

July 23-25, 2024  
NIST FMCP 2024

# Towards formal verification of the confidential computing framework for RISC-V

Wojciech  
Ozga

*IBM Research  
Zürich*

Lennard  
Gäher

*IBM Research  
Zürich  
&  
MPI-SWS,  
Germany*

Guerney D.H.  
Hunt

*IBM Thomas J.  
Watson Research  
Center*

Avraham  
Shinnar

*IBM Thomas J.  
Watson Research  
Center*

Elaine R.  
Palmer

*IBM Thomas J.  
Watson Research  
Center*

Michael V.  
Le

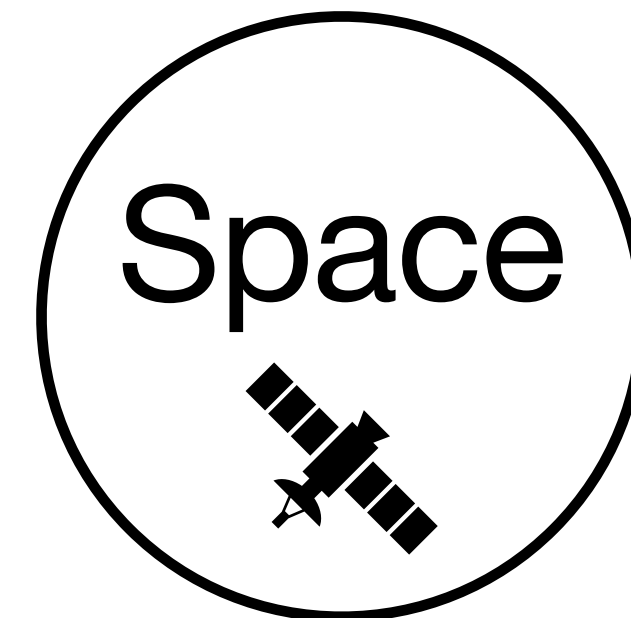
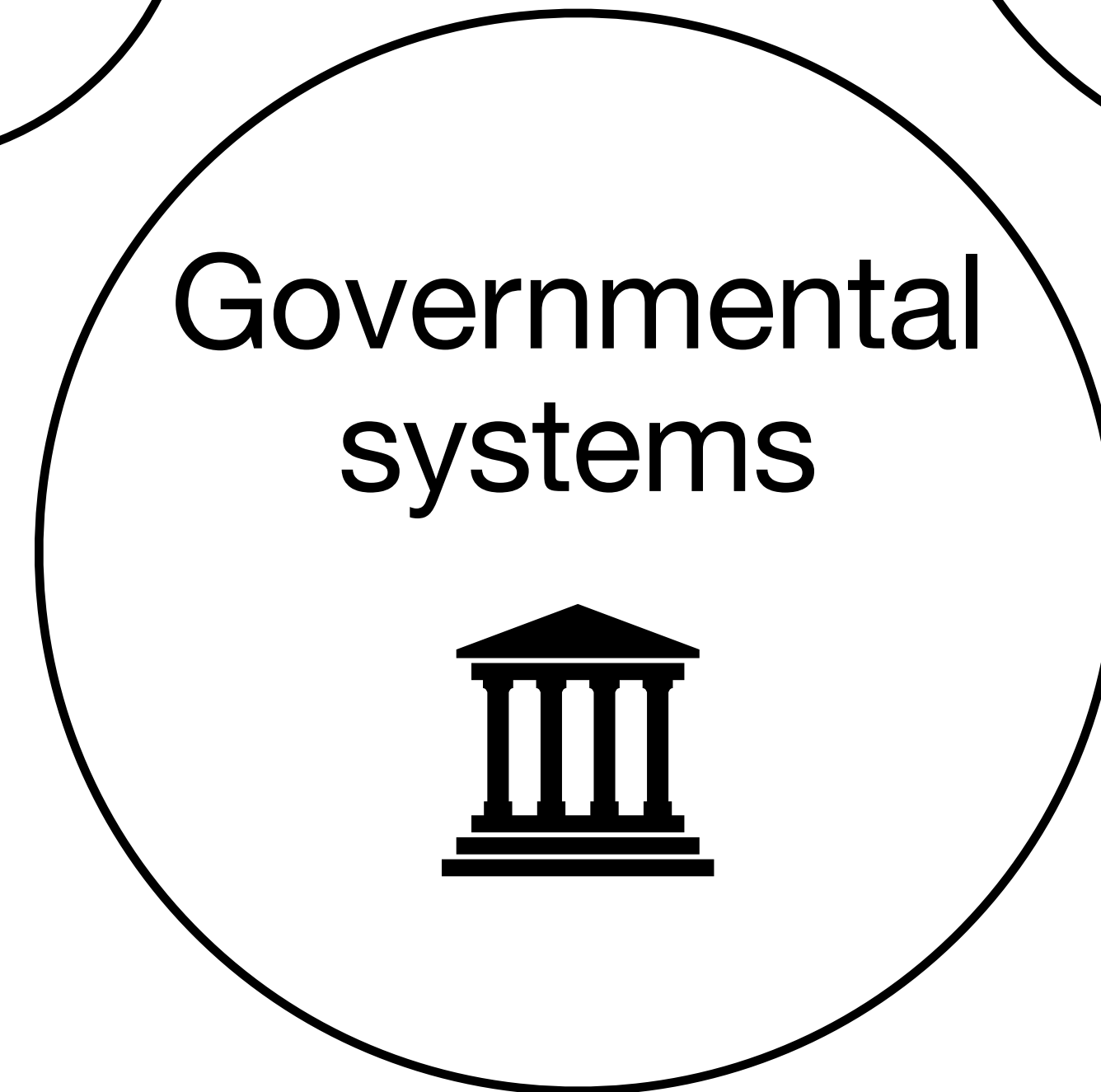
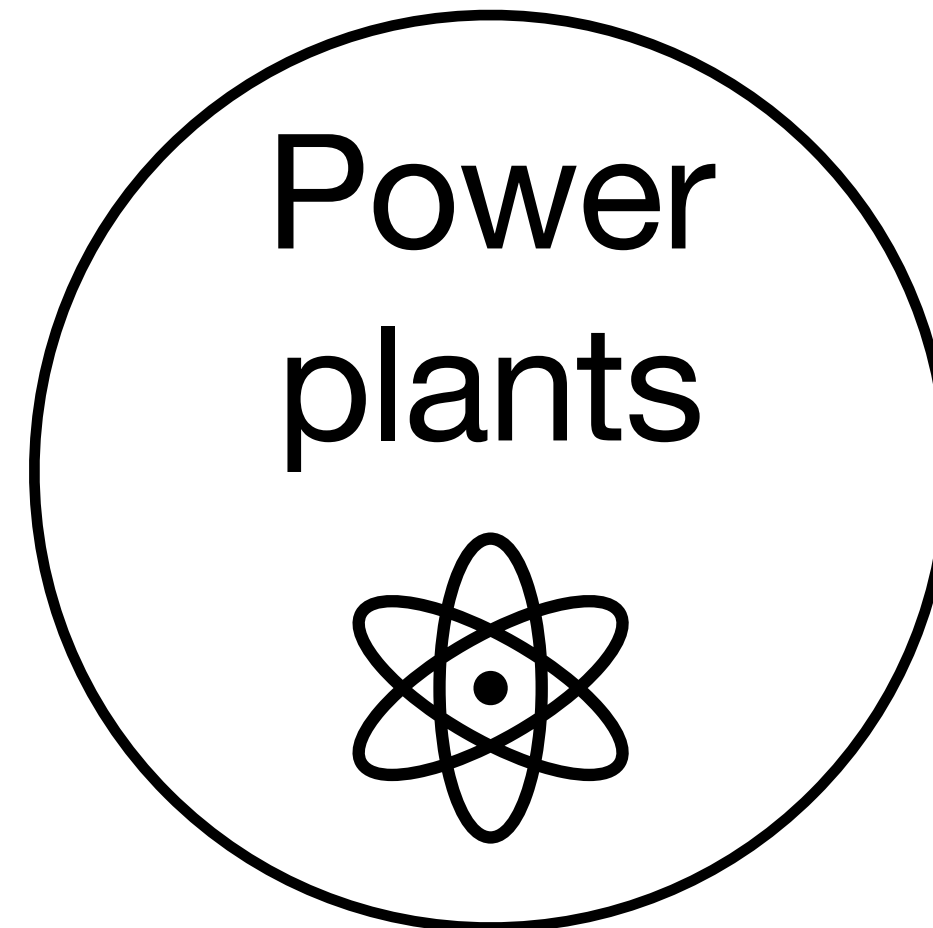
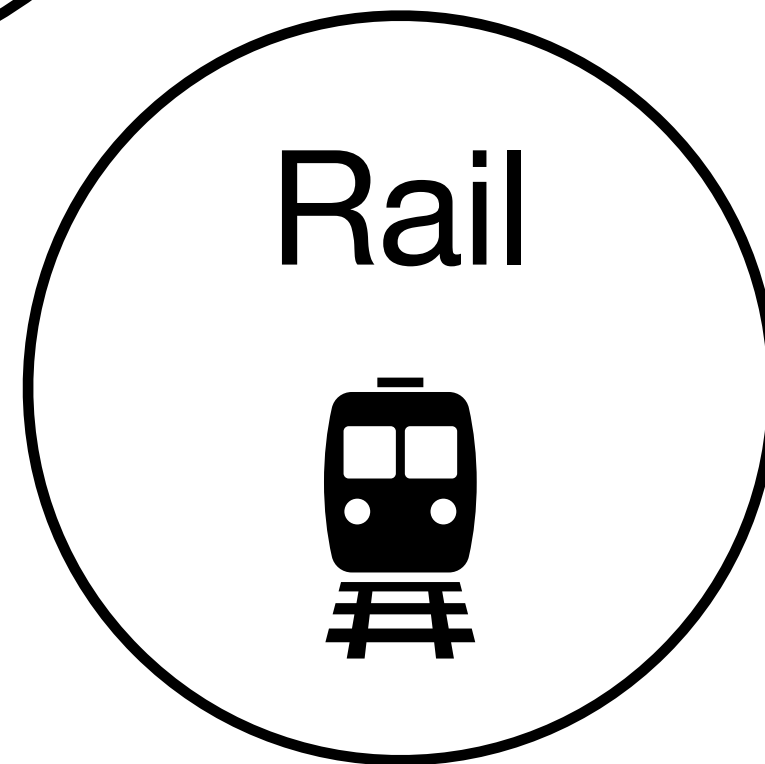
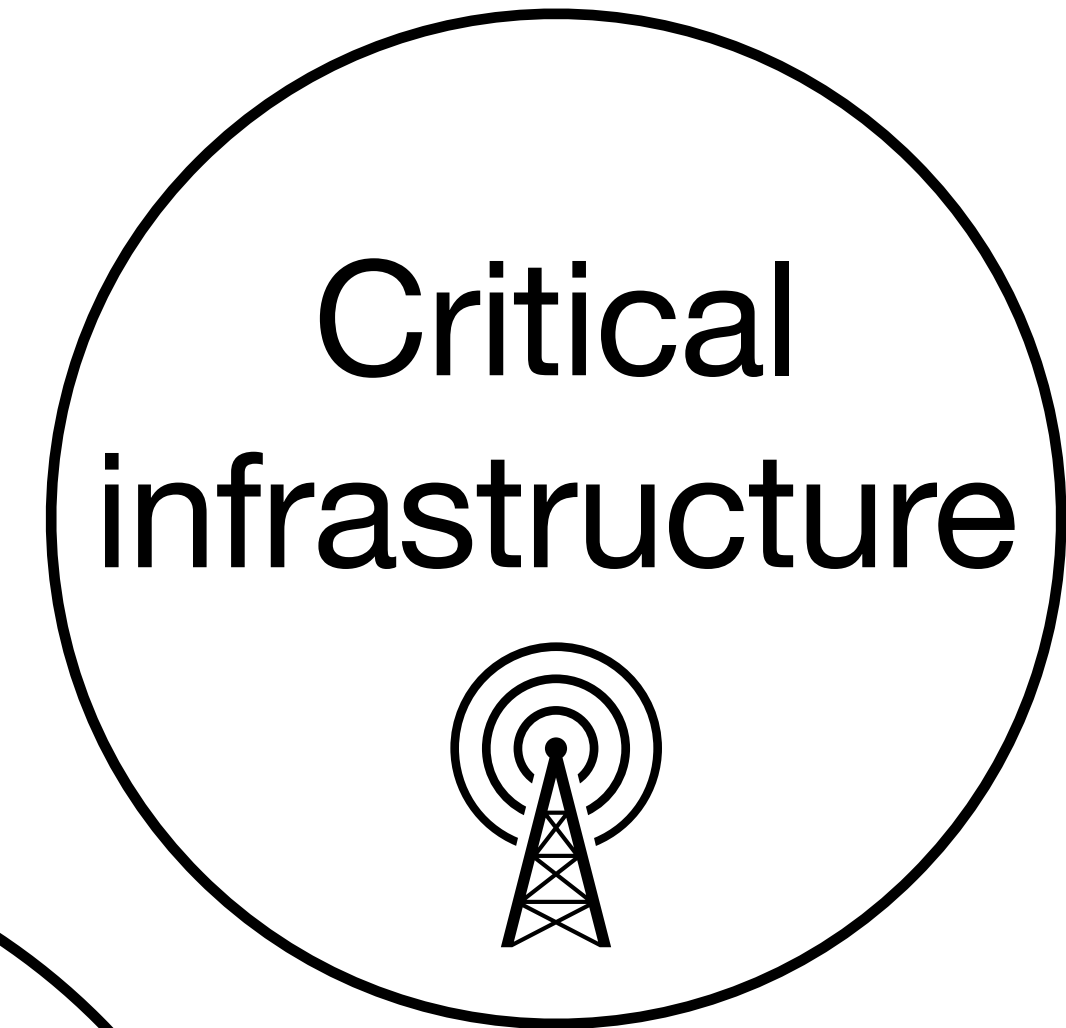
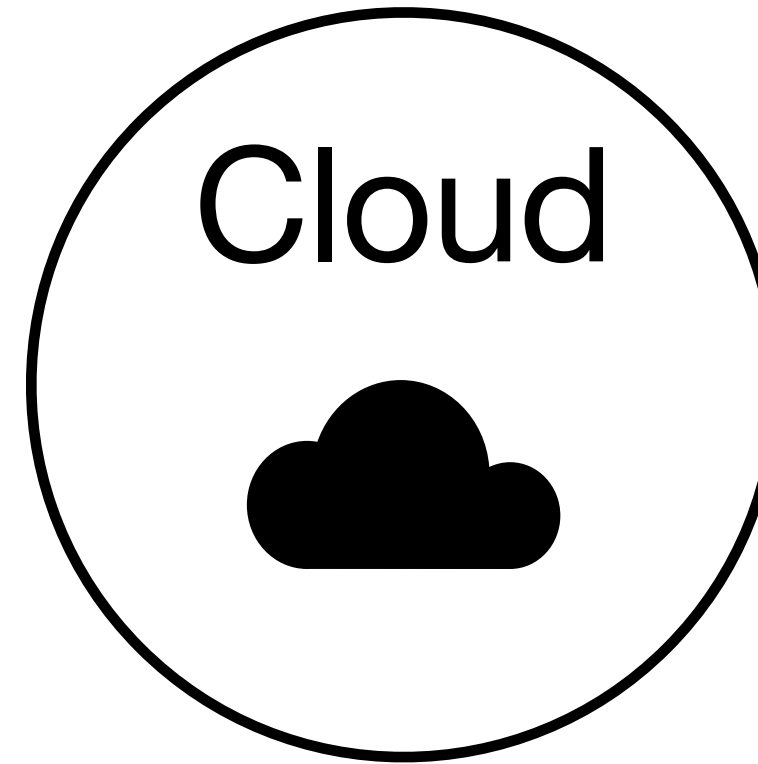
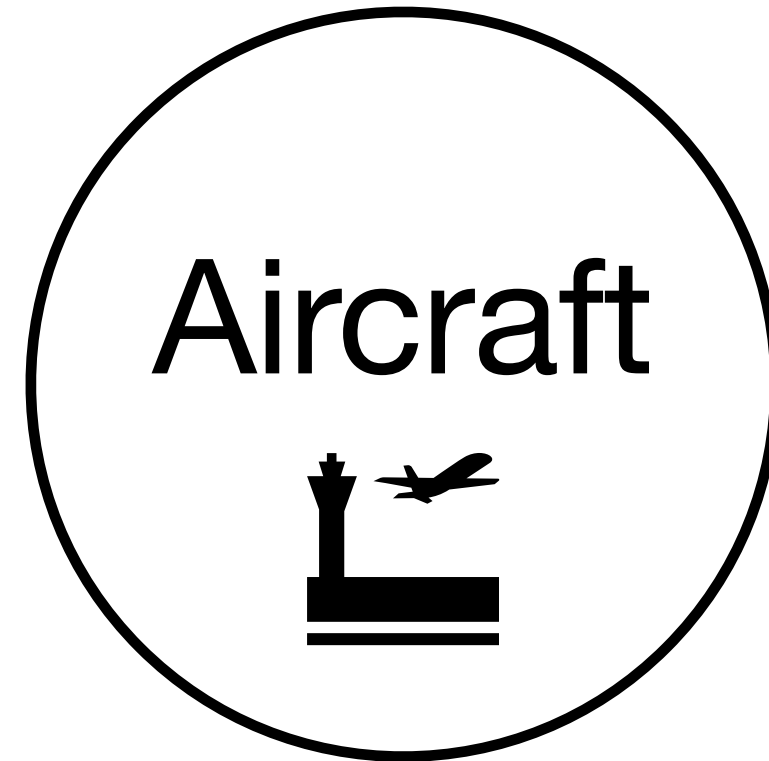
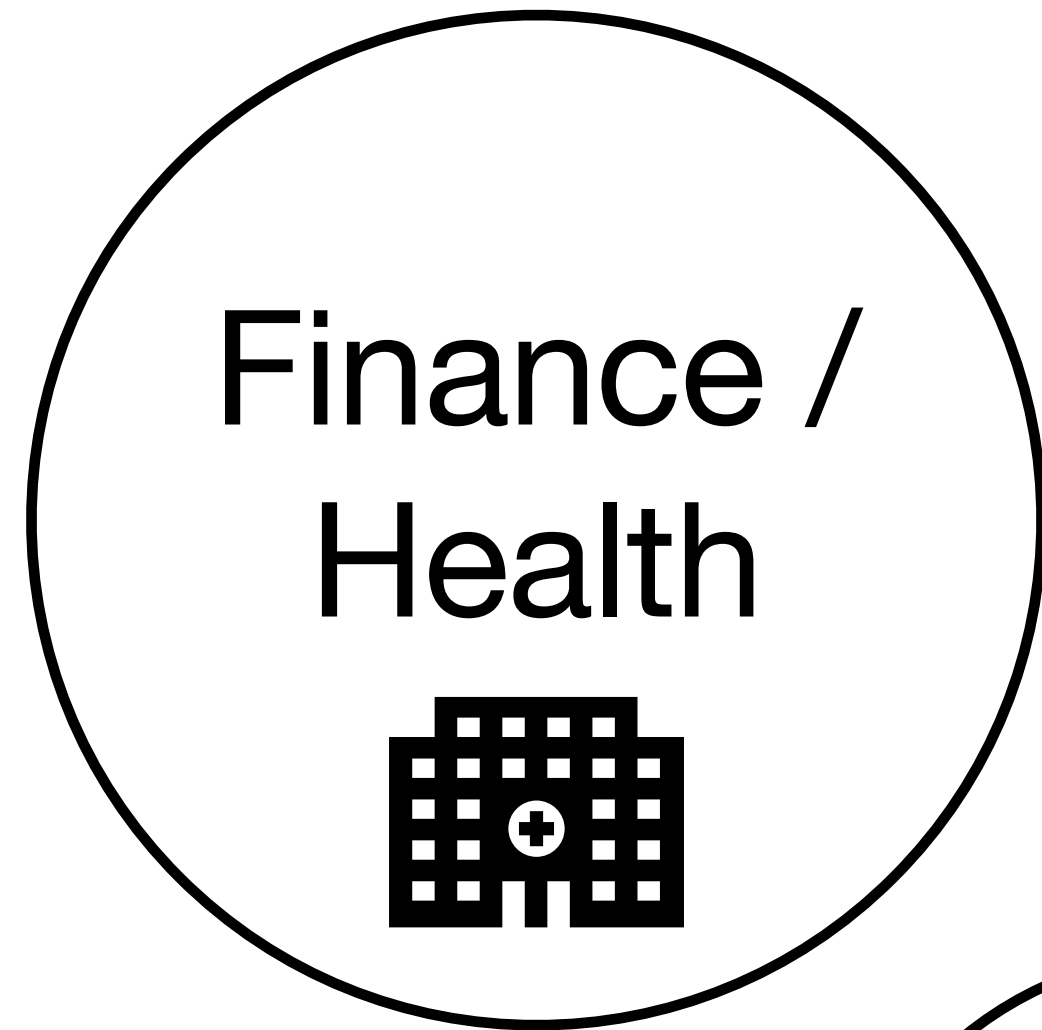
*IBM Thomas J.  
Watson  
Research Center*

Silvio  
Dragone

*IBM Research  
Zürich*

**How much does your life  
and security depend on  
computers?**

# Problem: **Security of high-assurance systems**



Successful attacks on **high-assurance systems** might lead to catastrophe, social disturbances, political instability.

