Design and Verification of the Arm Confidential Compute Architecture

Xupeng Li
Columbia University

Xuheng Li
Columbia University

Christoffer Dall
Arm Ltd

Ronghui Gu
Columbia University

Jason Nieh
Columbia University

Yousuf Sait
Arm Ltd

Gareth Stockwell
Arm Ltd

Abstract

The increasing use of sensitive private data in computing is matched by a growing concern regarding data privacy. System software such as hypervisors and operating systems are supposed to protect and isolate applications and their private data, but their large codebases contain many vulnerabilities that can risk data confidentiality and integrity. We introduce Realms, a new abstraction for confidential computing to protect the data confidentiality and integrity of virtual machines. Hardware creates and enforces Realm world, a new physical address space for Realms. Firmware controls the hardware to secure Realms and handles requests from untrusted system software to manage Realms, including creating and running them. Untrusted system software retains control of the dynamic allocation of memory to Realms, but cannot access Realm memory contents, even if run at a higher privileged level. To guarantee the security of Realms, we verified the firmware, introducing novel verification techniques that enable us to prove, for the first time, the security and correctness of concurrent software with hand-over-hand locking and dynamically allocated shared page tables, data races in kernel code running on relaxed memory hardware, integrated C and Arm assembly code calling one another, and untrusted software being in full control of allocating system resources. Realms are included in the Arm Confidential Compute Architecture.

1 Introduction

The use of sensitive private data in many applications from advertising to healthcare, often in the context of machine learning models, has raised concerns regarding the privacy of data in computing. These applications increasingly run on commodity cloud providers. For example, data and computation may be contained in virtual machines (VMs) running on shared hardware in the cloud, relying on a hypervisor to preserve VM isolation to protect applications and their data in VMs.

Software stacks generally require applications to trust system software which they rely on, such as hypervisors and operating systems (OSes). Although hypervisors and OSes are supposed to protect applications and their private data, their large codebases contain vulnerabilities that can risk data confidentiality and integrity. Vulnerable system software running at more privileged levels that can access application data is a significant security issue.

To address this problem, we introduce the *Arm Confidential Compute Architecture (Arm CCA)*. CCA provides *Realms*, secure execution environments that are completely opaque to privileged, untrusted system software such as OSes and hypervisors. CCA retains the ability of existing system software to manage hardware resources for Realms while preventing it from violating Realm confidentiality and integrity. For example, a hypervisor should retain its ability to dynamically allocate memory to or free memory from a Realm VM, but must never be allowed to access the protected memory contents of a Realm VM. CCA guarantees the confidentiality and integrity of Realm code and data in use, that is data in CPU registers and memory, but makes no guarantees regarding their availability. Confidentiality means that any change that a Realm makes to its private data cannot be observed by other Realms or untrusted system software. Integrity means that a Realm will not observe any changes to its private data that it did not make. Because CCA does not guarantee availability, a Realm data access is allowed to halt Realm execution.

CCA avoids hardware complexity by only introducing core hardware mechanisms for attestation and basic address space protection, then relying on firmware to manage the use of those mechanisms. Specifically, CCA introduces *Realm world*, a new physical address space for Realms orthogonal to privilege levels and separate from the existing *Non-Secure (NS) world* used today for running software stacks. Within each world, the normal privilege levels apply and instructions retain their existing semantics, but software in NS world cannot access CPU state and memory used by software in Realm world. CCA introduces a new *Realm Management Monitor (RMM)*, firmware which runs in Realm world at a higher privilege level than Realms. Untrusted system software such as a hypervisor running in NS world can then make requests to RMM to manage Realms, including creating and running Realms. RMM protects the confidentiality and integrity of Realms while handling such requests. System software in NS world is expected to retain full control of the dynamic allocation of hardware resources to Realms, including memory allocation and CPU scheduling.

Because any compromise of RMM could violate the security guarantees of Realms, it is crucial to formally verify its security and functional correctness. However, verifying RMM poses at least four significant challenges. First, RMM employs fine-grained synchronization mechanisms such as hand-over-hand locking to improve performance. Second, RMM has data races and runs on Arm multiprocessor hardware with relaxed

memory behavior. Third, RMM contains both C and Arm assembly code integrated together which call one another freely. Finally, RMM must protect the confidentiality and integrity of Realms even though untrusted system software has full control over the dynamic allocation of Realm resources. Previous verification approaches have not been able to verify system software with these properties [10, 11, 13, 26, 42, 43, 50, 62]

To verify RMM, we introduce VIA (Verification Infrastructure for Armv9-A), which supports four key verification techniques. First, VIA introduces *mover oracle queries* to combine a local CPU model with mover types [44]. These queries encapsulate how operations on other CPUs are interleaved with local CPU operations and can be reordered using mover types to group local CPU operations together. Along with the local CPU model, this allows easier sequential reasoning and modular verification. This makes it possible for the first time to verify hand-over-hand locking with dynamically allocated shared multi-level page tables in system software.

Second, VIA decomposes concurrent code into data race free (DRF) and not-DRF components, then introduces *permutation conditions* for the latter such that proofs on a sequentially consistent memory model will hold on relaxed memory hardware for all concurrent code. Instead of having to verify all of the code directly on relaxed memory hardware, all that is required is to prove that the code satisfies the permutation conditions, which ensure equivalent behavior on sequentially consistent and relaxed memory hardware. VIA allows any permutation conditions to be defined, supporting verification of a broad class of programs.

Third, VIA bridges incompatibilities between C and assembly code due to CPU register state being hidden by the former but explicitly used by the latter. To accomplish this without dependencies on a specific compiler, VIA introduces a register accounting mechanism to correctly verify integrated C and Arm assembly code. It leverages the machine-level procedure call standard for the Arm instruction set to specify how registers are potentially used when assembly code calls a C function or is called by a C function. VIA tracks CPU register state across invocations of both C and assembly code primitives, capturing any information flow through CPU registers even if hidden by C semantics.

Finally, VIA introduces an ideal/real paradigm for verifying security properties that can be applied to Realms, even though untrusted system software is in full control of system resources and can reclaim system resources such as memory without Realm permission, breaking noninterference. VIA defines an *idealized secure machine model* that supports declassification. Realm private data is stored in physically isolated memory and CPU registers. Data channels, governed by security policies, are used to exchange information between Realms and untrusted software. We can then prove the security guarantees of Realms by verifying that the implementation refines its specification and the real system captured by the specification simulates the idealized secure machine model. This approach allows us to prove, for the first time, the integrity and confidentiality of Realms. A key feature of the proof is that it only needs to trust the specification of the small idealized secure machine model; the much larger specification of the real system does not need to be trusted.

We implemented, evaluated, and verified an early prototype of CCA firmware. Although CCA hardware is not yet available, we demonstrated CCA on a functionally accurate Arm Fast Model with CCA support. We modified the Linux KVM hypervisor [19–21] to run on CCA and manage Realm VMs, and ran various VM workloads on the model. We also ported CCA firmware to current Arm hardware to obtain preliminary data on CCA performance, which shows that KVM on CCA incurs modest overhead versus vanilla KVM on real application workloads. We verified the correctness of both the C and Arm assembly CCA firmware implementation, including RMM, proving its implementation refines its specification through 43 abstraction layers. We then proved the specification is equivalent to the behavior of the idealized secure machine model to verify the confidentiality and integrity guarantees of Realms. The proof only needs to trust roughly 200 lines of Coq specification, making the formal security guarantees easy to read and understand. This is the first proof of the security guarantees of a confidential computing architecture. Realms will be included in Armv9-A, the next version of the Arm architecture.

2 Threat Model

We consider an attacker without physical access to the machine and assume the attacker's goal is to compromise the confidentiality and integrity of VM data. Confidentiality and integrity attacks in scope include compromising the hypervisor or any other software to read or modify private VM memory or register state, including by controlling DMA-capable devices, or via memory remapping and aliasing attacks. We assume a VM does not voluntarily reveal its own private data whether on purpose or by accident, but attacks from other compromised VMs, including confidentiality and integrity attacks, are in scope. Availability attacks by a compromised hypervisor are out of scope. Protection against known software error injection attacks and side-channel attacks require appropriate usage of architectural mitigations and are beyond the scope of this paper. DRAM attacks, such as cold boot attacks, live probing, or replay, require additional hardware and are outside of the scope of the threat model.

3 CCA Design

A key challenge with introducing Realms is how to provide backwards compatibility with a widely-used existing architecture that, like other CPU architectures, was designed based on the fundamental assumption that more privileged levels have greater control and access than less privileged levels of

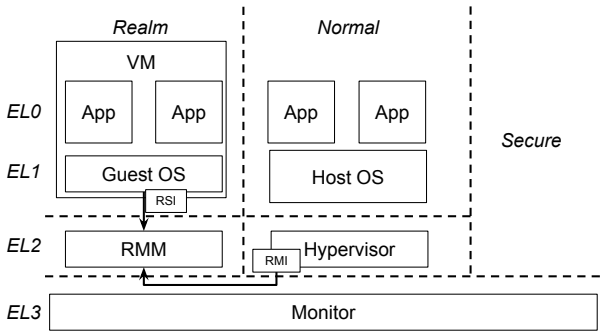


Figure 1: Arm Confidential Compute Architecture.

software. One issue is understanding the potential interactions of Realms with all the features in the Arm architecture. For example, debug registers defined in the Arm architecture are explicitly designed to allow hypervisors to peer into VM state, which is fundamentally at odds with Realms. The behavior of each instruction could be redefined in the context of Realms, but this would be an enormous undertaking with unclear compatibility implications, given that the Arm instruction set was designed over multiple decades.

Another issue is how to provide memory protection and isolation for Realms. The way this works for VMs is that hypervisors manage *nested page tables (NPTs)* [9] to isolate physical memory between VMs and protect hypervisor memory from VMs. The physical addresses perceived by a VM are *intermediate physical addresses (IPAs)*, which are translated by an NPT to physical addresses for the hardware. Physical memory not mapped to the NPT is not accessible to the VM. However, NPTs are under full control of the untrusted hypervisor, providing no protection against hypervisor access to VM data. While it would be possible to introduce an additional data structure to track memory ownership for each frame of physical memory [3], this approach comes with several problems. First, the amount of information required for each frame of memory would be substantial and significantly impact TLB design and performance. Second, this data structure would have to be managed either via a separate more privileged software entity than the hypervisor or via complex instructions capable of capturing measurements of data assigned to a Realm. Such complex CISC-like instructions would almost certainly require introducing extensive microcode into an architecture, which does not currently use any.

CCA avoids these problems by only introducing simple hardware mechanisms orthogonal to existing privilege levels and then relies on firmware to manage the use of those mechanisms. This reduces hardware complexity at the cost of depending on the firmware for the security guarantees of the architecture. As a result, verifying CCA firmware is of crucial importance.

