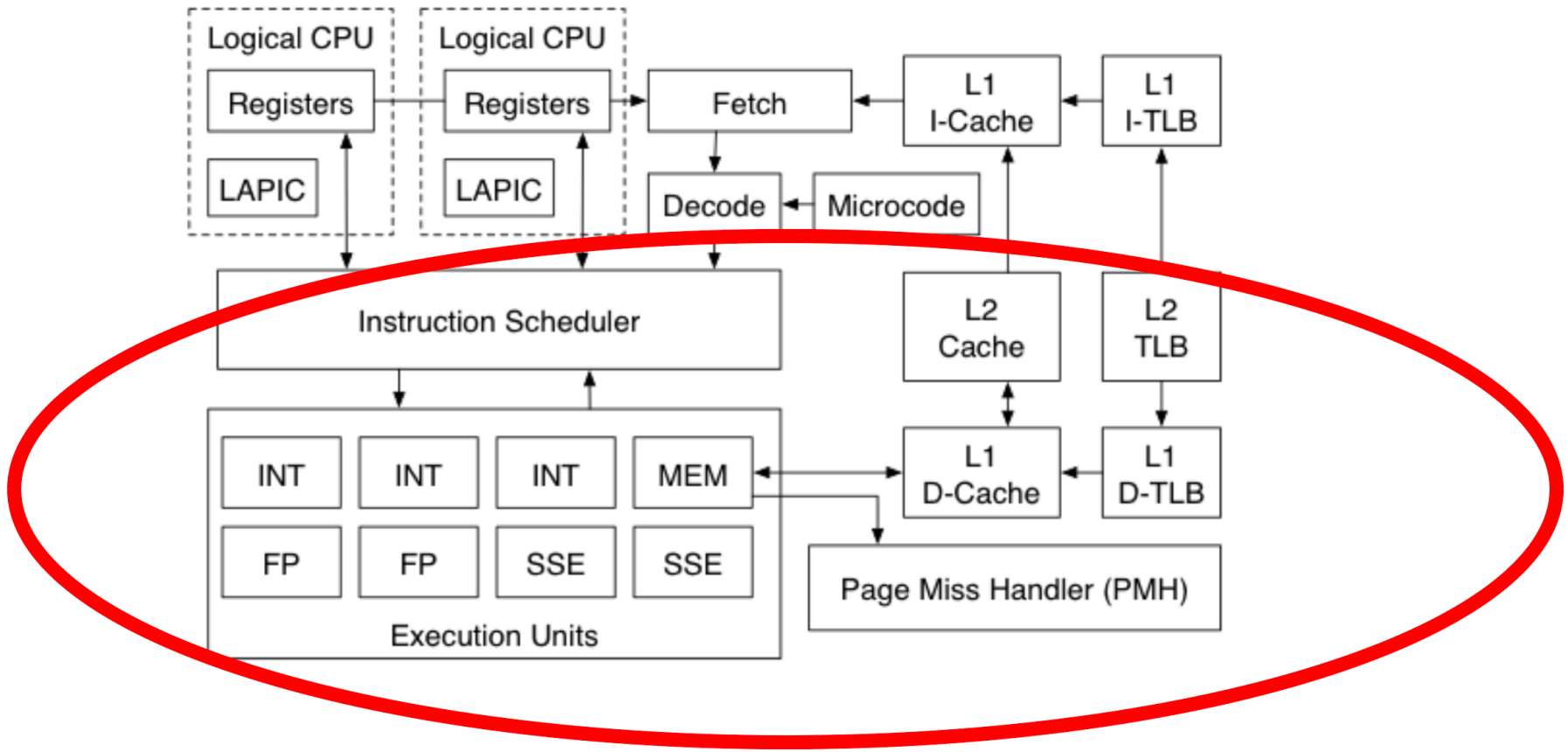
Related Work: Sanctum



DRAM partitioning

Sanctum: Minimal Hardware Extensions for Strong Software Isolation, Costan, *et al.*

Related Work: Sanctum



No isolation for hyper-threading

Sanctum: Minimal Hardware Extensions for Strong Software Isolation, Costan, *et al.*

Related Work: Sanctum

No runtime integrity checks

Sanctum: Minimal Hardware Extensions for Strong Software Isolation,
Costan, *et al.*

