Enabling Realms with 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

The Armv9 architecture will have a new secure execution environment, called Realms. Realms includes minimal hardware changes, relying on firmware for most of its features, so that existing applications can run unchanged yet be secured. In this article, we explain how Realms works, how we used Coq to prove the correctness of the firmware, run benchmarks to compare performance using Realms, and contrast Realms with existing secure enclaves.

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, the *Arm Confidential Compute Architecture (Arm CCA)* introduces *Realms*, secure execution environments that are completely opaque to privileged, untrusted system software such as OSes and hypervisors. Arm 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, Arm CCA allows a hypervisor to dynamically allocate memory to or free memory from a Realm VM, but disallows it from accessing the protected memory contents of a Realm VM. Arm 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.

Arm 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, Arm CCA relies on an Arm architecture feature called the *Realm Management Extension (RME)*. RME 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. Arm CCA introduces the *Realm Management Monitor (RMM)*, a new firmware component which runs in Realm world at a higher privilege level than Realms. Untrusted system software such as a hypervisor running in NS world can 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.

We implemented, evaluated, and verified an early prototype of Arm CCA firmware. Although RME hardware is not yet available, we demonstrated Arm CCA on a functionally accurate Arm Fast Model with Arm CCA support. We modified the Linux KVM hypervisor [15–17] to use RMM interfaces to manage Realm VMs, and ran various VM workloads on the model. We also ported Arm CCA firmware to current Arm hardware to obtain preliminary data on Arm CCA performance, which shows that KVM on Arm CCA incurs modest overhead versus vanilla KVM on real application workloads.

Because CCA relies on firmware to guarantee the security of Realms, we verify that firmware using the Coq proof assistant [37]. We verified the correctness of both the C and Arm assembly Arm CCA firmware implementation, including RMM, proving its implementation refines its specification. We proved the specification has equivalent behavior to an idealized secure machine model to verify the confidentiality and integrity guarantees of Realms. This is the first proof of the security guarantees of a confidential computing architecture. RME is an optional feature of the Arm A-Profile architecture as of Armv9.3-A, and CCA firmware is open source.

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 article. 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 Arm 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 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.

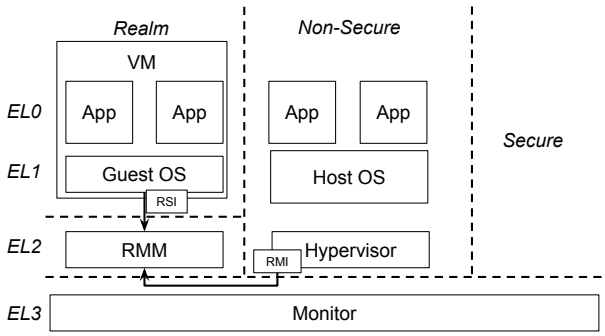

Figure 1: Arm Confidential Compute Architecture.

Arm 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 Arm CCA firmware is of crucial importance.

Figure 1 shows how Arm CCA extends the Arm architecture. Armv8-A provided two statically partitioned worlds, NS world used by most software stacks and Secure world to host platform security services, with an orthogonal Monitor Mode at EL3 [5]. RME 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. RME 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, RME 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.

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: RME access control policy. The entity accessing a granule belongs to a security state, while the Physical Address Space(PAS) is a property only of the granule being accessed.

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. RME 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. RME 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.

Arm 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 Arm 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 Arm CCA, the Realm isolation boundary is at the level of entire VMs. This solution can be used to secure containers, but within virtual machines, as demonstrated by the Confidential Containers project (Coco) [40]. Application level containerisation is not available in the current version of Arm CCA but might be considered in future work [38]. 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

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).

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)*.

The IPA space of a Realm includes a *Protected Address Range (PAR)*, 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 within the PAR 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 Arm 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. 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.

Arm 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 Arm 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 Verification and Evaluation

We have implemented, verified, and evaluated an early Arm CCA prototype [29]. Verification was done in Coq and used to guarantee the correctness of the firmware and the security of Realms. We proved 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. 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. Verifying the RTT implementation required us to develop new verification techniques to verify the correctness of hand-over-hand locking in a real system for the first time. To verify security properties even though untrusted system software is in full control of system resources, we proved the specification simulates an idealized secure machine model whose definition forbids behaviors that compromise Realm confidentiality or integrity. A key feature of the proof is that it only needs to trust roughly 200 lines of Coq specification, making the formal guarantees of the confidentiality and integrity of Realm code and data in use easy to read and understand. 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. The implementation has since been open sourced and continues to evolve [4], and formal methods continue to be applied to its development [19].