Figure 1 shows how CCA extends the Arm architecture. Armv8-A provided two statically partitioned worlds, NS world used by most software stacks and Secure world to host

Security State	PAS			
	NS	Secure	Realm	Root
NS	Allow	Block	Block	Block
Secure	Allow	Allow	Block	Block
Realm	Allow	Block	Allow	Block
Root	Allow	Allow	Allow	Allow

Table 1: CCA access control policy. The entity accessing a granule belongs to a security state, while the PAS is a property only of the granule being accessed.

platform security services [4]. CCA introduces Realm world, which is fully compatible with NS world so that existing software stacks that run in NS world can also run in Realm world. CCA provides three privilege levels in each of the NS, Realm and Secure worlds: EL0 for user, EL1 for kernel, and EL2 for hypervisor. Because Realm and Secure worlds are mutually distrusting, CCA introduces a fourth, more privileged Root world to manage switching between the other worlds.

Each world has its own *Physical Address Space (PAS)*. Each 4 KB frame of physical memory, which we refer to as a *memory granule*, belongs to one PAS at any given time. Individual memory granules can be dynamically transitioned from NS PAS to Realm PAS; there is no static partitioning of resources between NS and Realm worlds. Hardware performs a PAS check on each memory access against a *Granule Protection Table (GPT)* that tracks the PAS of each memory granule and enforces the access control policy shown in Table 1, forbidding invalid accesses. NS world can only access its own memory. Realm and Secure worlds can access their own respective memory and NS memory, but cannot access each other's memory. CCA hardware requires all DMA accesses be subject to GPT checks, protecting the Realm PAS against DMA-based attacks. We focus on the interactions between NS and Realm worlds and omit further discussion of Secure world due to space constraints.

CCA relies on two trusted firmware components: RMM and the *EL3 Monitor (EL3M)*. RMM runs at EL2 in Realm world. It controls the execution of Realms and provides services to untrusted system software running in NS world. It isolates Realms from each other using existing virtualization technologies such as NPTs and CPU register save/restore sequences. Because RMM only enforces the security guarantees of CCA, it can be orders of magnitude smaller than bare-metal hypervisors which must also provide virtualization functionality. For example, to run Realm VMs, RMM protects the confidentiality and integrity of Realms while relying on existing hypervisors for everything else, including resource allocation and scheduling, physical hardware support, and complex device emulation.

EL3M runs in Root world at EL3, the highest level of privilege. It is responsible for context switching CPU execution among the three other worlds and managing the GPT. EL3M can access memory in any PAS. Only EL3M can change the PAS of a granule, which involves updating its entry in the GPT. Software running in the three other worlds can issue a *Secure Monitor Call (SMC)* to EL3M to request a PAS change.

In the current version of CCA, the Realm isolation boundary

Command	Description
`Version`	Query RMI ABI version.
`Granule.Delegate`	Change granule (from NS) to Delegated.
`Granule.Undelegate`	Change granule (from Delegated) to NS.
`Realm.Create`	Create Realm Descriptor (RD).
`Realm.Destroy`	Destroy Realm identified by RD.
`Realm.Activate`	Change Realm (from New) to Active.
`REC.Create`	Create Realm Execution Context (REC).
`REC.Destroy`	Destroy REC.
`REC.Run`	Enter REC (i.e. run VCPU).
`Data.CreateUnknown`	Change granule to Data with unknown content.
`Data.Create`	Change granule to Data, copy NS content.
`Data.Destroy`	Change Data granule to Delegated, zeroed.
`RTT.Create`	Create Realm Translation Table (RTT).
`RTT.Destroy`	Destroy RTT.
`RTT.MapProtected`	Map Data granule in RTT.
`RTT.UnmapProtected`	Remove mapping from RTT.
`RTT.MapUnprotected`	Map NS granule in RTT.
`RTT.UnmapUnprotected`	Remove NS mapping from RTT.
`RTT.ReadEntry`	Return content of an RTT entry.

Table 2: RMM Realm Management Interface (RMI).

is at the level of entire VMs; applying Realms to secure other entities such as containers [59] is future work. Similar to normal VMs, a Realm VM can concurrently run multiple virtual CPUs (VCPUs) and the number of Realm VMs on a system is only limited by the amount of physical memory available, not by any arbitrary limits. The untrusted hypervisor always has the ability to stop scheduling a Realm and can always reclaim memory assigned to a Realm, but in no circumstances does it have access to Realm CPU or memory state.

This split of responsibility between an untrusted hypervisor and RMM, where the untrusted hypervisor allocates memory, and RMM provides integrity and confidentiality guarantees for the data and code stored in that memory, is accomplished through a simple but powerful delegation concept. The hypervisor *delegates* memory to Realm world, and *undelegates* memory back to NS world. All memory used by Realms must first be delegated by the hypervisor; RMM does not itself manage a pool of memory for Realms. Once memory is delegated to Realm world, the hypervisor can request RMM to use it for various purposes, such as storing metadata or data for a Realm. Whenever a memory granule is delegated to Realm world but not used by RMM, RMM ensures that the granule contains only zeros, reducing the risk of accidental information flow when a granule is reused or undelegated.

RMM provides a *Realm Management Interface (RMI)* for the hypervisor to request RMM to delegate memory, create Realms, execute Realms, and allocate memory to Realms. Each RMI command is implemented as an SMC, so when the hypervisor invokes the command, it traps to EL3M, which in turn switches execution to RMM in Realm world to handle the command. Upon completion of the RMI command, RMM issues an SMC to EL3M, which switches execution back to the hypervisor in NS world. Table 2 lists the RMI commands.

RMM must know the state of each memory granule on the system to uphold the security guarantees of Realms, which it accomplishes by maintaining its own *Granule Status Table (GST)* to track the delegation status and current use of each granule. RMM uses the GST to ensure that a granule is in a valid state to perform the requested action. For example, when the hypervisor delegates a memory granule, RMM checks its GST to confirm the granule has not already been delegated, then issues an SMC to EL3M to request a change to Realm PAS. EL3M checks that the granule is currently in NS PAS, then updates the GPT to move it to Realm PAS. Finally, RMM updates its GST to record that the granule has been delegated. If the hypervisor attempts to delegate a granule which is already delegated, or undelegate a granule which is in active use by RMM, RMM returns an error code to the untrusted hypervisor. This pattern of checking valid states and either performing a discrete action or returning an error is used for all RMI commands, allowing RMM to remain in overall control of the consistency of the system, while complex logic for policy and resource allocation remains in the hypervisor. Unlike the GPT, the GST is not checked by hardware and is only a software bookkeeping mechanism. By maintaining a separate GST from the GPT, the GPT can be kept simple so that it only needs to contain information required for hardware-enforced checks.

The hypervisor creates Realms, *Realm Execution Contexts (RECs)*, and *Realm Translation Tables (RTTs)* using the respective commands in Table 2. RECs correspond to VCPUs and RTTs correspond to NPTs for normal VMs. RTTs are Arm stage 2 page tables that translate from an IPA to a physical address. RTTs use the same format and topological layout in Realm world as NS stage 2 page tables, but also provide a bit which allows Realms to access NS granules under the control of RMM, for example, for virtual I/O between a Realm and the hypervisor. On each of the Realm, REC, and RTT create commands, RMM checks the GST entry for the address provided to confirm the granule is already delegated, and updates the GST entry to track that it is being used for Realm, REC, and RTT metadata, respectively. We refer to a Realm's metadata as its *Realm Descriptor (RD)*.

A Realm provides a *Protected Address Range (PAR)* within its IPA space, which RMM ensures can only be mapped to Realm PAS granules. For accesses within the PAR, RMM guarantees confidentiality and integrity to the Realm; outside the PAR, the hypervisor is free to map NS PAS granules or emulate accesses. This provides an OS running inside a Realm VM with a reliable mechanism to determine whether it is accessing its own private memory, or memory which can be shared with untrusted agents, for example, buffers used for untrusted DMA with virtual or physical network and block devices.

During Realm creation, the hypervisor can assign a granule to the Realm at a specific IPA and copy data to it from an NS granule. The IPA and data are cryptographically hashed and the hash is included in the attestation token of the Realm. The attestation token allows a Realm owner to reason about its initial state and content. Once a Realm has been activated, the measurement is fixed, and memory can only be added to otherwise unused IPAs with unknown content. We refer

to delegated granules used to store data for a Realm as *Data granules*. The hypervisor can request that RMM maps NS granules outside the PAR at any time. Physically contiguous delegated memory can be mapped to a Realm in blocks larger than 4 KB granules to optimize TLB usage.

The hypervisor can reclaim memory from a Realm at any time. RMM zeros a granule before undelegating it and returning it to the hypervisor. Subsequent accesses from a Realm to the IPA where the memory was reclaimed result in a stage 2 abort to RMM which prevents further execution of the Realm and preserves the CCA integrity guarantee. The hypervisor cannot subsequently map a granule to a previously-backed IPA within a PAR without Realm permission.

As a system designed to scale to many cores, RMM makes extensive use of fine-grained locking to support a high degree of concurrent operation. For example, each memory granule has its own lock so many granule operations can be done in parallel. Similarly, an RTT is a multi-level page table, for which each level has its own lock, and hand-over-hand locking is used to support concurrent operations on RTTs, as discussed in Section 4.1. For example, two Realm VCPUs can each cause a stage 2 page fault at the same time but at different IPAs, which can be resolved by the hypervisor in parallel on two CPUs to improve performance. This is a key requirement to support large Realms. Although most of RMM is written in C, Arm assembly code is also used to implement memory accesses with acquire/release semantics where lockless concurrent accesses are used for performance reasons, and to implement the locking primitives themselves.

CCA firmware is designed for security following best practices. Systems such as Linux map all physical memory to the kernel page table. RMM and EL3M do not. RMM's own page table statically maps code and metadata exclusively accessed by RMM, such as the GST and locks for each granule. Additional entries in RMM's page table are used to statically assign a virtual address range to each physical CPU in the system, resulting in a fixed number of virtual address slots per CPU. Memory is then mapped on demand when needed. RMM maps Data granules and metadata granules, such as RD and REC, on demand, and unmaps them once the respective operation is completed. EL3M's own page table only statically maps the EL3M code, a small fixed size stack, and the GPT; no other memory is mapped to its page table. Furthermore, SMC parameters are only interpreted as values in EL3M, never as pointers used to access memory. Even if a bug is introduced in some future version of CCA firmware that is not completely verified, these defense-in-depth measures make it much harder for a return-oriented or jump-oriented programming attack to succeed.

4 VIA Framework

Because CCA relies on firmware to guarantee the security of Realms, we verify that firmware, namely RMM and EL3M. We prove the CCA firmware implementation refines its layered specification in Coq, then use the top-level specification to prove the system's security properties hold for the implementation. To accomplish this, we developed the VIA verification framework, which supports layered verification of CCA firmware. VIA introduces four key verification techniques: mover oracle queries, relaxed memory support via permutation conditions, register accounting for C and assembly code integration, and a new ideal secure system model for proving security properties that cannot be verified using traditional noninterference-based approaches.

