A Verification Methodology for the Arm® Confidential Computing Architecture

From a Secure Specification to Safe Implementations

ANTHONY C. J. FOX, GARETH STOCKWELL, and SHALE XIONG, Arm, United Kingdom
HANNO BECKER, DOMINIC P. MULLIGAN, and GUSTAVO PETRI*, Amazon Web Services, United Kingdom
NATHAN CHONG, Amazon Web Services, United States

We present Arm's efforts in verifying the specification and prototype reference implementation of the Realm Management Monitor (RMM), an essential firmware component of Arm Confidential Computing Architecture (Arm CCA), the recently-announced Confidential Computing technologies incorporated in the Armv9-A architecture. Arm CCA introduced the Realm Management Extension (RME), an architectural extension for Armv9-A, and a technology that will eventually be deployed in hundreds of millions of devices. Given the security-critical nature of the RMM, and its taxing threat model, we use a combination of interactive theorem proving, model checking, and concurrency-aware testing to validate and verify security and safety properties of both the specification and a prototype implementation of the RMM. Crucially, our verification efforts were, and are still being, developed and refined contemporaneously with active development of both specification and implementation, and have been adopted by Arm's product teams.

We describe our major achievements, realized through the application of formal techniques, as well as challenges that remain for future work. We believe that the work reported in this paper is the most thorough application of formal techniques to the design and implementation of any current commercially-viable Confidential Computing implementation, setting a new high-water mark for work in this area.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Software verification**; **Model checking**; **Dynamic analysis**.

Additional Key Words and Phrases: Arm Confidential Computing Architecture (Arm CCA), formal methods, separation kernel, Confidential Computing, operating system verification

ACM Reference Format:
Anthony C. J. Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P. Mulligan, Gustavo Petri, and Nathan Chong. 2023. A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 88 (April 2023), 30 pages. https://doi.org/10.1145/3586040

*All work by Becker, Chong, Mulligan, and Petri carried out whilst at Arm.

Authors' addresses: Anthony C. J. Fox, anthony.fox@arm.com; Gareth Stockwell, gareth.stockwell@arm.com; Shale Xiong, shale.xiong@arm.com, Architecture and Technology Group, Arm, United Kingdom; Hanno Becker, beckphan@amazon.co.uk; Dominic P. Mulligan, dommul@amazon.co.uk; Gustavo Petri, gfpetri@amazon.co.uk, Automated Reasoning Group, Amazon Web Services, United Kingdom; Nathan Chong, ncchong@amazon.com, Automated Reasoning Group, Amazon Web Services, United States.

2475-1421/2023/4-ART88
https://doi.org/10.1145/3586040

1 INTRODUCTION

This paper describes Arm's use of a range of formal techniques in validating the specification of, and thereafter verifying a C-language implementation of, a new security-critical firmware called *Realm Management Monitor* (RMM). This firmware is a central component of the recently-announced Arm® *Confidential Computing Architecture* (Arm CCA); a technology comprising a reference software architecture and an architectural extension for the Armv9-A profile. Arm CCA will eventually be deployed in hundreds of millions of devices worldwide. Formal techniques were used to not only validate and verify design and implementation decisions throughout development, but also *influenced* the way that the RMM was specified and designed from the outset, acting as a guiding principle by Arm's product teams. We will provide a detailed description of our validation and verification methodology, but start by providing additional context needed to understand the significance of our decision to deploy formal techniques in this domain.

1.1 The Rise of Confidential Computing

Most mainstream computer architectures are now adopting architectural support for *Confidential Computing*, largely driven by the rise of Cloud computing, wherein potentially sensitive computations and datasets are now routinely *delegated* to a Cloud host. This delegation is not without risk, as it requires sensitive computations and datasets be shared with a third-party, introducing a route through which data may be leaked, and computations manipulated, by an attacker. Whilst the Cloud-host is one obvious potential attacker, most hosted computations take place on *co-tenanted* machines, wherein multiple computations, owned by mutually-mistrusting parties, are executed concurrently, with co-tenants potentially trying to exploit security vulnerabilities in the isolating hypervisor. Note that this is not a theoretical risk, as severe security vulnerabilities have been identified in hypervisors in the past [Cook et al. 2020; Pék et al. 2013]. Given this, industry has turned to *Confidential Computing* technologies rooted in trusted hardware as the primary solution, with a blend of features driven by security concerns associated with delegated computation.

Protected execution environments. Each of these solutions introduces a *protected execution environment* as a new primitive, called Secure Enclaves, Protected Virtual Machines, Realms, Trusted Execution Environments, or similar. (Henceforth, we use "Realm" for the Arm CCA-specific primitive and "protected execution environment" as a generic collective noun.) These protected execution environments are intended to provide strong *confidentiality* and *integrity* guarantees to the code and data they host, preventing spying or interference by unauthorized parties. Moreover, these guarantees should hold even in the face of a privileged adversary: untrusted operating systems, hypervisors, and software executing in other protected execution environments are all assumed malicious, with adversaries assumed capable of using system features—such as high-precision timers—out-of-reach for a typical attacker. The *Trusted Computing Base* (TCB) of a protected execution environment is therefore minimized, comprising the contents of the protected execution environment and its implementation in hardware and firmware.

Remote Attestation. Each Confidential Computing technology is usually accompanied with support for *Remote Attestation* [Coker et al. 2011]; a mechanism supported by cryptography through which a skeptical third-party may deduce, with high-confidence, that a legitimate protected execution environment, loaded with a known initial software image, correctly configured, exists on an otherwise untrusted third-party's machine.

Together, these aspects allow a skeptical party to securely delegate sensitive computations to untrusted, third-party machines and receive high-levels of assurance that these computations are protected from prying or undetected interference by co-tenants and machine owner alike.

