ACE: Confidential Computing for Embedded RISC-V Systems

Wojciech Ozga
IBM Research — Zürich

Guerney D. H. Hunt
IBM T.J. Watson Research Center

Michael V. Le
IBM T.J. Watson Research Center

Lennard Gäher
MPI-SWS, Germany

Avraham Shinnar
IBM T.J. Watson Research Center

Elaine R. Palmer
IBM T.J. Watson Research Center

Hani Jamjoom
IBM T.J. Watson Research Center

Silvio Dragone
IBM Research — Zürich

June 10, 2025

1 Abstract

Confidential computing plays an important role in isolating sensitive applications from the vast amount of untrusted code commonly found in the modern cloud. We argue that it can also be leveraged to build safer and more secure mission-critical embedded systems. In this paper, we introduce the Assured Confidential Execution (ACE), an open-source and royalty-free confidential computing technology targeted for embedded RISC-V systems. We present a set of principles and a methodology that we used to build ACE and that might be applied for developing other embedded systems that require formal verification. An evaluation of our prototype on the first available RISC-V hardware supporting virtualization indicates that ACE is a viable candidate for our target systems.

2 Introduction

Building mission-critical systems and confirming their safety and security properties has become increasingly challenging, sometimes impossible, because of growing software size, shrinking budgets, the pressures of time-to-market, and talent shortages. The growing frequency of cyberattacks [59, 62, 50, 15, 61, 7] confirm the need for a method to increase safety and security of low-level systems without sacrificing productivity or increasing development costs.

Evolving demands of the market drive the reuse of software and hardware components to accelerate development. Moreover, different companies deliver hardware and software components and clients license applications from various vendors. Some vendors may also desire to protect their intellectual property and sensitive data because the execution environment is potentially shared with competitors or regulated. In addition, reasoning about the correctness of the entire system composed of hundreds to thousands of dependencies written and maintained by a variety of vendors and open-source contributors is cumbersome if not impossible.

During verification, the interactions between all components have to be checked, causing an exponential increase in cost and time. Consequently, the verification phase may be shortened or eliminated to meet deadlines or cost targets, leaving unidentified memory safety bugs or security back doors in the system [45, 42]. We argue that confidential computing [54, 10, 56, 1, 28] can be leveraged to build safer, more secure, low-level systems by compartmentalizing software into isolated security domains. The correctness and safety of these security domains can be then individually assessed due to the limited and well-defined communication interfaces.

To address the need for confidential computing for embedded systems in regulated environments, we introduce ACE—a confidential computing technology targeted for the RISC-V open architecture that enables software compartmentalization to minimize the number and types of interactions between security domains to streamline their verification. In contrast to existing systems, ACE targets systems that operate on limited power and silicon budget and, due to costs, can run on regular off-the-shelf processors. Commercial confidential computing implementations for x86 [10, 56], IBM Z [5], or IBM POWER [28] target primarily high-end processors, typically used in data centers and cloud computing, while ARM, which applies to both desktop and embedded processors, incurs additional licensing costs and requires additional hardware for its confidential computing technology [1]. ACE, on the other hand, aims at broad adoption and is therefore open-source and royalty-free, making it easily reusable and deployable without additional licensing costs. The ACE design has already influenced the RISC-V confidential VM extension (CoVE) specification [6], broadening its scope to embedded systems.

The goal of ACE is to run safety and security-critical systems that must conform with regulations [20, 44, 14, 16]. Specifically, these regulations aim at ensuring system correct-

ness and mandate extensive testing while recommending or sometimes requiring use of formal methods for specification, design, verification or testing phases. For example, EAL7, the most stringent Common Criteria [14] evaluation level, requires rigorous mathematical proofs of system properties that give high assurance on a system's security. Achieving such levels of certification is challenging because proof sizes are expected to grow with the square of the specification size [40] that itself grows with the size of the codebase. For example, simply ensuring the *memory safety* is challenging because a single instruction in memory unsafe language can violate this property.

As part of our work on ACE, we developed a set of principles and a methodology that we believe are universal and provide a basis for the design and development of firmware that requires verification. These principles enable system designers and developers to choose a tradeoff between resources invested in verification and the obtained assurance level. For example, they might rely on strongly typed, memory-safe programming languages and their compilers for memory safety while harnessing proper encapsulation techniques with deductive verification to prove correctness of a small set of traditionally error-prone unsafe operations.

We applied these principles during the design and development of ACE, achieving a relatively high level of trust by design—even before the full formal verification is complete. We defined informal invariants early during the design phase and aimed for simpler feature-reduced system design to streamline subsequent verification efforts. We chose the Rust [41] programming language to benefit from its memory safety guarantees and rich type system that simplifies reasoning about memory ownership [33]. We followed the principle of encapsulating unsafe code, such as low-level pointer manipulation, within simple, verifiable interfaces, which serve as a foundation for proving the correctness of higher-level components that rely on them. We used RefinedRust [21] to formalize invariants directly in the Rust code and derive formal representation that allowed us to prove memory safety of core parts of the system in the Rocq prover [60], providing initial evidence for the usefulness of our approach.

To demonstrate maturity and functional readiness of the ACE implementation, we ran and evaluated multi-processor Linux-based confidential virtual machines (VMs) on the first RISC-V hardware available on the market that supports virtualization. The results show that ACE incurs low performance overhead for process-intensive workloads and up to 50% overhead for multi-vcpu network intensive workloads.

Our contributions:

- Defined principles and methodology to guide design of high-assurance embedded systems (§4).
- Designed (§4.2) and implemented (§5) ACE, an open-source confidential computing for embedded systems for RISC-V processors with virtualization support.
- Formally proved memory safety of a core part of the ACE implementation (§5.5).

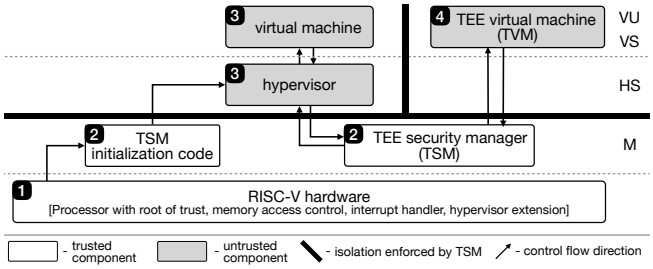

Figure 1: High-level overview of ACE, a VM-based confidential computing architecture for embedded RISC-V processors. The TSM (❷) leverages hardware features (❶) to multiplex execution of different security domains (❸) and (❹) on top of the same hardware while preserving security guarantees.

- Evaluated ACE on the first RISC-V hardware that implements the hypervisor extension (§6).
- Extended the RISC-V CoVE specification and patches for Linux kernel (patches not yet upstreamed).

3 Overview of ACE

The main goal of ACE is to bring confidential computing capabilities to embedded systems. As in other virtual machine (VM)-based approaches to confidential computing, the hypervisor manages VMs' lifecycle but is removed from the trusted computing base (TCB). To achieve the required level of isolation, ACE utilizes software and hardware components to establish a trusted execution environment (TEE) [22].

Figure 1 shows the high-level overview of the architecture, which consists of four components: (❶) hardware, (❷) TEE security manager (TSM), (❸) hypervisor with virtual machines (VMs), and (❹) TEE virtual machines (TVMs)[1]. The TSM's goal is to isolate security domains from each other. The hypervisor and normal VMs constitute one security domain and each TVM constitutes a distinct security domain. To enforce proper access control, the TSM relies on controlling hardware components. Intuitively, one might think of the TSM as a firewall that controls all interactions between a TVM and the outside world. The TSM is a piece of code that switches execution of a security domain's (a TVM or a hypervisor) context and ensures that only allowed information is exchanged. It achieves this by reconfiguring the hardware to enable proper access controls and clearing the execution state so that there are no execution traces left when another security domain resumes.

To maintain full control over the hardware's state and configuration, the TSM asserts control of the system during the early boot process and retains its privileged role during the system's lifetime. This role allows it to enforce access controls by

[1]We follow the nomenclature used in RISC-V CoVE [6] and TDISP [48] specifications. In literature one can find also terms confidential VMs (CVMs), secure VMs (SVMs), or enclaves, as well as, security monitor, security manager, or isolation monitor.

switching processor privilege modes, applying memory access controls, and taking control over interrupts.

3.1 Open Source and Compliance

ACE complies with the RISC-V CoVE specification [6] and builds on top of the canonical architecture [47], which specifies the minimal set of features needed for confidential computing system. The simplicity of this design is intended to streamline the formal verification process, while compliance with standards should facilitate faster adoption. The open source nature of the project should encourage community involvement which may lead to extensions that will enhance performance for specific use cases.