SIGN IN The Register

{\* SECURITY \*}

# No big deal... Kremlin hackers 'jumped air-gapped networks' to pwn US power utilities

'Hundreds' of intrusions, switch could be pulled anytime

Richard Chirgwin

80

The US Department of Homeland Security government hackers of penetrating America

Uncle Sam's finest reckon Moscow's agent networks within US electric utilities – to the have virtually pressed the off switch in cont Yanks, and plunged America into darkness

The hackers, dubbed Dragonfly and Energy 2016, and continued throughout 2017 and i

HOME > TECH

# The hackers that attacked a major US oil pipeline say it was only for money — here's what to know about DarkSide

Natasha Dailey May 10, 2021, 5:49 PM

npr

# A 'Worst Nightmare' Cyberattack: The Untold Story Of The SolarWinds Hack

April 16, 2021 · 10:05 AM ET

Heard on [All Things Considered](#)



An NPR investigation into the SolarWinds attack reveals a hack unlike any other, launched by a sophisticated adversary intent on exploiting the soft underbelly of our digital lives.

Zoë van Dijk for NPR

REUTERS

INTERNET NEWS

JULY 10, 2017 / 1:57 PM / UPDATED 5 YEARS AGO

# Foreign hackers probe European critical infrastructure networks: sources

By Mark Hosenball

LONDON (Reuters) - Cyber attackers are regularly trying to attack data networks connected to critical national infrastructure systems around Europe, according to current and former European government sources with knowledge of the issue.

The New York Times

# Hackers Are Targeting Nuclear Facilities, Homeland Security Dept. and F.B.I.



The Wolf Creek Nuclear power plant in Kansas in 2000. The corporation that runs the plant was targeted by hackers. David Eulitt/Capital Journal, via Associated Press

By Nicole Perlroth

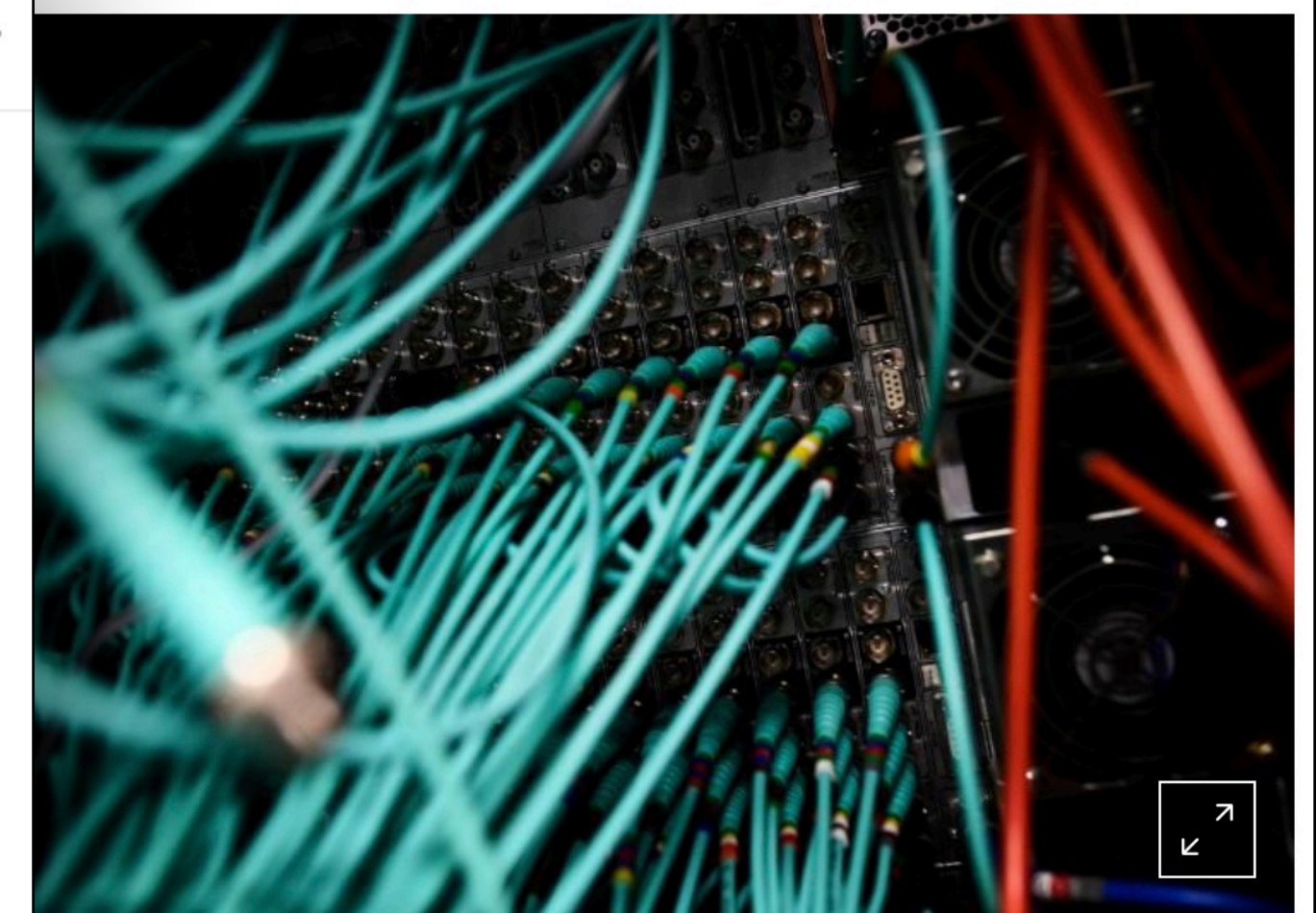
July 6, 2017

# Erro do Ministério expõe dados pessoais de mais de 200 milhões de brasileiros

Erro em sistema federal de registro de casos de covid permitiu acesso, durante seis meses, a informações pessoais de todos os brasileiros cadastrados no SUS e clientes de plano de saúde

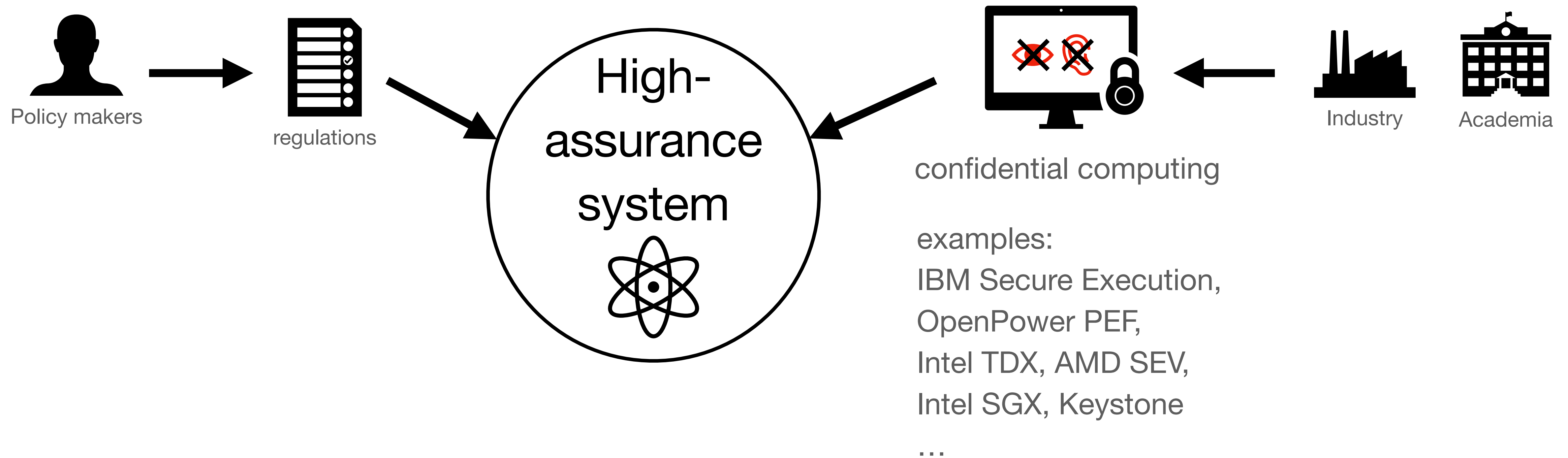
Fabiana Cambricoli, O Estado de S.Paulo

02 de dezembro de 2020 | 05h00



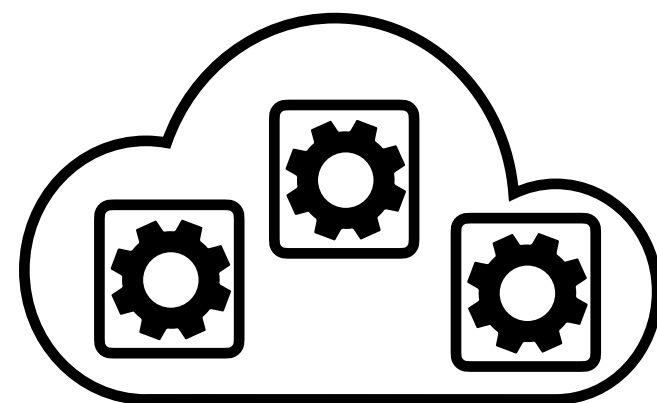
and computers are seen inside a data centre at an office in the heart of the financial in London, Britain May 15, 2017. REUTERS/Dylan Martinez

# Problem: How to formally verify security properties of confidential computing systems?



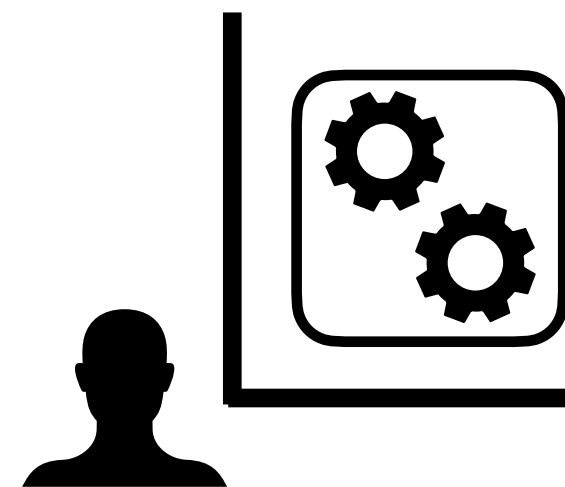
Security-critical systems are subject to **regulations**. Certification might require **formal verification**.

# Use Cases



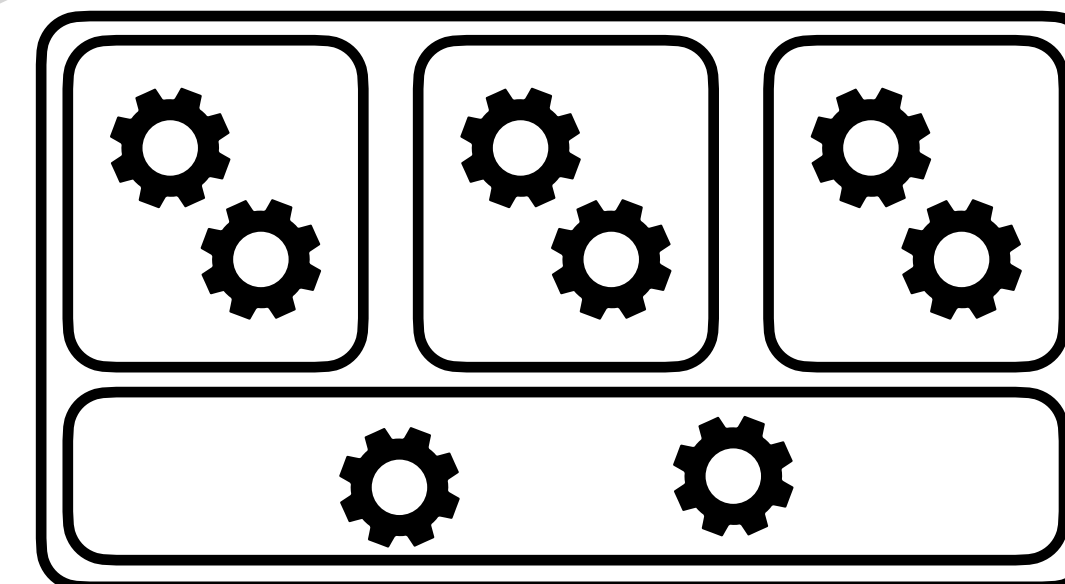
Multi-tenant cloud

e.g., IBM Cloud



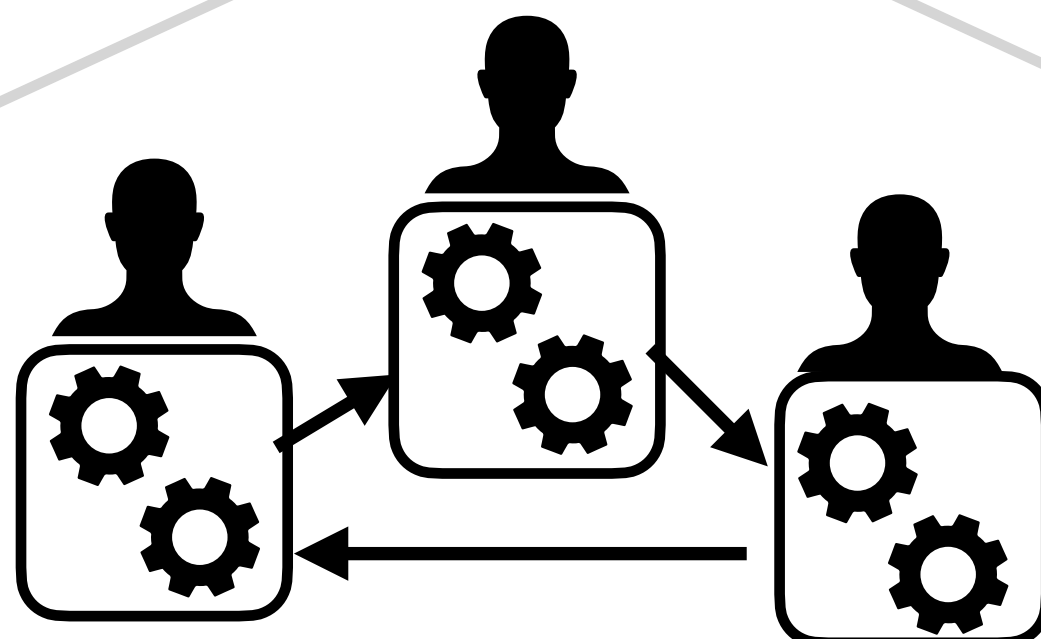
Edge systems

e.g., processing requests from smart cars



Multi-domain /  
Embedded systems

e.g., automotive, smartphones,  
sandboxing



Secure multi-party computation

e.g., ML, LLM trainings/inference

**Goal: Build an open-source  
formally verified confidential  
computing technology.**

# Agenda

## **Part I - Confidential Computing Architecture**

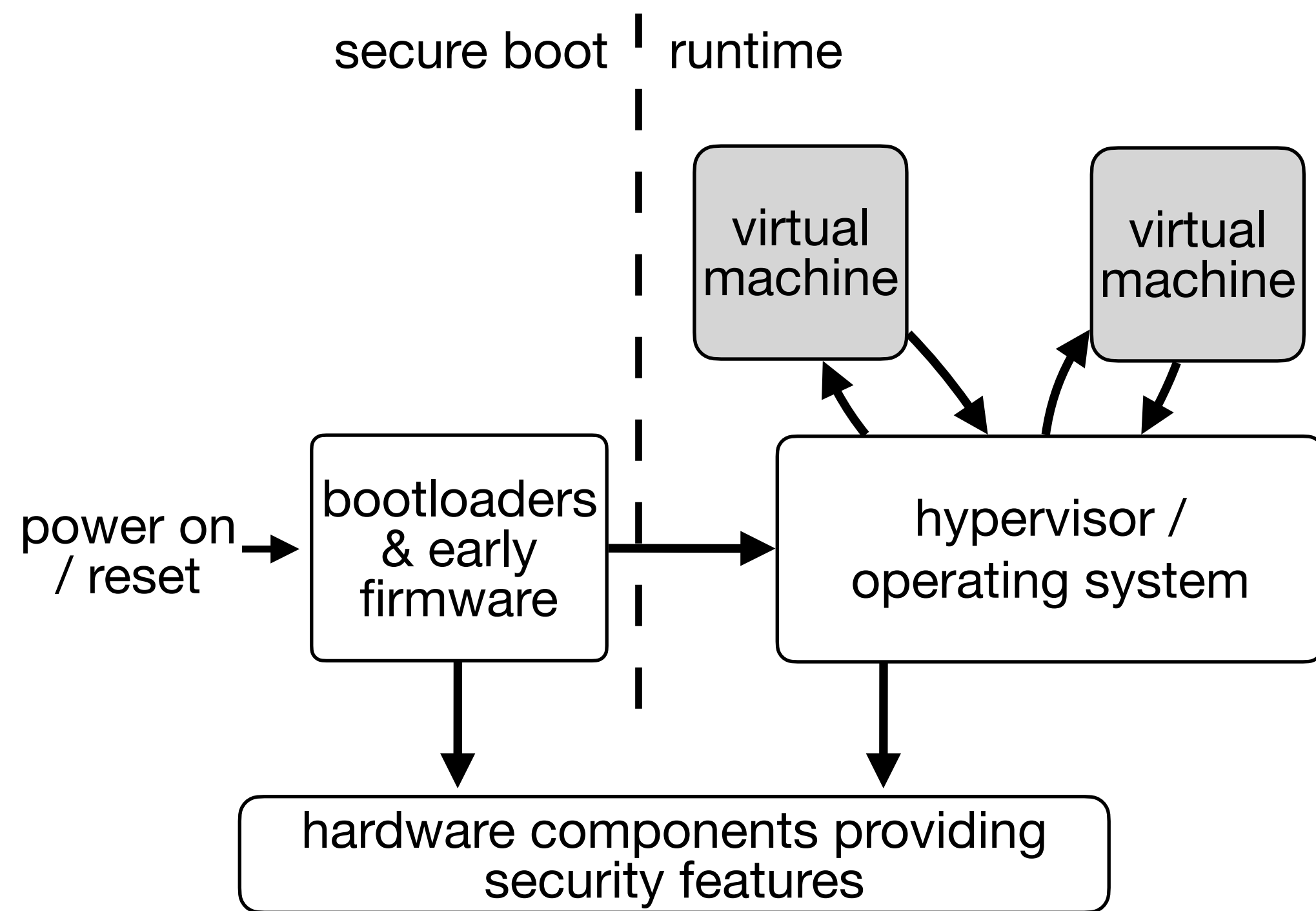
- Traditional vs confidential computing architecture
- Canonical architecture
- ACE: Implementation for RISC-V

## **Part II - Formal Verification**

- Methodology & verification approach
- What has to be proven?
- Demo
- Towards proving security properties