4.1 Mover Oracle Queries

To verify RMM, it is essential to simplify reasoning about possible interleavings of executions of concurrent software across multiple CPUs. For example, RMM uses hand-over-hand locking to synchronize access to RTTs, which are 4-level page tables, allowing multiple CPUs to manipulate the same page table concurrently. Figure 2 shows the steps to allocate delegated granules as new level `T1`, `T2`, and `T3` tables of a Realm's RTT using `RTT.Create` and then, in step 4, allocate a delegated granule to the Realm for its data and map its physical address to the leaf-level `T3` table using `RTT.MapProtected`, which would typically occur on a page fault. Figure 2 also shows how step 4 uses hand-over-hand locking, in which RMM first acquires `T0`'s lock so it can lookup and acquire `T1`'s lock and release `T0`'s lock. It can then lookup and acquire `T2`'s lock and release `T1`'s lock, so it can lookup and acquire `T3`'s lock and release `T2`'s lock, and finally update `T3`'s page entry. At the same time, RMM running on other CPUs can do other page table operations, such as acquiring `T0`'s lock to work on a different level 1 table.

To verify the page table operations with hand-over-hand locking, we need to reason about the correctness of all possible interleavings of operations. However, reasoning about all possible interleavings of all operations all at once is too difficult to do for a system as complex as RMM. To address this problem, VIA introduces mover oracle queries, a new mechanism that combines the power of local CPU reasoning with mover types [44], building on previous work on CertiKOS [24–27] and CSPEC [10].

To explain how mover oracle queries work, consider first an explicit multiprocessor machine model, whose machine state consists of per-physical CPU private state (e.g., CPU registers) and a global logical log, a serial list of events generated by all CPUs throughout their execution. Instead of explicitly modeling shared objects, events incrementally convey interactions with shared objects, whose state may be calculated by replaying the logical log. An event is emitted by a CPU and appended to the log whenever that CPU invokes a primitive that interacts with a shared object. Our abstract machine is formalized as a transition system, where each step models some atomic computation taking place on a single CPU; concurrency is realized by the nondeterministic interleaving of steps across all CPUs. However, reasoning

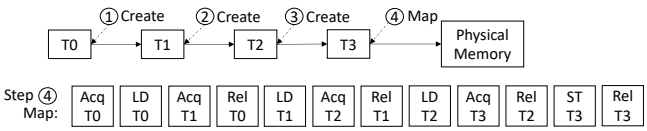


Figure 2: Page table creation and hand-over-hand locking execution.

about interleavings directly with multiple CPUs is difficult.

To simplify reasoning about all possible interleavings, we instead lift multiprocessor execution to a local CPU model, which distinguishes execution taking place on a particular CPU from its concurrent environment [27, 36, 42]. All effects coming from the environment are encapsulated by and conveyed through an *event oracle*, which yields events emitted by other CPUs when queried. Querying the event oracle can be thought of in the context of the explicit multiprocessor machine model as returning events from the global log generated by all other CPUs; only new events since the last query are returned. How the event oracle synchronizes these events is left abstract, its behavior constrained only by rely-guarantee conditions [35]. Since the interleaving of events is left abstract, our proofs do not rely on any particular interleaving of events and therefore hold for all possible concurrent interleavings.

A CPU captures the effects of its concurrent environment by querying the event oracle between local CPU steps. A CPU only needs to query the event oracle when interacting with shared objects, since its private state is not affected by these events. In other words, the CPU repeatedly performs two steps when interacting with shared objects: querying the event oracle to obtain events from other CPUs, then generating a local CPU event. The result is a composite log of events from other CPUs interleaved with events from the local CPU. This is equivalent to the logical log in the explicit multiprocessor model, but without the complexity of directly reasoning about multiple CPUs.

If possible, we would like to move the interleaved event oracle queries out of the way of the local CPU events so we can use sequential reasoning regarding the local execution of any given CPU. By using mover types, we can identify how we can reorder event oracle queries with respect to local CPU events without changing the machine's behavior. Thus, these queries are mover oracle queries. We classify all local CPU events in the composite log as RightMover, LeftMover, or NoneMover. Mover oracle queries can be reordered before a RightMover and after a LeftMover. For example, acquiring a lock is a RightMover because if other CPUs do something after acquiring the lock on the local CPU, they must be able to do the same thing before acquiring the lock. The oracle queries which capture the other CPUs' events can be reordered before acquiring the lock. Mover oracle queries cannot be reordered with a NoneMover. For example, an oracle query followed by a NoneMover then a LeftMover cannot be reordered after the LeftMover.

VIA can then reduce the interleaving of events in the log that need to be considered in two ways, which we refer to as *log refinement*. First, we can reorder oracle queries with local CPU

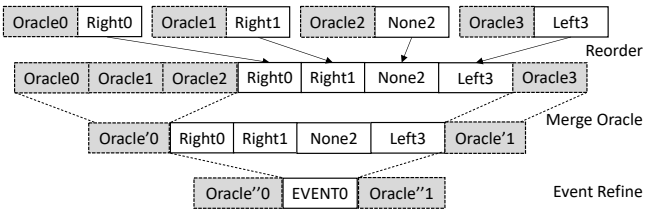


Figure 3: Log refinement with mover oracle queries.

events based on the local events' mover types. By reordering, consecutive oracle queries will be merged to one. Second, we can prove local sequences of events generated by the machine refine an aggregate local event generated by a higher-level machine. This refinement can be applied to any arbitrary CPU, therefore, it applies to all CPUs, so that the entire log of events refines the log of the higher-level aggregate events.

Figure 3 shows an example of log refinement to reduce interleavings of events across CPUs into an atomic event. We identify the mover type of each local event, i.e. [`Right 0`, `Right 1`, `None 2`, `Left 3`], and initially query the oracle before each event. Based on the mover types, we can reorder all oracle queries before the NoneMover to the beginning, and all remaining queries to the end, such that the log before and after reordering have the same machine behavior. We then define a new oracle that can be queried to return the consecutive events from the previous oracle queries [`Oracle 0`, `Oracle 1`, `Oracle 2`], allowing those events to be merged into a single oracle query [`Oracle' 0`]. We then refine the local sequence of events [`Right 0`, `Right 1`, `None 2`, `Left 3`] into a single higher-level aggregate local event `EVENT 0`. This can be done for all CPUs so we can reason further only using the higher-level aggregate event `EVENT 0` with oracle queries `Oracle" 0` and `Oracle" 1` that also return higher-level aggregate events, instead of the many `Left/Right/None` events of lower-level machine.

4.2 Permutation Conditions

To verify RMM, we must account for the relaxed memory behavior of the Arm architecture on code that is not data race free (DRF). For example, Figure 4 shows how a Realm's list of `RECs` is updated in `REC.Create`, `REC.Destroy`, and `Realm.Destroy` without holding a common lock. Each Realm's `RD` has a `RECLIST` (`rd->rec_list`), an array that stores the pointers to all its `RECs`. The `RECLIST` can be referenced from both the Realm's `RD` and each of the Realm's `RECs` (`rec->rec_list`). Each `REC` records its index in the `RECLIST` (`rec->id`). `RD`'s `counter` keeps tracking of how many `RECs` are in a Realm. The hypervisor must destroy all `RECs` of a Realm before destroying its `RD` because once `RD` is destroyed, the Realm can no longer be referenced. Access to the `RECLIST` is not synchronized by its own lock, to avoid potential deadlock issues due to needing to hold multiple locks. Instead, in `REC.Create`, the `RD`'s lock must be held to insert a new `REC` in `RECLIST` to ensure mutual exclusion. However, in `REC.Destroy`, the `REC`'s lock is held instead of the `RD`'s

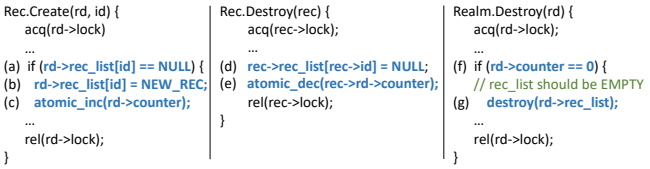


Figure 4: Pseudo code of `RECLIST` data races, marked in bold blue.

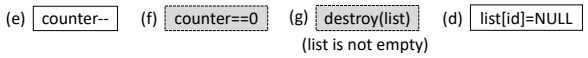
locks when clearing the `REC`'s entry from the `RECLIST` so that multiple CPUs can destroy different `REC`s of the same Realm concurrently. Furthermore, the `RD`'s `counter` is increased or checked in `REC.Create` and `Realm.Destroy` while holding `RD`'s lock, but it is decreased in `REC.Destroy` without holding any lock. As a result, data races can occur when concurrently executing `REC.Destroy` with `REC.Create` or `Realm.Destroy`.

To address this problem, VIA builds on VRM [57]. VRM verifies programs on Arm relaxed memory hardware that are DRF except for synchronization methods and virtual memory hardware. VRM verifies a program on a sequentially consistent (SC) multiprocessor hardware model, defines and proves that a fixed set of conditions hold for the program running on relaxed memory hardware, and proves that the conditions guarantee that the program has the same behavior on SC and relaxed memory hardware so that its SC proofs also hold for relaxed memory hardware.

VIA generalizes this approach for programs that are not DRF. It ensures that such a program will have the same behavior on SC and relaxed memory hardware by first decomposing the program into components that are DRF and not DRF. Previous work already shows that the DRF components will have the same behavior on SC and relaxed memory hardware [57]. VIA then introduces *permutation conditions* $\mathcal{P}$ on the non-DRF components such that $\mathcal{P}$ can be verified to hold for the program on relaxed memory hardware, and $\mathcal{P}$ can be proven to guarantee that the non-DRF components will have the same behavior on SC and relaxed memory hardware. Our experience suggests that even for programs that are not DRF, only a small percentage of the code in these programs is not DRF, so non-DRF programs can be verified on relaxed memory hardware by only proving a small number of permutation conditions in practice. This observation holds for RMM, in which almost all of the code is DRF.

VIA uses VRM's extended Promising Arm model [57] to model Arm's relaxed memory hardware, such that $\mathcal{P}$ needs to be verified against all instruction permutations of the program allowed by VRM's Promising Arm model. Unlike VRM which defines a fixed set of conditions that do not all hold for RMM, VIA allows any condition $\mathcal{P}$ to be specified for non-DRF components that will result in their behavior being in the same on SC and relaxed memory hardware and that can be proven to hold for the program on relaxed memory hardware. The condition is essentially a constraint based on the program's semantics that restricts the possible instruction reorderings that can occur on relaxed memory hardware so that resulting program behavior is the same on SC and relaxed memory hardware.