3.2 Threat Model

We assume a software-level adversary whose goal is to compromise a victim's TVM, specifically (1) by tampering with the integrity of the TVM's execution by modifying its control or data flow execution, (2) by reading the protected TVM's data, or (3) by impersonating the victim's TVM to its owner.

We assume an adversary who controls the hypervisor with root privileges and might run her own TVMs concurrently and in parallel with the victim's TVM. She can also fully control the lifecycle of the victim's TVM and input data via hypervisor-controlled registers, shared memory buffers, virtual I/O devices, or the initial TVM's image state and the flattened device tree (FDT).

We aim to provide security down to the architectural level, *i.e.*, instruction set architecture (ISA)-level. We assume that the hardware ACE runs on is correctly implemented [36, 9] and all micro-architectural traces of executions are correctly cleared as part of the system's context switch. We assume software in the TVM is correct and bug-free.

We exclude from the threat model physical attacks on memory, buses, and processors, because of the existence of well known counter-measures, like fault-detection mechanisms like lockstep, error correction code-protected memory, or encrypted buses. These technologies can be implemented separately for target systems exposed to physical adversaries without requiring changes to the TSM's implementation.

4 Design

The design principles outlined below arise from the need to satisfy three competing constraints: resource limitations of embedded systems, complex security needs of confidential computing, and tractability of formal verification.

Embedded systems: Embedded systems typically have limited power and silicon budget and are designed to execute specific workloads with a fixed and well defined runtime requirement. ACE targets mid to high-end systems that in contrast

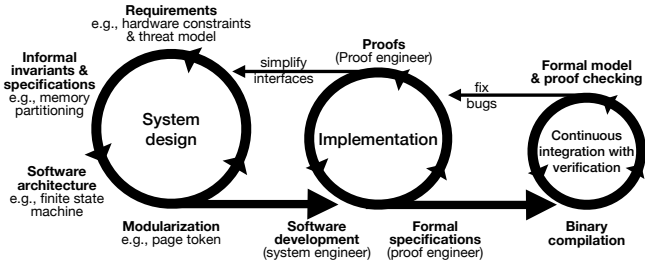

Figure 2: Our methodology to develop formally verified low-level computing systems.

to low-end embedded systems often contain multiple security domains. These systems can be but are often not Internet-connected.

Confidential computing: Confidential computing builds from hardware and firmware components, whose implementation and composition require rigorous security analysis. While increased complexity of these components might enhance the performance and capabilities of target systems, it also poses greater challenges for ensuring desired security.

Formal verification: Formal verification scales best when the verification can be modularized with clear abstraction boundaries, making reasoning about system invariants local. Reasoning often becomes easier with simpler representation invariants, *e.g.*, low-level memory hacks can complicate verification significantly.

Achieving ACE's goal while satisfying the above constraints requires tradeoffs. To navigate these tradeoff decisions effectively, we define the following guiding principles.

- P1: Design a small modular core with minimal hardware requirements that can be extended for more complex use cases and hardware.
- P2: Minimize cross-domain interactions (to make formal verification achievable)
- P3: Encapsulate complex operations with clearly defined boundaries (modularity, extensibility to satisfy formal verification requirements)
- P4: Treat verification as a first-class citizen—designing and implementing with verification in the loop, using verification considerations to guide the system design (to reduce verification time, improve coverage).

4.1 Design Methodology

Figure 2 shows our methodology for designing and developing ACE, consisting of three phases interacting with each other: system design, implementation, and continuous integration with proofs. For the remainder of this section we will focus on the system design decisions, the other two phases in our process will be discussed in Section 5.

We started our design phase by evaluating the requirements such as hardware constraints and the threat model (Subsection 3.2). Those requirements, together with our design prin-

ciples, then motivated our key design decisions and high-level invariants (Subsection 4.2).

4.2 Key Design Choices & Invariants

By adhering to our design principles and applying our methodology, we made several key design decisions that make the ACE design unique. These are (1) prefer commonly used, simple hardware primitives with easily abstracted functionality, (2) favor static versus dynamic configurations for simplified reasoning about system state, (3) avoid unnecessarily complex interfaces to reduce intermediate states, (4) prioritize simplicity over performance and memory usage to facilitate formal verification.

Simple hardware and abstractions Since ACE primarily targets mid- to high-end embedded systems, its design must support commodity, ideally off-the-shelf hardware. Following principle **P1**, we do not require the presence of sophisticated TEE-optimized hardware components for memory isolation, such as for example RISC-V supervisor domains access protection [52] or an interrupt controller that supports interrupts directly injected into TVMs [4]. We aim at design that builds on a simpler, general-purpose hardware components [38, 18, 47] that can be further extended with additional hardware and software components to improve performance or support use case-specific features.

We decided to lay the ACE's design on a virtualization layer which provides a good abstraction over execution environment (**P3**) and is a natural boundary for isolation with a smaller attack surface (**P2**) compared with process-based TEEs [13, 38, 32]. Moreover, the RISC-V hypervisor extension can be completely emulated in firmware [64], allowing ACE to potentially run on much simpler hardware at the cost of reduced performance (**P1**). However, we expect that most of the targeted systems will be powerful enough to support a RISC-V processor with virtualization capabilities, like the hypervisor extension and a two-level address translation memory management unit (MMU).

Statically partitioned confidential memory. The TSM uses memory access control hardware to prevent unauthorized accesses to protected memory regions. Embedded systems usually run a fixed well-understood set of applications. This makes them uniquely suited to static memory partitioning because the memory requirements are predictable.

There are different hardware-based mechanisms used for memory access control. More complex mechanisms enable fine (page-level) access control at the cost of more complex configuration setup and runtime walks. Alternatively, simpler mechanisms generally have a limited number of coarse-grained access control rules for physical memory but are simple to set up, implement, and consequently verify.

Following **P1** and **P2**, ACE statically partitions memory into non-confidential and confidential memory. This is a simple one-time operation that utilizes simple hardware which performs address range checking. This approach simplifies the mathematical modeling and verification of the system because of the simplified ownership reasoning. This simplicity comes at the cost of potential memory over- or under-allocation. We argue that this is acceptable for our target systems, because their resource requirements and applications are known upfront.

Single-step TVM creation. Existing confidential computing frameworks are designed to run arbitrary code. As a result, there is no way to predict the memory requirements on the system in advance. In addition, to fulfill unknown confidential memory requirements, existing frameworks allow the dynamic conversion of free memory from one state to another. This contributes to the large application binary interface (ABI) between the hypervisor and the TSM that allows the hypervisor to construct a TVM in a series of calls to the TSM. These calls cause the TSM's size to grow, increasing the TCB. They also increase the attack surface during the transition from normal VM to TVM, which makes formal verification more complex. Multiple calls also result in larger TVM creation latency due to superfluous context switches between the hypervisor and the TSM.

Following **P2** and **P3**, ACE requires only a single call from the hypervisor to create a TVM. In this approach, the hypervisor sets up an instance of a normal VM from a TVM image and then requests the TSM to *promote* it to a TVM. This decision reduces the size of the TSM's implementation. For example, there is no need to implement a complex ABI to manage page table mappings. ACE's approach is also more secure, because there is no intermediate (incomplete) state of a TVM that an adversary might try to rollback to.

There are some disadvantages to this approach. Since the TSM's execution is non-uninterruptible, the hardware thread (hart in RISC-V terminology) is blocked for the time that the TSM takes to copy a VM to the confidential memory. We argue, however, that this is acceptable for the targeted systems that tend to fully initialize early in the boot process because of auditability and real-time requirements. In other words, TVMs are created during the system initialization phase to not influence the system's execution during the operational phase. Further, the targeted embedded systems are not running arbitrary code at arbitrary points in time. They often start a limited number of functions that are well understood and run for a long period of time.

Formal Verification and Rust. Our target systems might operate in regulated industries that require formal proofs. Figure 3 shows what properties the TSM aims to guarantee. These are presented in the form of a pyramid: to reason about properties in one layer, one must first prove the properties in layers below. ACE assumes the hardware is correct [36](Subsection 3.2).