Outline

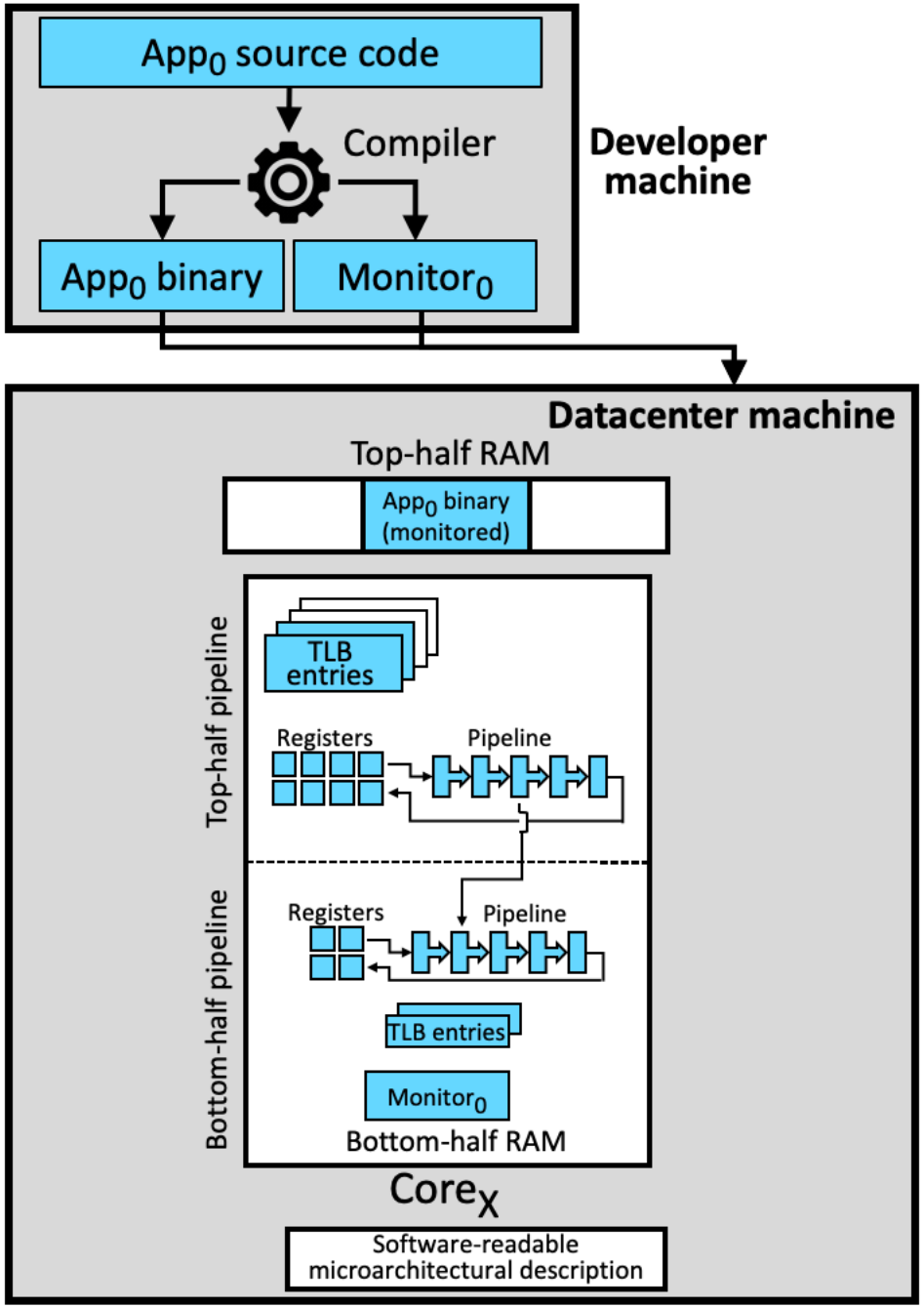
- Trusted Computing Overview
- IME Design
- Security Monitor Policies
- Implementation
- Related Work
- Open Challenges

Open Challenges

- Side-channels involving monitor-induced stalling
- Multi-threaded applications
- Syscalls
- High-performance network IO
- Dyad Migration
- Additional die area

IME Design: Side-channels?

- Constant-time execution →runtime of top-half code is tailored to the worst-case execution time of monitor checks
- Early launch of monitors→ The monitor check for instruction i can start during an earlier instruction i-n



Open Challenges

- Side-channels involving monitor-induced stalling
- High-performance network IO
- Multi-threaded applications
- Syscalls
- Dyad Migration
- Additional die area

Open Challenges

- Side-channels involving monitor-induced stalling
- High-performance network IO
- Multi-threaded applications
- Syscalls
- Dyad Migration
- Additional die area

Open Challenges

- Side-channels involving monitor-induced stalling
- High-performance network IO
- Multi-threaded applications
- Syscalls
- Dyad Migration
- Additional die area

Open Challenges

- Side-channels involving monitor-induced stalling
- High-performance network IO
- Multi-threaded applications
- Syscalls
- Dyad Migration
- Additional die area

Open Challenges

- Side-channels involving monitor-induced stalling
- High-performance network IO
- Multi-threaded applications
- Syscalls
- Dyad Migration
- Additional die area

IME: Conclusion

1. Isolate a secure computation from other local computations (including privileged ones!)
2. Enforce dynamic runtime checks to demonstrate to users that their secure computations are actually secure
3. Provide a software readable description of the microcode

Thank you!

Authors

Varun Gandhi
vgandhi@g.harvard.edu

James Mickens
mickens@g.harvard.edu