For example, to handle the non-DRF code in Figure 4, we identify $\mathcal{P}$ to be when `Realm.Destroy` finds `rd->counter` equals 0, `rd->rec_list` must be empty. This is necessary because `rd->rec_list` must be empty when destroying it in `(g)`, otherwise the system may crash due to reclaiming non-empty memory. Since `REC.Create` and `Realm.Destroy` use the same lock, data races can only occur when either runs concurrently with `REC.Destroy`. We prove each function always behaves the same on SC and relaxed memory. For `REC.Create`, since `(b)` and `(c)` cannot be reordered with `(a)` due to the branch dependency, as required by Promising Arm, its possible executions are `(a)(b)(c)` or `(a)(c)(b)`. Since `(a)` confirms that `rec_list[id]` is empty, all concurrent `REC.Destroy` on other CPUs must destroy slots other than `id` because `REC.Destroy` will only work if the `rec` exists, which must be a non-empty slot in the `rec_list`. Therefore, swapping `(b)` and `(c)` will never change any CPU's behavior and `(a)(c)(b)` is equivalent to `(a)(b)(c)`, which is the order on SC. For `REC.Destroy`, if `(e)` executes before `(d)`, $\mathcal{P}$ will be broken because when `Realm.Destroy` checks `counter` concurrently on other CPUs, it may find `counter` is 0 but `rec_list` is not empty, as shown below:



This was actually a real bug in the prototype implementation of RMM. Therefore, we must enforce that `(d)` always executes before `(e)` by adding a barrier between them so it must follow program order as on SC. For `Realm.Destroy`, the proof is trivial because the branch dependency between `(f)` and `(g)` guarantees that they execute in program order as on SC. Therefore, this non-DRF code will not generate more behavior on relaxed memory hardware than on SC.

4.3 Register Accounting

To verify CCA firmware with both C and assembly code, we must account for the interactions of C and assembly code primitives that call one another across language boundaries. However, C code hides the details of how it uses CPU registers, as the use of registers during C code execution is decided by the implementation of specific C compiler used. Although register behavior is not expressed by C language semantics, ignoring it causes problems when attempting to verify programs in which C and assembly code call one another, as shown in Figure 5, which illustrates a real bug in the original prototype RMM implementation detected during our verification. Existing verification approaches cannot support bidirectional calls between C and assembly code, such that the example in Figure 5 would be erroneously verified without detecting the information leakage [10, 11, 23, 26, 37, 42, 43, 46].

To address this problem, VIA introduces a novel register accounting mechanism to correctly verify integrated C and Arm assembly code while making minimal assumptions

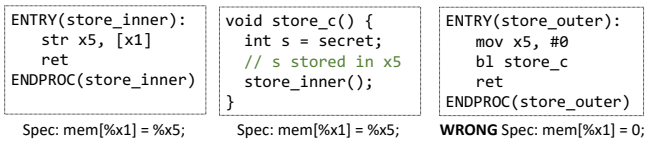


Figure 5: An example of incorrectly combining C and assembly specifications. Assembly function `store_outer` clears register `x5` to 0, then calls C function `store_c`. `store_c` calls assembly function `store_inner`, which stores register `x5` into memory. The intended behavior is that the value 0 will be stored to memory. The actual behavior is that `x5` stores C temporary variable `s` which contains secret data, resulting in undetected information leakage.

regarding compiler behavior. VIA leverages the *Arm64 Procedure Call Standard (AAPCS64)* [7] to specify how registers are potentially used when assembly code calls a C function or is called by a C function. It then conservatively marks all registers used by C code whose values cannot be determined based on AAPCS64 as of `Unknown` value, and requires assembly code to not depend on registers with `Unknown` values.

AAPCS64 constrains how some Arm registers are used. In CCA firmware, C functions pass no more than eight integer or pointer parameters and return an integer or pointer. For such functions, AAPCS64 specifies that a C compiler will only pass parameters through registers `r0-r7` and save the return value in `r0`. It also specifies registers that must have their values preserved through a function call, namely all callee-saved registers `r19-r29` and the stack register `sp`. The use of other general-purpose registers (GPRs) may depend on the specific C compiler implementation.

For an assembly function that calls a C function, VIA checks that the assembly code does not read any `Unknown` registers. Legal assembly code can either keep such `Unknown` registers untouched or overwrite them before using them. VIA uses AAPCS64 to model the register behavior of the C function by identifying register `r0` as containing the return value, and registers `r19-r29` and `sp` as preserving the values. It marks the values of other registers after the C function call as `Unknown`, including caller-saved registers `r1-r18` and the link register `lr`.

For an assembly function that can be called from a C function, VIA checks that its behavior does not depend on `Unknown` registers, and that it obeys AAPCS64 C calling conventions so that it will not cause unexpected behavior in its caller. VIA checks that (1) callee-saved registers `r19-r29` and `sp` preserve the values; (2) the program counter `pc` after the call is equal to `lr` before the call so the assembly primitive returns like a function call; (3) if the caller expects a return value, `r0`'s value is never `Unknown`; and (4) the assembly code behavior remains the same if we initialize all GPRs to `Unknown` except for those carrying parameters. The last condition implies that the assembly code does not read any `Unknown` registers, except for saving and restoring callee-saved registers.

VIA also supports GNU Compiler Collection (GCC) inline assembly extensions within a C function. This is used in inline assembly memory accessors in RMM which guarantee

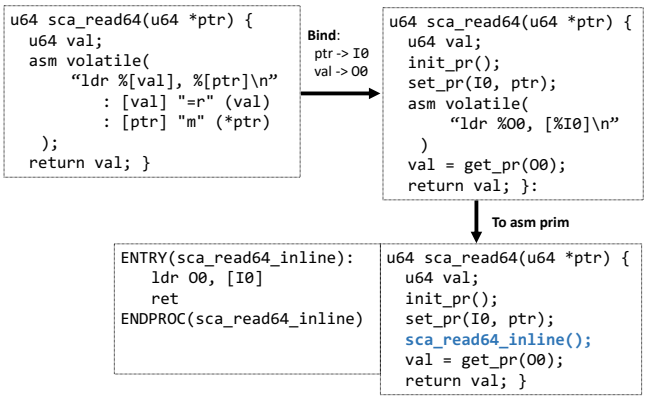


Figure 6: Translation of parameterized inline assembly.

atomicity or memory order semantics, as shown in the `sca_read64` example in Figure 6. `sca_read64` implements a 64-bit single-copy-atomic read in one line of assembly code plus an interface, which can specify a list of input registers, output registers and clobbered registers. VIA translates inline assembly code into an assembly function according to the interface constraints; "r", "Q", and "m" constraints are currently supported. It then checks its correctness like any other assembly function.

Translation is done using a set of logical registers `I0-In` for inputs and `O0-On` for outputs so that verification does not depend on the specifics of GCC register assignment. Input registers are defined read only. VIA also defines abstract accessors `init_pr`, which initializes all logical registers to `UNKNOWN`, `set_pr`, which writes to a register, and `get_pr`, which reads from a register. As shown in Figure 6, the translated `sca_read64` function first calls `init_pr` for initialization, saves parameters to input registers by calling `set_pr`, uses the input and output registers in the assembly code, and gets the return value from the output register by calling `set_pr`.

For simplicity, VIA imposes additional requirements to guarantee GCC generates correct machine code whose behavior is the same as VIA's translated code. VIA forbids inline assembly code from explicitly using any GPRs or goto labels. For inline assembly with multiple instructions, VIA enforces that all output registers are constrained by "&" or "+". Thus, an output-only register never doubles as an input register, and the same register is used for input and output of an operand. This avoids any unexpected overlap in the assignment of input and output registers [53].

Finally, because assembly code functions may be at the interface to outside programs that are untrusted, VIA enforces that all register values are not `Unknown` when returning from those assembly functions. This ensures that there is no unintentional information leakage from assembly code functions to untrusted programs through registers with `Unknown` values.

4.4 Ideal Secure System Model

CCA protects the confidentiality and integrity of Realms' private data during their lifetime. Confidentiality means any

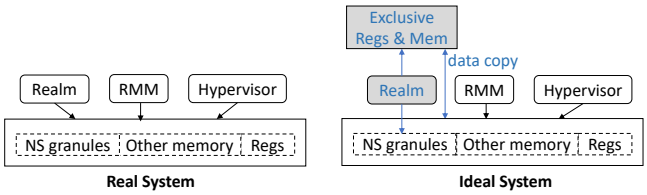


Figure 7: The real and ideal secure system model.

change a Realm makes to its private data is only observable by that Realm. Integrity means a Realm will not observe any changes to its private data that it did not make, but does not imply availability; data access should either fail or return the data previously stored. The confidentiality definition is standard, but the integrity definition allows untrusted software to modify a Realm's private data as long as the Realm does not observe the change. For example, to reclaim memory from Realms, a hypervisor can unmap a Realm's private data without the Realm's permission. This is allowed because the Realm's access to the unmapped data will trigger a page fault so the Realm cannot observe future changes to the data content. However, this breaks noninterference, which therefore cannot be used to to prove security as is done for other verified systems [16, 23, 29, 34, 42, 49, 55].

To address this problem, VIA introduces an ideal/real paradigm, shown in Figure 7, inspired by the idea from formal verification of separation kernels [22, 30]. The *real system* is defined by the RMM top-layer specification, which builds on and incorporates EL3M, in which all memory and CPU registers are shared by Realms, RMM, and the hypervisor. The *ideal system* is defined by an ideal system model specification, in which each Realm has its own exclusive memory, and each REC of the Realm has its own exclusive CPU registers, while other software can only access the same non-exclusive memory and registers as in the real system.

If each Realm only accesses its exclusive memory and registers in the ideal system, we could then show that RMM guarantees confidentiality and integrity by proving that the real system simulates the ideal system. This would mean that each Realm only accesses its exclusive memory and registers in the real system as well, so nothing other than a Realm can access its own data. However, such a simplistic model does not work in practice. For CCA, we need a model that allows declassification so Realms can access NS granules for initialization and I/O, and CPU registers can be used to pass parameters between Realms and RMM, or Realms and the hypervisor.

VIA introduces a new ideal system model for Armv9-A that supports declassification of memory and registers based on a set of well-designed rules that define when declassification is allowed. The model has six declassification rules, listed in Table 3. In this model, Realm exclusive memory consists of all memory in its PAR and exclusive CPU registers consists of all registers accessible by a Realm or that can affect its execution, such as system registers. A Realm will only access its exclusive memory and registers, unless it accesses a granule outside its PAR or it accesses a granule or register that is `Unknown`. If it accesses memory outside its PAR, the Realm will access non-exclusive memory directly. If it accesses a granule or register that is `Unknown`, the data will be copied from a special initialization buffer or non-exclusive register, respectively, before accessing it. A granule is `Unknown` if it is not yet initialized. A register is `Unknown` if it is used by the Realm to communicate with RMM or the hypervisor. For example, when a Realm invokes a hypercall, it exposes the arguments in registers `r0-r6`, which RMM will provide to the hypervisor, then return the results back in those registers. Marking a granule or register as `Unknown` is used to represent declassification in the model.

Type	Rule
Mem	When a Realm accesses an IPA within its PAR but it is `Unknown`, the Realm will copy the data from a special initialization buffer in memory to exclusive memory before accessing the IPA. This can only be done once per granule. The buffer is populated before the Realm is activated, and cannot be changed once it has been activated.
Mem	When a Realm accesses an IPA outside of its PAR, it will directly access memory, not exclusive memory.
Reg	On any trap from a Realm to the RMM, a Realm exposes the contents of various exclusive system registers, marking them `Unknown`, and marks various timer-related exclusive registers `Unknown`.
Reg	If a trap is due to system register emulation, a Realm will mark a specified exclusive GPR as `Unknown`.
Reg	If a trap is due to a hypercall, a Realm will expose and mark the seven exclusive GPRs `r0-r6` used for parameter passing as `Unknown`.
Reg	If a trap is due to an RMM call, a Realm will expose and mark the four exclusive GPRs `r0-r3` used for parameter passing as `Unknown`.