The most basic property the TSM implementation needs

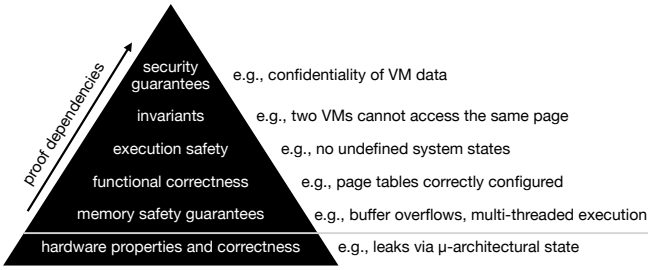

Figure 3: Pyramid of proof dependencies.

to satisfy is memory safety. Memory safety errors continue to make up a large part of the security-related bugs [11, 23]. Thus, following **P4**, we decided to use Rust [41], which aims to statically rule out memory safety errors, as the implementation language for the TSM. Rust has a rich type system that helps to statically eliminate memory-safety issues without getting formal verification engineers involved, thus shifting a significant part of the formal verification burden to the compiler. By applying sound software engineering practices—such as encapsulating unsafe Rust and creating well-defined abstractions with the help of modularization (**P3**)—developers can gain greater confidence in the correctness of their programs, even in multi-threaded environments.

While ensuring memory safety eliminates many common bugs, it is just a step towards our ultimate goal: achieving security guarantees, as illustrated at the top of the pyramid in Figure 3. Achieving these security properties requires the high expressiveness offered by deductive verification frameworks like the Rocq prover [60] that enable us to formally define security properties and mechanically prove them. Rust code can also be formally verified in the Rocq proof assistant using tools such as RefinedRust [21], with proofs being continuously checked for every code change in a continuous integration (CI) system.

Local attestation. It is essential for TEEs to provide a way for clients to detect unauthorized modifications to the firmware, TSM, or TVM. However, embedded systems might operate in environments with limited or no network access, like controllers used in operational technology or automobile industry.

To support disconnected systems following **P1**, ACE supports local attestation [28] which uses information delivered together with a TVM's image to verify the integrity and authenticity of the TVM during its creation. In local attestation, a TVM owner uses his key to encapsulate (*i.e.*, cryptographically protect) a TVM attestation payload (TAP), which is an object (file) that stores TVM integrity measurements and TVM-specific secrets. Only the expected TSM running on the correct hardware can read the contents of a TAP because it has access to the decapsulation key. The presence of local attestation does not exclude remote attestation for systems with network connectivity.

5 Implementation

We implemented the TSM in Rust [41] and open sourced it under: `https://github.com/IBM/ACE-RISCV`. The TSM is statically linked with OpenSBI [51] firmware to which it delegates some requests, like handling unaligned memory accesses or inter processor interrupts (IPIs). Both OpenSBI and the TSM run in the most-privileged RISC-V M mode, but OpenSBI is used as a library to simplify its future replacement or de-privileging as reported by [8]. We use Linux KVM as the hypervisor and QEMU as a virtual machine monitor. We patched Linux kernel with the CoVE patches, which were developed by Rivos and then extended by us with functionality required for running the deployment model 3 of the RISC-V CoVE spec.

The TSM's core implementation consists of two parts: initialization code executing during secure boot (§5.1) and runtime code executing as a finite state machine (§5.2) reacting to interrupts. We then discuss how formal verification is imbued in our design and implementation of the TSM in (§5.5) and outline how a key TSM component can be verified with our approach.

5.1 System Initialization

At power on, the hardware root of trust initializes the secure boot [2] process and transfers control to the TSM's initialization code. This code configures the system to ensure that the TSM maintains full control until the next power-cycle. Specifically, it reconfigures hardware to protect the TSM's code, data, and hardware configuration from tampering by less-privileged software that will execute during runtime. Additionally, it ensures that the execution of less-privileged software traps to the TSM at well-defined entry points. In more detail, the TSM (1) configures the memory access control mechanisms so that its code and data cannot be accessed by software executing in less privileged processor modes and by non-processor memory accesses, *e.g.*, direct memory access (DMA), (2) sets up the hart to trap into the TSM's entry preamble on selected interrupts and exceptions, and (3) executes the hypervisor's code in a less privileged mode. The configuration is identical across all physical harts.

Memory partitioning. During the system initialization, the TSM divides main memory into two contiguous regions: non-confidential memory and confidential memory. Later, at runtime, the hypervisor will have complete ownership of the non-confidential memory but will not be able to access the confidential memory. That allows the TSM to store in confidential memory the security-sensitive data required to maintain confidential computing functionality, such as the TSM's stack areas (for every physical hart), the TSM's heap area, save-state areas (for every physical and virtual hart), and code and data of TVMs.

To enforce memory access protection to confidential memory, the TSM configures hardware-based memory access control mechanisms. Specifically, it uses RISC-V physical memory protection (PMP) [29] and IOPMP [34][2]. The RISC-V architecture guarantees that only software running in M-mode is authorized to reconfigure it.

Page allocator and page tokens. Confidential memory is further partitioned into pages that the TSM assigns to TVMs to store their code and data, as well as pages to manage the TSM's internal data structures. To ensure correct allocation of pages, *i.e.*, that pages belong to confidential memory and have a single owner, the TSM instantiates a component called the *page allocator*. The page allocator is a software submodule of the TSM. During system initialization, for every confidential memory region corresponding to a physical page, it creates a logical *page token*, a Rust object that lives on the TSM's heap. Initially, the page allocator owns all page tokens. Whenever the TSM allocates a new page of memory, for example to create a TVM's data page, the TSM retrieves a free page token from the page allocator and stores it within the Rust data type corresponding to the final entity, *e.g.*, a page table associated with the TVM. In the above example, the ownership transfer would correspond to the move of the page token's *self* reference from page allocator to the object of the TVM's page table type. Rust's type system ensures that a page token is not spent twice. In the above example, it would ensure that every physical memory region is only assigned to one TVM.

5.2 Runtime

The firmware executing the secure boot eventually transfers control to the hypervisor, marking the transition of the device into its operational state called *runtime*. During runtime, the hypervisor, running in the RISC-V HS-mode, manages which software runs on the processor and for how long. The hypervisor has limited privileges and must request the TSM to perform security-sensitive operations, for example, accessing certain memory mapped input/output (MMIO) or running a TVM.

To limit the attack surface, the TSM exposes only a narrow ABI to the hypervisor. In addition to the standardized RISC-V ABI calls, such as read-only calls to discover the system configuration and a call to shutdown/reboot the system, the TSM adds three calls to manage a TVM: promote, run, and destroy. In comparison, a full RISC-V CoVE defines additional sixteen complex calls for page table management and two extra calls for constructing a TVM in multiple steps. With help of OpenSBI, the TSM supports optional symmetrical multiprocessing calls (like start/stop/suspend hart, execute remote fence), access to hardware devices via MMIO, and emulated access to misaligned load/stores.

[2]The current implementation of ACE runs on a RISC-V evaluation board that does not support IOPMP because IOPMP has not been ratified at the time when the board was designed.

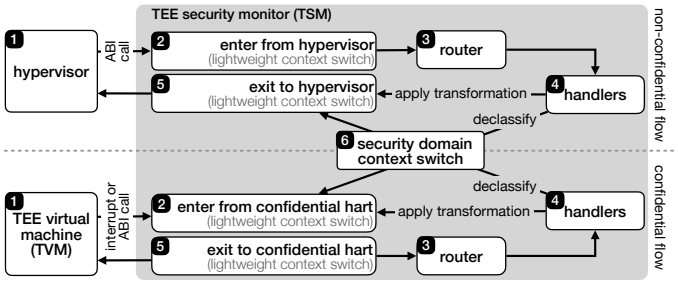

Figure 4: Finite state machine (FSM) shows the execution flow of the TSM on a single physical core on which the hypervisor and a TVM execute concurrently.

Finite state machine. To enhance the correctness of the TSM's implementation and simplify the verification process, the TSM operates as a finite state machine (FSM), as shown in Figure 4. The hart that the TSM executes upon taking control is in one of two states: executing a non-confidential flow (upper part) or executing a confidential flow (bottom part). This split simplifies the analysis of information flow between the hypervisor and the TVM, while minimizing the potential for programming errors. Specifically, we leveraged Rust modularization capabilities to statically ensure that the TSM's functionalities are accessible exclusively to the hypervisor or a TVM.

In the non-confidential flow (upper part of the FSM), the TSM handles requests on behalf of the hypervisor and the hardware remains configured to enforce hypervisor-specific access control permissions. For example, the memory access control mechanism enforces that the hypervisor can access only non-confidential memory. Therefore, no additional security measures must be taken to return execution to the hypervisor. Analogously, in the confidential flow (bottom part), the TSM processes requests of a TVM and the hardware is configured to enforce access permissions of a TVM. When the TVM resumes execution, the memory management unit (MMU) is using the page tables associated with the TVM which allows the TVM to access only memory pages it owns. If there is a need to switch from the current TVM, the TSM switches execution to the hypervisor as the hypervisor retains scheduling responsibility.