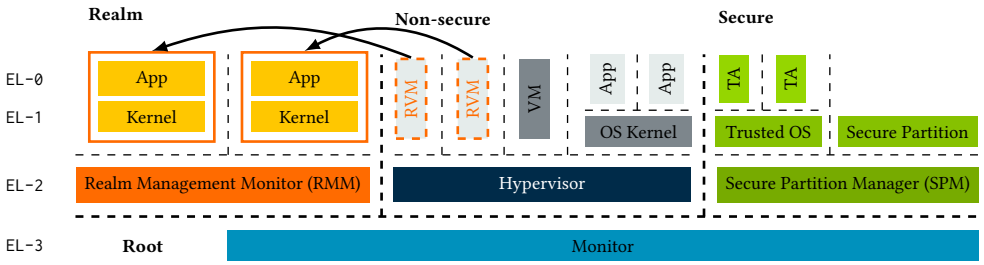

Fig. 1. An architectural schematic of an Arm CCA-enabled PE

This is exactly what is required for solving security and privacy challenges associated with Cloud computing. These assurances, however, only hold if the protected execution environment is correctly specified and implemented. Whilst the semiconductor industry in general, and Arm in particular, makes wide use of formal methods for checking hardware implementations, one interesting aspect of Confidential Computing technologies (including Arm CCA) is the use of both *hardware* and *firmware* in their implementations. As a result, whilst hardware engineers are already primed to think of formal verification as a routine aspect of any engineering flow, new and unfamiliar techniques and tools need to be deployed to accommodate firmware verification.

1.2 Arm Confidential Computing Architecture

We now introduce relevant aspects of Arm CCA; more can be learned from [Arm Ltd. 2021]. Execution on an Armv8-A and Armv9-A *processing element*[1] (PE) is associated with an *exception level*, `EL0` through to `EL3`, denoting relative privilege. User-space executes at `EL0`, operating system kernels at `EL1`, hypervisors at `EL2`, and privileged, low-level firmware at `EL3`. Executing at an elevated privilege level allows, for example, modifications to important system registers. Memory is partitioned into *granules* of some fixed size, the unit of memory managed by the architecture.

The Arm TrustZone® [Arm Ltd. 2008] extension introduced two architectural security states: *Secure* and *Non-secure*, with physical memory correspondingly bifurcated into *Secure* and *Non-secure Physical Address Spaces* (PASs). An additional PAS bit propagates throughout the memory subsystem, enforcing memory access controls based on the combination of the current PE security state and the PAS being accessed. An Arm A-profile PE is therefore split into two logical *worlds*, with the Secure world hosting *trusted applications* executing under a trusted operating system [Linaro Ltd. 2022a]; and the Non-secure world hosting a *rich* operating system (typically Android or Linux). Trusted applications offer sensitive services, such as cryptographic key management, to less trusted software executing in the Non-secure world, and remain protected from it. Communication and context switching between worlds is handled by the *monitor* [Linaro Ltd. 2022b], privileged firmware executing at `EL3`.

Typical Arm TrustZone deployments are largely *static*: memory is assigned to the Secure world on device boot, and the Secure world cannot be dynamically resized thereafter. The Secure world also tends to be memory-constrained, since it hosts a small number of trusted applications which are provisioned onto the device by the device manufacturer or system integrator.

In Figure 1 we provide a schematic of a future Arm CCA enabled PE under Armv9.1-A. The hardware changes associated with Arm CCA are collectively called the *Realm Management Extensions* (RME); intuitively, these generalize Arm TrustZone. The split between the Secure and Non-secure world is still present (protecting trusted applications) but an additional *two* security

[1]Processing elements generalize CPUs, as other components in the Arm system architecture can also compute.

states and associated PASs have been introduced: *Realm* and *Root*. As such, Arm CCA-enabled PEs are logically split into *four* worlds instead of two. RME also introduces additional memory access controls, called *Granule Protection Checks* (GPCs), which gate access to PASs depending on the Security state of the accessing PE. RME also enables memory to be dynamically transitioned between PASs, importantly for Arm CCA between the Non-secure and the Realm PASs, allowing system resources to be partitioned according to the demands of workloads in different Security states. This transitioning is performed by the monitor, which is now moved into its own dedicated Root world.

Realm world hosts eponymous *Realms*—Arm CCA's protected execution environments—with each Realm hosting a protected virtual machine spanning `EL1` and `EL0` of the Realm world. Realms can be *dynamically* spawned and torn down, marking a departure from the static trusted applications of Arm TrustZone, and are provisioned and administered by software executing in the Non-secure world, rather than by the platform manufacturer, to support Confidential Computing use-cases. Realms are assumed *mutually distrusting*, as are Realm and Secure world, with both also distrusting the Non-secure world. The architecture provides protections for attacks coming from other Realms or from Secure or Non-secure worlds.

To administer Realms, Arm CCA introduces a new privileged firmware component called the *Realm Management Monitor* (RMM), executing at `EL2` in the Realm world, and depicted in orange in Figure 1. The RMM acts as a *separation kernel* isolating Realms from each other. Responsibility for allocation of memory and CPU resources used by Realms remains with the untrusted hypervisor executing in the Non-secure state, which we call the *Host*. Arm CCA splits *policy* from *mechanism*, with the Host retaining control of resources, whether allocation of memory to Realm, or scheduling CPU cycles to a Realm, making use of the RMM proxy to enact its requests. To do this, the RMM presents an ABI, called the *Realm Management Interface* (RMI), which enables the Host to manage and schedule Realms indirectly. Note that as all scheduling and resourcing decisions for Realms remain with untrusted code executing in the Non-secure world, and Realms do not, in general, have any form of *availability* guarantee. A separate ABI, called the *Realm Services Interface* (RSI), is presented to the Realm. For example, the RSI can generate attestation evidence. For the majority of this paper we shall concentrate on the RMI interface, only mentioning RSI when necessary.

1.3 Scope and Contributions

Here we focus exclusively on the specification and implementation of the RMM and its interactions with hardware, establishing important guarantees derived from the specification, as well as the adherence of Arm's prototype RMM implementation to the specification. We ignore Remote Attestation: Arm CCA commits to a common attestation token format, but as an *architecture* does not mandate any particular protocol. RME hardware changes are handled by Arm's established hardware verification flows.

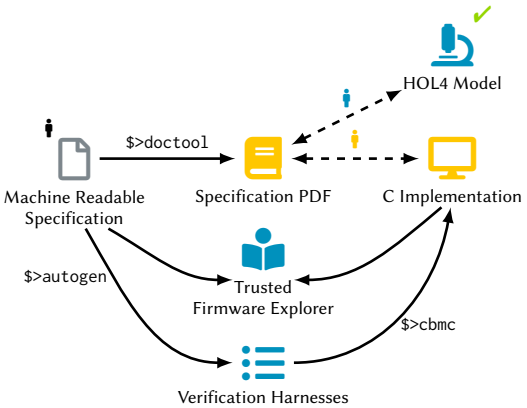

Fig. 2. The RMM verification flow and associated tools. Arm-published artifacts are shown in yellow, whilst our verification artifacts are shown in blue. Steps that require user input are labeled with a stick figure.