Table 3: Declassification rules.

We can then use this ideal system model with declassification to verify that RMM guarantees Realm confidentiality and integrity. The key is to establish a simulation relation in which all machine states are equivalent between the ideal and real systems and show that, at any step in the two systems satisfying the simulation relation, the same data is obtained when accessing memory or registers. This involves proving a one-to-one mapping of data between the two systems. With declassification, the mapping will change such that a different mapping will be used depending on whether the data is declassified or not. For example, if a granule within a Realm's PAR is not declassified, we will want to show that accessing that granule in non-exclusive memory in the real system correponds to accessing it in exclusive memory in the ideal system to get the same data. On the other hand, if a granule within a Realm's PAR is declassified, because its contents were initialized from an NS granule, we will want to show that first accessing that granule in non-exclusive memory in the real system correponds to accessing it in non-exclusive memory in the ideal system since the respective exclusive memory is initially `Unknown` so the data is first copied from non-exclusive to exclusive memory.

5 CCA Implementation and Verification

We used VIA to verify an early prototype implementation of CCA firmware, which includes both RMM and EL3M as

Description	LOC	Description	LOC
Machine model	1.4K	RMM refinement proofs	6.1K
Lock proof	1.7K	Top-level specification	1.1K
EL3M layer specifications	.2K	Ideal secure system model	.2K
EL3M refinement proofs	.9K	Security simulation proofs	3.4K
RMM layer specifications	4.4K	Permutation condition proofs	1.2K
Total			**20.6K**

Table 4: Lines of Coq code for verifying CCA firmware.

described in Section 3. The verification outcomes, including the discovery of several latent bugs, were confirmed by Arm's development team and used to further improve the firmware implementation. RMM contains 3.2K lines of code (LOC) in C and .3K LOC in assembly. The runtime critical parts of EL3M contain .1K LOC in C and .7K LOC in assembly; all of the C code is for updating the GPT. All RMM and EL3M code is verified, except for the portion of assembly code for initialization (.1K LOC in RMM and .5K LOC in EL3M). For remote attestation, RMM also uses functions provided by a crypto library, which was not verified, though a verified crypto library could be ported and used instead [42, 61].

Table 4 shows our proof effort, measured in LOC in Coq. 45 abstraction layers were used. The bottom layer machine model is based on VRM's Promising Arm model [57] to model Arm's relaxed memory. Another layer was used to verified the spinlock implementation on the relaxed memory model and lift it to an SC model. We verify the EL3M implementation refines its layered specification through three layers. On top of that, we verify the RMM implementation refines its layered specification through 39 layers. The top-level specification reflects RMM's interface, combining both RMM and EL3M functionality. Another layer defines the ideal secure system model. We verify that the top-level specification simulates the ideal secure system model.

5.1 Concurrent Multi-level Page Tables

The most challenging refinement proofs were for verifying RMM's RTT implementation. RTT primitives use hand-over-hand locking to synchronize access to dynamically allocated 4-level page tables, allowing fine-grain concurrent operation on different page table levels. This required nine layers. We leverage mover oracle queries and log refinement, discussed in Section 4.1, to refine all of RMM's page table operations to atomic operations, verifying the correctness of hand-over-hand locking in a real system for the first time.

Figure 8 visualizes the proof. Since acquiring a lock is a RightMover, releasing a lock is a LeftMover, and reading the page table entry is both a LeftMover and RightMover, we can reorder mover oracle queries to refine the procedure of walking the page table until acquiring the lock of `T1` into an atomic step. We group the local CPU events into a single higher-level aggregate "walk until level 1" event. Similarly, we can group events together from creating a level 1 table into a "create level 1 table' event, and destroying a level 1 table

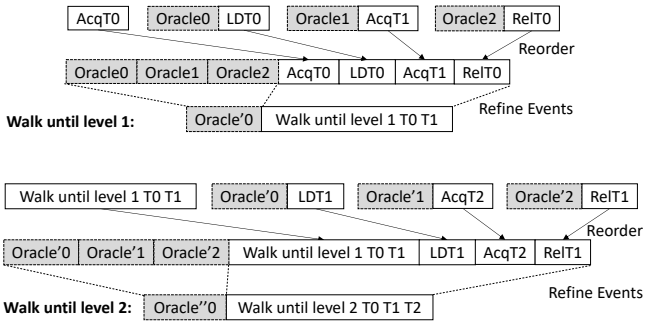


Figure 8: Proving atomicity for page table operations.

into a "destroy level 1 table" event.

We then refine the procedure of walking the page table until acquiring the lock of `T2` into an atomic step. We first prove that "walk until level 1" is a RightMover because any subsequent events at this layer from other CPUs can be reordered with it, i.e., "create level 1 table", "destroy level 1 table", "walk until level 1", and `acq`/`rel`/`LD`/`ST` events for `T2` and `T3` level tables. A "create level 1 table" from other CPUs is irrelevant to the local "walk until level 1" because it can only create other level 1 tables and cannot overwrite `T1` since RMM only allows creating a table that does not exist yet. Events "destroy level 1 table" and "walk until level 1" from other CPUs are irrelevant because they cannot hold `T1`'s lock so can only access other level 1 tables, not `T1`. Other events are also irrelevant because they do not manipulate `T0` and `T1` tables. Therefore, "walk until level 1" is a RightMover and all subsequent mover oracle queries can be reordered before it. Thus, we refine "walk until level 2" into an atomic step, as shown in the bottom of Figure 8. In a similar fashion, we prove "walk until level 2" to be a RightMover and refine the steps of "walk until level 3." Continuing in this manner, we eventually refine all RTT operations into atomic steps.

Proving RTT operations to be atomic allows us to prove desired properties about RMM's RTT management. The key property to prove is that each non-empty entry in the RTTs, including both intermediate entries pointing to lower-level RTTs and leaf mappings, uses a unique delegated granule. This prevents page remapping attacks while still allowing fine-grained access to the RTTs for improved performance. The proof is straightforward because every operation on an RTT entry is proved to be atomic, only the PA of a delegated granule is used to populate a previously empty RTT entry, and each such granule is guaranteed to be unused and zeroed. Once a granule is used for an RTT entry, its state changes from delegated to RTT or Data, preventing it from being used for other RTT entries. By using mover oracle queries and log refinement, we complete the first proof of hand-over-hand locking in a real system, and the first proof of a system with fully dynamically allocated shared page tables.

5.2 Relaxed Memory

We prove permutation conditions as discussed in Section 4.2 to verify the proofs hold on Arm relaxed memory hardware. Veri-

fying CCA firmware only requires six permutation conditions, the `RECLIST` empty condition discussed in Section 4.2, and five conditions previously introduced by VRM, namely (1) NO-BARRIER-MISUSE, (2) TRANSACTIONAL-PAGE-TABLE, (3) SEQUENTIAL-TLB-INVALIDATION, (4) WRITE-ONCE-KERNEL-MAPPING, and (5) MEMORY-ISOLATION. NO-BARRIER-MISUSE requires that barriers are correctly placed. We verified that all lock acquisitions have `acquire` memory semantics and all lock releases have `release` memory semantics. We also proved that memory accesses to shared objects outside critical sections have `release` semantics so that they cannot be reordered, preserving program ordering and SC behavior.

TRANSACTIONAL-PAGE-TABLE requires that shared page table writes within a critical section are transactional. This ensures that page table writes will not result in any behavior on relaxed memory hardware that cannot be produced on an SC model. In RMM and EL3M, each critical section contains at most one page table write, so they are obviously transactional.

SEQUENTIAL-TLB-INVALIDATION requires that a page table unmap or remap be followed by a TLB invalidation, with a barrier between them. This precludes relaxed memory behavior in TLB management code. There are no remaps in RMM or EL3M. We verified that all page table unmaps are followed by a TLB invalidation with a barrier between them.

WRITE-ONCE-KERNEL-MAPPING requires that if RMM or EL3M's own page tables are shared, they can only be written once—only empty page table entries can be modified. This precludes relaxed memory behavior due to out-of-order reads of these page tables. For EL3M, this holds as it uses a statically reserved hardcoded page table shared across all CPUs that is never changed after booting. For RMM, although its kernel page table is shared across all CPUs and can be changed, we prove that it is logically partitioned into two tables, as discussed in Section 3. We prove one table is shared but never changed once initialized, and the other table is not shared because it is statically divided into per-CPU ranges private to each CPU.

MEMORY-ISOLATION requires that the memory space accessible by RMM and EL3M is partially isolated with Realms and NS hypervisors. This ensures that any relaxed memory behavior of Realms or NS hypervisors cannot be propagated to RMM or EL3M. We verify that Realms and the hypervisor will only access `Data` and `NS` granules. Realms' memory accesses are managed by RTTs, We prove RTTs will only map `Data` granules and `NS` granules. A hypervisor's memory accesses are controlled by the GPT. We prove all delegated granules are in the Realm PAS state in the GPT so the hypervisor cannot access them. We further prove that RMM and EL3M behavior do not rely on what Realms or the hypervisor may do with `Data` or `NS` granules. We prove EL3M never accesses memory other than its own, RMM will not access the contents of `Data` granules, and whenever RMM accesses `NS` granules, it may obtain arbitrary data because the hypervisor can make arbitrary changes to the data. Thus, we show RMM's proof on SC does not rely on the concrete implementation of Realms or NS hypervisors.

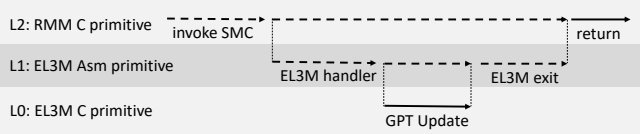


Figure 9: Verify RMM and EL3M GPT update operations. Solid arrows represent C code and dashed arrows represent assembly code.

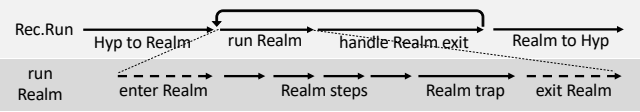


Figure 10: Verify `REC.Run` and its inner `run_realm` loop. Solid arrows represent C code and dashed arrows represent assembly code.

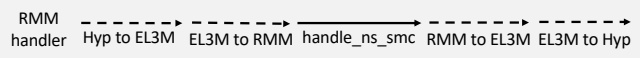


Figure 11: Verify `rmm_handler` in the top layer. Solid arrows represent C code and dashed arrows represent assembly code.

5.3 C and Assembly Code Integration

Another key aspect of the refinement proofs was verifying the interactions between RMM and EL3M, RMM and Realms, and RMM and the hypervisor, which required the C and assembly code integration techniques discussed in Section 4.3. For RMM and EL3M, we verified the correctness of GPT updates. Figure 9 shows how to verify a C primitive in RMM which issues an SMC to EL3M to update the GPT. Layer `L0` verifies the C code for EL3M's GPT operations. Layer `L1` verifies EL3M's assembly code handler, which handles traps from RMM and calls the GPT operations in C. Finally, layer `L2` verifies the C code in RMM that traps to EL3M's assembly code handler.