Lightweight context switch. The TSM executes only in response to interrupts and ABI calls from the hypervisor and TVMs (❶). The lightweight context switch (❷) starts by storing the minimal set of the architectural state that the TSM could overwrite during its execution (*i.e.*, general-purpose registers (GPRs) and control status registers (CSRs)) in the save-state area in main memory. Then, the router component (❸) parses the request and invokes the appropriate handler. The handler (❹) decides if the request will return to the same security domain or a different security domain. Handlers implement the logic that decides how to transform the architectural state to fulfill the request. This transformation is then

applied to the hardware architectural state and/or state in the save-state area. If the security domain will not be changed, the lightweight context switch (❺) is called to return to the security domain.

Security domain context switch. One security domain can request assistance from another security domain to perform an operations. For example, a TVM that cannot access a physical network card directly uses the VirtIO protocol, so that the hypervisor emulates such access. The corresponding handler (❹) routes then the call to the *security domain context switch* (❻) and the TSM crosses the boundary between the non-confidential and confidential security domains.

The security domain context switch is a piece of code that reconfigures the hardware to prepare for execution of a different security domain. It saves all architectural state of one security domain in the save-state area in the main memory and loads the architectural state of the other security domain from the main memory into hardware. All state for the security domain context switch is stored to or read from confidential memory. The security domain context switch also reconfigures the memory access control mechanism to protect the current security domain from the one that will execute next, and clears micro-architectural caches, *e.g.*, translation lookaside buffer (TLB) caches. Our design of the TSM implementation in Rust ensures that every execution path between non-confidential and confidential flows goes through the security domain context switch.

Handlers and Reclassification. Handlers implement the TSM's functionality exposed to the hypervisor and TVMs via the ABI. To reduce the risk of implementation errors creating security bugs, each handler operates only on a subset of the architectural state of a security domain. For example, a TVM's attestation handler will never have write access to the hypervisor's state, so that it can never leak attestation-related data to the hypervisor. To do so, we leveraged Rust's constructs to ensure the limited visibility scope and privacy of data structures in the implementations of handlers. Specifically, each handler consists of three phases: a *constructor*, a *transformation method*, and a *destructor*. The *constructor* has read-only access to the hart state, which permits it to read information required to process the request. The *transformation method* has access only to the TSM's core functionalities, such as the page allocator, the interrupt controller or the TVM's metadata storage. Finally, the *destructor* has write access to the target security domain's architectural state to which it can apply or reclassify processed information. The Rust-enforced constraints placed on handlers and accessibility of each domain's architectural state simplify how a TVM's owners and verification tools can verify that the TSM correctly implements the specification.

Importantly, the TSM cannot guarantee that TVMs use handlers correctly. Therefore, it is the security domain's responsibility to use them securely. For example, for VirtIO, the TSM enables a TVM to have a shared page with the hypervisor, but does not enforce any form of data secrecy. Thus, to maintain security, a TVM itself must encrypt data before sending them to disk over VirtIO and decrypt when reading back.

5.3 Selected Non-confidential Flow Handlers

Promote to TVM. As explained in Subsection 4.2, the TSM implements a single-step TVM creation procedure. In this novel approach, the TSM atomically converts an existing instance of a VM into a TVM as part of the process called *promotion*. In contrast to other architectures [10, 28, 5] which convert the VM memory page-by-page, there are no intermediate states corresponding to a TVM that could be exploited by an adversary.

We leave the decision when to promote a VM into a TVM to a TVM owner. A VM must execute a hypercall that will then be forwarded by the hypervisor to the TSM. The TSM receives as input the initial state of the VM (boot hart), a pointer to the root page table, a pointer to the flattened device tree, and a pointer to the attestation payload. It creates save-state areas for the new confidential harts and traverses the VM's page table hierarchy copying non-zeroed pages from non-confidential memory into confidential memory. After measuring the initial TVM state, it performs local attestation to decide whether or not the TVM is allowed to execute. The Above process can also be initialized by the hypervisor. In all cases, a TVM owner must establish trust with the TVM using an attestation mechanism.

Local attestation. Local attestation provides attestation for TVMs that execute with limited or no network access. When the hypervisor requests the TSM to create a TVM, it passes to the TSM both the TVM's image and a dedicated file called TEE attestation payload (TAP). The TSM verifies the TVM's integrity by comparing it with reference measurements from the TAP and stores TVM's secrets retrieved from the TAP in confidential memory. A verified TVM can use a dedicated ABI call to retrieve secrets from the TSM.

The TVM's integrity measurement includes the cryptographic hashes of the TVM's code and data, flattened device tree, and the initial confidential hart's state. This is stored in dedicated registers analogous to platform configuration registers (PCRs) [3]. The TVM's data integrity is a single hash, the value of which uniquely represents the initial content of the TVM's code and data pages.

The TVM's owner creates a TAP by encrypting a payload, which contains integrity measurements and secrets, and concatenates it with a *lockbox*. The lockbox stores the symmetric key encrypted with the public portion of the TSM's attestation key, which is an asymmetric cryptographic key. Only the TSM running in the target hardware can then decrypt the symmetric key and the payload. A TVM owner can permit running a TVM image on different hardware by attaching multiple lock-

boxes to the TAP.

The TAP format supports post-quantum cryptography (PQC), so that an adversary possessing a powerful enough quantum computer is not able to decrypt confidential TAP information. Specifically, the TSM uses quantum-safe algorithms, such as ML-KEM [46] to encapsulate the TAP symmetric key in the lockbox, SHA-384 to calculate the TVM's integrity measurements, and AES-GCM-256 to encrypt the payload. The TAP format also supports cryptographic agility, because each lockbox defines the type of algorithm that encrypts the TAP symmetric key to allow a TVM owner to choose a PQC algorithm of their choice.

5.4 Selected Confidential Flow Handlers

Symmetrical multiprocessing. ACE supports symmetrical multiprocessing which enables a TVM to have multiple confidential harts.

When creating a TVM, the TSM enables only the TVM's boot hart and sets other confidential harts to the powered off state. The boot hart can then request the TSM to start another confidential hart at the indicated guest physical address and with specific initial arguments. The TSM follows the finite state machine defined in the hart state management (HSM) extension to RISC-V supervisor binary interface (SBI) [43] to track and ensure proper lifecycle state changes. The TSM also permits confidential harts to send inter processor interrupts (IPIs), as well as dedicated commands to clear remote caches, to other confidential harts as defined in the respective SBI extensions [43].

Timer programming and interrupt handling. The ACE implementation relies on the hypervisor to manage a TVM's external interrupts and on hardware to provide the per-hart timers.[3] ACE also supports systems with a basic hardware interrupt controller [24] without requiring support for the complex RISC-V AIA specification [4].

During the TVM's execution, all of the guest's interrupts (RISC-V VS-mode) trap directly into the TVM, because of the TSM interrupt delegation setup. Other interrupts, like an external interrupt or a software interrupt, trap into the TSM. For external interrupts, the TSM returns execution to the hypervisor so that it can decide who is the recipient of the interrupt. If the interrupt targets a TVM, the hypervisor requests the TSM to resume the TVM's execution and inject the interrupt whose identifier is in a dedicated CSR. After the security domain context switch, the TSM decides whether to inject the interrupt to a TVM by checking which interrupts the TVM agreed to receive. If the TSM is not executing, all interrupts go either to the hypervisor or the VM as indicated by the configuration.

[3]The evaluation board we used to measure the performance of ACE did not support Sstc. Consequently, we added support for hardware without a VS-level hardware timer to enable evaluating ACE's performance. The CoVE spec mandates that hardware provide the RISC-V Sstc extension.

A TVM manages its own timer by programming a dedicated CSR specified by the RISC-V Sstc extension [64]. The TSM ensures proper resumption of the timer after the security domain context switch. Following the CoVE spec, the TSM discloses by reclassifying the timer value to the hypervisor for scheduling purposes.

5.5 Formal Verification Foundation

We have designed ACE with the goal of providing a high-trust implementation in mind. This influenced the design of the ABI, as well as its implementation, where unsafe code is minimized and safely encapsulated.

Rust memory safety Rust aims to provide memory safety guarantees, which hold as long as only the safe fragment of the language is used. However, for low-level software like the TSM, *unsafe* code is inherently necessary to interface with the hardware and do low-level memory manipulation. Unfortunately, a memory safety error in a piece of unsafe code can nullify the memory safety guarantees of all other code, including safe Rust code. Thus, it is important to both *minimize* the amount of unsafe code and to scrutinize it.