Importantly, we developed verification techniques for the RMM contemporaneously with the development of the specification and implementation prototypes, adopting formal techniques early in the design process. While our ultimate goal is to ensure the safety and security of the final specification and implementation, our

methodologies have excelled at revealing inconsistencies, errors, and imprecision in both specification and prototype implementation. Early adoption of formal techniques requires methodologies that can readily adapt to change, and our tooling must be a good match for the different artifacts under verification. In Figure 2 we present tools that we use to interface between the specification and the implementation, and verification artifacts that we produce to validate them. A *Machine Readable Specification* (MRS) is central to this methodology: Arm architects use the MRS (shown left in Figure 2) to specify the RMM ABI and an Arm-internal tool, called `doctool`, is used to generate the public RMM PDF documentation [Arm Ltd. 2022c] from the MRS. The behavior of the RMM ABI is specified using pre/postconditions, and is discussed in Sec. 2.

We have paid special interest to validating the RMM ABI, using a formal model in the HOL4 proof assistant (shown upper right in Figure 2); validating the coherence of the specification, proving that important invariants are enforced by the specification, as well as desirable security properties for Realms. This model, discussed in Sec. 3, is produced by hand, through careful interpretation of the specification. We have worked closely with product engineers to keep up-to-date with the specification and to validate updates thereof. Incomplete or under-specified details in the specification have been reflected as educated assumptions within the HOL4 model. We expect our formal model to be made publicly available alongside the RMM specification [Arm Ltd. 2022c].

Arm has developed a prototype C-language implementation of the RMM firmware (shown right in Figure 2) to explore and validate RMM ABI design choices. This prototype has provided a baseline contribution to the production version of the RMM, which is to be maintained by TrustedFirmware.org. For now, C remains the established systems language choice for implementing low-level firmware, such as the RMM. Viable alternatives are emerging, such as Rust, but Rust's in-built safety benefits must be carefully weighed against the downsides of having a much smaller developer community, combined with a comparatively immature and limited choice of development and verification tooling.

To verify the correctness of the RMM implementation with respect to the RMM specification, we introduce an automated model checking workflow based on the *C bounded model checker* (CBMC) [Kroening and Tautschnig 2014]. We automatically generate verification harnesses from the MRS of the RMM, using an internal tool named `autogen`, as shown in Figure 2. The invariants discussed in Sec. 3 are used to constrain the initial state, making model checking with CBMC tractable. We also discuss how the states of abstract specification and concrete implementation are related and how the result of a failed verification can aid in debugging the implementation and specification. Importantly, since our workflow is automated, we can use the model checker as an early error detection mechanism, as well as an enhanced debugging tool integrated into our product team's Continuous Integration (CI) flow. Our model checking workflow is expected to be released alongside the RMM implementation.

The RMM implementation maintains RMM-private objects and data structures used for bookkeeping and Realm administration, which may be manipulated and queried by many different concurrent callers, executing on different cores. For performance reasons, fine-grained locking of objects is used, and in Sec. 5 we address the maintenance of the coherency and safety of the internal RMM prototype implementation state. While the specification only describes commands as being atomic (through the use of pre/postconditions) the implementation must mediate concurrent accesses from different cores to the shared data structures maintained by the RMM, with locks not only serving as a coordination mechanism, but as a critical aspect of the security of the implementation. To avoid deadlocks, we impose a non-trivial strict lock ordering discipline for commands: this is not a static ordering, with the order in which new objects are *discovered* in the process and hence locks are taken dependent on the state of the RMM. To ensure the RMM implementation is deadlock-free we use a combination of HOL4 theorem-proving, to verify that

Table 1. Excerpted Realm abstract state specification (draft).

Name	Type	Description
ipa_width	UInt64	IPA width in bits
measurements	RmmRealmMeasurement[7]	Realm measurements
measurements_algo	RmmRealmMeasurementAlgorithm	Algorithm used to compute Realm measurements
rec_index	UInt64	Index of next REC to be created
rtt_base	Address	Realm Translation Table base address
rtt_level_start	Int64	RTT starting level
rtt_num_start	UInt64	Number of physically contiguous starting level RTTs
state	RmmRealmState	Lifecycle state
vmid	Bits16	Virtual Machine Identifier

the *discipline* ensures deadlock-freedom, and model checking, to ensure that all commands respect the discipline. Together, these results ensure the deadlock-freedom of the RMM implementation.

Finally, in Sec. 6 we present a debugging and introspection framework called *Trusted Firmware Explorer* (TFX) which we use to gain empirical confidence in the correctness of the RMM in situations which are not covered by our formal verification methodologies, such as interrupt injection and specific or random concurrent executions of RMI commands. TFX processes the MRS and the RMM C implementation, making it maintainable as the RMM specification and implementation evolve.

Together, these different verification efforts address the coherence of the specification and some security guarantees; they address the correspondence of our prototype implementation with the specification, as well as some non-functional correctness guarantees such as deadlock-freedom. For the hardware-software interface, we implement our own testing and concurrency-aware validation tools, which increase the confidence of the RMM implementations and can identify bugs early. Whilst we have not yet achieved a full end-to-end formal security statement for Arm CCA—not to mention that the RMM specification and prototype are still evolving—to the best of our knowledge this is the most comprehensive formal and semi-formal analysis of a Confidential Computing architecture, and in Sec. 8 we discuss future work that will further enhance our results.

2 RMM SPECIFICATION: PRECONDITIONS AND POSTCONDITIONS

The RMM specification [Arm Ltd. 2022c] provides a detailed description of the intent and operation of the firmware component of the Arm CCA architecture. To that end, it describes the abstract state managed by the firmware, as well as the ABI used to interact and manipulate this state. In this section we provide a brief, high-level description of the specification, focusing on the elements that are necessary to understand our work.

RMM-internal objects and data structures are queried and manipulated by the RMM in response to *syscall*-like commands, exposed as binary interfaces, either issued to it by the Host, through the *Realm Management Interface* (RMI), or by a Realm, through the *Realm Services Interface* (RSI). The RMM specification describes the high-level RMM internal state, together with the behavior of the RMM commands in terms of how they act on this state.