# Traditional (Non-Confidential) Computing Systems

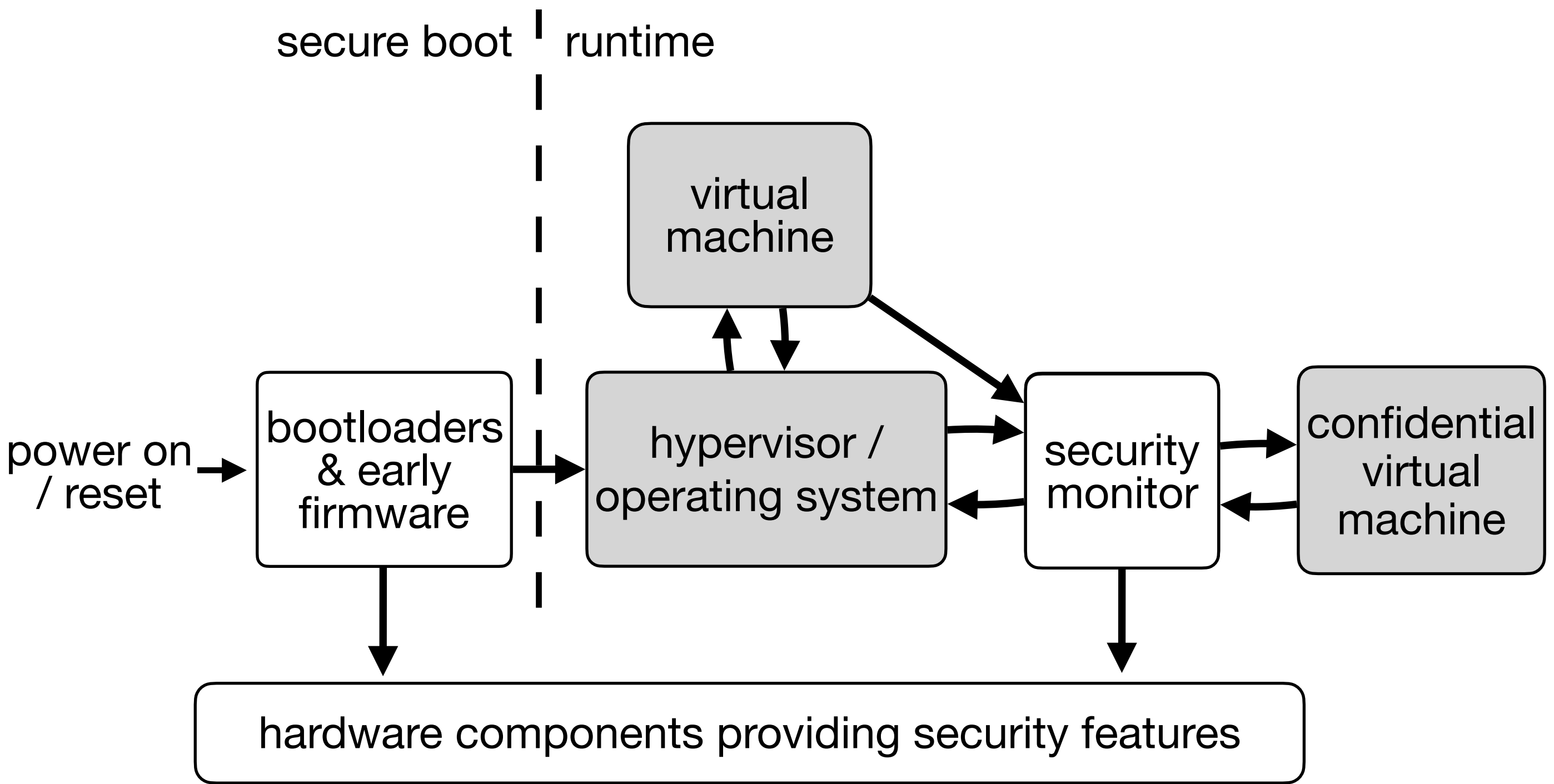


## Security guarantees:

- Isolate virtual machines and hypervisor from other virtual machines
- Hypervisor, firmware, drivers, and system administrator are trusted.
- Linux-based hypervisor consists of more than 10 millions lines of code written in unsafe language.

# Confidential Computing

is a technology that provides infrastructure to run computations confidentially.



### Minimal security guarantees:

- (Confidentiality), integrity of code and its execution.
- Confidentiality and integrity of data.
- No availability guarantees.
- Guarantees to runtime state (no leaks via architectural state or when information stored in the main memory).

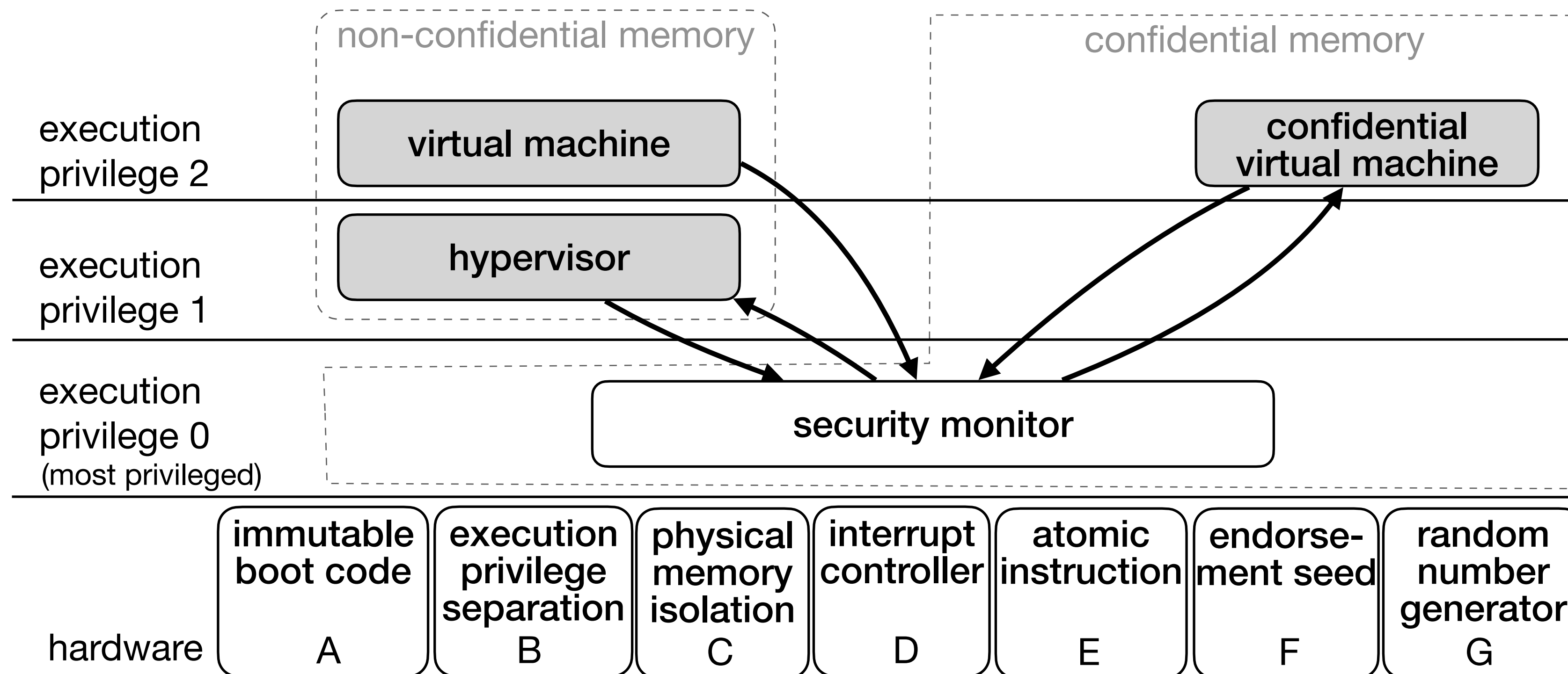
### Threat model:

- Software-level adversary controlling hypervisor, other VMs, confidential VMs, peripheral devices except for the protected confidential VM
- Protections against hardware-level adversary include, for example, memory encryption.

□ - trusted component, subject to the formal verification    ■ - untrusted component    ↗ - control flow direction

# Canonical Architecture

is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.

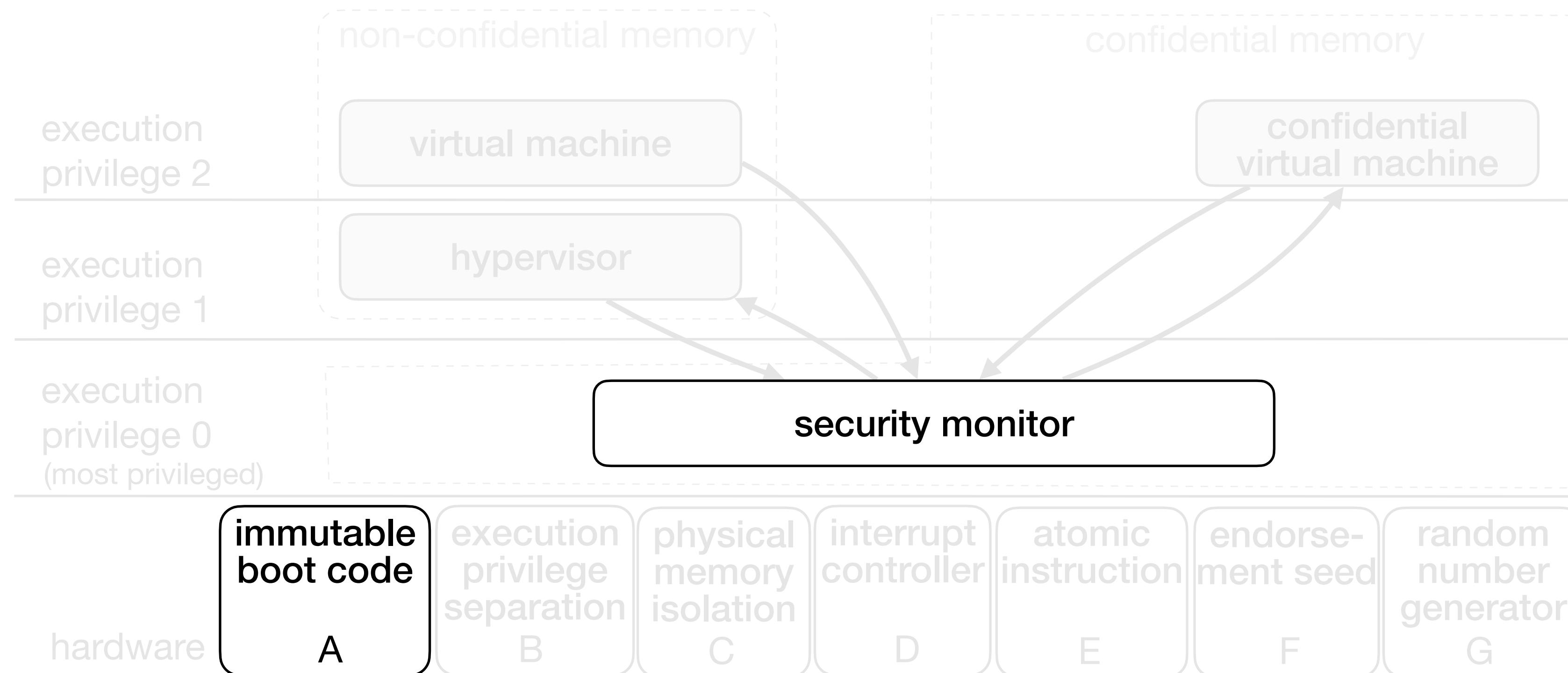


## Hardware components:

- A. Immutable boot code enables integrity- and authenticity-enforced boot of the security monitor.
- B. Execution privilege separation enables partitioning software to create, assign, and enforce roles and access control.
- C. Physical memory isolation allows isolating memory regions by setting and enforcing memory access control.
- D. Interrupt controller enables signalling and execution flow between execution privileges.
- E. Atomic instruction required on multi-core processors to implement synchronisation primitives.
- F. Endorsement seed required for attestation, used to derive attestation key.

# Canonical Architecture

is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.

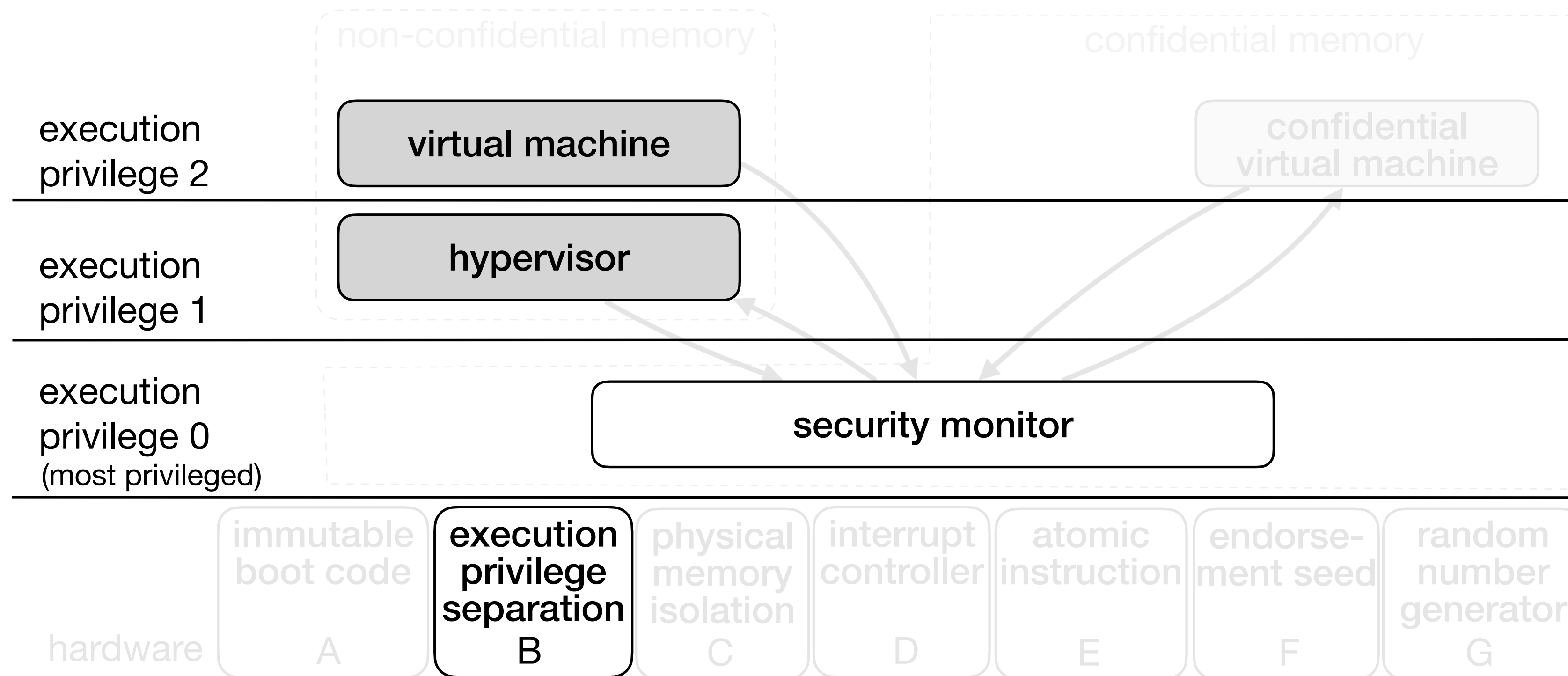


## Hardware components:

- A. Immutable boot code enables integrity- and authenticity-enforced boot of the security monitor.

# Canonical Architecture

is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.

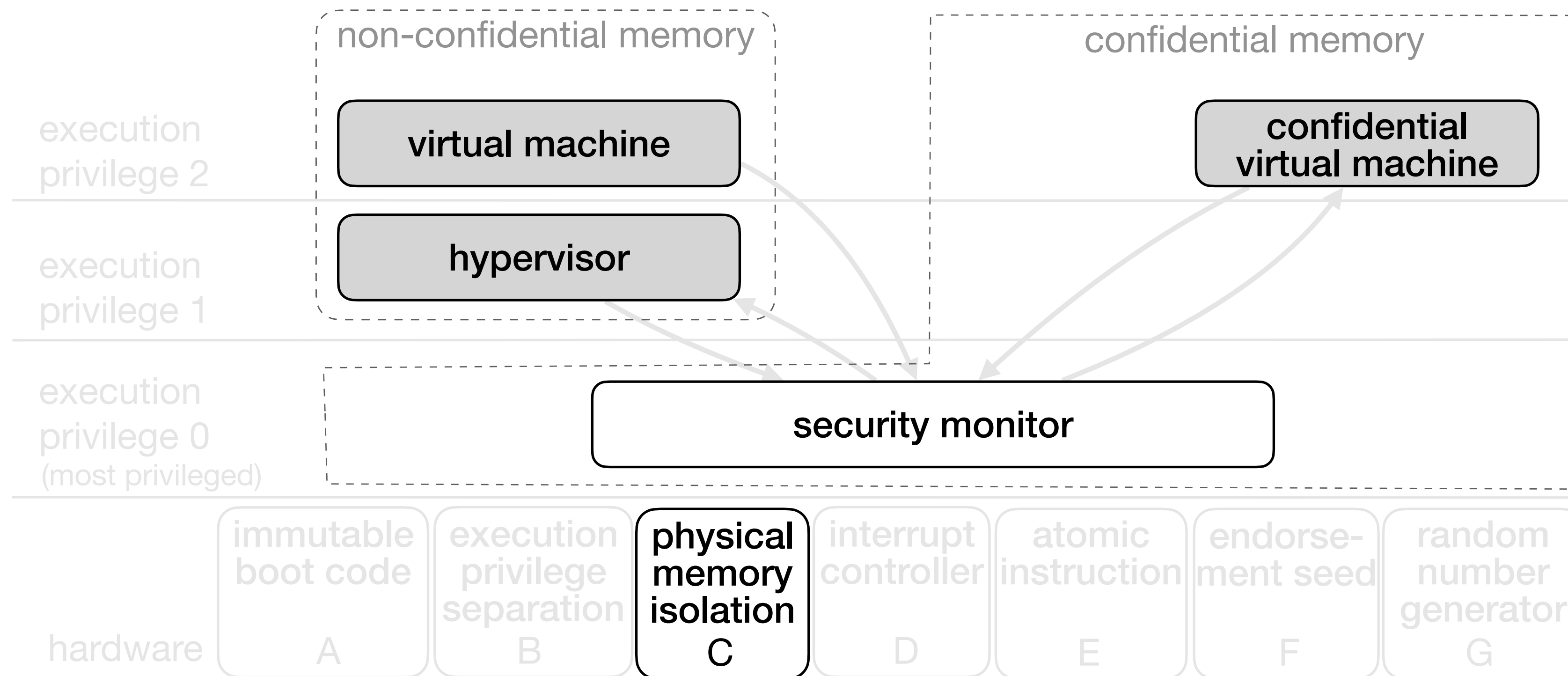


## Hardware components:

- A. Immutable boot code enables integrity- and authenticity-enforced boot of the security monitor.
- B. Execution privilege separation enables partitioning software to create, assign, and enforce roles and access control.

# Canonical Architecture

is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.