The TSM implementation tries to minimize unsafe code by developing *safe abstractions* over core sources of unsafety that can be re-used across the code – one such abstraction are the page tokens mentioned earlier. In total, the TSM code base has 55 unsafe blocks, each of which is usually very short, with an average size of just over 4 lines. Of these unsafe blocks, 12 are part of the TSM's core memory management, 11 are for interfacing with the hardware (*e.g.* reading CSRs), and 20 for flow handling and managing the state of confidential VMs. In total, about 240 of the 8000 lines of Rust code are unsafe; about 90 of the unsafe lines are for inline assembly backing up or modifying registers. Thus, we argue that it is possible to do effective low-level systems programming with relatively little unsafe code.

ABI design The primary attack surface of the TSM is the ABI. For that reason, in our design we minimized the ABI between the TSM and hypervisor, and between the TSM and a TVM. The *promote* call is the most complex ABI with the largest input. To handle this call, the TSM must traverse the multi-level page table hierarchy and parse individual intermediate page tables. Because the hypervisor creates these page tables, it can maliciously craft the content of these page tables to exploit a buggy TSM implementation.

For example, one of the attacks consists of defining a pointer to a data page that resides in the confidential memory. When handling the promote call, an incorrectly implemented TSM would copy the contents of an arbitrary confidential memory region to a new confidential page mapped to the attacker's TVM. To mitigate this class of attacks, the TSM allows a hypervisor and a TVM to pass as input arguments only memory

addresses they own, *i.e.*, non-confidential memory and guest physical addresses, respectively. To enforce this requirement statically during compile time, we define dedicated Rust types for each type of memory address. At ABI entry points, all memory addresses are validated in order to obtain valid (non-)confidential memory addresses that can be trusted in the rest of the security monitor. These kinds of protections are possible thanks to the rich Rust type system which guarantees non-trivial properties even without complex deductive verification tools.

Initial Formal Verification In ongoing work, we formally verify the TSM using RefinedRust [21]. RefinedRust verifies programs modularly, which means that individual functions are given a specification and verified independently of each other to then compose to a verification result for the whole program.

As explained in Section 4, we integrate formal verification methodology into our design and implementation process. During the design phase, we state individual module invariants that the implementation has to uphold. After the module has been implemented, we annotate the Rust code with specifications for individual functions and data structures. While writing the specifications, individual interfaces may be hard to specify. In this case, the proof engineer and system engineer iterate to simplify the code and make it more amenable to formal specification.

Based on the module's specifications, RefinedRust generates a formal model of the Rust code and specifications in the Rocq prover [60], as well as a proof template in Rocq using RefinedRust's automation tactics. If the automatic proof does not succeed, the proof engineer can interactively add hints to RefinedRust until the verification succeeds. The specification and proof are then checked into continuous verification, where RefinedRust continuously updates the model of the code when it is changed, and checks the specifications and proofs against the model. If a change to the code breaks the proof, the continuous verification rejects the code.

RefinedRust verifies functional correctness of the code against the specification, as well as memory safety of safe and unsafe code, and absence of panics. For unsafe code, we use RefinedRust to verify that functions adhere to Rust's safety contract for all inputs, ensuring that all clients only using safe Rust code cannot trigger memory unsafety.

To demonstrate our approach, let us take a look at a part of the memory safety proof for page tokens. Page tokens are crucial to ensure isolation between security domains, *i.e.*, no two security domains have access to the same page in confidential memory. Figure 5 shows excerpts of the invariant on page tokens. The invariant on `Page` specifies that page tokens are mathematically modeled by a type `page` we define in Rocq, with the following components: the memory location `page_loc`, the stored sequence of words `page_val`, and the size `page_sz`. Then, the core invariant is that a page token exclusively owns the memory region it spans, containing the list of words `page_val`. We state this by providing a RefinedRust type assignment for the `page_loc` location.

```
#[rr::refined_by("p" : "page")]
/// Invariant: A Page exclusively owns its memory region.
#[rr::invariant(#type "p.(page_loc)" : "p.(page_val)" @
  "array usize (page_sz_in_words p.(page_sz))")]
/// Invariant: The page is well-formed.
#[rr::invariant("page_wf p")]
/// Omitted: the page resides in confidential memory
#[rr::invariant("...")]
pub struct Page {
    #[rr::field("p.(page_loc)")]
    address: ConfidentialMemoryAddress,
    #[rr::field("p.(page_sz)")]
    size: PageSize,
}
impl Page {
/// Precond: offset is divisible by the size of usize.
#[rr::requires("size_of usize | off_bytes")]
/// Precond: a usize fits at the offset within page bounds
#[rr::requires("off_bytes + size_of usize ≤
    page_sz_in_bytes p.(page_sz)")]
/// Postcond (omitted): return value is read from page
#[rr::ensures("...")]
fn read(&self, off_bytes: usize) -> Result<usize, Error> {
    unsafe { .. }
}}
```

Figure 5: Definition of the page token invariant in the RefinedRust specification language.

The function `read` on `Page`s reads one machine word from the page. It does so by using *unsafe code* to read from the page's underlying memory. Nevertheless, the function as a whole is *safe* (*i.e.* not marked as unsafe), as `read` does appropriate bounds and alignment checks (*e.g.* that the read word is in confidential memory and inside the page) before performing the read, returning an error in case of an invalid input. The RefinedRust specification we annotate on the function specifies the conditions under which the read succeeds, and the expected result value.

The page token abstraction is useful, as it safely abstracts from the low-level memory operations that it is implemented with. With verification tools like RefinedRust, we can verify once and for all that this abstraction is sound and upholds the safety guarantees of Rust's type system. Other components of ACE like the page allocator can thus be assumed to be free of memory safety errors before fully verifying them for functional correctness and security – and once we fully verify them, Rust's safety proof can be re-used to simplify the full verification.

6 Evaluation

ACE has been designed from the ground up to be minimalistic, take advantage of commodity hardware features, and be amenable to verification. We evaluate next whether ACE hampers the TVM's runtime performance and memory overhead compared with a normal VM. Specifically, we analyze the impact of the TSM on the system's performance as the TSM plays

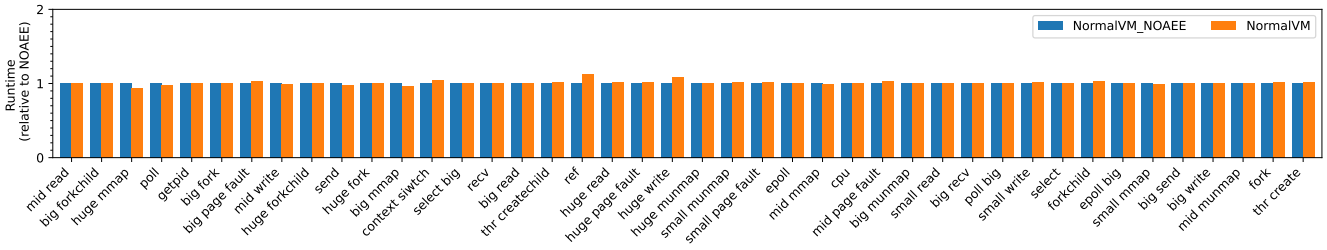

Figure 6: Overhead of running LEBench in a normal VM on system with ACE to a system without ACE.

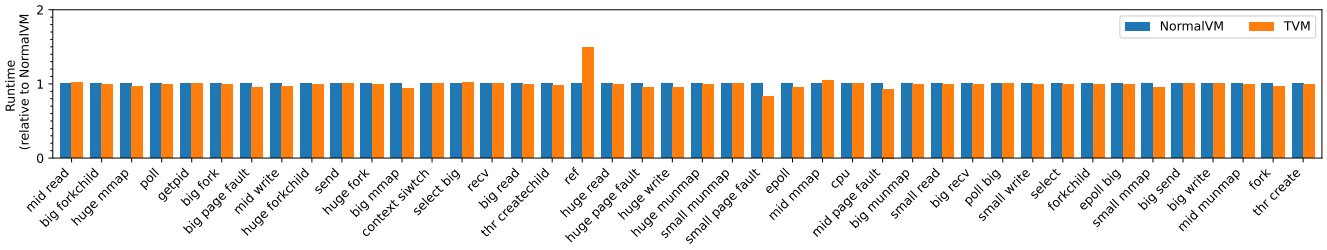

Figure 7: Overhead of running LEBench in a TVM vs a normal VM running on top of ACE.

a critical role during the booting of VMs and context switching of security domains such as during interrupt handling and direct invocation via the ABI by a TVM. We seek to answer the following key questions:

- What is the overhead of the TSM on TVM's execution?
- How efficient is I/O in TVMs?
- What is the impact on TVM's boot time?
- What is the implementation complexity and memory overhead of the page token mechanism?