Abstract State Specification. The RMM abstract state consists of a number of data structures managed by the RMM, including metadata that tracks and details memory granule usage within the RMM. Table 1 provides an example of metadata used by the RMM to administer Realms, presented as named fields or attributes of a structure, with a defined type, and a brief description of their purpose. The specification also defines a number of predicates over, and auxiliary functions that make modifications to, this abstract state. These predicates and functions are used in the specification of the commands, as well as in the description of important RMM invariants.

D3.2.5 RMI_GRANULE_DELEGATE

Delegates a Granule.

D3.2.5.1 Interface

D3.2.5.1.2 Input Values

Name	Register	Field	Type	Description
fid	X0	[63:0]	UInt64	Command FID
addr	X1	[63:0]	Address	PA of the target Granule

D3.2.5.1.3 Output Values

Name	Register	Field	Type	Description
result	X0	[63:0]	ReturnCode	Command return status

D3.2.5.2 Failure conditions

ID	Condition
gran_align	pre: !AddrIsGranuleAligned(addr)
	post: ResultEqual(result,RMI_ERROR_INPUT)

(–– continued in the right column)

(–– from the left column)

gran_bound	pre: !PaIsDelegable(addr)
	post: ResultEqual(result,RMI_ERROR_INPUT)
gran_state	pre: Granule(addr).state != UNDELEGATED
	post: ResultEqual(result,RMI_ERROR_INPUT)
gran_pas	pre: Granule(addr).pas != NS
	post: ResultEqual(result,RMI_ERROR_INPUT)

D3.2.5.3 Success conditions

ID	Post-condition
gran_state	Granule(addr).state == DELEGATED
gran_pas	Granule(addr).pas == REALM

D3.2.5.4 Footprint

ID	Value
gran_state	Granule(addr).state
gran_pas	Granule(addr).pas

Fig. 3. Excerpted RMI_GRANULE_DELEGATE command specification (draft).

A succinct description of the most important RMM data structures is provided in Table 2. Most structures are self-explanatory, with the possible exception of RECs, which represent the virtual PEs associated with a Realm. REC structures are used to enter and exit Realm execution when the virtual PE is scheduled.

Table 2. Main data structures managed by the RMM.

Acronym	Name	Function
RD	Realm Descriptor	Representation of a Realm (see Table 1)
RTT	Realm Translation Table	Contains Translation Tables of Realms
REC	Realm Execution Context	Represents a virtual PE associated to a Realm
Data	Realm's Data	Metadata for granules containing Realm's data

RMM ABI Specification. The behaviors of all RMM commands are defined via pre/postconditions, with execution of each command potentially modifying a defined *footprint* of state elements, thus making verification tractable. There are no undefined behaviors for command execution: all commands are totally defined, covering both success and error cases (this is discussed in Sec. 3.4). Commands are associated with an opcode, which uniquely identifies them, and the behavior of each command, which is a function of the values of its arguments and state of the RMM when the command is invoked.

In Figure 3 we show a simplified specification of the RMI_GRANULE_DELEGATE command, which takes a single granule address as input from the Host (the addr parameter, in general purpose register X1, is the physical address location of the granule to be delegated). If the corresponding granule metadata is UNDELEGATED, meaning that the granule is not in the REALM PAS, and moreover it is currently in the NS PAS, then the granule is moved to the REALM PAS.

The *Failure conditions* section of Figure 3 consists of pre/postcondition pairs, with one pair for each possible error condition. Whilst each command's specification could be manipulated into a traditional Hoare-logic style triple, the style used within the RMM documentation is easier to comprehend, with each condition dedicated to a single task, and it is also useful in the generation of our verification harnesses (see Sec. 4). For example, the first failure case verifies the alignment of the addr address, with the command returning a RMI_ERROR_INPUT error if this check fails. Other

failure conditions are similar. Importantly, commands should be *all-or-nothing* in the sense that error-raising commands do not modify the abstract state of the RMM, i.e., their footprint is empty.

The *Success conditions* section of Figure 3 describes the expected updates and state changes of successful commands. Note that a command without a failure condition has by omission a *true* precondition and therefore necessarily succeeds. Here, we see that the new PAS for the granule is `REALM`, and the state is updated to `DELEGATED`, indicating that it is not actively in use by any Realm. Note that the *Footprint* section of Figure 3 lists the set of RMM state attributes that are updated on a successful execution of the command, and there is a requirement that *all other RMM state components* are left unmodified by the command.

Machine Readable Specification. The example command specification presented in Figure 3 is an excerpt from a draft PDF of the RMM specification, generated from the Arm CCA MRS. Each command is ultimately defined in a YAML file specifying the pre/postconditions, and other information, such as a partial order describing the possible orders in which preconditions can be checked. This MRS is the source for the verification harnesses (discussed in Sec. 4), providing a *single source of truth*, and ensuring a tight correspondence between verification harnesses and specification. Consequently, any update in the specification is automatically reflected in our verification flows.

3 RMM MODEL AND PROOFS

Sec. 2 described how pre/postconditions have been used to specify the RMM ABI. These assertions are written in ASL [Reid 2016], Arm's internal specification language. The ASL for the MRS is parsed and type-checked, which helps to identify early issues within the specification. This section describes how we have used theorem proving in HOL4 to further validate the RMM specification and ensure that it satisfies important design requirements. Our HOL4 model operates at the level of the RMM ABI, which is the basis for *all* RMM implementations. It is essential to get this ABI right: flaws here could result in security vulnerabilities that affect every RMM implementation. The HOL4 formalization and proofs have uncovered dozens of issues (ambiguities, under-specified behavior and technical problems) within the RMM specification. We have been able to identify these issues early in the design process, and thus accelerate the RMM specification's evolution, which has in turn reduced the need for more disruptive changes later on.

The RMM specification is predominantly axiomatic in style (using assertions instead of code), and is intentionally highly abstract, prioritizing brevity, clarity and freedom from implementation detail. A prescriptive operational semantics of RMM commands is undesirable at the specification level, as it is important to not over-constrain the architecture or unduly influence *how* each RMM implementation satisfies the requirements of the ABI. Although Arm is not providing an operational model of the RMM, we would still like to explore, test and validate the high-level specification by running *flows* (evaluating sequences of command calls) representing typical ABI usage scenarios. This goal provided the initial motivation for constructing a HOL4 version of the RMM specification. Proof assistants, such as HOL4, are very versatile and well suited to rigorously formalizing (mirroring) the axiomatic MRS. We can develop and use custom written HOL4 *automation* to animate the specification, providing a reliable means to explore RMM flows and better understand the emergent behaviors of the RMM specification. In particular, we have proved that the RMM specification preserves important invariant properties. It is infeasible to achieve all of these results through the use of standard EDA tools.