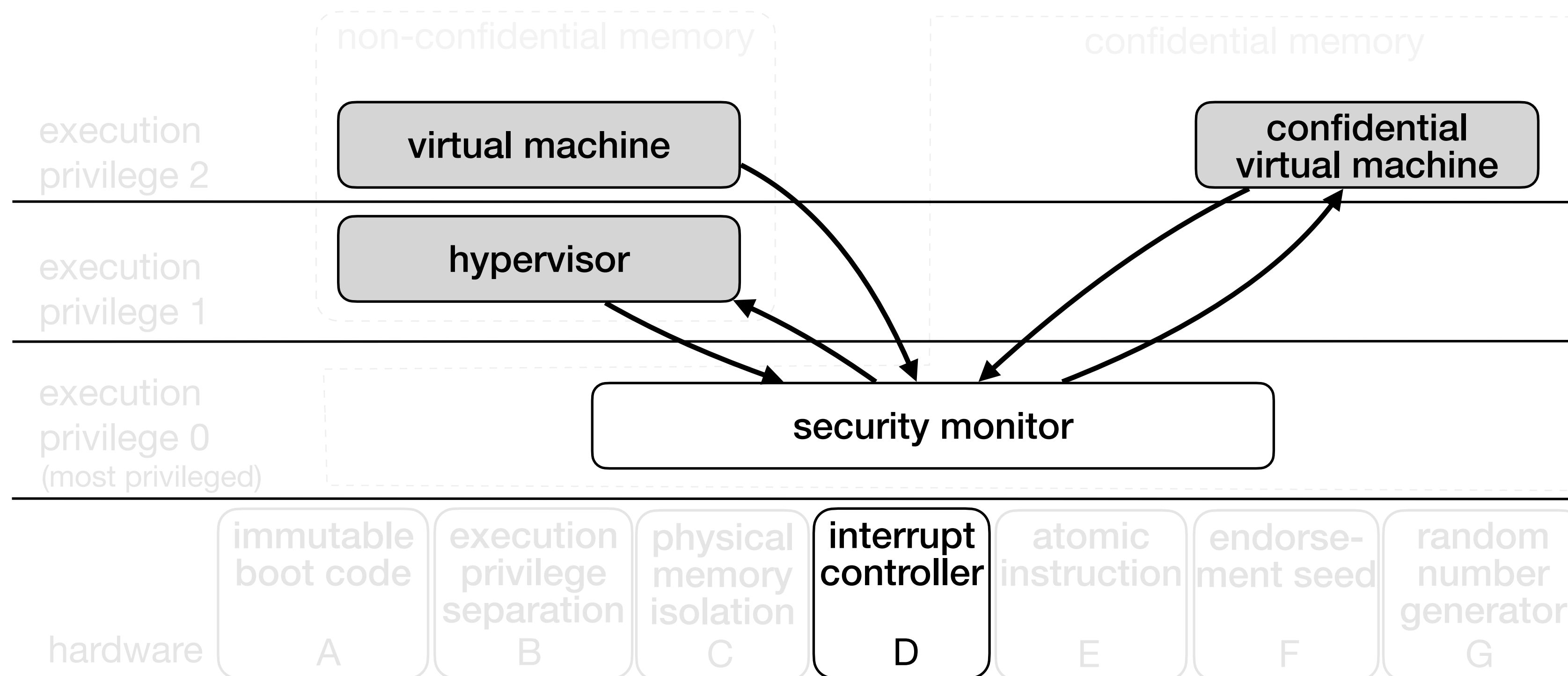


## Hardware components:

- Immutable boot code enables integrity- and authenticity-enforced boot of the security monitor.
- Execution privilege separation enables partitioning software to create, assign, and enforce roles and access control.
- Physical memory isolation allows isolating memory regions by setting and enforcing memory access control.

# Canonical Architecture

is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.

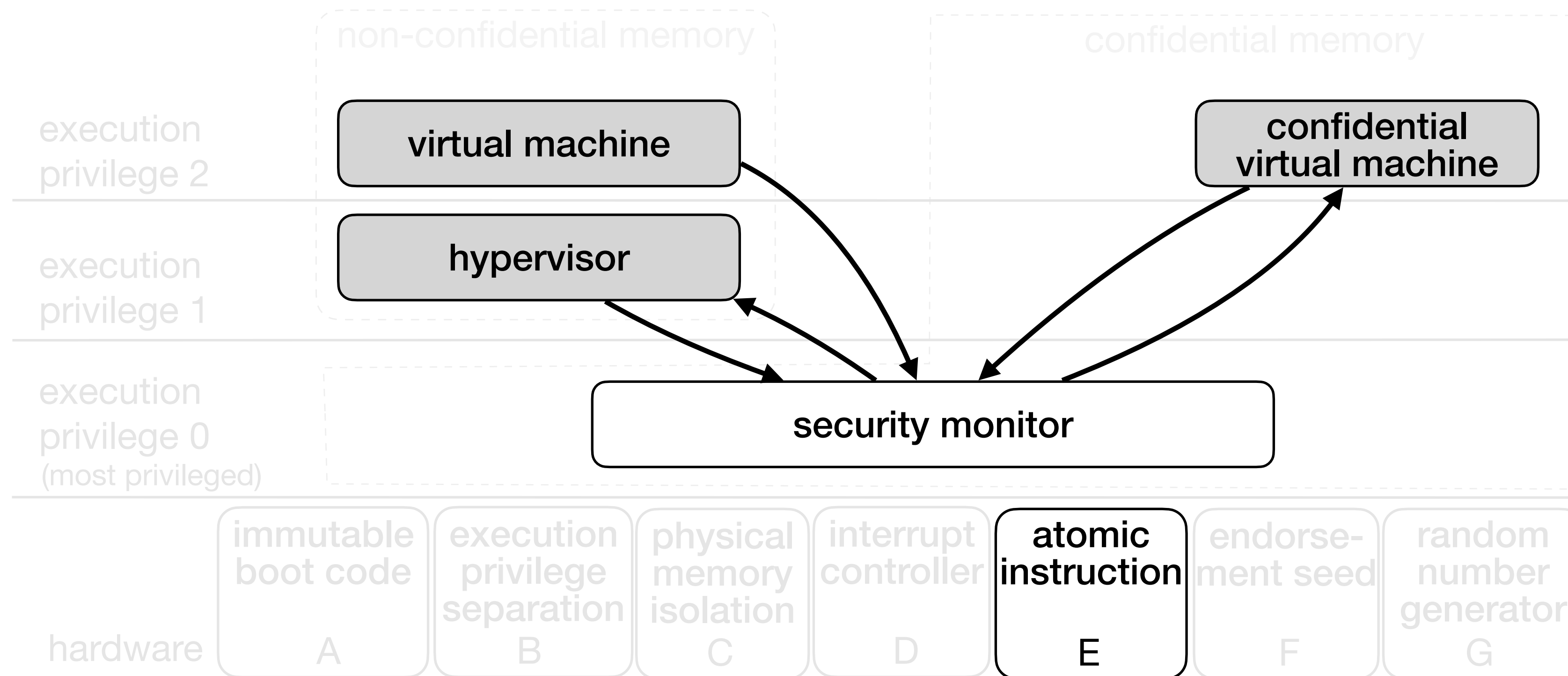


## Hardware components:

- A. Immutable boot code enables integrity- and authenticity-enforced boot of the security monitor.
- B. Execution privilege separation enables partitioning software to create, assign, and enforce roles and access control.
- C. Physical memory isolation allows isolating memory regions by setting and enforcing memory access control.
- D. Interrupt controller enables signalling and flow transition between execution privileges.

# Canonical Architecture

is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.



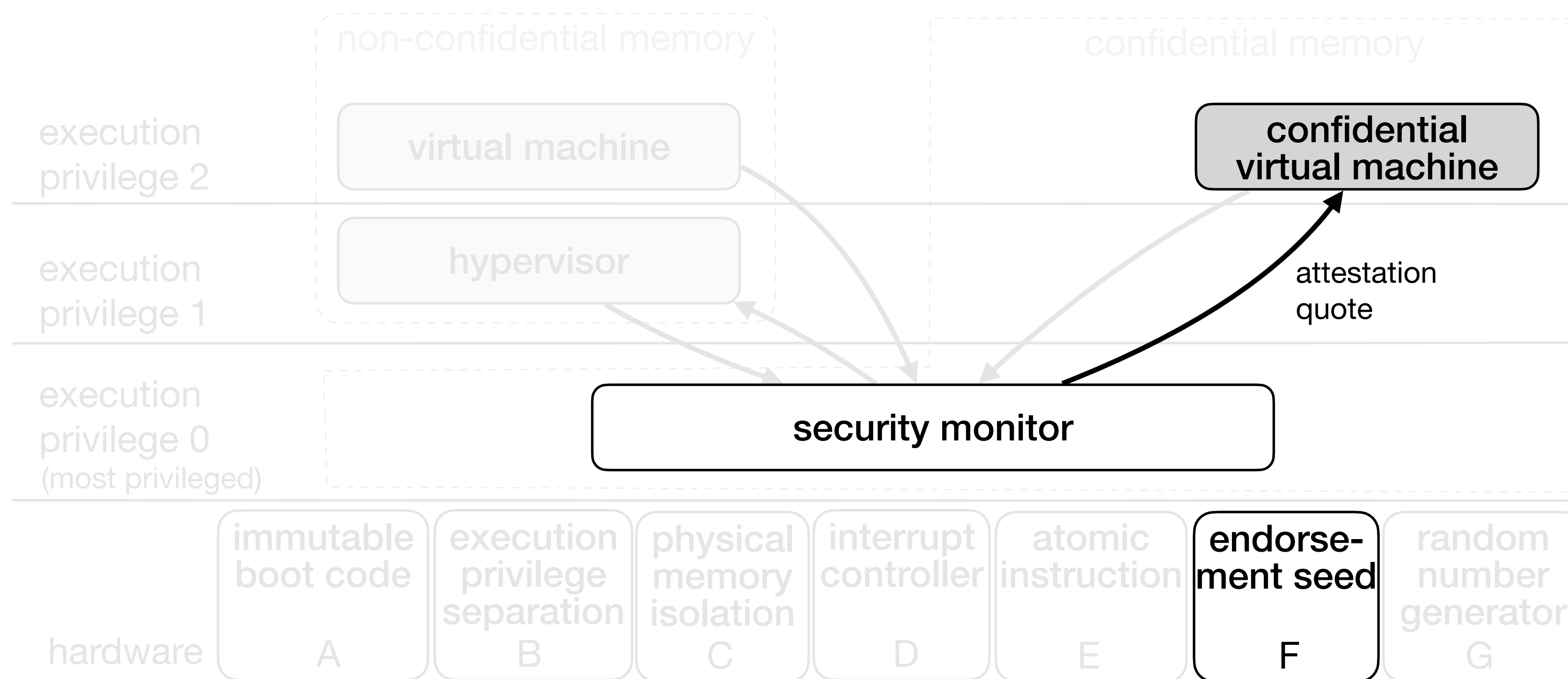
## Hardware components:

- Immutable boot code enables integrity- and authenticity-enforced boot of the security monitor.
- Execution privilege separation enables partitioning software to create, assign, and enforce roles and access control.
- Physical memory isolation allows isolating memory regions by setting and enforcing memory access control.
- Interrupt controller enables signalling and execution flow between execution privileges.
- Atomic instruction** required on multi-core processors to implement synchronisation primitives.



# Canonical Architecture

is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.

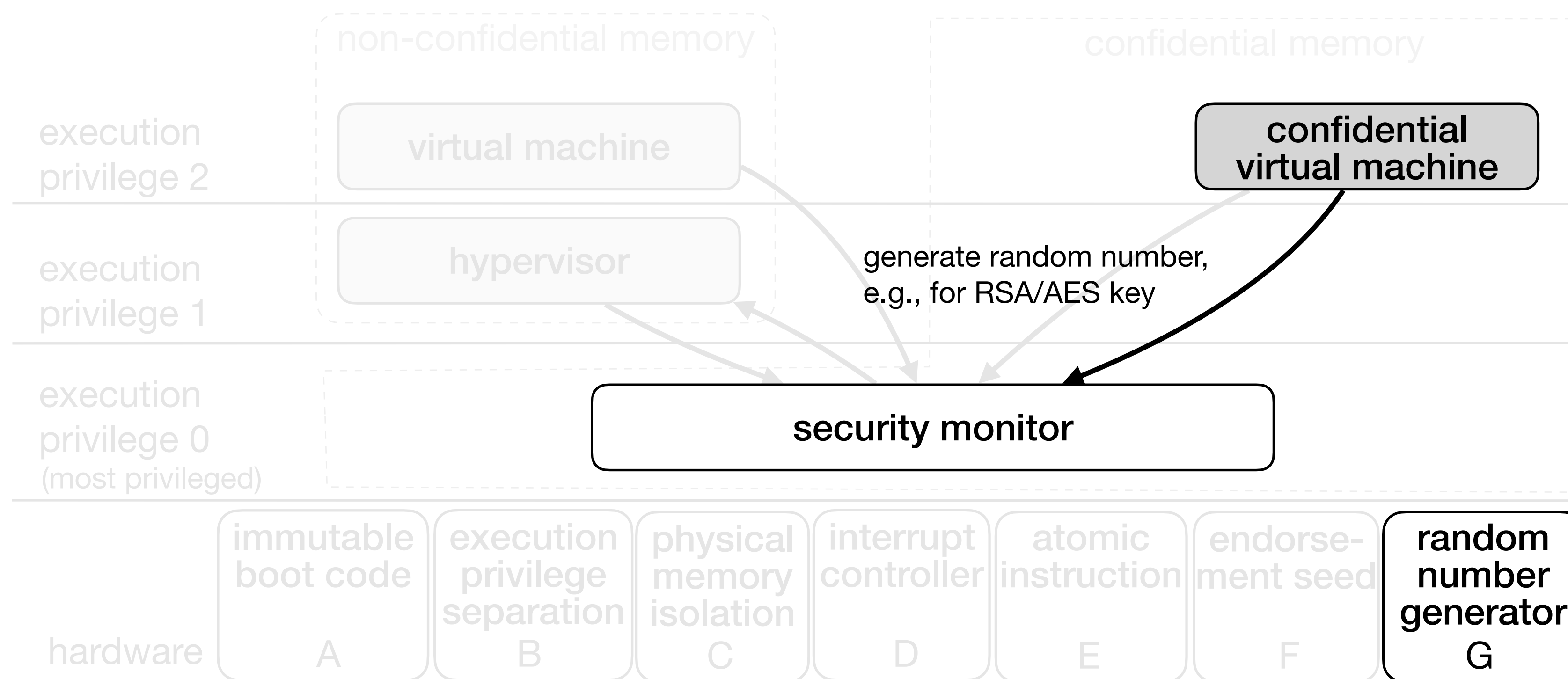


## Hardware components:

- Immutable boot code**  
enables integrity- and authenticity-enforced boot of the security monitor.
- Execution privilege separation**  
enables partitioning software to create, assign, and enforce roles and access control.
- Physical memory isolation**  
allows isolating memory regions by setting and enforcing memory access control.
- Interrupt controller**  
enables signalling and execution flow between execution privileges.
- Atomic instruction**  
required on multi-core processors to implement synchronisation primitives.
- Endorsement seed**  
required for attestation, used to derive an attestation key.

# Canonical Architecture

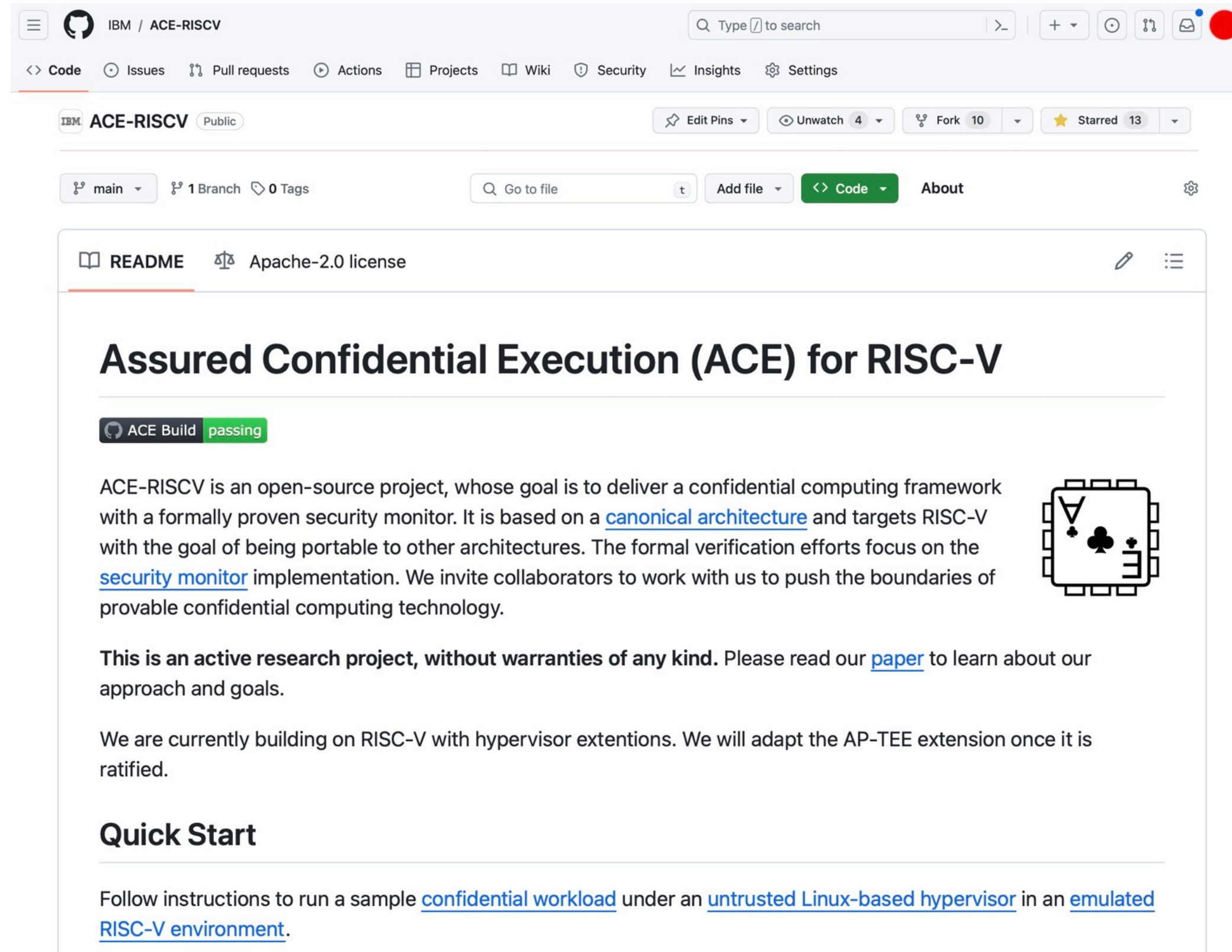
is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.



## Hardware components:

- A. Immutable boot code enables integrity- and authenticity-enforced boot of the security monitor.
- B. Execution privilege separation enables partitioning software to create, assign, and enforce roles and access control.
- C. Physical memory isolation allows isolating memory regions by setting and enforcing memory access control.
- D. Interrupt controller enables signalling and execution flow between execution privileges.
- E. Atomic instruction required on multi-core processors to implement synchronisation primitives.
- F. Endorsement seed required for attestation, used to derive an attestation key.
- G. Random number generator required for cryptographic operations, e.g., for attestation

<https://github.com/IBM/ACE-RISCV>



IBM / ACE-RISCV

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

ACE-RISCV Public

main 1 Branch 0 Tags

Go to file Add file Code About

README Apache-2.0 license

## Assured Confidential Execution (ACE) for RISC-V

ACE Build passing

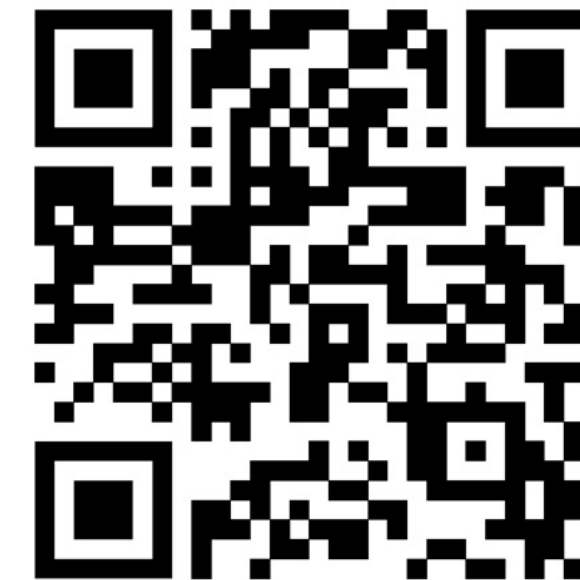
ACE-RISCV is an open-source project, whose goal is to deliver a confidential computing framework with a formally proven security monitor. It is based on a [canonical architecture](#) and targets RISC-V with the goal of being portable to other architectures. The formal verification efforts focus on the [security monitor](#) implementation. We invite collaborators to work with us to push the boundaries of provable confidential computing technology.

