Towards formal verification of the confidential computing framework for RISC-V
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.
Problem: How to formally verify security properties of confidential computing systems?

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

Examples: IBM Secure Execution, OpenPower PEF, Intel TDX, AMD SEV, Intel SGX, Keystone, ...
Use Cases

Multi-tenant cloud  
e.g., IBM Cloud

Multi-domain / Embedded systems  
e.g., automotive, smartphones, sandboxing

Secure multi-party computation  
e.g., ML, LLM trainings/inference

Edge systems  
e.g., processing requests from smart cars

based on: Confidential Computing for RISC-V-based Platforms - Ravi Sahita, Rivos Inc., RISC-V Summit North America 2022
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.
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:

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.
Canonical Architecture

is a set of hardware and software components sufficient to build a minimalistic but functional **processor-independent confidential computing architecture**.
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.
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.
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
Assured Confidential Execution (ACE) for RISC-V

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.
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
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

Aims to provide memory safety for free(*):
• no null-pointer accesses
• no use-after-free
• no data races
• ...

Brings modern programming paradigms to systems programming

(*) more work if you use unsafe code
Deductive verification using RefinedRust

**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

1. Code & Spec
   - User-annotated specifications
   - Rust code

2. Frontend (compilation)
   - Rust compiler + RefinedRust frontend
   - Generates RefinedRust’s refinement type system

3. Formal code representation (Coq)
   - Translated code in Coq
   - Radium operational semantics
   - Iris separation logic framework

4. Proof automation (Coq)
   - Specfications in RefinedRust’s refinement type system
   - RefinedRust proof rules + automation
   - Manual tactics and hints

Proof (Coq)
- Proof succeeded
- Proof failed

Translated code in Coq
- Lemma add_42_correct : ....
  - Proof.
    - ...
  - Qed.

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

Designed to be run continuously in CI

# [rr::args("i")]
#[rr::returns("i + 42")]

Proof succeeded

Proof failed
Example: specifying pre- and postconditions

```rust
/// Add 42 to the argument x and return the result.

fn add_42(x: i32) -> i32 {
    x + 42
}
```
How practical is RefinedRust?

“Research prototype”
- Verified Vec library
- Published paper

Q3’23

We got here
- Verified parts of security monitor

Q2’23

Actually usable

future plans

- Supports traits (≈ Rust interfaces)
- Supports closures (≈ Rust anonymous functions)
- Supports multi-module crates
- Runs in continuous integration (CI)
- Supports axiomatising Rust standard library
- Supports mixed manual-automatic proofs
- Supports Iris ghost state
- Supports statics, char, …
- Improved performance & documentation
- …. much more!
Making RefinedRust practical: CI & multi-crates

CI integration

Cargo integration

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

Axiomatising library functions

```rust
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

```rust
#[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:

```rust
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:

```rust
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

<table>
<thead>
<tr>
<th>Tech</th>
<th>Unsafe code</th>
<th>Traits &amp; closures</th>
<th>Verified proofs</th>
<th>Automated</th>
<th>User-friendly</th>
<th>Extensible</th>
<th>Concurrency Aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>RefinedRust</td>
<td>Coq</td>
<td>✅</td>
<td>✅</td>
<td>~</td>
<td>~</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Prusti</td>
<td>SMT</td>
<td>✗</td>
<td>~</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Creusot</td>
<td>SMT</td>
<td>✗</td>
<td>✅</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Aeneas</td>
<td>Mixed</td>
<td>✗</td>
<td>✗</td>
<td>~</td>
<td>~</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Verus</td>
<td>SMT</td>
<td>~</td>
<td>~</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>~</td>
</tr>
<tr>
<td>Kani</td>
<td>Model Checking</td>
<td>✅</td>
<td>~</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
</tbody>
</table>

Future Work: combine advantages of different systems
ACE: What has to be proven?

security guarantees
ACE: What has to be proven?

- **Hardware properties and correctness**
  - e.g., leaked information via micro-architectural state

- **Language memory safety guarantees**
  - e.g., buffer overflows

- **Functional correctness**
  - e.g., page tables correctly configured

- **Execution safety**
  - e.g., no undefined system states

- **Invariants**
  - e.g., two VMs cannot access the same page

- **Security guarantees**
  - e.g., confidentiality of VM data

**Proof dependencies**
ACE: What has to be proven?

- **security guarantees**: e.g., confidentiality of VM data
- **invariants**: e.g., two VMs cannot access the same page
- **execution safety**: e.g., no undefined system states
- **functional correctness**: e.g., page tables correctly configured
- **language memory safety guarantees**: e.g., buffer overflows
- **hardware properties and correctness**: e.g., leaked information via micro-architectural state

**Our focus (so far)**
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.
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.
Demo: Verifying page tokens with RefinedRust
ACE: What has to be proven?

- Security guarantees
  - e.g., confidentiality of VM data

- Invariants
  - e.g., two VMs cannot access the same page

- Execution safety
  - e.g., no undefined system states

- Functional correctness
  - e.g., page tables correctly configured

- Language memory safety guarantees
  - e.g., buffer overflows

- Hardware properties and correctness
  - e.g., leaked information via micro-architectural state

How do we get here?
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

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

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

2022
- Informal specification of invariants and security properties
- Work with RISC-V community on CoVE (AP-TEE) spec

2023
- Canonical architecture design
- ACE prototype implementation in Rust

Now
- Implementation of ACE for RISC-V with support for Linux VMs
- Formal specification of memory isolation subsystem
- Formally verified page tokens and partial proof for page allocator

EOY 2024
- Formally verified hardware memory isolation
- ACE compliance with RISC-V CoVE spec.
- Contribute to Linux kernel CoVE patches.

2025+
- Proving security properties (non-interference) using RefinedRust,
- Defining formal spec for a sample declassifier
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.

Thank you