Overall, HOL4 has been a good fit for verifying the RMM specification. While HOL4 lacks the comparative ease of use of some fully-automated tooling (e.g., those based on SMT), it does have a number of key advantages here: a flexible and *expressive* logic that supports a wide range of specification styles and verification tasks (it is able to accurately model and verify high-level system

properties); and the ability to overcome tractability issues through human guidance and ingenuity, which can present a significant risk when relying on the abilities of full-automated tooling.

We provide an overview of the HOL4 specification in Sec. 3.1 and discuss model animation in Sec. 3.2. We then present our main proof results in Sec. 3.3, which includes an important Realm isolation property in Sec. 3.3.2. Finally, we discuss the topic of proof maintenance in Sec. 3.5, which is especially relevant in a live product development setting.

3.1 HOL4 Model

There are three main parts to the HOL4 model:

(1) *RMM command predicates.* For example, `RMI_COMMAND` *pre_state post_state* is true iff an RMM state *pre_state* can transition to state *post_state* by making a call to an RMI command.
(2) *State assertions.* For example, `X 0` *addr* represents the set of all RMM states in which variable *addr* is the current content of register `X0`.
(3) *Primitive functions.* For example, `AddrIsGranuleAligned` *addr* is true if *addr* is 4 KiB aligned.

The RMM command predicates are defined using the state assertions and primitive functions. At the time of writing the specification was roughly 3.5 KLoC, and proof scripts were around 24 KLoC.

We have developed and maintained the HOL4 specification manually, with no auto-generation from the MRS. There are two main reasons for this approach. Firstly, HOL4 auto-generation would be time-consuming to develop and maintain, and potentially unreliable or disruptive. Using HOL4 expertise to manually fine-tune the specification has been important in facilitating proof work, which has been the main priority. Secondly, both the HOL4 and the MRS specifications have been under active development, and *by choice* have not always been fully aligned with each other: in some cases, the HOL4 model has contained more detail, providing full definitions for most primitive functions and including precise definitions for pre/postconditions that had only been sketched out (using comments) within the latest MRS. In other cases, the HOL4 model has temporarily trailed behind the evolving MRS while proofs and other work was completed. Allowing the HOL4 model to freely move ahead of the MRS at times has provided an important feedback mechanism which helped guide and inform specification choices within the MRS.

3.1.1 The State Space. The HOL4 model operates at a high level of abstraction; it does not incorporate low level details of the underlying Armv9-A architecture or micro-architecture. As such, the state space, representing the RMM's view of the world, contains just two components:

(1) An array of general purpose registers; and
(2) A finite collection of granules, represented by a finite map from PAs to a pair, consisting of granule attributes and objects.

The model contains no details with regard to the underlying memory system, system registers, exception levels or security states.[2] Nevertheless, the HOL4 model does contain sufficient detail to capture the key aspects of the RMM ABI and to reliably validate the MRS on which it is based.

For convenience, we use custom data types to represent various RMM granule *objects*. For example, we use the HOL4 function type `word9→RmmRttEntryState` to represent Realm Translation Table (RTT) objects, which are the (stage 2) page tables maintained by the RMM (see Sec. 3.1.3). We could have adopted a more general approach here, modeling granules as raw 4 KiB arrays, together with object encoding and decoding functions. However, this less abstract approach would have added complexity without obvious benefit.

[2]The RMM documentation contains sections of prose covering various architectural aspects of the design (for example, Realm interrupts and exceptions). These details are not captured in the HOL4 model.

Preconditions	
Shared	X 0 RMI_GRANULE_DELEGATE X 1 *addr* Granule *addr gran*
Failure	GranuleDelegateError (*addr*,*gran*) *result*
Success	NoGranuleDelegateError (*addr*,*gran*) *result*

Postconditions	
Shared	X 0 *result*
Success	Granule *addr* (*gran* with <|pas := REALM; state := DELEGATED|>)

Footprint			
Failure	{ GPR 0 }	**Success**	{ GPR 0; GRAN *addr*; OBJ *addr* }

Fig. 4. RMI_GRANULE_DELEGATE HOL4 specification.

A technicality of the HOL4 specification is that the actual RMM state space is represented by the set WELL_TYPED. For example, in a *well typed* state, every granule in state RTT will have an RTT object associated with it.[3]

3.1.2 Command Predicates. The HOL4 specification focuses on RMI commands, covering little material related to RSI commands, which we do not focus on here. We declare the predicate RMI_COMMAND using a conjunction of clauses of the form:

```
RMM_ASSERT (pre_state :State) (post_state :State) (pre_conds :(State set) set)
  (post_conds :(State set) set) (footprint :Component set) ⇒ RMI_COMMAND pre_state post_state
```

The helper predicate RMM_ASSERT makes sure that:

- State *pre_state* satisfies WELL_TYPED and *all* of the preconditions in the set *pre_conds*;
- State *post_state* satisfies WELL_TYPED and *all* of the postconditions in *post_conds*;
- any component *not* in *footprint* has the same value in *post_state* and *pre_state*; and
- the domain of the granule finite map is the same in *post_state* and *pre_state*.

Most RMI commands are declared using two RMM_ASSERT clauses: one for failure, and one for success, though some commands never fail (e.g., RMI_VERSION), so have just one clause.

The HOL4 specification for the RMI_GRANULE_DELEGATE command is shown in Figure 4. The first precondition constrains register X0 to hold constant value RMI_GRANULE_DELEGATE, which is the unique *function identifier* (FID) for this command (at the time of writing: RMI_GRANULE_DELEGATE=0xC4000151). The second precondition constrains variable *addr* to be the contents of register X1, which is a command argument. The third precondition *conditionally* constrains *gran* to be the attributes associated with the granule at location *addr*. An important feature of the HOL4 specification (see Sec. 3.4) is that, in use, such Granule assertions are universally true for all states *pre_state*. If the granule at address *addr* in not delegable (there is no granule meta-data) then the precondition is true and the value of *gran* is unconstrained.