**This is an active research project, without warranties of any kind.** Please read our [paper](#) to learn about our approach and goals.

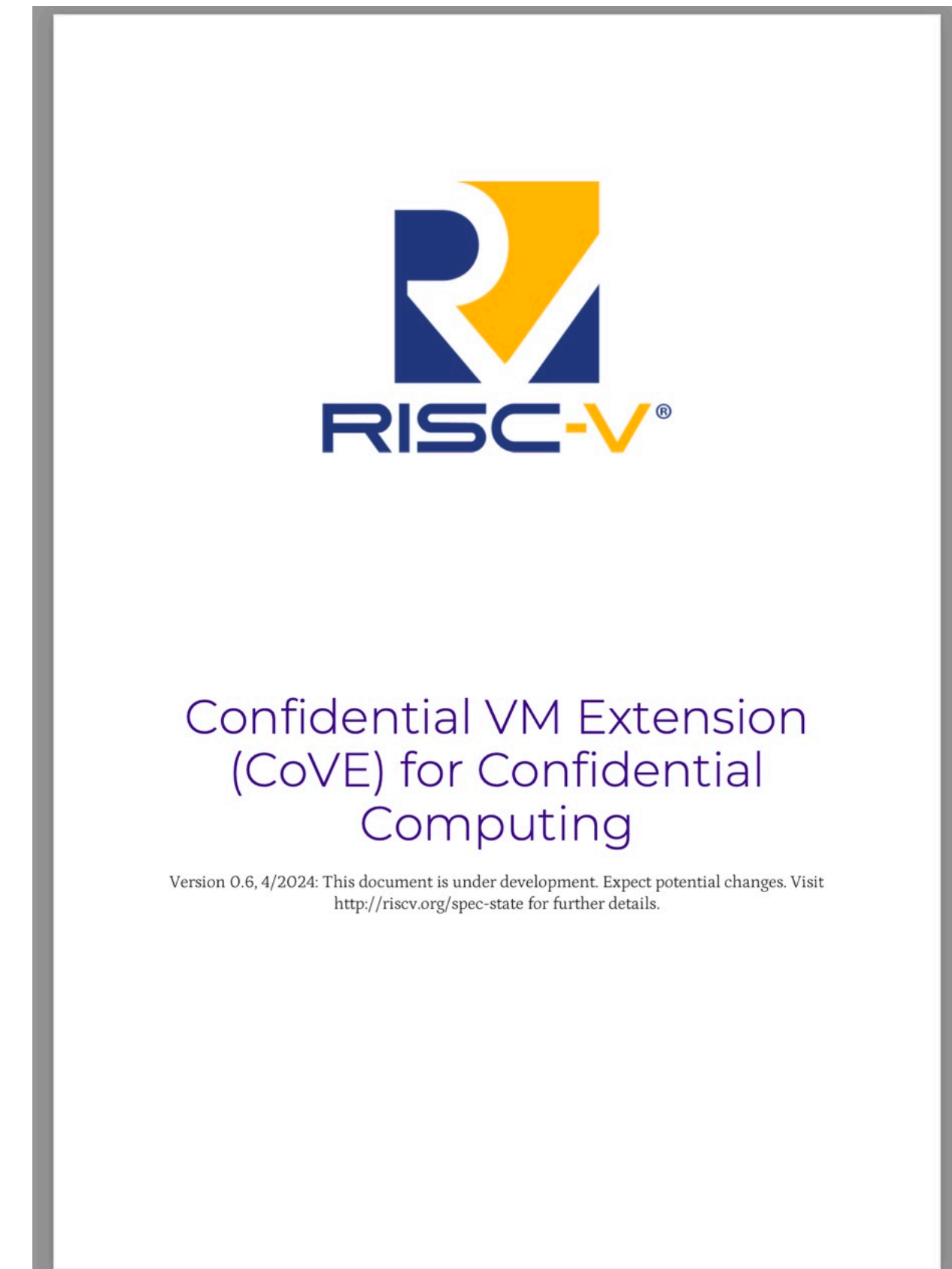
We are currently building on RISC-V with hypervisor extensions. We will adapt the AP-TEE extension once it is ratified.

### Quick Start

Follow instructions to run a sample [confidential workload](#) under an [untrusted Linux-based hypervisor](#) in an [emulated RISC-V environment](#).



<https://github.com/riscv-non-isa/riscv-ap-tee>



RISC-V®

## Confidential VM Extension (CoVE) for Confidential Computing

Version 0.6, 4/2024: This document is under development. Expect potential changes. Visit <http://riscv.org/spec-state> for further details.

# Agenda

## **Part I - Confidential Computing Architecture**

- Traditional vs confidential computing architecture
- Canonical architecture
- ACE: Implementation for RISC-V

## **Part II - Formal Verification**

- Methodology & verification approach
- What has to be proven?
- Demo
- Towards proving security properties

# Where are formal methods used?

**CompCert**

Formally verified C compiler

**seL4**

Verified microkernel

**IBM Formal ML**

Formalised probability theory

**HACL\***

High assurance cryptographic primitives

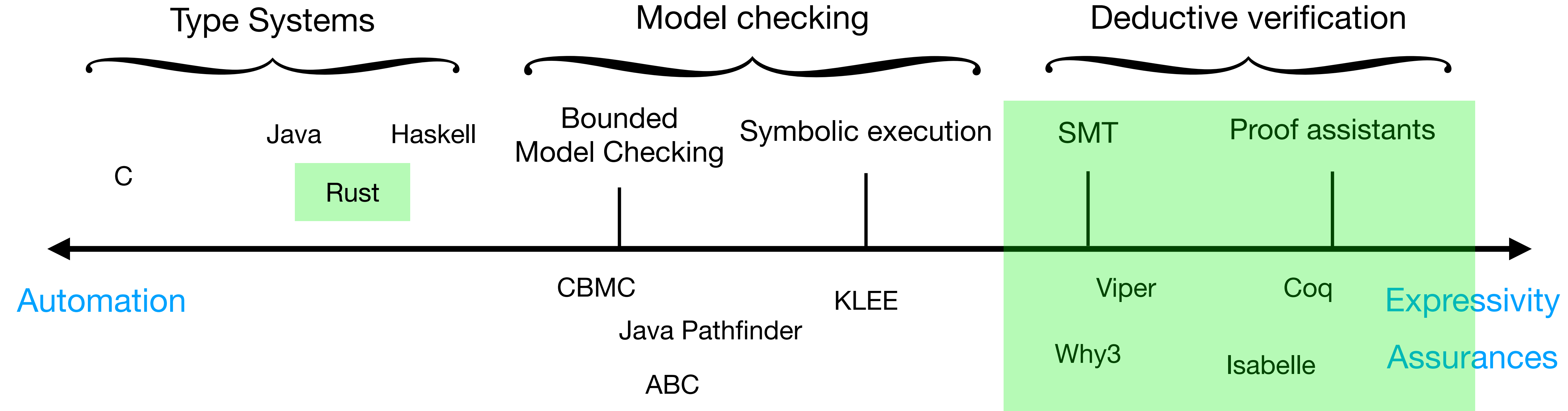
**IBM HSM**

Certified Hardware Security Module

**SLAM**

Property checks for Windows drivers

# Techniques in formal verification (non-exhaustive)

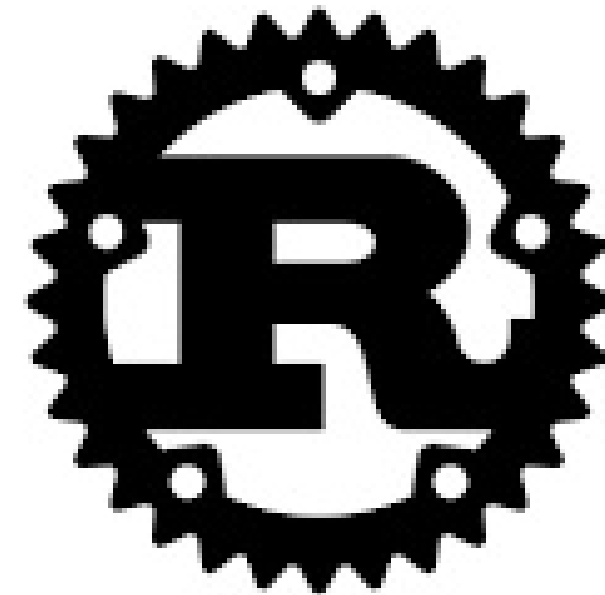


We use a two-pronged approach

# The Rust programming language provides safety

Systems programming  
with zero-cost abstractions  
for memory management

Growing ecosystem and  
increasing popularity



Brings modern programming paradigms  
to systems programming

Aims to provide memory safety for free(\*):

- no null-pointer accesses
- no use-after-free
- no data races
- ...

(\* ) more work if you use unsafe code

# Deductive verification using RefinedRust

To appear at  
PLDI'24

**Goal:** verify memory safety (of unsafe code) & functional correctness

**Automatic translation**

Rust  $\Rightarrow$  Radium

**Proof automation**  
guiding application of  
typing rules

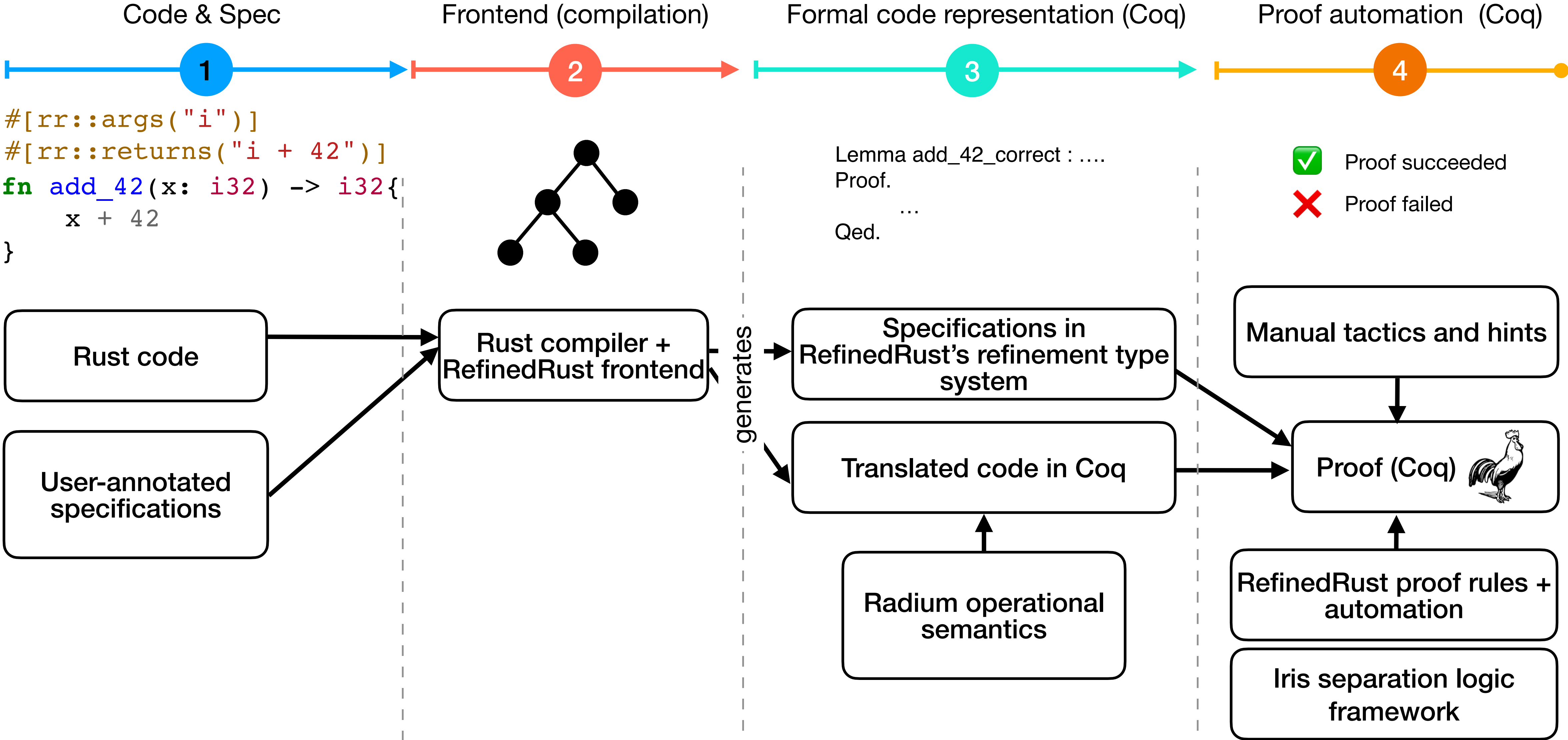
**Formal model of Rust:**  
**Radium** operational  
semantics

**Refinement type system**  
with semantic soundness  
proof

Coq proof assistant 



# Architecture of RefinedRust



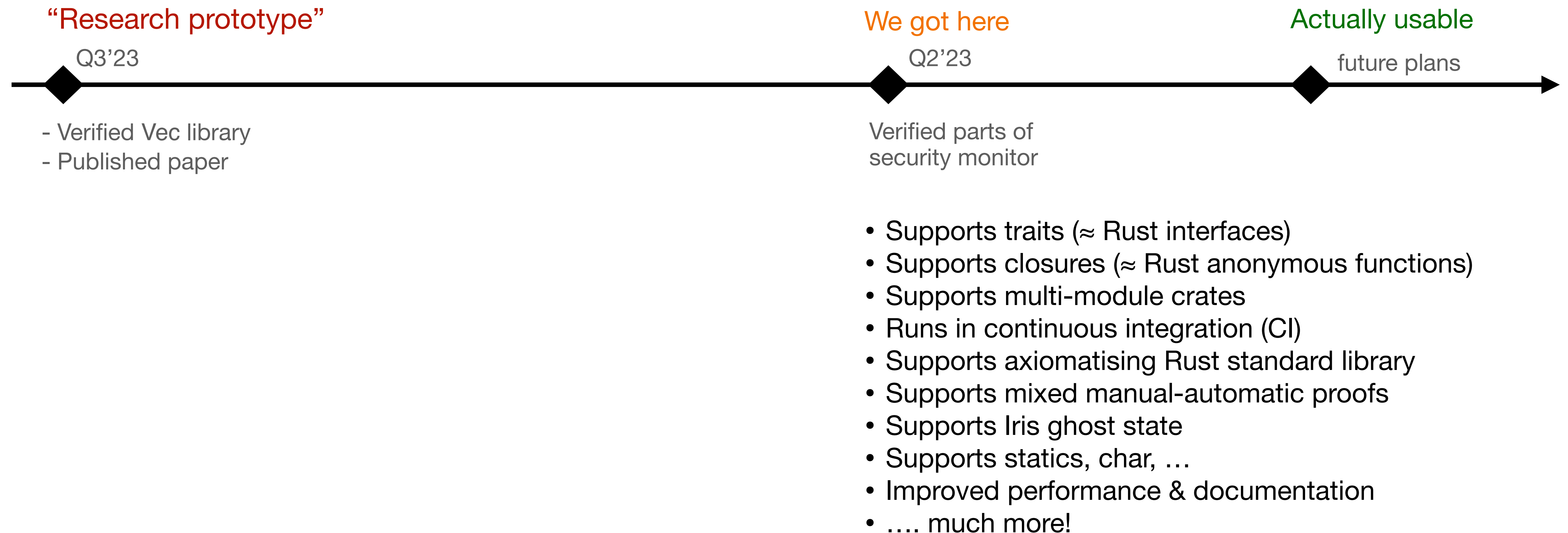
Designed to be run continuously in CI

# Example: specifying pre- and postconditions

*/// Add 42 to the argument x and return the result.*

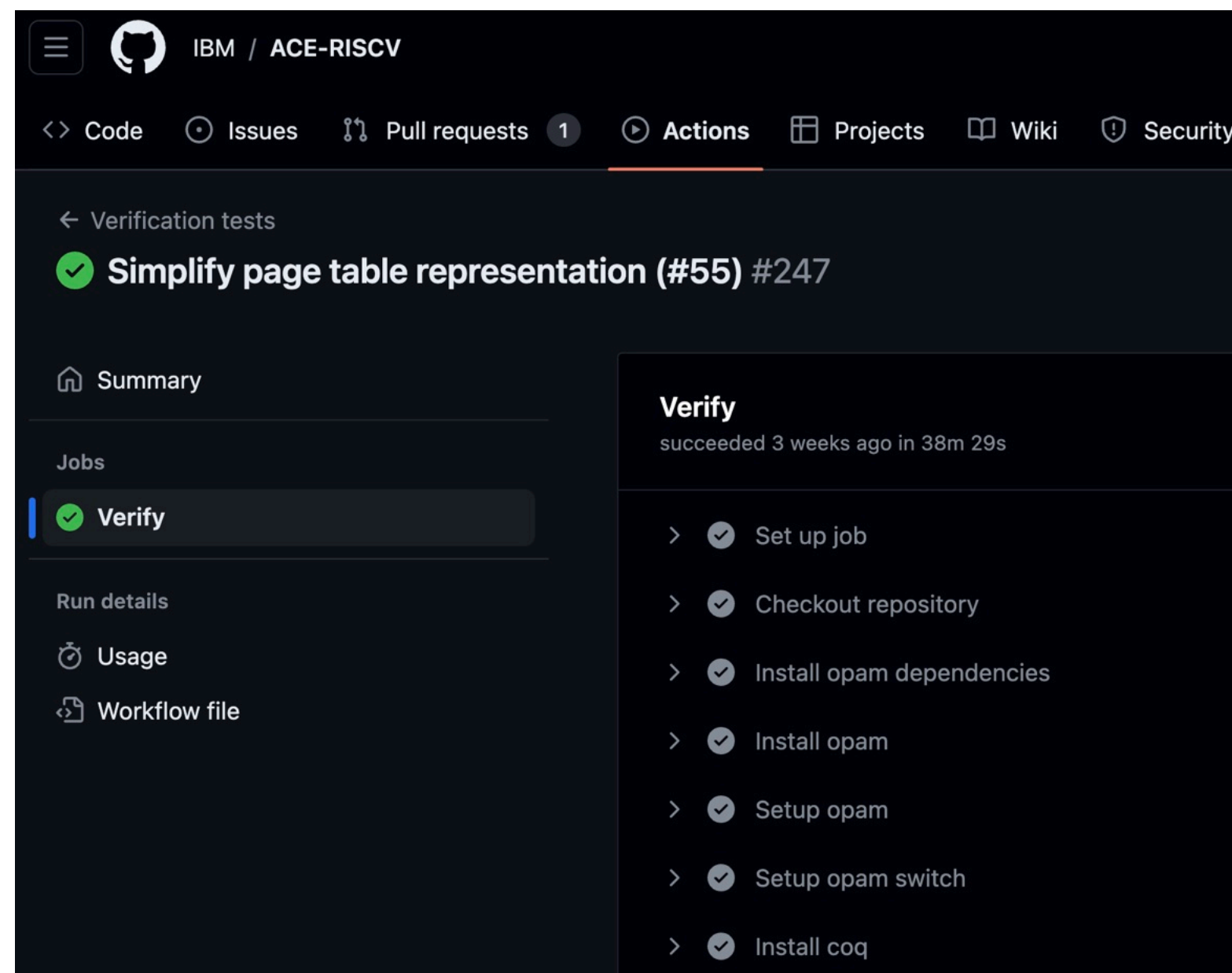
```
fn add_42(x: i32) -> i32 {  
    x + 42  
}
```

# How practical is RefinedRust?



# Making RefinedRust practical: CI & multi-crates

## CI integration



## Cargo integration

Just run ``cargo refinedrust`` to call RefinedRust on the whole crate

## Axiomatising library functions

```
#[rr::only_spec]
#[rr::export_as(spin::once::Once)]
#[rr::context("onceG  $\Sigma$  ({rt_of T})")]
impl<T, R> Once<T, R> {
    /// Creates a new [`Once`].
    #[rr::exists("η")]
    #[rr::ensures(#iris "once_status_tok η None")]
    #[rr::returns("η")]
    pub const fn new() -> Self {
        unimplemented!();
    }

    /// Creates a new initialized [`Once`].
    #[rr::params("x")]
    #[rr::args("x")]
    #[rr::exists("η")]
    #[rr::ensures(#iris "once_status_tok η (Some x)")]
    #[rr::returns("η")]
    pub const fn initialized(data: T) -> Self {
        unimplemented!();
    }
}
```