Evaluation setup. We run evaluation on a SiFive P550 evaluation board equipped with ESWIN EIC7700X system on chip with four 64 bit RISC-V 1.4 GHz cores supporting imafdch, zicsr, zifencei, zba, zbb, sscofpmf extensions. The board has 16 GB RAM, 8 PMPs, and Sv48x4 MMU. The host operating system runs Linux kernel 6.6.21 with CoVE and ESWIN patches that enable support for CoVE and P550 hardware.

The TSM runs with OpenSBI in M-mode and uses two PMPs to define the confidential memory in the upper half of the main memory. The evaluation board is not fully compliant with the requirements of the CoVE spec and implements the pre-ratified version of the RISC-V H extension. Thus, we had to emulate certain functionalities inside the TSM. Specifically, (1) the TSM reads the TVM's instruction that trapped into the TSM because such information is not provided in the mtinst register, (2) the TSM virtualizes the VS-level timer because the hardware does not implement the Sstc extension, (3) the TSM emulates TVM's access to the clock because hardware does not provide it via the CSR_TIME. Since hardware does not implement a root-of-trust for attestation, we utilized a hard coded attestation key to evaluate the local attestation mechanism.

6.1 ACE's Overhead

To evaluate the impact of CoVE patches on the Linux kernel and runtime overhead of the TSM, we ran LEBench [49] on VMs and TVMs with and without ACE. Since the evaluation board's hardware clock does not support nanosecond accuracy, we modified LEBench to measure a batch of operations as a single measurement to ensure that a single measurement runs longer than 1 μs.

Figure 6 shows that a normal VM executed on Linux KVM with CoVE patches and OpenSBI with the TSM performs mostly similarly to a normal VM running without a TSM. In the case with the TSM, normal VM has lower performance for read/write, page fault, and context switch operations. This could be a result of additional execution paths and branches inside Linux KVM that were added by the experimental patch with support for the RISC-V nested acceleration extension (NACL) and a level of indirection introduced by the TSM. The CoVE spec requires NACL to share CSRs values between the TSM and KVM and current the implementation affects both normal VMs and TVMs.

Figure 7 shows that core Linux kernel operations have similar performance for a TVM and a normal VM, except for mmap and page fault operations for which we observed between 5% and 16% performance increase for a TVM compared with a normal VM. It might be caused by the way the TSM manages TVM's address translation, timers, and caches. The TSM never pages out confidential memory pages, preventing G-stage page faults. Moreover, timer interrupt handling and scheduling happens directly in the TSM (M mode) while the regular VM's timer interrupts must be handled by KVM (HS mode) with help of OpenSBI (M mode).

For the ref benchmark, which measures the time to read the

clock value managed by the Linux kernel, we observe significant latency increase for TVMs. This might be caused by the way we compensated for the lack of hardware support for nanosecond accuracy clocks and guest (VS-level) timers. We anticipate that this problem will go away with production-grade hardware that better adheres to the RISC-V CoVE specification.

ACE's multi-VCPU overhead To get a better idea of whether the TSM causes any overhead when multiplexing between VCPUs within a VM, we use the parallel mode of CoreMark [12] to launch different numbers of instances of the benchmark inside a multi-VCPU VM. Each VM is given 4, 8, and 16 VCPUs to match the benchmark's level of concurrency (number of parallel processes). Table 1 shows that the TSM does not introduce any significant performance differences when multiplexing the different VCPUs within a VM across the different configurations.

6.2 Network I/O

We analyzed ACE's impact on performance of a Virtio device by measuring network I/O overhead in a TVM. To do so, we measured throughput and latency of an Nginx server for each VM type and varying number of VCPUs. The Nginx server and the ab benchmarking tool run on separate machines connected to the same switch. ab used 8 threads to generate 10k requests for the same 615 B file. VMs were configured with 5 GB of memory and run Nginx with the number of workers matching the number of VCPUs. We run experiment five times and present average with standard deviation.

Figure 8 shows large throughput variation for normal VMs (between 550 to 1350 req/sec) and drop in throughput when adding more than 2 VCPUs. Similarly, TVMs throughput decreases when adding more VCPUs and constitutes 48% of the normal VM's throughput, before overcommitment of VCPUs on physical harts. When the number of VCPUs exceeds number of harts, we observe increase in latency and drop in throughput (up to 8x). We do not anticipate this degradation to be a major impediment in real-world settings as resource oversubscription is not a typical deployment pattern for embedded systems.

The TVM's lower performance for Virtio is caused by ACE security measures: use of bounce buffers and security domain context switches. Bounce buffers are needed because ACE does not allow hypervisor to perform direct memory access (DMA) in confidential memory (similar to other confidential computing approaches [10, 56, 28]) and security domain context switches prevent covert and side channels. The latter results in cache misses on every context switch because the TSM clears all microarchitectural state when switching control between a TVM's VCPU and the hypervisor.

Table 1: CoreMark benchmark results when running inside the respective VMs. Reported values are means from three runs along with standard deviation.

# of proc	VM_NOACE	VM	TVM
4	30003 ± 776	30387 ± 1172	29373 ± 713
8	32191 ± 325	32368 ± 148	31614 ± 497
16	32952 ± 91	33055 ± 14	32822 ± 40

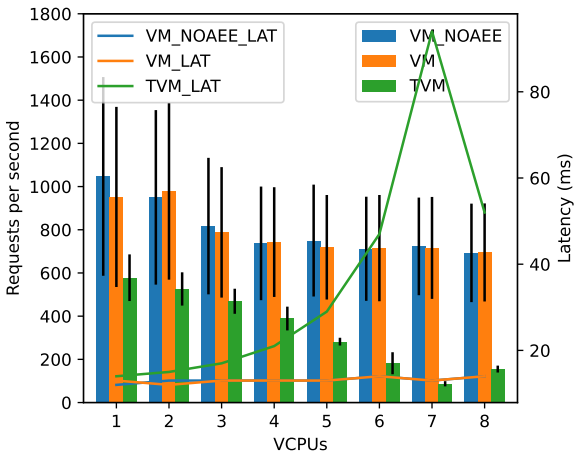

Figure 8: Network I/O performance when hosting Nginx server in VMs with different numbers of VCPUs.

6.3 Boot Time

We measured a VM boot time from the point we request its creation to the point we established an SSH connection to include overhead of local attestation, initialization of virtio devices, and network communication. The results show an average time calculated over eight measurements.

Figure 9 presents boot times of normal VMs and TVMs for different memory sizes and number of VCPUs. Since TVMs' zeroed pages are lazily loaded, boot times for TVMs, similarly to normal VMs, scale with the memory size. We see slower boot times (10 sec increased to up to 30 sec) caused by local attestation and slower I/O for TVMs. The increased boot time when adding more VCPUs to a TVM can be caused by the synchronization between VCPUs: additional IPIs trigger more context switches that result in cache misses due to security domain context switches. ACE does not impact a normal VMs boot time compared to a non ACE system.

6.4 Page Token Memory Overhead

A single page token occupies 9 B in main memory regardless of the page size it represents. The page allocator stores page tokens in a tree data structure, in which every level corresponds to a different architectural page size, adding extra 32 B of information per not-empty node. The presence of a page token in the tree indicates that a page can be allocated. To allow allocation of smaller pages, large page tokens are split into smaller ones. An analogous operation, merging, occurs when pages

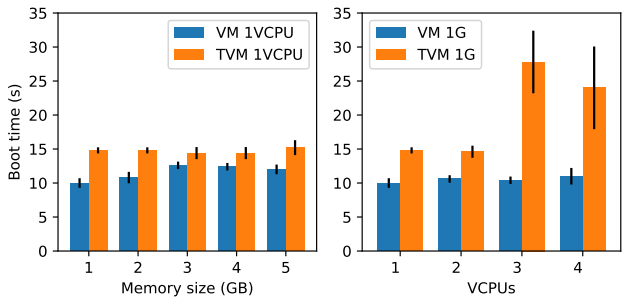

Figure 9: Average time to boot a normal VM and a TVM with different memory size and number of VCPUs. Bars show standard deviation.

are deallocated and returned to the page allocator. In such cases, multiple contiguous smaller page tokens are merged into a larger single page token, reducing the number of nodes in the tree. Thus, the latency of allocation/de-allocation operations is independent of fragmentation of the confidential memory, and the number of unallocated page tokens is minimized. In more detail, 1 GiB of unallocated confidential memory can be represented as a single 1 GiB page token, taking then only 9 B of the ACE heap. If all page tokens representing the smallest architectural page size (4 KiB) are allocated, then page allocator's tree is empty but allocated page tokens are stored in TVM's specific structures. In such case, the overhead of page tokens is the largest and results in occupancy of 2.25 MiB of ACE heap for every 1 GiB of confidential memory.