The last precondition differs between the failure and success clauses. GranuleDelegateError holds when a command failure precondition is true, and *result* is the associated error code. NoGranuleDelegateError holds when there is *no* failure condition that is true, and *result* is a success code (value 0). The failure and success cases cover *all* well typed states *pre_state* in which a RMI_GRANULE_DELEGATE command is called. The set of all errors is specified as follows:

```
GranuleDelegateErrors (addr,gran) pre_state ≝
      ErrorSet { ("gran_align",¬AddrIsGranuleAligned addr,RMI_ERROR_INPUT,0w);
                 ("gran_bound",¬PaIsDelegable pre_state addr,RMI_ERROR_INPUT,0w);
                 ("gran_state",gran.state ≠ UNDELEGATED,RMI_ERROR_INPUT,0w);
                 ("gran_pas",gran.pas ≠ NS,RMI_ERROR_INPUT,0w) }
```

The helper function ErrorSet constructs the set of errors by eliminating all failure conditions that are false. Only one error code (value of *result*) is possible in this example: the value of *result* must

[3]The RMM state space could have been represented directly using *constructors* within HOL4, but we wish to closely mirror the MRS specification here, which just has an *enumerated type* for granules, e.g., *gran*.state ≠ RTT not *gran*.state ≠ RTT(*obj*).

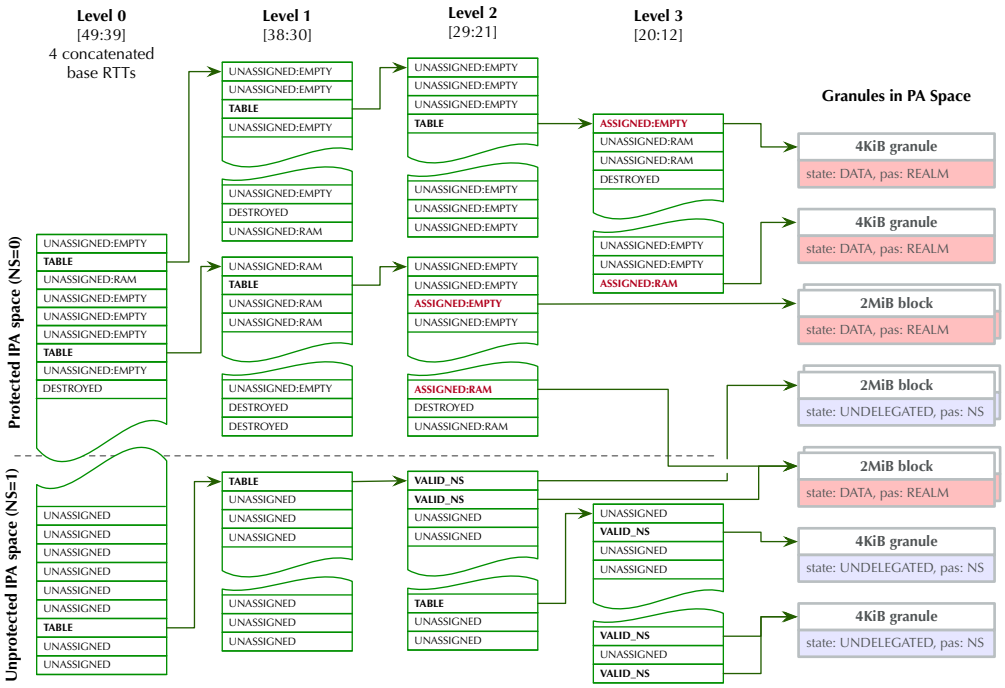

Fig. 5. Example of Realm page tables: configured with 50-bit IPA width and a start level of zero.

correspond with error (`RMI_ERROR_INPUT`, `0w`). For some commands and inputs, there may be a choice of error code, and here the HOL4 model asserts that the error code must be valid with respect to a declared partial ordering on errors. For brevity, we omit details of these partial orderings.

The first postcondition constrains `X0` to have value *result*, which is the error/success code. The success case has an extra postcondition: a `Granule` assertion that specifies the required updates to the granule attributes. The footprints contain the component `GPR 0`, which specifies that register `X0` is permitted to change value. The success case footprint has two additional components: `GRAN` *addr* for changing the granule attributes; and `OBJ` *addr* for changing the object associated with the granule. The `OBJ` footprint component is a technicality of the HOL4 specification: a delegate command does not actually change the memory contents of the delegated granule, however, in the model, the object associated with the granule must be allowed to change to remain well typed.

The footprints in the HOL4 specification are more coarse grained than those presented in the MRS specification; they work over whole granules, rather than individual record fields. In the HOL4 model, unmodified record fields are preserved by updating the pre-state value using the `with` construct. This approach helps avoid excessive complexity within the specification and proofs.

3.1.3 Page Table Walks. The RMM manages stage 2 page tables[4] for Realm world, thus guaranteeing memory isolation between Realms. The stage 2 tables translate *Intermediate Physical Addresses* (IPAs) into *Physical Addresses* (PAs). The RMM documentation works over an abstraction of Arm's *Virtual Memory System Architecture* (VMSA) [Arm Ltd. 2022a].

An example of this page table abstraction is shown in Figure 5. An abstract type is used to represent page table entries. There are three main entries: a `TABLE` entry points to a next level

[4]Stage 1 page tables translate Virtual Addresses (VAs) into IPAs, and are managed by the Realm OS kernel at EL-1.

Preconditions
X 0 RMI_RTT_CREATE
X 1 *rtt*
X 2 *rd*
X 3 *ipa*
X 4 *level*
Granule *rtt gran_rtt*
Granule *rd gran_rd*
Realm *rd realm_rd*
RttWalk *rd ipa* (*level* − 1w) *walk*
NoRttCreateError (*rd*,*gran_rd*,*realm_rd*,*rtt*,*gran_rtt*,w2i *level*,*ipa*, *walk*) *result*

Postconditions
X 0 *result*
Granule *rtt gran_rtt′*
RTT *rtt rtt_rtt*
RttWalk *rd ipa* (*level* − 1w) *walk′*
K (*gran_rtt′* = *gran_rtt* with state := RTT ∧ *walk′*.entry = TABLE *rtt* ∧ *rtt_rtt* = RttUnfold *walk*.entry)

Footprint
{ GPR 0; GRAN *rtt*; OBJ *rtt*; OBJ *walk*.rtt_addr; WALK *realm_rd*.ipa_width *walk walk′ ipa* }

Fig. 6. RMI_RTT_CREATE success case.

table; an ASSIGNED entry (on the protected half) points to Realm data, and a VALID_NS entry (on the unprotected half) can point to any PA. Consider walking the example page tables (Figure 5) to a requested depth of level 3, using the 50-bit IPA addr with octal value 0001_002_003_000_0000 (split at level boundaries). We start at level 0, which is an extra large (concatenated) base page table with $4 \times 512 = 2048$ entries. We lookup the entry at index addr[49:39]=1, which is a TABLE entry that points to the topmost table at level 1 (a regular 512 entry table). We now perform a (level 1) lookup at index addr[38:30]=2, which is another table entry, this time pointing to the topmost table at level 2. The walk proceeds with a (level 2) lookup at index addr[29:21]=3, which then takes us to a level 3 table. The final (level 3) lookup at index addr[20:12]=0 gives us an ASSIGNED entry that maps to the first protected data granule in the PA space. By contrast, a (level 3) walk for IPA 0x0 would terminate early at level 0, since addr[49:39]=0 and this gives an UNASSIGNED entry, which indicates that this address range is unpopulated.