# Making RefinedRust practical: traits

Traits are Rust's way of specifying generic interfaces

- Using many basic elements of the standard library requires support for traits
- We allow specialisation of specifications for particular implementations

```
#[rr::context("Ordered {rt_of Self}")]  
pub trait Ord: Eq + PartialOrd {  
    // Compare the two elements.  
    #[rr::params("x", "y")]  
    #[rr::args("#x", "#y")]  
    #[rr::returns("ord_cmp x y")]  
    fn cmp(&self, other: &Self) -> Ordering;  
}
```

# Making RefinedRust practical: closures

Closures are anonymous functions:

```
fn mul_two(x: &mut Vec<i32>) {
    x.iter_mut().map(
        #[rr::args("(x, x')")]
        #[rr::observe("x'": "2*x")]
        |x| *x *= 2
    ).for_each(drop);
}
```

Closures may (mutably) capture their context:

```
fn add_indices(x: &mut Vec<i32>) {
    let mut count = 0;
    x.iter_mut().map(
        #[rr::args("#x, x'")]
        #[rr::capture("count": "i" -> "1+i")]
        #[rr::observe("x'": "i+x")]
        |x| { *x += count; count += 1; }
    ).for_each(drop);
}
```

# RefinedRust vs other Rust verification approaches

	Tech	Unsafe code	Traits & closures	Verified proofs	Automated	User-friendly	Extensible	Concurrency Aware
<b>RefinedRust</b>	Coq	✓	✓	✓	~	~	✓	✓
<b>Prusti</b>	SMT	✗	~	✗	✓	✓	✗	✓
<b>Creusot</b>	SMT	✗	✓	✗	✓	✓	✗	✓
<b>Aeneas</b>	Mixed	✗	✗	~	~	~	✓	✓
<b>Verus</b>	SMT	~	~	✗	✓	✓	~	✓
<b>Kani</b>	Model Checking	✓	~	✗	✓	✓	✗	✗

Future Work: combine advantages of different systems

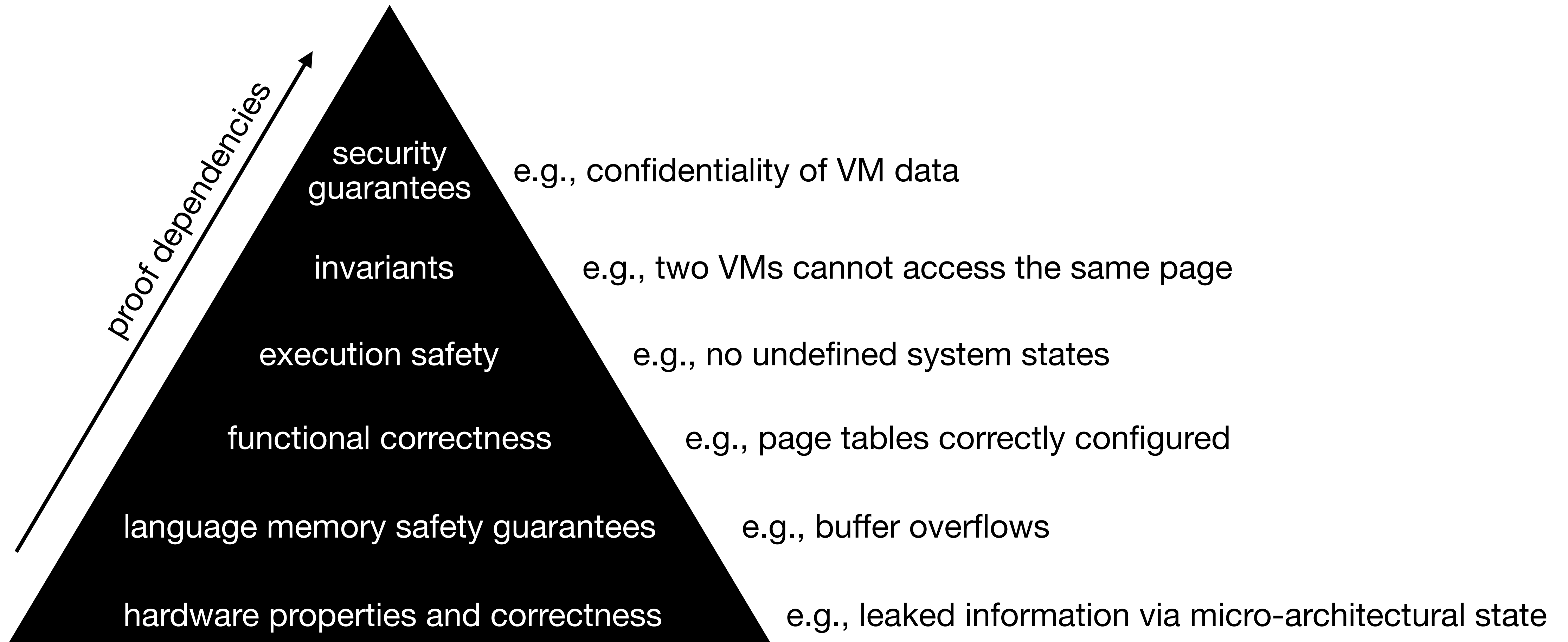
# ACE: What has to be proven?



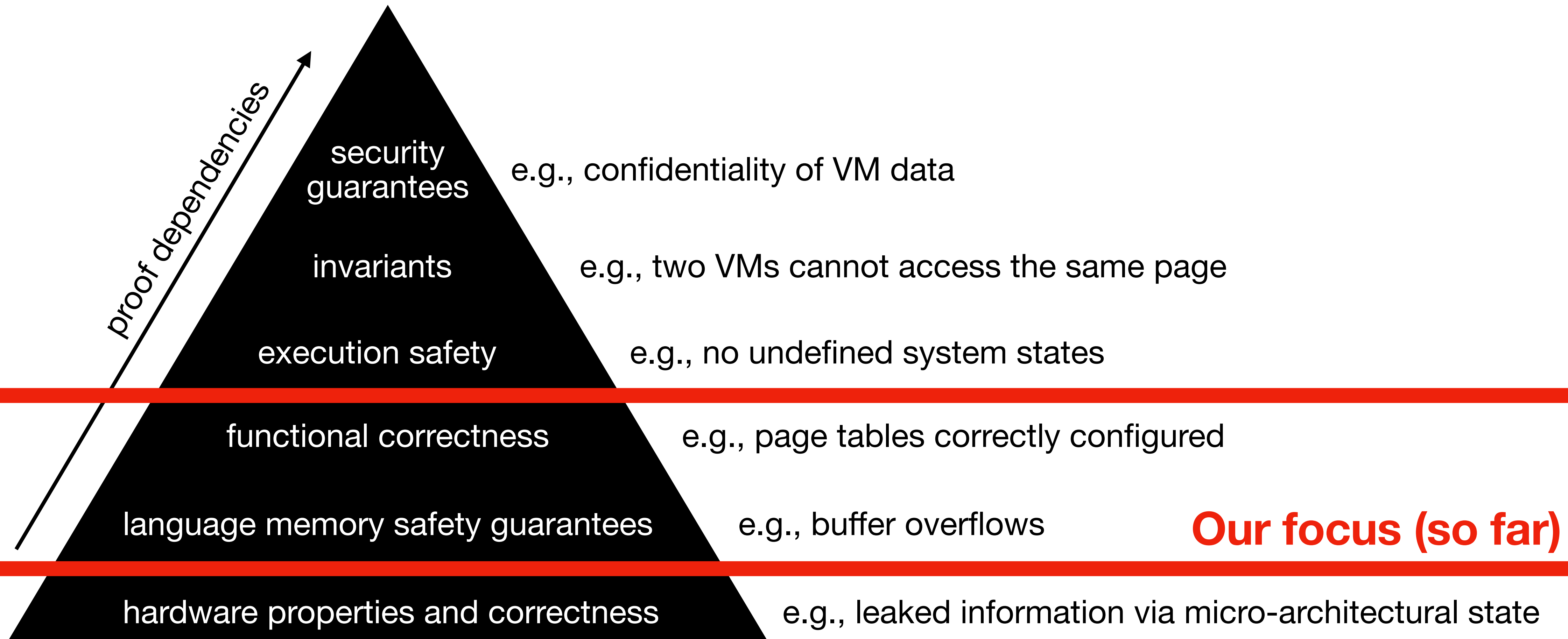
security  
guarantees



# ACE: What has to be proven?

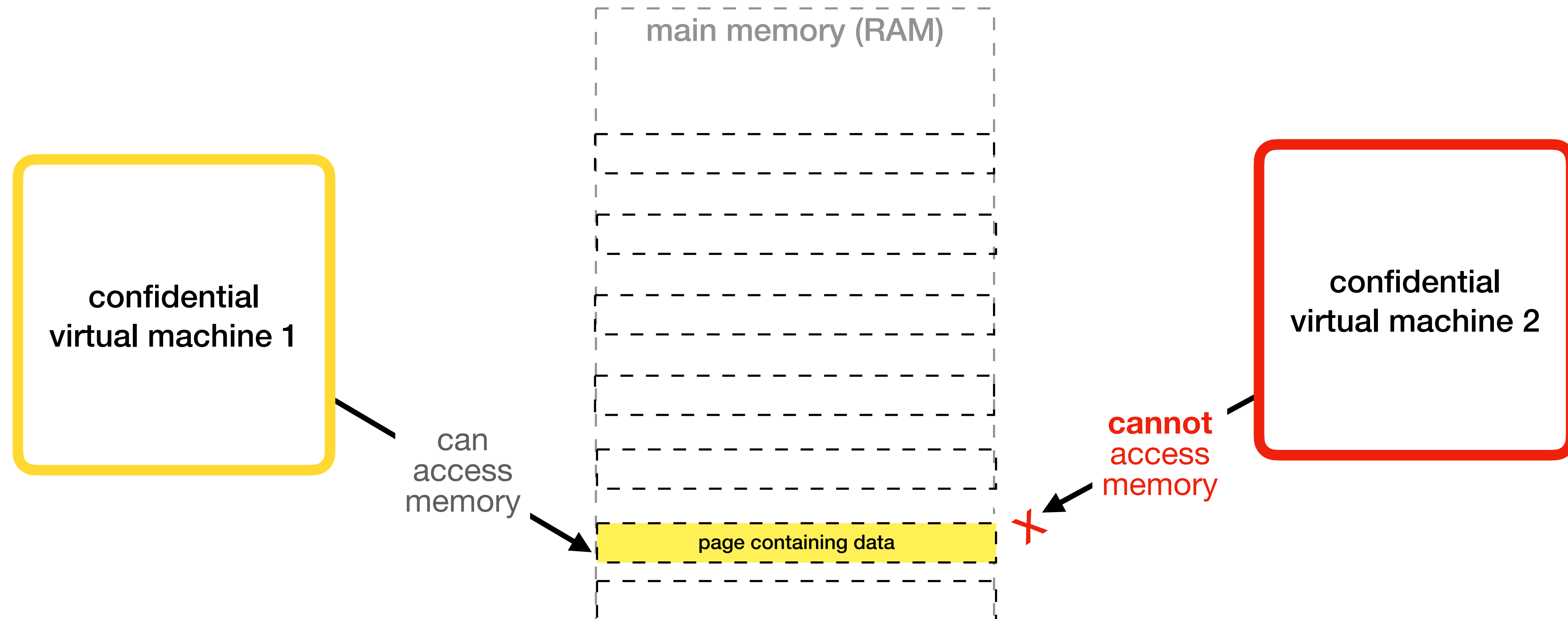


# ACE: What has to be proven?



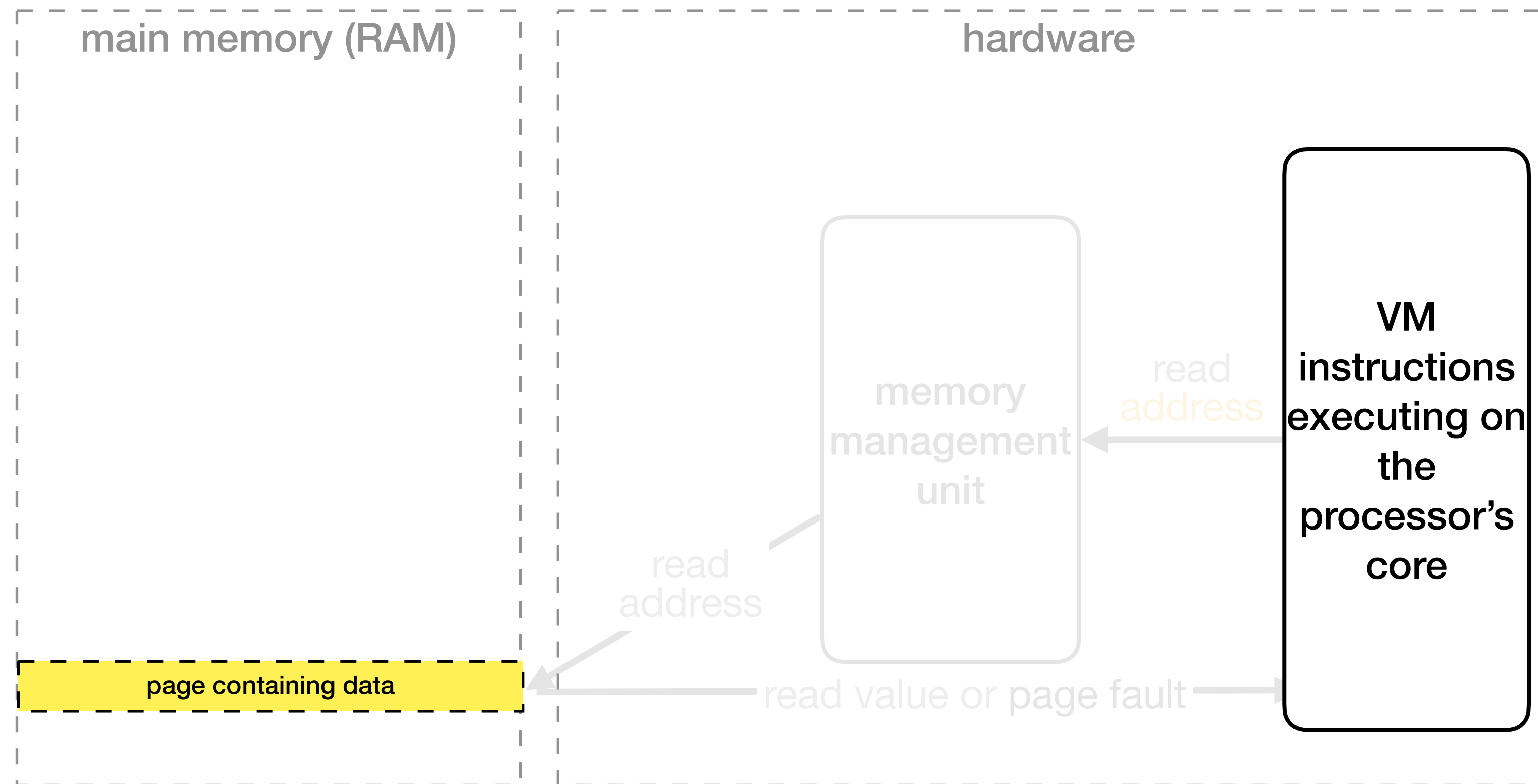
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



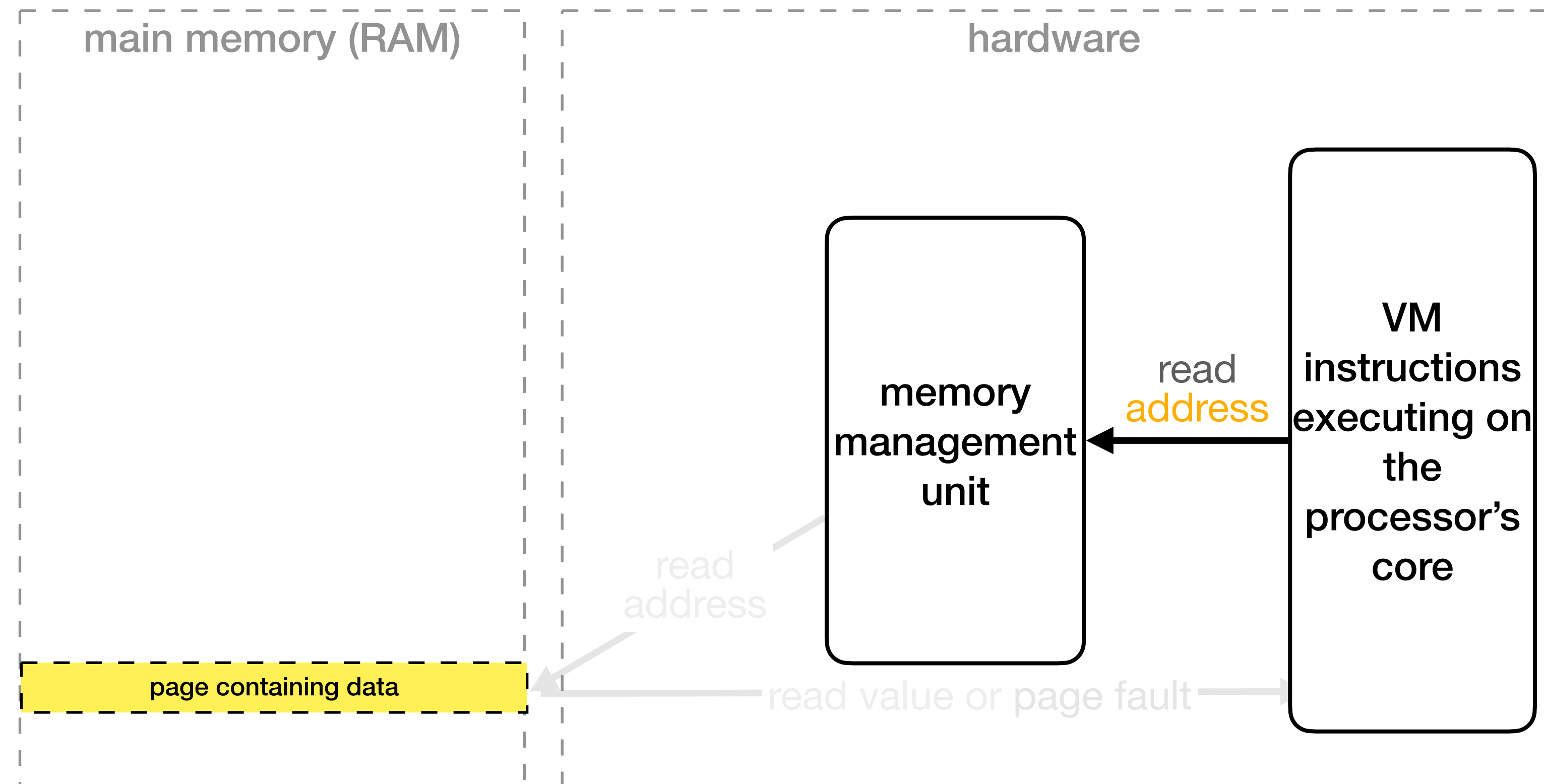
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