7 Related Work

Target use cases. ACE is tailored for mid- to high-end, off-the-shelf, embedded RISC-V processors. Other VM-based confidential computing systems target AI, cloud, and multi-tenant high-end systems. A summary of confidential computing technology for high-end systems (*e.g.*, from AMD, ARM, and Intel) appears in the Related Work section of [54]. Some design concepts from one such system, IBM's Protected Execution Facility [28] were incorporated into ACE: open source software, secure boot, TAP data structures for local attestation, security domains, and hardware-enforced memory access control managed by privileged firmware.

Commodity Hardware. Confidential computing systems rely on hardware to isolate resources, protect the execution state, and meet performance targets. ACE builds upon, but does not rely on custom hardware, nor on hardware that requires license fees. The CoVE architecture [54] with its Salus TSM reference implementation [31] requires extensions to the RISC-V ISA such as a protected execution state and scalable isolation of resources [53]. IBM Z Secure Execution for Linux [30] uses powerful hardware-accelerated cryptography to protect memory from physical attacks. ACE does not protect against physical attacks.

Process-based TEEs. Other commercial and academic systems target process-based confidential computing for RISC-V: Cerberus [37], Keystone [38], OP-TEE [63], Penglai [17], Servas [58], Timber-V [65], Sanctum [13], SPEAR-V [55], and Elasticlave [66].

Formal verification of systems. CertiKOS [25] is a framework for verification of operating system kernels. However, it is still written in an inherently unsafe combination of C and x86 assembly, putting greater needs for abstraction on the verification methodology. The verification proceeds in layers that progressively abstract implementation details in the Rocq proof assistant. It was used to verify the mCertiKOS hypervisor.

ARM CCA [1] is a confidential computing architecture for ARM, in which realm management monitor (RMM) has a similar role to ACE's TSM. RMM is implemented in C and assembly, and thus there is a higher risk for vulnerabilities caused by memory unsafety. Li et al. [39] verify an early snapshot of the RMM implementation using CertiKOS' approach, incrementally abstracting to a small top-level specification in Rocq. However, their proof cannot be easily updated when the code evolves. Fox et al. [19] propose continuous verification of the RMM implementation, using a combination of interactive theorem proving in the HOL4 proof assistant for a core model, and bounded model checking and concurrency-aware testing for the actual system. This results in less high assurances, but enables a larger-scale integration.

Verismo [67] functionally verifies firmware implemented in Rust and running inside a confidential VM on AMD SEV-SNP [56]. It relies on the isolation boundary provided by the AMD platform security processor (PSP), while ACE's TSM is functionally closer to firmware running inside PSP. Their approach to formal verification differs from ours by using Verus [35]: Verus is faster and more automated for proving functional specifications, while RefinedRust offers higher assurances due to its foundational verifier and smaller trusted computing base. Verus trusts unsafe Rust code, whereas we verify unsafe code. Additionally, Verus's proofs require significant modifications to the Rust source code, while we aim to verify idiomatic Rust code.

Komodo [18] is a process-based TEE that builds on top of ARM TrustZone. Due to its implementation in assembly it has limited potential to scale in terms of verification for VM-based TEEs. NOVA [57] is a microhypervisor that, similarly to ACE, follows a philosophy of minimizing the critical code base and reducing to the core functionality. Verification of its variant in ongoing work [26]. GlobalPlatform [22] publishes security standards for IoT devices through its Common Criteria Protection Profile (PP) for TEEs. ProveriT [27] is a verified formal specification of the GlobalPlatform PP for TEEs. In contrast, we do not aim to just verify the specification, but the concrete implementation of ACE.

8 Conclusion

We introduced principles and a methodology to design and implement high-assurance embedded systems. We applied these to build ACE, an open-source and royalty-free confidential computing for embedded RISC-V processors. Thus far we have formally verified the memory safety of a core part of the ACE implementation. The formal verification of the rest of the implementation and security properties is work in progress. Our work on ACE was used to extended the RISC-V CoVE spec with a deployment model targeting embedded systems.

We are first to evaluate VM-based confidential computing on RISC-V hardware. The results show that a confidential VM running on this hardware, which does not have confidential computing-specific extensions, is practical. ACE incurs low performance overhead for process-intensive workloads and up to 50% overhead for multi-vcpu network intensive workloads.

Acknowledgement

We would like to thank SiFive for making a P550 evaluation board available to us, as well as Warren Lew and John Chasko for their support during the initial setup. We thank also our colleagues from the RISC-V community for great collaboration on the RISC-V specifications, especially Ravi Sahita and Atish Kumar Patra for feedback on our design and extensions to the Linux KVM's CoVE patches.

References

[1] Arm confidential compute architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Accessed on May 2025.

[2] William A Arbaugh, David J Farber, and Jonathan M Smith. A secure and reliable bootstrap architecture. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*, pages 65–71. IEEE, 1997.

[3] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 2015.

[4] Krste Asanovic, Paul Donahue, Greg Favor, John Hauser, James Kenney, David Kruckemyer, Shubu Mukherjee, Stefan O'ear, Vernon Pang, Anup Patel, Josh Scheid, Ved Shanbhogue, and Andrew Waterman. The risc-v advanced interrupt architecture, version 1.0. https://github.com/riscv/riscv-aia/releases/download/1.0/riscv-interrupts-1.0.pdf, 2023.

[5] Christian Bornträger, Jonathan D Bradbury, Reinhard Bündgen, Fadi Busaba, Lisa Cranton Heller, and Viktor Mihajlovski. Secure your cloud workloads with ibm secure execution for linux on ibm z15 and linuxone iii. *IBM Journal of Research and Development*, 64(5/6):2–1, 2020.

[6] Andrew Bresticker, Andy Dellow, Atish Patra, Atul Khare, Beeman Strong, Christian Bolis, Dingji Li, Dong Du, Dylan Reid, Eckhard Delfs, Fabrice Marinet, Guerney Hunt, Jiewen Yao, Kailun Qin, Manuel Offenberg, Nicholas Wood, Nick Kossifidis, Osman Koyuncu, Qing Li, Rajnesh Kanwal, Ravi Sahita, Rob Bradford, Samuel Ortiz, Steven Bellock, Vedvyas Shanbhogue, Wojciech Ozga, and Yann Loisel. Confidential vm extension (cove) for confidential computing. https://github.com/riscv-non-isa/riscv-ap-tee/releases/download/v0.7/riscv-cove.pdf, accessed on May 2025.

[7] Fabiana Cambricoli. Nova falha do Ministerio da Saude expoe dados pessoais de mais de 200 milhoes de brasileiros. https://saude.estadao.com.br/noticias/geral,nova-falha-do-ministerio-da-saude-expoe-dados-70003536340, accessed on May 2025.

[8] Charly Castes, Neelu S. Kalani, Sofia Saltovskaia, Noé Terrier, Abel Vexina Wilkinson, and Edouard Bugnion. Kicking the firmware out of the tcb with the miralis virtual firmware monitor. In *Proceedings of the 2nd Workshop on Kernel Isolation, Safety and Verification*, KISV '24, 2024.

[9] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. uCFI: Formal Verification of Microarchitectural Control-flow Integrity. In *CCS*, October 2024.

[10] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *arXiv preprint arXiv:2303.15540*, 2023.

[11] Chromium. The chromium projects - memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/, accessed on May 2025.

[12] Embedded Microprocessor Benchmark Consortium. coremark. https://github.com/eembc/coremark, 2025.

[13] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[14] Common Criteria. Common criteria for information technology security evaluation: Part 5: Pre-defined packages of security requirements. https://www.commoncriteriaportal.org/files/ccfiles/CC2022PART5R1.pdf, 2022.

[15] Natasha Dailey. The hackers that attacked a major US oil pipeline say it was only for money — here's what to know about DarkSide. https://www.businessinsider.com/pipeline-cyber-attack-darkside-hacker-group-shutdown-ransomware-money-politics-oil-2021-5?op=1&r=US&IR=T, accessed on May 2025.