For RMM and Realms, we verified `REC.Run`, which runs a VCPU of a Realm and required five layers. Figure 10 shows this C primitive, which calls the `run_realm` assembly code primitive, which restores the Realm's VCPU contexts and enters the Realm. We proved that all GPRs are correctly restored such that there is no information leakage from RMM to the Realm through registers with `Unknown` values.

For RMM and the hypervisor, we verified the RMM handling of RMI calls from the hypervisor. Figure 11 shows when the hypervisor invokes an RMI call, it traps to EL3M first, then jumps to RMM and calls the C function `handle_ns_smc` to execute the RMI call. Eventually, RMM returns to EL3M and then the hypervisor. We proved that when returning to the hypervisor, there is no information leakage to the hypervisor through GPRs with `Unknown` values.

5.4 Security

We prove that the real system specified by the RMM top-level specification simulates the ideal system model with declassification, as discussed in Section 4.4. We discuss the simulation relation in three parts: all machine states except for `Data`

granules, CPU registers, and VCPU contexts stored in REC granules (**Rel 1**), Data granules (**Rel 2**), and CPU registers and VCPU contexts (**Rel 3**). Each relation is proved by induction, in which we assume the relation is initial true at machine boot and prove that it is preserved during RMM, hypervisor, and Realm execution so that the same data is obtained when accessing memory or registers in both real and ideal systems.

We prove that **Rel 1** is preserved during execution and all data accessed from memory is the same. **Rel 1** concerns NS granules, delegated granules, and granules containing Realm metadata including RTTs, none of which involve declassification. We prove two invariants: (1) all RTTs only map IPAs within the respective Realm's PAR to Data granules and IPAs outside its PAR to NS granules; and (2) the GPT only labels NS granules in the NS PAS while all delegated granules are labeled in the Realm PAS. The first invariant ensures that Realms will only access Data and NS granules, and the former will not affect **Rel 1**. The second invariant ensures that the hypervisor can only access NS granules. Since Realms and the hypervisor access NS granules in the same non-exclusive memory in both real and ideal systems, they will obtain the same data. All other granules for **Rel 1** can only be accessed by RMM. Since RMM accesses NS and other granules in the same non-exclusive memory in both real and ideal systems, it will obtain the same data; the VCPU contexts that are part of REC granules are excluded here and considered in **Rel 3**.

We prove that **Rel 2** is preserved during execution. The invariant above ensures that the hypervisor cannot access Data granules, and we prove that RMM does not access Data granules, so **Rel 2** is preserved for both the hypervisor and RMM. Data granules are only accessed by Realms. From **Rel 1**, the RTTs must be the same in both real and ideal systems. If an RTT maps an ipa within a Realm's PAR to a Data granule at host physical address hpa, the Realm will access the same data at exclusive memory ipa in the ideal system as at hpa in the real system, so **Rel 2** is preserved. To ensure that an hpa cannot be mapped to ipas in different Realms, we prove an invariant that if an RTT maps ipa to hpa, then the Data granule at hpa inversely maps to (Realm, ipa). Because there is a one-to-one mapping for each Data granule to (Realm, ipa), any changes at hpa can only be observed by the specific Realm at the specific ipa as is the case in the ideal system, so **Rel 2** is preserved for all other data. If an an ipa within a Realm's PAR is Unknown, the Realm will access the same data at non-exclusive memory hpa in the ideal and real system, so **Rel 2** is preserved.

We prove that **Rel 3** is preserved during execution. We prove if a Realm's VCPU V is running, its register r in the real system equals the corresponding exclusive register r if not Unknown or the non-exclusive register r if Unknown in the ideal system. We prove if a Realm's VCPU V is not running, V's REC context of r in the real system equals the corresponding exclusive register r if not Unknown or the V's REC context of r if Unknown in the ideal system. In the ideal system, Realm's register data is always stored in the exclusive registers except for those being declassified. Exclusive registers are not involved in context switches. We then prove that RMM indeed correctly saves and restores Realms' VCPU contexts, so that **Rel 3** is preserved.

Finally, we note that our simulation proofs between the real system and ideal secure system model verify Realm confidentiality and integrity without even trusting the correctness of the RMM or EL3M specifications. The proofs only need to trust the specification of the ideal secure system model, which encodes the declassification rules and consists of only .2K LOC in Coq. Furthermore, as shown in Table 3, the declassification rules only allow a Realm to disclose its data in two ways, by writing NS granules outside of its PAR or via the eight GPRs used for hypercalls, making the security policy formalization easy to understand.

5.5 Bugs Found

We identified several bugs in the CCA firmware prototype implementation during verification. Through refinement proofs, we detected common bugs such as incorrect boundary checking for some variables and misuse of locks; some locks were released without previously holding them. More importantly, verification of C and assembly code integration identified a serious security bug that neither EL3M nor RMM clear the caller-saved registers when returning to the hypervisor. These registers may carry RMM's private execution states and leak information. For example, RMM saves and restores Realms' VCPU contexts, and some contexts may remain in caller-saved registers and leak to the untrusted hypervisor. Another bug identified was in the REC execution handler. The hypervisor provides an NS granule to communicate entry and exit information with RMM. RMM locks and checks that the given granule is indeed an NS granule, accesses its contents, unlocks the granule, and enters the Realm. However, when exiting from the Realm, RMM did not lock and check the granule state before accessing it. This may lead to RMM unexpectedly receiving a Granule Protection Fault (GPF) from the hardware when accessing the granule using the NS PAS, if the granule was delegated by another CPU. This could lead to a denial of service of RMM or have worse consequences if GPF handling was not properly implemented in RMM.

Through permutation condition proofs, we identified an RMM bug that REC.Destroy does not implement "counter−−" with the release semantics (instruction (e) in Figure 4) such that it can be reordered with (d) on Arm's relax hardware. This may cause Realm.Destroy to wrongly set the RECLIST to be reusable before REC.Destroy clears it because when counter is zero, all RECs in the list should have been destroyed, which was not true due to this relaxed memory bug.

Through security proofs, we identified an RMM bug that allows the hypervisor to create two Data granules for the same memory address of a Realm. Thus, RMM can unmap one Data granule from an IPA of a Realm and map another Data granule to the same IPA, violating the Realm integrity guarantee,

because the Realm could observe a change in Realm data not caused by a Realm memory access.

5.6 CCA KVM

CCA provides a standard application binary interface (ABI) to allow hypervisors to communicate their intents to RMM via RMI commands, which is suitable for adoption by commodity hypervisors. However, existing hypervisors do require some modifications to use CCA to support Realm VMs. Regardless of whether a hypervisor is modified to use CCA, it cannot compromise the confidentiality and integrity of Realms. Without modifications, existing hypervisors cannot run Realm VMs, but can still run non-Realm VMs.

We modified the Linux KVM hypervisor to use CCA, which we refer to as CCA KVM. The modifications involved roughly 3K LOC in C to KVM, including .5K LOC for RMI commands, .4K LOC for handling exits from Realms, .8K LOC for creating and destroying Realms, and 1.1K LOC for stage 2 page table management using RMI commands. The modifications also required roughly .5K LOC in C to QEMU, mostly related to VM boot, initialization, and exit handling. Finally, roughly 40 LOC in C of modifications to the virtio driver in the Linux guest kernel were required so that it uses a bounce buffer to communicate I/O data with the hypervisor. This is needed because the ring buffer normally used by the virtio driver in the VM is in memory not accessible to the hypervisor when using Realms. Our experience with KVM indicates that the modifications required for a commodity hypervisor to use CCA are quite modest and involve changes to a very small percentage of its existing codebase.

6 Performance Evaluation

We have run the CCA software stack, including RMM, EL3M, and modifications to the Linux KVM hypervisor to use Realms, on an Arm Fast Model which implements the Realm Management Extensions (RME) CPU architecture. The Fast Model is a valid software emulation of the CPU architecture, allowing us to demonstrate that the CCA software stack provides the desired security guarantees and system functionality. However, Fast Models do not provide any cycle accurate measure of real performance and are too slow to run real application workloads. While CCA will be available in Armv9-A, Armv9-A hardware is not yet available.

To provide a preliminary measure of CCA performance, we have ported the CCA software prototype to run on currently available Arm hardware, an Arm N1 System Development Platform (N1SDP) [5] with an Armv8.2-A Neoverse N1 SoC. This version of EL3M is based on the the Trusted Firmware-A (TFA) codebase. The N1SDP does not provide GPT or Realm world hardware, so it cannot enforce the security guarantees of Realms, but we can use it to mimic the performance costs of Realms by modifying the EL3M code. Context switching between NS and Realm worlds is mimicked by modifying EL3M to switch between two separate contexts within NS world. EL3M is further modified to support the RMI as well as handle GPT update requests from RMM. We did not include EL3M code that controls GPT registers as they do not exist on the N1SDP, but all data written to the GPT memory can be done, although without any effect.

This setup necessarily will have some performance differences from real CCA hardware, but it provides a useful approximation of actual Realm performance. The cost of GPT checks by CCA hardware are not included since no GPT hardware is available, but are expected to exhibit good caching behavior and will not affect the relative performance of VMs versus Realm VMs since they apply equally in NS and Realm worlds. The cost of some hypervisor operations, such as those that require exiting to userspace, will be overly conservative as controlling timer interrupt behavior requires those operations to write to the Arm Generic Interrupt Controller (GIC) on the N1SDP which is slow, whereas real CCA hardware will have system registers that can be used by RMM to achieve the same functionality. Finally, the current prototype lacks support for directly injecting virtual interrupts without hypervisor intervention, which is expected to be available in future CCA hardware.

We ran both microbenchmark and application workloads in VMs on unmodified KVM and CCA KVM in Linux 5.12 on the N1SDP, which has two dual-core 2.6 GHz Neoverse N1 CPUs, 6 GB RAM, a 240 GB SATA3 SSD and a Intel 82574L 1 Gbps NIC. We used QEMU 4.2.0 [8] to run VMs, with the modifications discussed in Section 5.6 to support CCA KVM. VMs were run using KVM or CCA KVM with 4 cores and 1 GB RAM with the VM capped at 2 VCPUs and 512 MB RAM; VCPUs were pinned to individual cores. VHOST networking was used and virtual block storage devices were configured with `cache=none` [28, 38, 56]. Arm VHE [6, 17, 18] was used for all measurements. For client-server workloads, clients ran on an x86 machine with a 16-core Intel Xeon E5-2690 2.9 GHz CPU, 378 GB RAM and an Intel I350 1 Gbps NIC, connected to the N1SDP via a Linksys LGS108 1 Gbps switch.