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. However, Fast Models do not provide any cycle accurate measure of real performance. To provide a preliminary measure of Arm CCA performance even though Armv9-A hardware with RME support is not yet available, we have ported the Arm CCA software prototype to run on currently available Arm hardware, an Arm N1 System Development Platform (N1SDP) [6] 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 data is still written to the GPT, although without any effect.

This setup necessarily will have some performance differences from real RME hardware, but it provides a useful approximation of actual Realm performance. The cost of GPT checks by RME 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 RME hardware will have system registers that can be used by RMM to achieve the same functionality. Finally, the prototype evaluated for this article lacks support for directly injecting virtual interrupts without hypervisor intervention, but this is expected to be available in future RME hardware.

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 [34] 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 [32] v1.2.11 client with default parameters.
MongoDB	MongoDB server v3.6.8 handling requests from a remote YCSB [11] 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) [33] running `GET/SET` with 50 parallel connections and 12 pipelined requests.

Table 3: Application benchmarks.

We ran 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. 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` [20, 24, 35]. Arm VHE [7, 13, 14] 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.

Table 3 lists the application workloads we ran. 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 2 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.

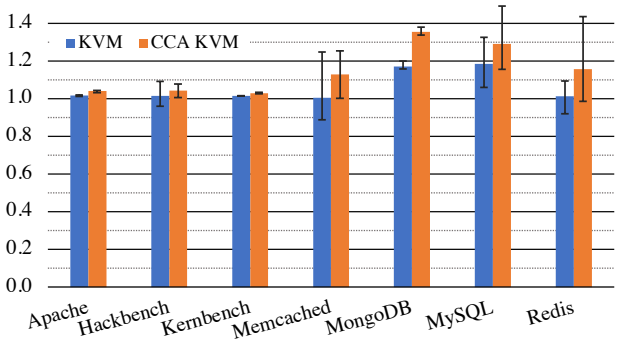

Figure 2: Application benchmark performance.

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. An exit to the hypervisor is more expensive on CCA KVM, taking 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 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 RME hardware to be less than what we measured on the N1SDP hardware. Exits are expected to occur less frequently on real RME hardware when support for direct virtual interrupt injection is added. Exits that go to userspace are expected to cost less on real RME 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 a future version of Arm CCA. Overall, our measurements indicate that Arm CCA's security guarantees can be delivered with acceptable performance overheads for real application workloads.

5 Related Work

Hardware-enforced trusted execution environments have become an important feature of major computer architectures. Arm TrustZone [5] 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) [23] 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) [22] provide protection at the level of VMs with similar threat models to Arm 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 [21, 25, 30, 31, 39]. 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 coprocessor. 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 Arm CCA, the security of SGX, SEV, SEV-SNP and TDX relies on complex implementations in unverified microcode and firmware [10, 12]. They are difficult to update, either to patch security flaws or introduce new features.

Komodo [18] 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. Arm 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 [26–28, 36]. 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. Arm 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, RME 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, Arm 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.

6 Conclusions

Arm 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 RME 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 Arm CCA firmware, demonstrating the feasibility of relying on trustworthy firmware for the security guarantees of the architecture. Arm CCA provides its security guarantees with only modest performance overhead compared to running VMs with the Linux KVM hypervisor without verified VM protection.

7 Acknowledgments

Charles Garcia-Tobin and Mark Knight provided helpful feedback 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. TF-RMM: the Trusted Firmware Implementation of the Realm Management Monitor (RMM). https://www.trustedfirmware.org/projects/tf-rmm/.

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

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

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

[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] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] Anthony C. J. Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P. Mulligan, Gustavo Petri, and Nathan Chong. A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023.

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

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 465–484, Carlsbad, CA, July 2022.

[30] 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.

[31] 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.

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

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

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

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

[36] 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.

[37] The Coq development team. The Coq Proof Assistant. http://coq.inria.fr. Accessed on December 13, 2022.

[38] 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.

[39] 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.

[40] Willen Yang, Arm Ltd. Confidential Containers (Coco) on Arm CCA. https://linaroconnect2023.sched.com/event/1K86S/lhr23-315-confidential-containerscoco-on-arm-cca, April 2023.