[16] RTCA / EUROCAE. Formal methods supplement to do-178c [ed-12c] and do-178a [ed-109a]. *DO-333/ED-218*, 2011.

[17] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 275–294. USENIX Association, July 2021.

[18] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *26th ACM Symposium on Operating Systems Principles*, pages 287–305. ACM, October 2017. The Komodo specification, prototype implementation, and proofs are available at https://github.com/Microsoft/Komodo.

[19] Anthony C. J. Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P. Mulligan, Gustavo Petri, and Nathan Chong. A verification methodology for the arm® confidential computing architecture: From a secure specification to safe implementations. *Proc. ACM Program. Lang.*, 7(OOPSLA1):376–405, 2023.

[20] BSI Bundesamt für Sicherheit in der Informationstechnik. Railway applications -communication, signalling and processing systems - software for railway control and protection systems. In *European Standard EN 50128*, 2011.

[21] Lennard Gaeher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. RefinedRust: Towards high-assurance verification of unsafe Rust programs. In *Rust Verification Workshop*, 2023.

[22] GlobalPlatform. TEE protection profile v1.3. https://globalplatform.org/specs-library/tee-protection-profile-v1-3.

[23] Google. Google security blog - memory safe languages in android 13. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html, accessed on May 2025.

[24] RISC-V Task Group. Risc-v platform-level interrupt controller specification, version 1.0. https://github.com/riscv/riscv-plic-spec/releases/download/1.0.0/riscv-plic-1.0.0.pdf, 2023.

[25] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In *OSDI*, pages 653–669. USENIX Association, 2016.

[26] Hoang-Hai Dang, David Swasey, and Gregory Malecha. Towards modular specification and verification of concurrent hypervisor-based isolation. https://www.bluerock.io/formal-methods-publications/towards-modular-specification, 2024.

[27] Jilin Hu, Fanlang Zeng, Yongwang Zhao, Zhuoruo Zhang, Leping Zhang, Jianhong Zhao, Rui Chang, and Kui Ren. Proverit: A parameterized, composable, and verified model of tee protection profile. *IEEE Trans. Dependable Secur. Comput.*, 21(6):5341–5358, November 2024.

[28] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. Confidential Computing for OpenPOWER. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, 2021.

[29] Sander Huyghebaert, Steven Keuchel, Coen De Roover, and Dominique Devriese. Formalizing, verifying and applying isa security guarantees as universal contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2083–2097, 2023.

[30] IBM. Linux on IBM Z and IBM LinuxONE. https://www.ibm.com/docs/en/linuxonibm/pdf/lx24se04.pdf, 2024.

[31] Rivos Inc. salus. https://github.com/rivosinc/salus, accessed on May 2025.

[32] Intel. Intel software guard extensions (intel sgx). https://www.intel.com/

content/www/us/en/developer/tools/software-guard-extensions/overview.html, Accessed on May 2025.

[33] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2017.

[34] Pau Ku and Channing Tang. RISC-V IOPMP Specification Document: version 1.0.0-draft1. https://github.com/riscv-admin/iopmp/blob/main/specification/riscv_iopmp_specification.pdf, accessed on May 2025.

[35] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023.

[36] Stella Lau, Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. Specification and verification of strong timing isolation of hardware enclaves. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, 2024.

[37] Dayeol Lee, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanovic. Cerberus: A formal approach to secure and efficient enclave memory sharing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1871–1885, New York, NY, USA, 2022. Association for Computing Machinery.

[38] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020. Repository: https://github.com/orgs/keystone-enclave/repositories.

[39] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *OSDI*, pages 465–484. USENIX Association, 2022.

[40] Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. Empirical study towards a leading indicator for cost of formal software verification. In *International Conference on Software Engineering*, page 11, Firenze, Italy, February 2015.

[41] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of HILT*, 2014.

[42] NIST. Cve-2021-44228 detail. https://nvd.nist.gov/vuln/detail/cve-2021-44228, 2021.

[43] RISC-V Platform Runtime Services Task Grouphttps://github.com/riscv non-isa/riscv-sbi-doc/releases/download/v2.0/riscv sbi.pdf. Risc-v supervisor binary interface specification. https://github.com/riscv-non-isa/riscv-sbi-doc/releases/download/v2.0/riscv-sbi.pdf, 2024.

[44] National Institute of Standards and Technology (NIST). Security requirements for cryptographic modules. https://csrc.nist.gov/pubs/fips/140-3/final, 2019.

[45] National Institute of Standards and Technology (NIST). Cve-2024-3094 detail. https://nvd.nist.gov/vuln/detail/cve-2024-3094, 2024.

[46] National Institute of Standards and Technology (NIST). Fips 203: Module-lattice-based key-encapsulation mechanism standard. https://csrc.nist.gov/pubs/fips/203/final, 2024.

[47] Wojciech Ozga, Guerney D. H. Hunt, Michael V. Le, Elaine R. Palmer, and Avraham Shinnar. Towards a formally verified security monitor for vm-based confidential computing. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP2023, 2023.

[48] PCI-SIG. Tee device interface security protocol (tdisp). https://members.pcisig.com/wg/PCI-SIG/document/18268, 2022.

[49] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of linux's core operations. Huntsville, Ontario, Canada, October 2019.

[50] Reuters. Foreign Hackers Probe European Critical Infrastructure Networks: Sources. https://www.reuters.com/article/us-britain-cyber-idINKBN19V1C7, accessed on May 2025.

[51] RISC-V International, Western Digital Corporation or its affiliates. RISC-V Open Source Supervisor Binary Interface (OpenSBI). https://github.com/riscv-software-src/opensbi, accessed on May 2025.

[52] Ravi Sahita, Andy Dellow, Dean Liberty, Deepak Gupta, Guerney Hunt, Krste Asanovic, Mark Hill, Nick Wood, Osman Koyuncu, Paul Elliott, Ved Ortiz, Samuel abd Shanbhogue, and Wojciech Ozga. RISC-V SmmTT: Supervisor Domain Access Protection. https://

github.com/riscv/riscv-smmtt/, accessed on May 2025.

[53] Ravi Sahita and Atish Patra. Enabling new security frontiers: Deep dive into confidential computing on risc-v. https://lsseu2024.sched.com/event/1ebVO/enabling-new-security-frontiers-deep-dive-into-implementing-confidential-computing-on-ris accessed on May 2025.

[54] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. Cove: Towards confidential computing on RISC-V platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 315–321, 2023.

[55] David Schrammel, Moritz Waser, Lukas Lamster, Martin Unterguggenberger, and Stefan Mangard. Spear-v: Secure and practical enclave architecture for risc-v. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, page 457–468, New York, NY, USA, 2023. Association for Computing Machinery.

[56] AMD Sev-Snp. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 53(2020):1450–1465, 2020.

[57] Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *EuroSys*, pages 209–222. ACM, 2010.

[58] Stefan Steinegger, David Schrammel, Samuel Weiser, Pascal Nasahl, and Stefan Mangard. Servas! secure enclaves via risc-v authenticryption shield. In *Computer Security – ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part II*, page 370–391, Berlin, Heidelberg, 2021. Springer-Verlag.

[59] Dina Temple-Raston. A 'worst nightmare' cyberattack: The untold story of the solarwinds hack. https://www.npr.org/2021/04/16/985439655/a-worst-nightmare-cyberattack-the-untold-story-of-the-solarwinds-hack, accessed on May 2025.

[60] The Coq Team. The Coq proof assistant. https://coq.inria.fr/, accessed on May 2025.

[61] The New York Times. Hack of Saudi Petrochemical Plant Was Coordinated From Russian Institute. https://www.nytimes.com/2018/10/23/us/politics/russian-hackers-saudi-chemical-plant.html, accessed on May 2025.

[62] The New York Times. Hackers Are Targeting Nuclear Facilities, Homeland Security Dept. and F.B.I. Say. https://www.nytimes.com/2017/07/06/technology/nuclear-plant-hack-report.html, accessed on May 2025.

[63] TrustedFirmware.org. About OP-TEE. https://optee.readthedocs.io/en/latest/general/about.html#about-op-tee.

[64] Andrew Waterman, Krste Asanovi, and John Hauser. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20241101. https://github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-f32140c-2025-05-05/riscv-privileged.pdf, accessed on May 2025.

[65] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. 01 2019.

[66] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E. Carlson, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4111–4128, Boston, MA, August 2022. USENIX Association.

[67] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. Verismo: A verified security module for confidential vms. In *OSDI*, pages 599–614. USENIX Association, 2024.