6.1 Microbenchmarks

We ran KVM unit tests [39], which execute common micro-level hypervisor operations, plus an additional system register access microbenchmark, as listed in Table 5. For each test, we ran it 2^{16} times and report the average latency. Table 6 shows the microbenchmark measurements in nanoseconds for unmodified KVM and CCA KVM. The measurements show that the security benefits of CCA design do come with a performance cost on most micro-level hypervisor operations, because the cost of transitioning between a VM and the hypervisor is much more expensive on CCA KVM than unmodified KVM, which is most clearly shown for Hypercall.

Hypercall simply traps from the VM to the hypervisor in EL2 and returns for KVM, but involves additional operations

Name	Description
Hypercall	Trap from a VM to the hypervisor and return to the VM immediately. Measures base transition cost of hypervisor operations.
I/O Kernel	Trap from a VM to the emulated interrupt controller in the host OS kernel and return to the VM. Measures cost of accessing I/O devices supported in kernel space.
I/O User	Trap from a VM to read the device ID of virtio mmio device then return to the VM. Measures base cost of operations that access I/O devices emulated in user space.
Virtual IPI	Issue virtual IPI to another VCPU on a different CPU. Measures time from sending virtual IPI until receiving VCPU handles it.
Sysreg	Trap from a VM to emulate access to system register `ID_AA64PFR0_EL1` in the hypervisor and return to the VM. Measures system register access cost.

Table 5: Microbenchmarks.

for CCA KVM: (1) trap from VM in EL1 to RMM in EL2; (2) map NS granule to copy exit info to NS world, unmap granule; (3) trap from RMM to EL3M in EL3; (4) save Realm context, restore NS context; (5) exception return from EL3M to hypervisor in EL2; (6) trap from hypervisor to EL3M in EL3; (7) save NS context, restore Realm context; (8) exception return from EL3M to RMM in EL2; (9) map NS granule to copy entry info from NS world, unmap granule; (10) map and read data in REC and RD granules, unmap granules; (11) exception return from RMM to VM in EL1. The additional operations result in Hypercall costing an additional 1.5 μs on CCA KVM than vanilla KVM. Roundtrip transitions between RMM and the hypervisor take roughly 700 ns, and roundtrip transitions between the VM and RMM take roughly 60 ns. Saving and restoring system registers when transitioning between the VM and RMM takes roughly 200 ns per transition, or 400 ns total. The four map/unmap operations take roughly 100 ns each, 400 ns total. The remaining roughly 250 ns is due to other bookkeeping code, including saving and restoring GPRs and error checking.

I/O Kernel and I/O User include the same transition from the VM to the hypervisor and back as the Hypercall, so they also require more than 1.5 μs to execute on CCA KVM than vanilla KVM. Although the difference between CCA KVM and vanilla KVM is roughly 1.5 μs for I/O Kernel, the difference for I/O User is roughly 2.3 μs. This is because on the N1SDP, CCA KVM must write to the GIC when going to userspace, which is quite slow and takes an extra 800 ns.

Virtual IPI is more expensive on CCA KVM versus vanilla KVM because it involves multiple transitions between a VM and the hypervisor. Sending the virtual IPI involves the source vCPU writing to a system register, causing a trap to the RMM, which forwards the operation to the hypervisor (1). The hypervisor issues a physical IPI to the CPU running the destination vCPU, then returns to the source vCPU (2). The physical IPI causes an exit from the destination vCPU (3). On taking this exit, the hypervisor detects that there is a pending virtual IPI, and returns to the destination vCPU (4). Of these four transitions, approximately two occur in parallel, so the cost is roughly twice that of a Hypercall on CCA KVM for the transitions, plus the cost of the actual operation. Because Hypercall is much faster for unmodified KVM, its Virtual IPI cost is not dominated by the transition cost between VM and hypervisor.

Benchmark	Hypercall	I/O Kernel	I/O User	Virtual IPI	Sysreg
KVM	362	549	1,761	1,806	437
CCA KVM	1,865	2,060	4,049	4,324	70

Table 6: Microbenchmark performance (ns).

The one microbenchmark that is much faster on CCA KVM than KVM is Sysreg. Accessing system registers is roughly 5 times as expensive on KVM versus CCA KVM. On CCA KVM, RMM handles this register access directly without returning to the hypervisor. RMM's system register trap handling mechanism is simpler than KVM's because it does not need to support KVM's more general hypervisor functionality that requires synchronizing accesses to hypervisor-related data structures and additional conditional checks.

6.2 Application Benchmarks

We next ran the application benchmarks listed in Table 7 to measure performance on more realistic workloads. We also ran the workloads on native hardware running the same kernel to provide a baseline for comparison, restricting the system to use 2 CPUs and 512 MB RAM to provide a comparable configuration to the VMs. For each platform, we ran each workload 50 times and measured the average, worst, and best performance.

Figure 12 shows the average performance for each benchmark for unmodified KVM versus CCA KVM, with error bars indicating worst and best performance. Performance was normalized to average native execution on the N1SDP hardware; lower is better. Unlike microbenchmark performance, the application benchmark performance shows that CCA KVM and KVM have much more modest performance differences on more realistic workloads.

CCA KVM has less than 8% overhead versus unmodified KVM for most workloads, but in the worst case, overhead was 18% for MongoDB, an I/O intensive workload. The I/O intensive workloads have higher overhead for a couple reasons. The main reason is because the VM exits more frequently, so the cost of exits has a more significant impact on performance. Exits are more expensive on CCA KVM as shown by the Hypercall microbenchmark results in Table 6, in which an exit to the hypervisor costs an extra 1.5 μs. If there are many exits as will be case for I/O intensive workloads, this additional cost can become significant. For example, Memcached incurs roughly a million VM exits to the hypervisor. This results in roughly 1.5 s of additional overhead, or .75 s of overhead per core if the exits are split evenly across cores for a VM with 2 VCPUs. Memcached takes 9 s to run on vanilla KVM, so this is 8% overhead due to the extra latency for exits on CCA KVM, which roughly matches the actual overhead measured for Memcached on CCA KVM versus vanilla KVM.

A secondary reason is because CCA KVM needs to use a bounce buffer while vanilla KVM does not. CCA KVM needs a bounce buffer to support virtio because Realm memory is protected from the hypervisor. KVM uses the default virtio

Name	Description
Apache	Apache server v2.4.41 handling 100 concurrent requests via TLS/SSL from remote ApacheBench [1] v2.3 client, serving the index.html of the GCC 7.5.0 manual.
Hackbench	Hackbench [54] using Unix domain sockets and 20 process groups running in 500 loops.
Kernbench	Compilation of the Linux kernel v4.18 using allnoconfig for Arm with GCC 9.3.0.
Memcached	Memcached v1.5.22 handling requests from a remote memtier [51] v1.2.11 client with default parameters.
MongoDB	MongoDB server v3.6.8 handling requests from a remote YCSB [14] v0.17.0 client running workload A with 16 concurrent threads and operationcount=500000.
MySQL	MySQL v8.0.27 running sysbench v1.0.11 with 32 concurrent threads and TLS encryption.
Redis	Redis v4.0.9 server handling requests from a remote redis-benchmark client (redis-tools v5.0.7) [52] running `GET/SET` with 50 parallel connections and 12 pipelined requests.

Table 7: Application benchmarks.

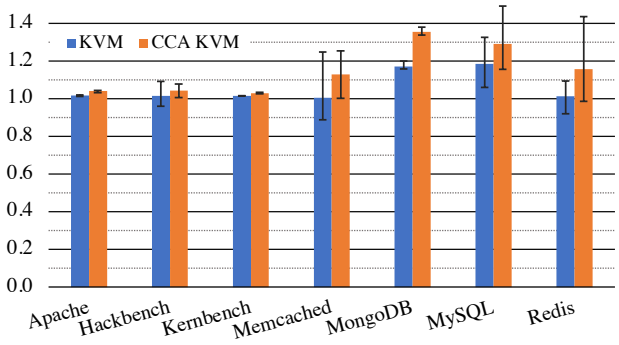


Figure 12: Application benchmark performance.

mechanism to directly access VM memory, so it does not require bounce buffers and does not need to perform the additional data copying. Since KVM can also be configured to use a bounce buffer, we also measured KVM with this configuration to isolate the impact of using a bounce buffer on performance. The overhead with versus without a bounce buffer was negligible in most cases, but in the worst case as high as 3-4% for the more disk I/O intensive workloads, MongoDB and MySQL.

We expect the overheads for I/O intensive workloads on real CCA hardware to be less than what we measured on the N1SDP hardware. Exits are expected to occur less frequently on real CCA hardware when support for direct virtual interrupt injection is added. Exits that go to userspace are expected to cost less on real CCA hardware as the expensive GIC writes required for N1SDP hardware will be eliminated, though this was not a dominant factor in our results with the use of VHOST networking. This cost can be further mitigated by using device passthrough instead of paravirtual I/O, which will largely avoid these exits and their associated performance overhead. Support for Realm device passthrough will be added to future CCA hardware. Overall, our measurements indicate that CCA's security guarantees can be delivered with acceptable performance overheads for real application workloads.

7 Related Work

Hardware-enforced trusted execution environments have become an important feature of major computer architectures. Arm TrustZone [4] can be used to statically partition and isolate a memory region in Secure world, but most implementations only support a small number of such memory regions, limiting its scalability. Intel Software Guard Extensions (SGX) [33] can be used by application developers to protect userspace memory from other programs, including a potentially malicious OS or hypervisor. SGX is not suitable for securing VMs.

AMD Secure Encrypted Virtualization (SEV) [2] and Intel Trust Domain Extensions (TDX) [32] provide protection at the level of VMs with similar threat models to CCA. The initial version of SEV ensured confidentiality by encrypting VM memory at runtime, but did not ensure memory data integrity, which has been utilized as an attack vector such that a compromised hypervisor can tamper with or steal private VM data [31,40,47,48, 60]. Secure Nested Paging (SNP) [3] now provides the previously missing integrity protection capability. SEV-SNP allows an untrusted hypervisor to directly manage NPTs, but checks accesses against a reverse map table, an additional data structure managed by a security co-processor. In contrast, Intel TDX runs a TDX module in a privileged SEAM (Secure-Arbitration Mode) root CPU mode. The firmware manages NPTs used by protected VMs in response to requests issued by the untrusted hypervisor. Unlike CCA, the security of SGX, SEV, SEV-SNP and TDX relies on complex implementations in unverified microcode and firmware [12, 15]. They are difficult to update, either to patch security flaws or introduce new features.

Komodo [23] draws on ideas from SGX, but is implemented as a software monitor in verified Arm assembly code on top of TrustZone instead of requiring hardware to support complex enclave-manipulation instructions. This avoids hardware complexity and enables deployment of new enclave features independently of CPU upgrades. Komodo does not support multiprocessor execution, largely due to the challenge of verifying low-level concurrent code. CCA retains the advantages of Komodo's approach by relying on a verified software monitor to implement Realms, but supports verified VM protection and multiprocessor execution.