The IPA space is split into two halves: *Protected* and *Unprotected*. Realm confidentiality and integrity guarantees only apply to the protected half. In our 50-bit IPA example, bit 49 is 0 for protected addresses, and 1 for unprotected addresses.

At the time of writing, there are twenty-three RMI commands, of which eleven involve page table walks. Walks are handled using assertions of the form: RttWalk *rd ipa level walk*. This assertion *conditionally* constrains *walk*, of type RmmRttWalkResult, to be the result of attempting a page table walk, under Realm *rd*, for IPA address *ipa*, to a requesting depth of *level*. If *rd* is not a valid Realm descriptor then the assertion will still be true, but *walk* will not be constrained.

Figure 6 shows the HOL4 specification for the success case of an RMI_RTT_CREATE command, which adds a new page table at *level*. The specification works by asserting that a walk is possible to *level* − 1 in state *pre_state*, and then asserting that the same walk in state *post_state* gives a new TABLE entry, which points to the new page table granule at address *rtt*.[5] The footprint now contains a special WALK component, which makes sure the modified page table at address *walk*.rtt_addr has just just one modified entry (no other walks are affected).

3.1.4 Primitive Functions. Most primitive functions within the HOL4 specification are relatively simple, which means we can readily give them an operational semantics, supporting evaluation within the logic. A significant feature of the HOL4 specification is that we also provide an operational semantics for performing page table walks, albeit at the same high level of abstraction presented in the MRS. There are two main reasons for this: it supports model animation (Sec. 3.2); and it enables precise/sound reasoning about the behavior of memory management commands (Sec. 3.3.1).

We define some *implementation dependent* constants using HOL4's constant specification facility, which is cleaner than polluting the state space with constants. For example, the HOL4 constant

[5]In the HOL4 sources bit-vector literals are identified with a w suffix, so value 1 appears as 1w.

RMM_MAX_S2SZ represents the maximum Realm IPA address width supported by the RMM and is defined by the following theorem: $\vdash$ 16w $\leq_+$ RMM_MAX_S2SZ $\wedge$ RMM_MAX_S2SZ $\leq_+$ 52w.

The set of delegable granules is determined by the function PaIsDelegable (used in the failure precondition in Figure 4), which represents all of the physical memory that the Host presents to the RMM. This function is not given a precise ASL definition within the MRS. In HOL4, a granule is considered delegable if its address is in the domain of the granule finite map. Hence, PaIsDelegable(addr) within the MRS corresponds with PaIsDelegable *state addr* within HOL4.

3.2 Model Animation

We use custom HOL4 automation as a means to explore the emergent behavior of the RMM specification. Prior to completing invariant proofs, the automation has helped in understanding the mechanics of, and rationale behind, the RMM specification. Given a concrete initial state *pre_state* and sequence of commands *cs*, the automation can derive a theorem of the form:

$$\vdash \texttt{COMMANDS}\ \mathit{pre\_state}\ \mathit{cs}\ \mathit{post\_state}$$

where *post_state* is an RMM state reachable through the sequence of commands *cs*. A simple abstract data type is used to represent commands, which include RMI and RSI calls.[6]

The automation works by iterating over the current state, and the next command, at each step constructing a suitable (witness) post state and proving that it satisfies the RMM semantics. This automation has acted as a *smoke test* for the formalization, helping identify early mistakes, inconsistencies and regressions. We have used the automation to confirm that important RMM flows work as expected. The main flow involves: creating a Realm; initializing the Realm IPA space; creating page tables; creating Realm data; creating Realm Execution Contexts (RECs); activating the Realm; and finally destroying everything (data, RECs, page tables and the Realm). This flow of nearly 300 commands takes a few minutes to complete, as HOL4 checks all inferences with its kernel. Here, we have prioritized ease of maintenance (see Sec. 3.5) over optimising the performance.

3.3 RMM Invariant

Model animation provides a good means to explore possible behaviors, but it does not provide deep insights. On the other hand, invariants can precisely characterize properties that hold for *all reachable states*, which is useful in establishing that certain *bad* (insecure) states are not reachable. Sec. 3.3.1 will briefly outline the definition of an RMM invariant that has been verified in HOL4, and Sec. 3.3.2 will present a memory isolation result that follows directly from this invariant definition.

The main invariant result is as follows:

$$\vdash \mathit{pre\_state} \in \texttt{INITIAL\_STATE} \wedge \texttt{RMI\_COMMAND}^+\ \mathit{pre\_state}\ \mathit{post\_state} \Rightarrow \mathit{post\_state} \in \texttt{RMI\_INVARIANT}$$

Here, RMI_COMMAND$^+$ is the transitive closure of the RMI command relation, and RMI_INVARIANT is our main RMM invariant.[7] This theorem shows that if we start in a valid initial state and the Host calls any number of RMI commands then the resultant state will always satisfy the invariant.

An initial RMM state is defined to be any well typed state in which all delegable granules are in state UNDELEGATED (they are not managed by the RMM):

$$\texttt{INITIAL\_STATE} \stackrel{\text{def}}{=} \texttt{WELL\_TYPED} \cap \{\, \mathit{state} \mid \forall\, \mathit{addr}.\ \mathit{state} \in \texttt{GranuleState}\ \mathit{addr}\ \texttt{UNDELEGATED} \,\}$$

This represents the set of all RMM states following a system boot.

[6] In reality interactions with the RMM would arise through PEs running sequences of Arm instructions.

[7] It is also important to establish that this invariant cannot be violated by any other means (e.g., Realm interface commands), though this is mostly trivially true.