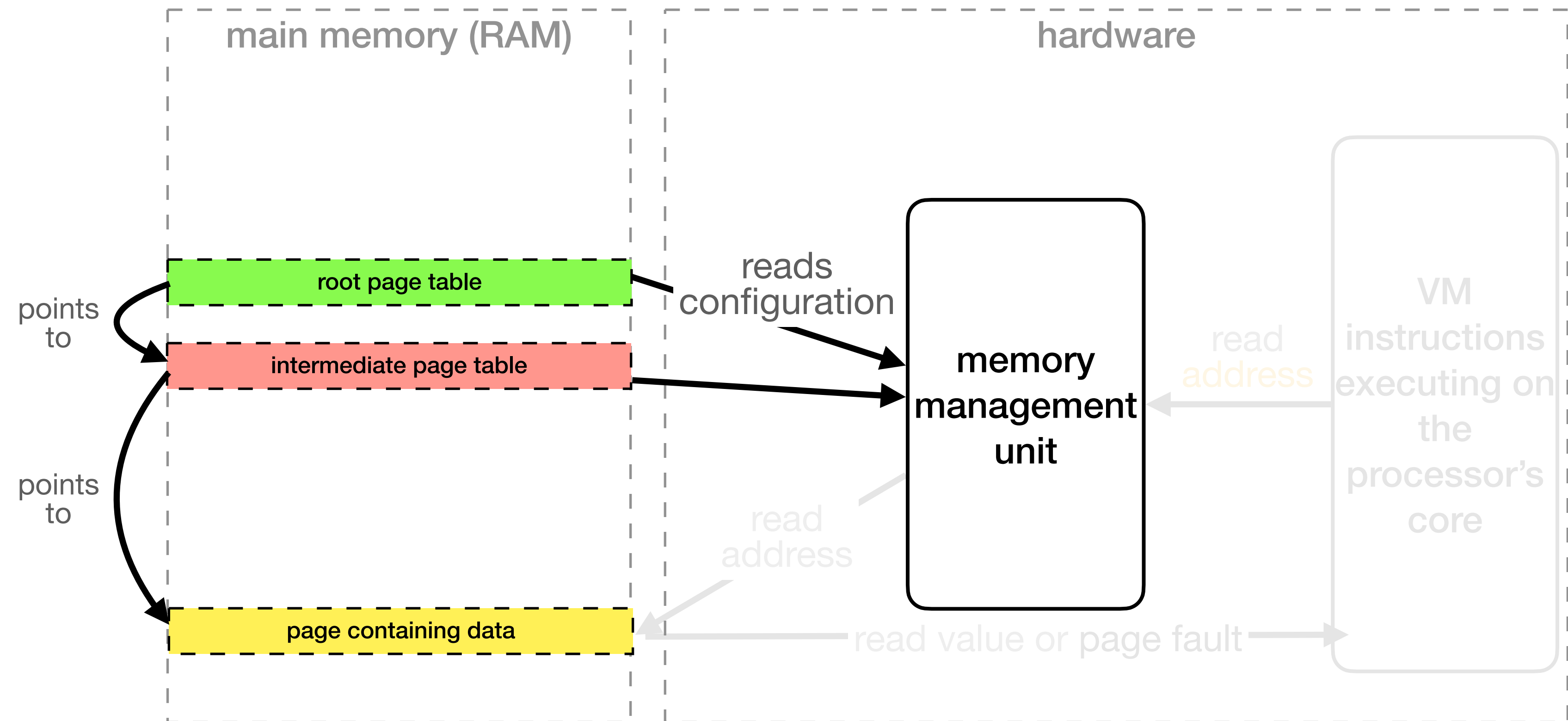
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



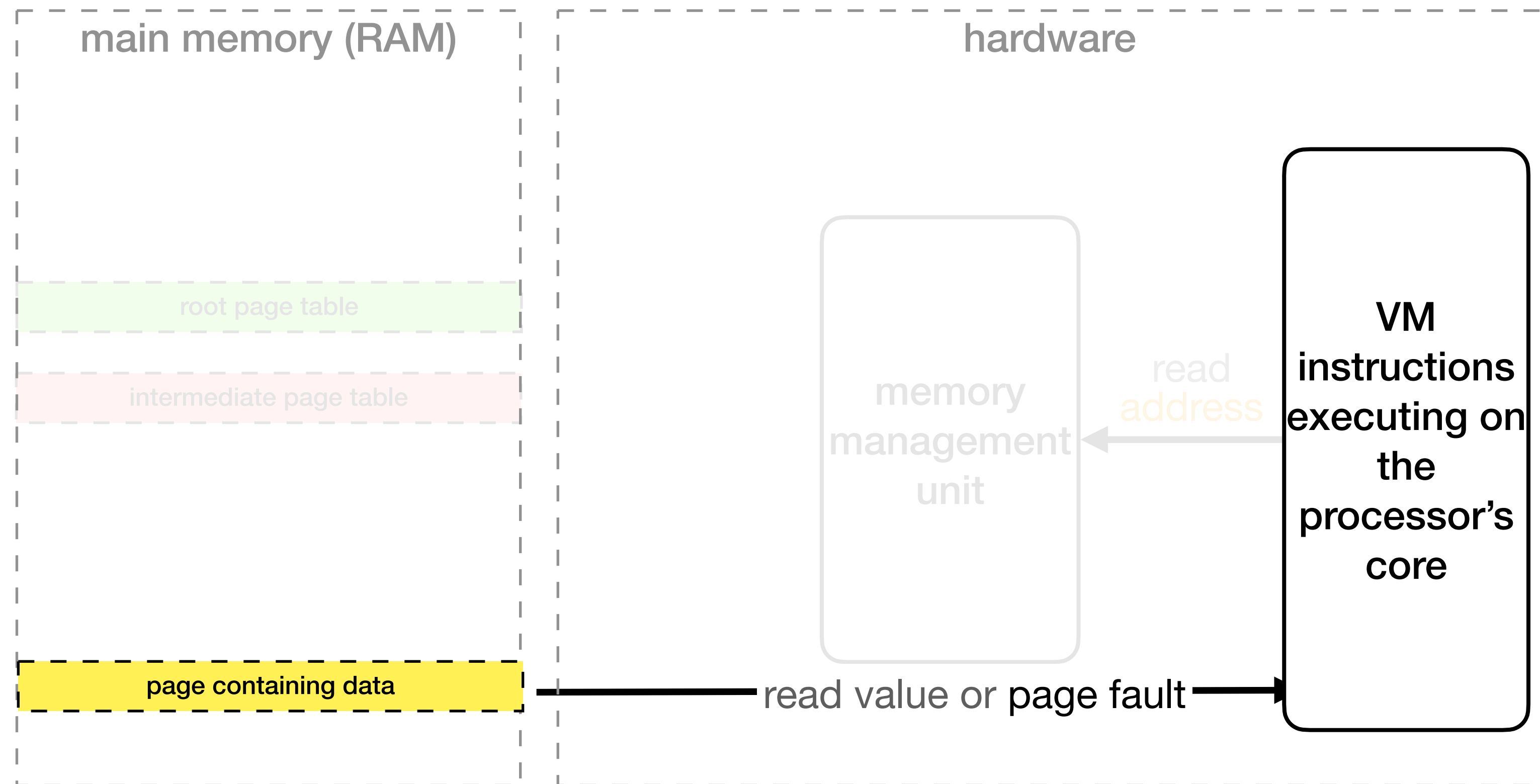
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



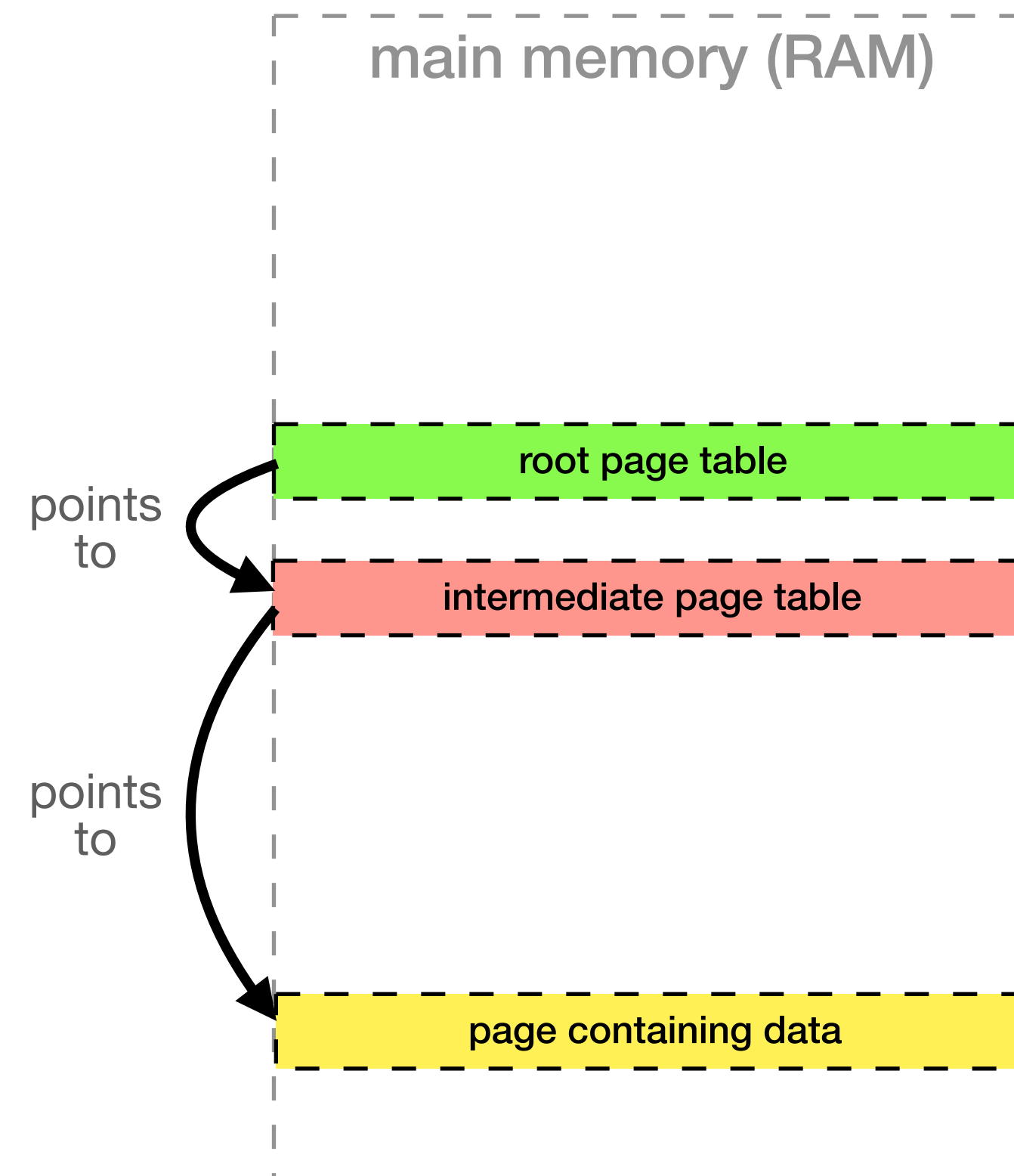
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



We must formally verify the functional correctness of the page table configuration.

Let's leverage Rust's type system with its ownership and memory safety guarantees!



# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.

initialization procedure executed at boot time

main memory (RAM)

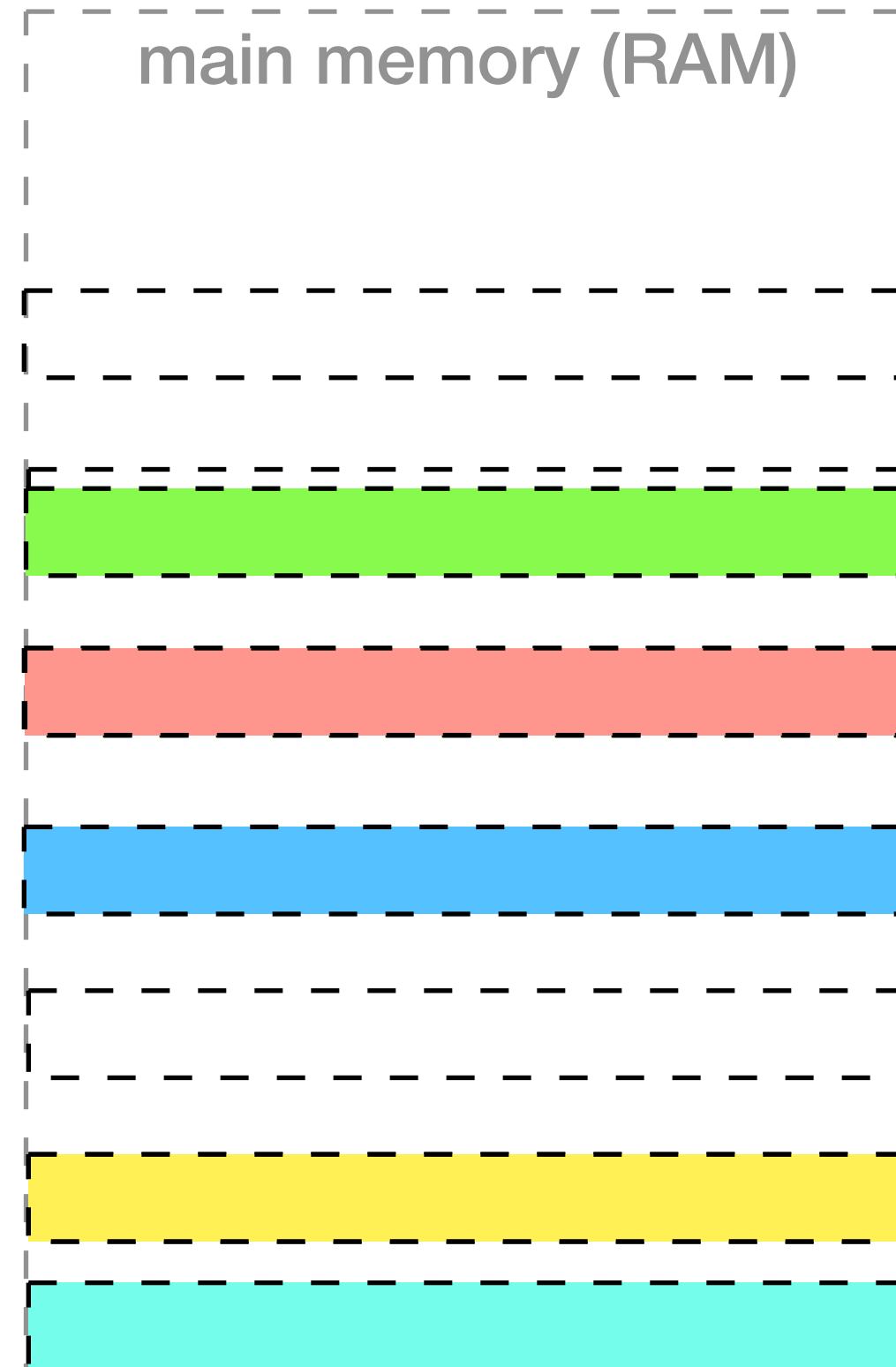
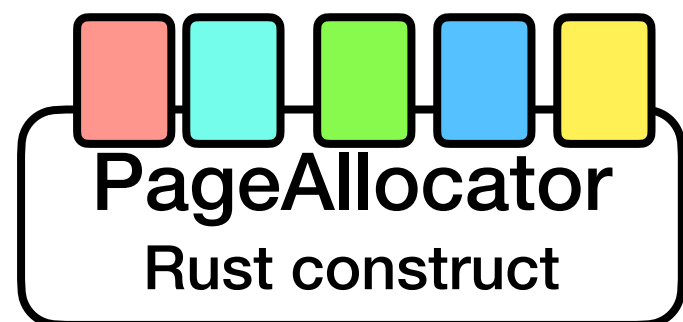
**PageAllocator**  
Rust construct

# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.

initialization procedure executed at boot time

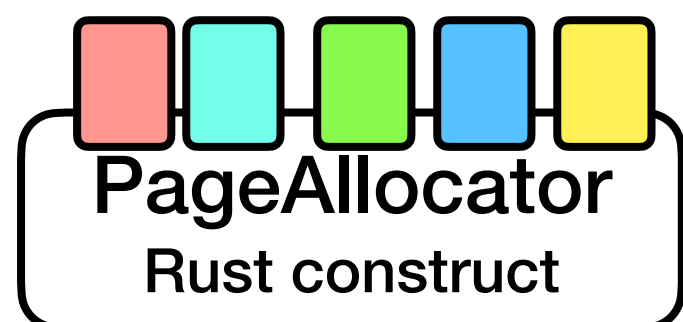
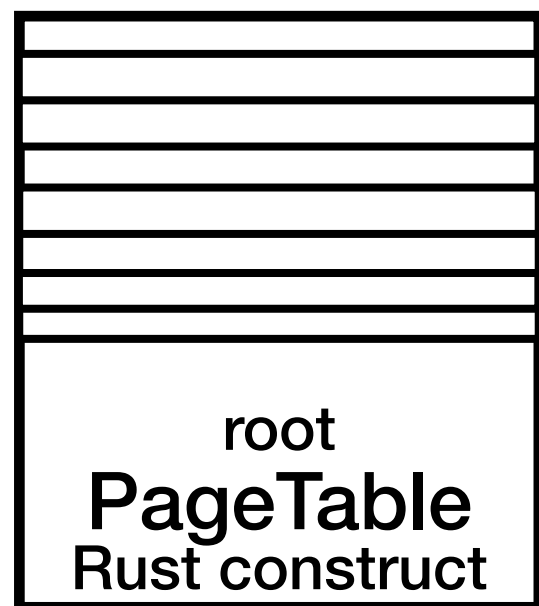
main memory (RAM)



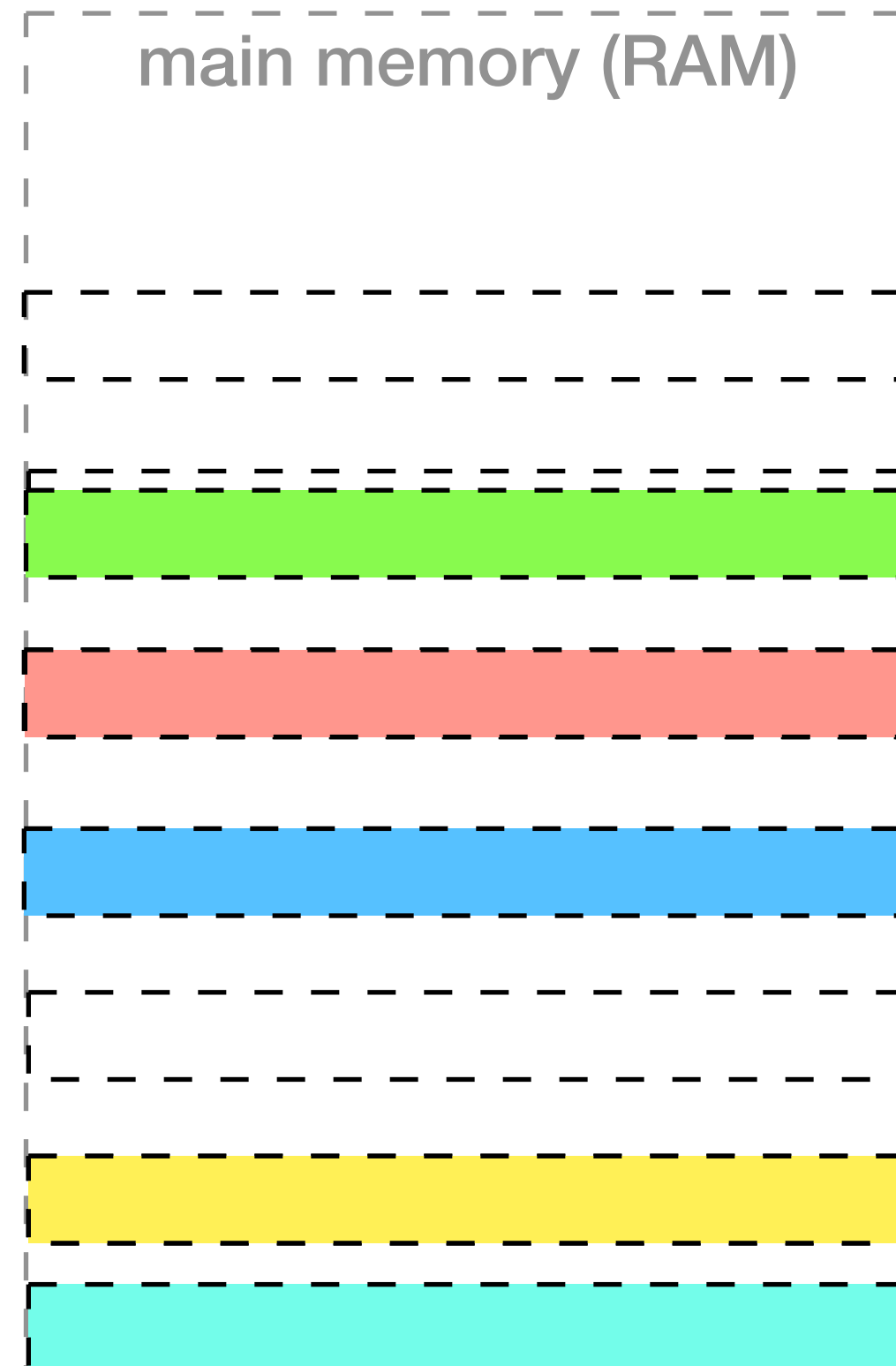
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.