The idea of retrofitting a commodity hypervisor so that its security guarantees are enforced by a small trusted core was first explored by SeKVM [41–43, 57]. SeKVM was the first to show how this retrofitting approach, known as microverification, makes it possible to verify that a commodity hypervisor guarantees the confidentiality and integrity of VMs. CCA allows hypervisors to be modified to support Realm VMs, whose confidentiality and integrity are protected by a verified monitor, reminscient of SeKVM. While SeKVM uses existing Arm hardware, CCA introduces new hardware mechanisms that protect VMs from untrusted software running in both NS and Secure world, and allow hypervisors to make full use of Arm virtualization features such as VHE for better

performance. Furthermore, CCA firmware is designed to support a higher degree of scalability and concurrent operation by allowing data races, leveraging fine-grain synchronization, and enabling the hypervisor to provide fully dynamic memory allocation for all VM-related metadata.

While verifying CCA firmware required new VIA verification techniques, many of them build on previous work. Various concurrent systems have been verified, including CertiKOS [26, 27, 45], SeKVM, and CMAIL using CSPEC [10]. CertiKOS and SeKVM support sequential reasoning with a local CPU model and encapsulate other CPUs' behavior by rely/guarantee conditions, but do not support reordering using mover types, making proving hand-over-hand locking infeasible. Although hand-over-hand locking can theoretically be proved using rely/guarantee reasoning [58], the approach is not machine-checkable or scalable to a real system like RMM. CSPEC provides proof patterns with mover types, but lacks a local CPU model and does not verify C code; it offers little help for RMM code not reducible by movers (e.g. `REC.Destroy` in Figure 4) that still need rely/guarantee reasoning to verify. VIA builds on CertiKOS, SeKVM, and CSPEC to combine a local CPU model with mover types.

Some programs have been previously verified on relaxed memory hardware. Armada [46] supports verifying programs on the x86-TSO memory model, but their approach of verifying the entire program on a relaxed memory model has not been shown to scale to real systems such as RMM. VRM [57] instead allows proofs on an SC model to hold on relaxed memory hardware by ensuring certain conditions hold, making possible the verification of SeKVM, the first machine-checked proof for concurrent systems software on Arm relaxed memory hardware. VIA generalizes VRM to arbitrary non-DRF programs.

Verifying programs with both C and assembly code has been done to varying degrees, but none support bidirectional calls between them. seL4 [37] verifies C code, but its assembly code is unverified. CertiKOS relies on a verified x86 C compiler to verify assembly primitives invoking C primitives by compiling the invoked C primitives into assembly primitives, but cannot verify C primitives that invoke assembly primitives. Since no verified Arm C compiler exists, this approach cannot be used for CCA. SeKVM verifies C and Arm assembly code separately, but does not link the proofs, in part because no verified Arm C compiler exists. Komodo is written entirely in assembly code which is then verified, but this is difficult to scale to a large system as it is hard to write and maintain a large codebase in assembly. Ironclad [29] conducts verification at the assembly level by compiling programs in a high-level language down to assembly. This is also difficult to scale as it is harder to verify the much larger generated assembly code than the original high-level language implementation. VIA allows most proofs to be done at the C level while verifying interactions between C and assembly code are safe.

Noninterference has been frequently used to prove information-flow security [16, 23, 29, 34, 42, 49, 55], but cannot be applied to RMM given the definition of data integrity and confidentiality supported by Realms. While most of these approaches rely on some static partitioning of memory to simplify their noninterference proofs, RMM imposes no such scalability limitations. The ideal/real simulation paradigm has been used to verify information-flow security of a simple 750 LOC two-user uniprocessor separation kernel without page tables [22], but we show for the first time how it can be applied in the presence of declassification to verify data confidentiality and integrity of a real system that supports modern multiprocessor and MMU hardware with page tables.

8 Conclusions

Arm CCA is the first confidential compute architecture backed by verified firmware that is correct and secure. CCA introduces Realms, secure execution environments that protect the confidentiality and integrity of VMs against untrusted system software such as hypervisors. Realms are made possible by hardware support for Realm world, a new physical address space for Realms inaccessible to untrusted system software, and a firmware monitor that runs in Realm world to control CCA hardware to secure and manage Realms, including handling requests from untrusted hypervisors to create Realms, run Realms, and allocate memory to Realms. This design maintains compatibility with the Arm architecture without introducing complex hardware mechanisms by relying on firmware, and avoids complexity in the firmware by relying on existing hypervisors to provide virtualization functionality.

We formally verified CCA firmware, demonstrating the feasibility of relying on trustworthy firmware for the security guarantees of the architecture. We introduced various verification techniques to make it possible to verify for the first time concurrent firmware with data races running on relaxed memory hardware, fine-grain synchronization such as hand-over-hand locking, dynamically allocated shared multi-level page tables, and integrated C and assembly code. We also prove the security guarantees despite untrusted software being in full control of resource allocation decisions. The proof only needs to trust roughly two hundred lines of Coq specification, making the formal security guarantees easy to read and understand. CCA provides its security guarantees with only modest performance overhead compared to running VMs with the Linux KVM hypervisor without verified VM protection.

9 Acknowledgments

Andrew Baumann and Charles Garcia-Tobin provided helpful comments on earlier drafts. This work was supported in part by Arm, OPPO, an Amazon Research Award, a Guggenheim Fellowship, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080. Ronghui Gu is the Founder of and has an equity interest in CertiK.

References

[1] ab, The Apache Software Foundation. http://httpd.apache.org/docs/2.4/programs/ab.html, April 2015.

[2] Advanced Micro Devices. Secure Encrypted Virtualization API Version 0.16. https://support.amd.com/TechDocs/55766_SEV-KM%20API_Spec.pdf, February 2018.

[3] Advanced Micro Devices. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, January 2020.

[4] ARM Ltd. ARM Security Technology Building a Secure System using TrustZone Technology. https://documentation-service.arm.com/static/5f212796500e883ab8e74531, April 2009.

[5] ARM Ltd. Arm Neoverse N1 Core Technical Reference Manual. https://developer.arm.com/documentation/100616/0400/, April 2019.

[6] ARM Ltd. Virtualization Host Extensions. https://developer.arm.com/documentation/102142/0100/Virtualization-Host-Extensions, January 2019.

[7] ARM Ltd. Procedure Call Standard for the Arm® 64-bit Architecture (AArch64). https://github.com/ARM-software/abi-aa/releases/download/2022Q1/aapcs64.pdf, April 2022.

[8] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track (FREENIX 2005)*, pages 41–46, Anaheim, CA, April 2005.

[9] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, February 2017.

[10] Tej Chajed, M. Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 306–322, Carlsbad, CA, October 2018.

[11] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 243–258, Huntsville, ON Canada, October 2019.

[12] Anrin Chakrabortid, Reza Curtmola, Jonathan Katz, Jason Nieh, Ahmad-Reza Sadeghi, Radu Sion, and Yinqian Zhang. Cloud Computing Security: Foundations and Research Directions. *Foundations and Trends in Privacy and Security*, 3(2):103–213, February 2022.

[13] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 431–447, Santa Barbara, CA, June 2016.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*, pages 143–154, Indianapolis, IN, June 2010.

[15] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, January 2016. https://ia.cr/2016/086.

[16] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 648–664, Santa Barbara, CA, June 2016.

[17] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.

[18] Christoffer Dall, Shih-Wei Li, and Jason Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 221–234, Santa Clara, CA, July 2017.

[19] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences Building the Linux ARM Hypervisor. Technical Report CUCS-010-13, Department of Computer Science, Columbia University, June 2013.

[20] Christoffer Dall and Jason Nieh. Supporting KVM on the ARM Architecture. *LWN Weekly Edition*, pages 18–22, July 2013.

[21] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, March 2014.

[22] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS 2013)*, pages 223–234, Berlin, Germany, November 2013.

[23] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*, pages 287–305, Shanghai, China, October 2017.

[24] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, and Haozhong Zhang. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL 2015)*, pages 595–608, Mumbai, India, January 2015.

[25] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Wu, Vilhelm Sjöberg, and David Costanzo. Building Certified Concurrent OS Kernels. *Communications of the ACM*, 62(10):89–99, September 2019.

[26] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, pages 653–669, Savannah, GA, November 2016.

[27] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 646–661, Philadelphia, PA, June 2018.

[28] Stefan Hajnoczi. An Updated Overview of the QEMU Storage Stack. In *LinuxCon Japan 2011*, Yokohama, Japan, June 2011.

[29] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 165–181, Broomfield, CO, October 2014.

[30] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. Formal Specification and Verification of Data Separation in a Separation Kernel for an Embedded System. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 346–355, Alexandria, Virginia, October 2006.

[31] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2017)*, pages 129–142, Xi'an, China, April 2017.

[32] Intel Corporation. Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, October 2014.

[33] Intel Corporation. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, May 2021.

[34] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing Browser Security Guarantees through Formal Shim Verification. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, pages 113–128, Bellevue, WA, August 2012.

[35] C. B. Jones. Tentative Steps toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, October 1983.

[36] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and Liveness of MCS Lock—Layer by Layer. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS 2017)*, pages 273–297, Suzhou, China, November 2017.

[37] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220, Big Sky, MT, October 2009.

[38] KVM contributors. Tuning KVM. http://www.linux-kvm.org/page/Tuning_KVM, May 2015.

[39] KVM contributors. KVM Unit Tests. http://www.linux-kvm.org/page/KVM-unit-tests, August 2020.

[40] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1257–1272, Santa Clara, CA, August 2019.

[41] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1357–1374, Santa Clara, CA, August 2019.

[42] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, pages 1782–1799, San Francisco, CA, May 2021.

[43] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, pages 3953–3970, Vancouver, BC Canada, August 2021.

[44] Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[45] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, December 2019.

[46] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*, pages 197–210, London, UK, June 2020.

[47] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting Secrets from Encrypted Virtual Machines. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY 2019)*, pages 221–230, Dallas, TX, March 2019.

[48] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec 2018)*, pages 1–6, Porto, Portugal, April 2018.

[49] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (IEEE S&P 2013)*, pages 415–429, San Francisco, CA, May 2013.

[50] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 225–242, Huntsville, ON Canada, October 2019.

[51] Redis Labs. Memtier Benchmark. https://github.com/RedisLabs/memtier_benchmark, January 2018.

[52] Redis Labs. Redis Benchmark. https://redis.io/docs/reference/optimization/benchmarks/, March 2022.

[53] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc-12.1.0/gcc.pdf, May 2022.

[54] Rusty Russell. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, January 2008.

[55] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 287–305, Carlsbad, CA, October 2018.

[56] SUSE. Performance Implications of Cache Modes. https://www.suse.com/documentation/sles11/book_kvm/data/sect1_3_chapter_book_kvm.html, September 2016.

[57] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP 2021)*, pages 866–881, Virtual Event, Germany, October 2021.

[58] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving Correctness of Highly-Concurrent Linearisable Objects. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006)*, pages 129–136, New York, NY, March 2006.

[59] Alexander Van't Hof and Jason Nieh. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, Carlsbad, CA, July 2022.

[60] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (IEEE S&P 2020)*, pages 1483–1496, San Francisco, CA, May 2020.

[61] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1789–1806, Dallas, TX, October 2017.

[62] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 259–274, Huntsville, ON Canada, October 2019.