confidential VM creation at runtime

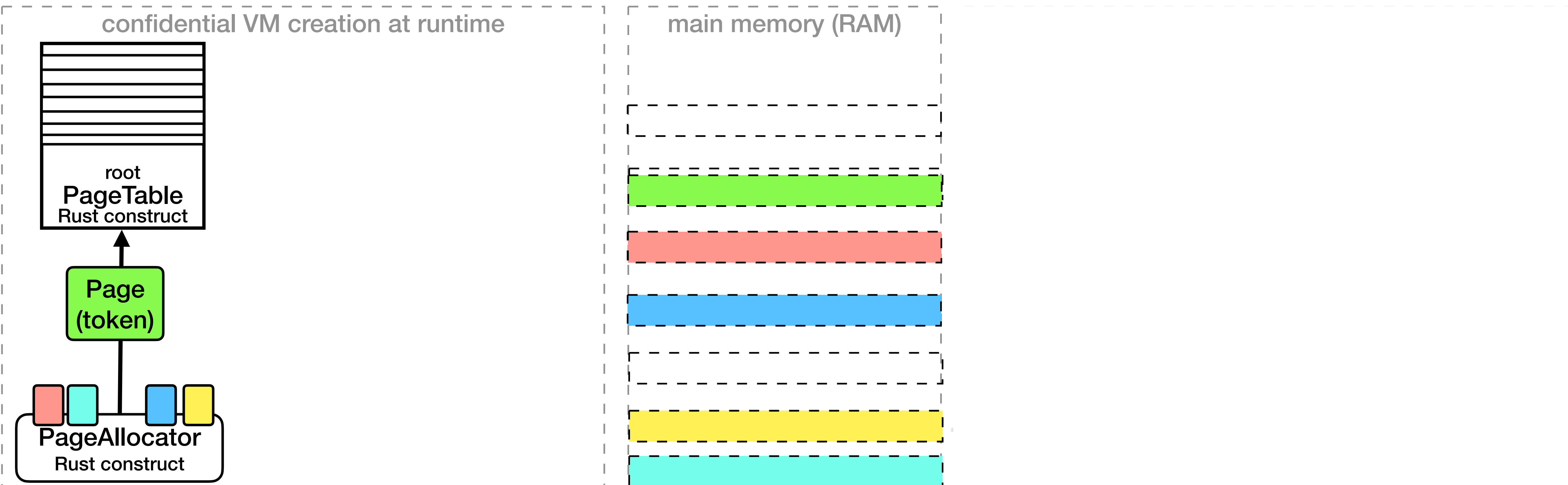


main memory (RAM)



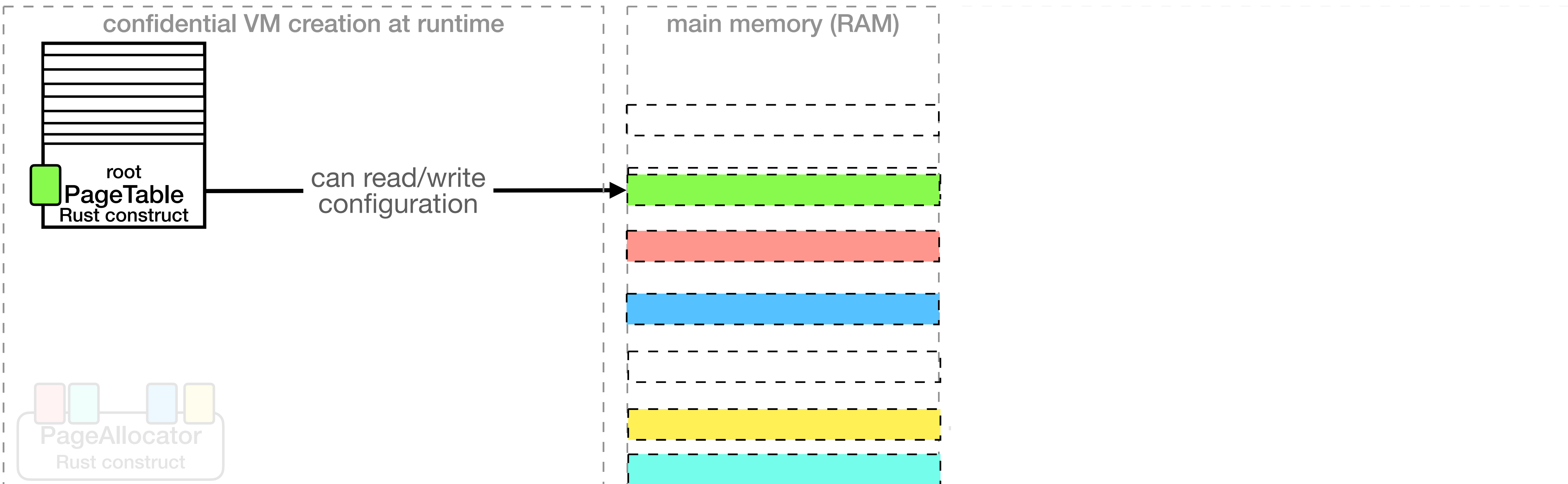
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



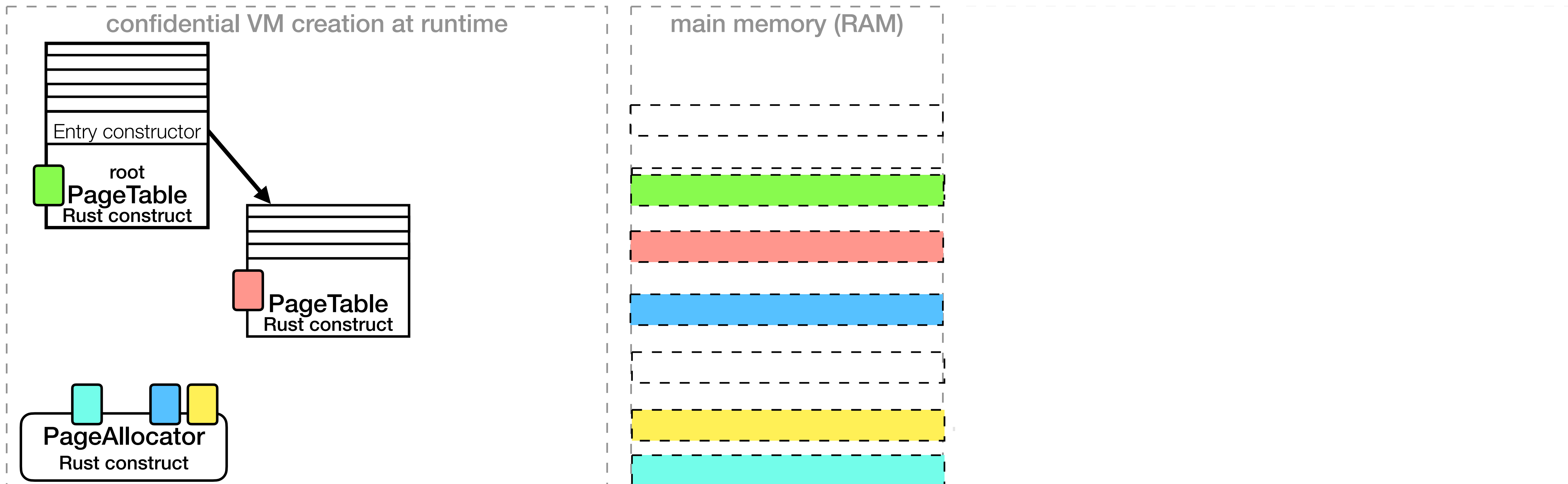
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



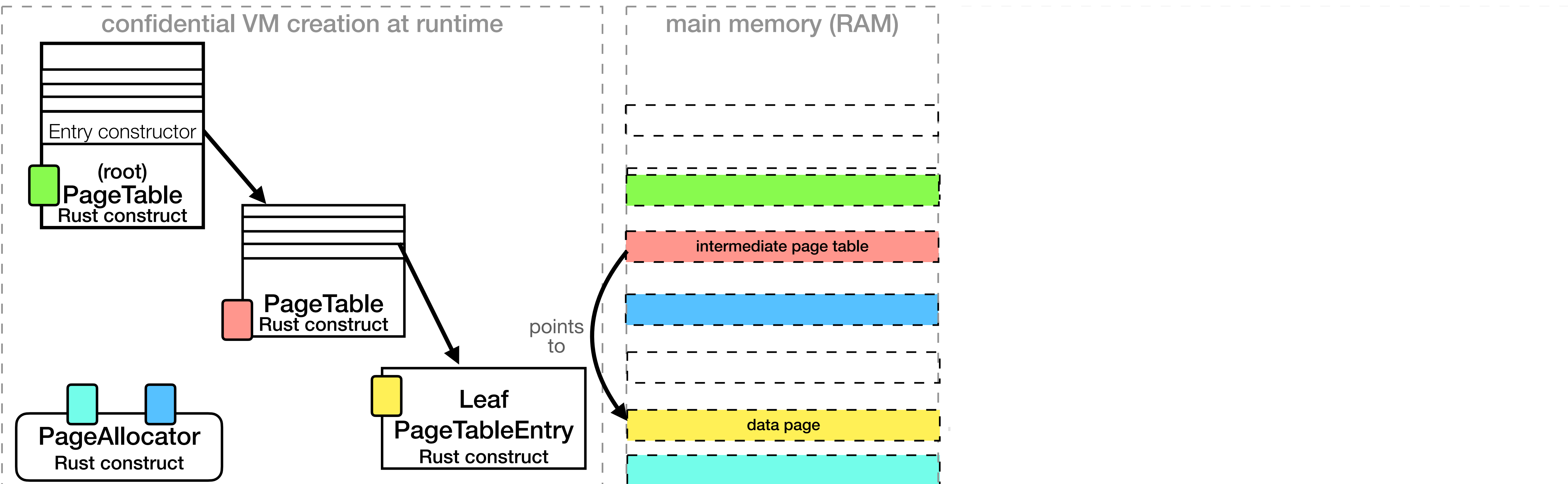
# Example: Memory Allocation

Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



# Example: Memory Allocation

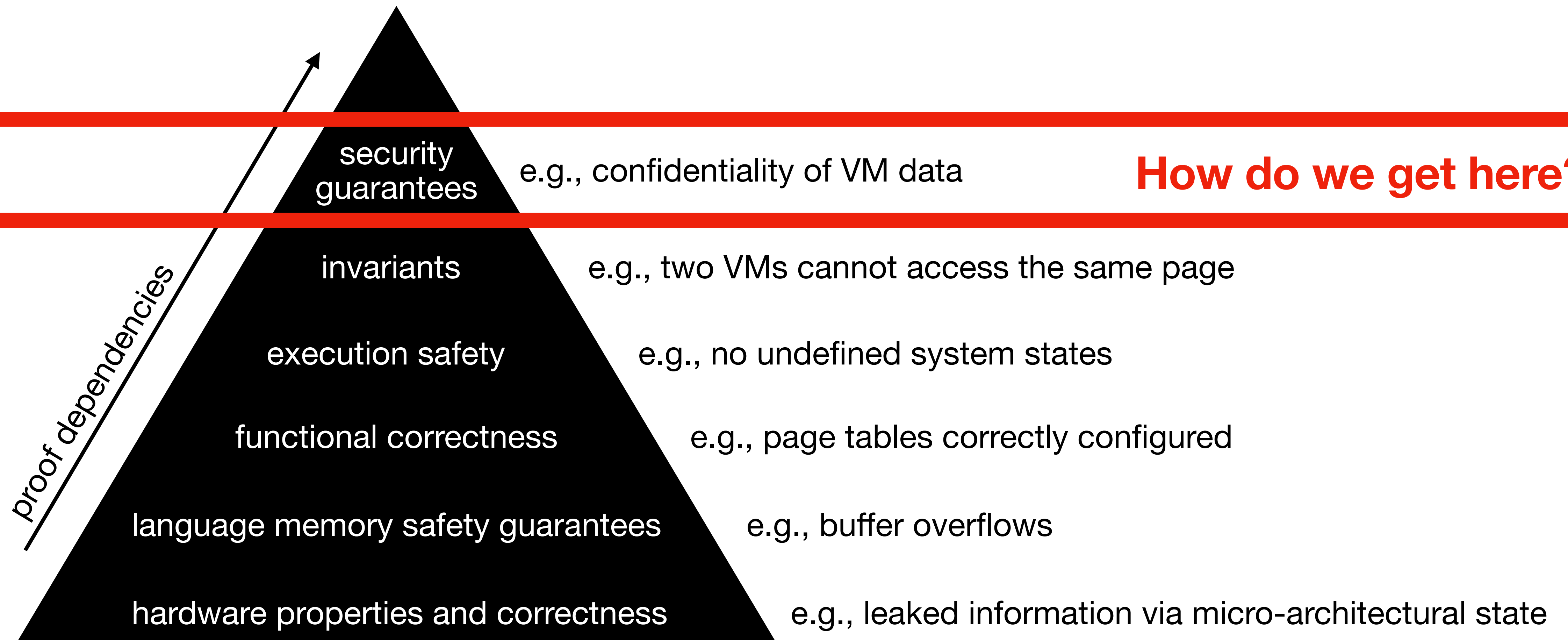
Goal: To prove that two different confidential VMs cannot access the same physical memory region in the confidential memory.



# Demo: Verifying page tokens with RefinedRust



# ACE: What has to be proven?

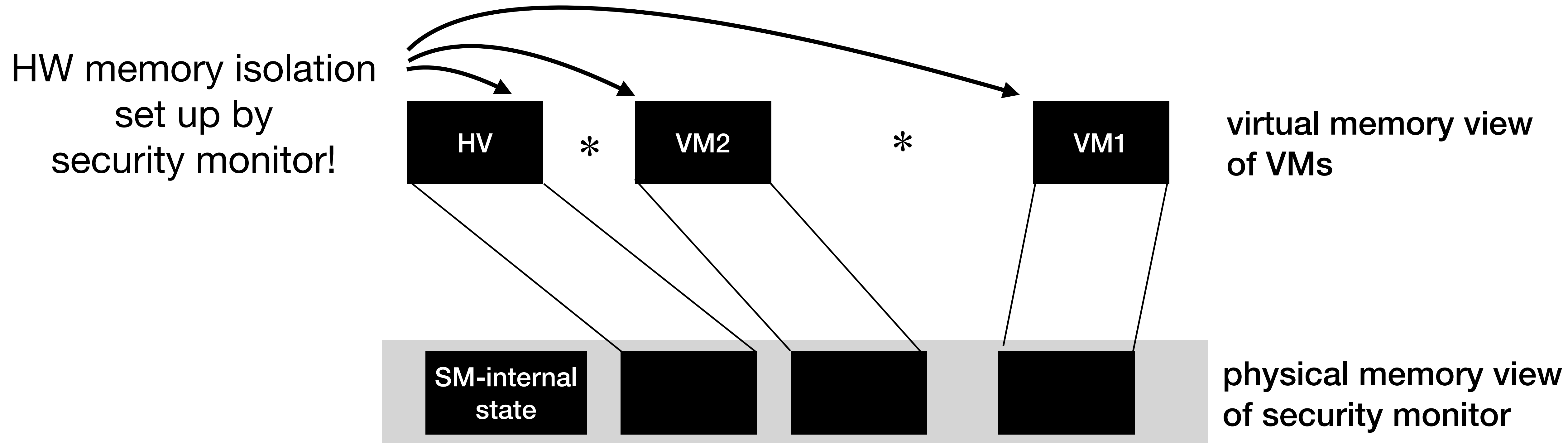


# Towards security properties

- Goal: prove *non-interference* (wrt. memory; not timing etc.)
  - No secrets from confidential VMs are leaked
- In a realistic system: *relaxed* non-interference
  - Confidential VMs can selectively declassify information
  - e.g. by requesting to share a page

**For now: focus on strict non-interference**

# Part I: Isolating memory regions



If we prove the page table setup correct, we know that any process (a VM or the hypervisor) cannot access another process memory.

# Part II: Proving non-interference for the security monitor

- The security monitor has access to the full physical memory
  - We cannot prove that it is physically isolated!
- Security monitor could open side channels:
  - Read memory of one VM and behave differently depending on the value

**How to prove non-interference for the security monitor?**

# How do we prove non-interference?

- Typically: proved by relating two executions

$$\forall s_1, s'_1, s_2, s'_2, i . \text{related } p_i s_1 s_2 \rightarrow \text{exec } p_i s_1 s'_1 \rightarrow \text{exec } p_i s_2 s'_2 \rightarrow \text{related } p_i s'_1 s'_2$$

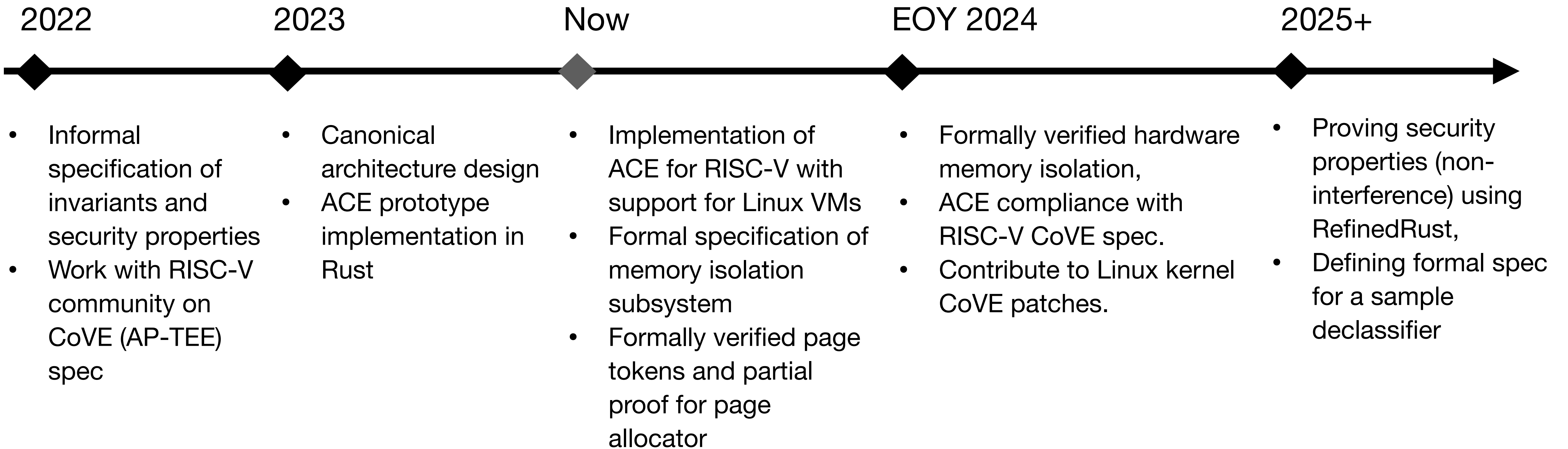
where `related` states that the two states are equivalent on  $p_i$ 's memory

- But our main verification of the security monitor (in `RefinedRust`) reasons about only one execution!

Standard trick: **information flow tracking**

**Future Work:** add information flow tracking to **RefinedRust**

# Ongoing & Future Work



# Summary

- We designed a canonical architecture for confidential computing.
- We implemented it and open sourced its implementation called ACE.
- We defined the formal specification for the memory isolation subsystem
- We formally verified part of the memory isolation subsystem.
- We conceptualized an approach for formalizing security properties.



<https://github.com/IBM/ACE-RISCV>

# Thank you