3.3.1 Invariant Definition. The invariant RMI_INVARIANT is quite complex, and the full invariant was only *discovered* in the process of completing the proof. The invariant is split into three main parts, covering: basic Realm configuration, REC properties, and page table walks.

We provide a concise, mostly complete, informal description of the invariant in the supplementary material. The most pertinent aspect of the invariant concerns the set of possible stage 2 page table configurations (as illustrated by the example in Figure 5). The keys points are:

- RTTs are not aliased — they can only be reached by one Realm (the *owner*) through a single unique IPA prefix. This means that page tables form a *tree* structure.
- All ASSIGNED page table entries are at level 2 or 3. They map Protected IPAs to Protected Data granules that are not aliased by any other ASSIGNED entry. This means that Data granules have a single *owner* and can only be reached through a single unique IPA prefix.
- All VALID_NS entries are at level 2 or 3. They map Unprotected IPAs to PAs, and aliasing *is* possible (see Sec. 3.3.2).

The main challenge in verifying the invariant is reasoning about page table walks, which are constrained by the footprint and possibly by RttWalk assertions. For each command and Realm it is necessary to consider whether or not walks are preserved, or perhaps modified in a constrained way. For example, RMI_RTT_CREATE modifies one walk, and enables additional walks. By contrast, the commands RMI_RTT_DESTROY and RMI_RTT_FOLD modify one walk and remove other walks. The invariant proof has provided a powerful means to detect and avoid issues with these complex parts of the specification, which involve *block mappings* that alter possible walk depths.[8] In particular, it is important that all of the invariant properties associated with ASSIGNED entries are correctly preserved when RMM commands successfully introduce or remove a block mapping.

3.3.2 Realm Memory Isolation. The invariant above can be used to derive an important Realm memory isolation result. The following set represents the set of all states in which no two distinct Realms can access the same protected data by walking to an ASSIGNED page table entry:

$$\texttt{DISJOINT\_DATA} \stackrel{\text{def}}{=} \{\, state \mid \forall\, rd_1\ realm_1\ rd_2\ realm_2 .\ (rd_1, realm_1) \in \texttt{AllRealms}\ state \wedge (rd_2, realm_2) \in \texttt{AllRealms}\ state \wedge rd_1 \neq rd_2 \Rightarrow \texttt{DISJOINT}\ (\texttt{AllWalkData}\ state\ rd_1)\ (\texttt{AllWalkData}\ state\ rd_2) \,\}$$

where AllWalkData is the set of all Realm owned protected data. It is easy to prove

$$\vdash \texttt{RMI\_INVARIANT} \subseteq \texttt{DISJOINT\_DATA}$$

and this means the DISJOINT_DATA property holds for the RMM specification (all reachable states satisfy this property). This isolation result follows naturally from the definitions of AllWalkData and RMI_INVARIANT, since the invariant enforces the property that all ASSIGNED entries point to protected data granules that can only be accessed by the Realm performing the walk (which is the *owner* of the data).

The DISJOINT_DATA result only considers ASSIGNED page table entries in the Protected IPA space. Note that VALID_NS entries in the Unprotected IPA space *can* point to data from another Realm. (This can be achieved using the command RMI_RTT_MAP_UNPROTECTED.) This is why our invariant does not preclude aliasing for VALID_NS entries. When Realms access unprotected memory, a Non-secure flag is set (NS=1), which means that a Granule Protection Fault (GPF) exception will occur when Realms use unprotected IPAs to access granules that are not Non-secure. The HOL4 proof of DISJOINT_DATA, when combined with the granule protection checks of RME hardware, guarantee that specification compliant RMMs will always maintain the required Arm CCA world-to-world memory isolation requirements, i.e., *protected* Realm data cannot be *directly* accessed by any other world or by any other Realm.

[8] A block mapping is where 512 *homogeneous* level 3 page table entries are *folded* into a single level 2 entry.

3.4 No Undefined Behavior

The invariant of Sec. 3.3.1 provides us with good assurance that certain bad states are not reachable. However, we also need to guard against inconsistencies (*undefined behaviors*) accidentally creeping into the specification. For example, if we accidentally have conflicting assertions, which can never be true at the same time, e.g., `X 1` *level* $\wedge$ `X 1 (`*level* $-$ `1w)`, in either the preconditions or the postconditions of a command then state transition is precluded. In the worst case no state transitions are possible, in which case the state would always remain in `INITIAL_STATE`, and any invariant that is true for an initial state could be verified without correctly considering the intended command semantics. Unfortunately, inconsistencies can be hard to spot by eye, since they may only arise in subtle corner cases.

Model animation (Sec. 3.2) provides a limited protection against undefined behavior, since we demonstrate that certain expected states are reachable. However, we really require *no* undefined behavior, and so have proved the following theorem:

$$\begin{aligned}\vdash\ & \mathit{pre\_state} \in \texttt{WELL\_TYPED} \wedge \mathit{pre\_state} \in \texttt{X 0}\ r_0 \wedge \\ & r_0 \in \{\texttt{RMI\_FEATURES; RMI\_GRANULE\_DELEGATE; RMI\_GRANULE\_UNDELEGATE;} \\ & \qquad \ldots \texttt{; RMI\_VERSION}\} \Rightarrow \exists \mathit{post\_state}.\ \texttt{RMI\_COMMAND}\ \mathit{pre\_state}\ \mathit{post\_state}\end{aligned}$$

This theorem shows that there is no undefined behavior for RMM commands without table walks: all well typed states have a post state. A similar theorem, with some extra (minor) side-conditions, has been verified for all the remaining commands (with page table walks). Proving these theorems identified corner-cases missing from some command failure conditions.

If register zero does not hold a valid RMI FID then it is not possible to make a state transition. This legitimate undefined case is considered acceptable for the HOL4 specification because it corresponds with an error case, which has no impact in terms of invariant verification.

3.5 Tracking and Maintenance

A known challenge with theorem proving is the problem of *proof maintenance*. Minor and seemingly innocuous changes to specifications can have a disproportionate impact on proofs and proof automation. Simplistically, this is often reflected in script sizes; in the case of the RMM, seven times as much code is needed for proofs compared with definitions. To avoid maintenance being untenable, it helps if definitions remain mostly stable and all changes are carefully assessed to minimize proof impact. As such, theorem proving at scale tends to happen either with *post-hoc* verification (designs are modeled after they are stable/complete) or where verified designs are the primary output of theorem proving projects.

The RMM specification is amenable to verification, and Arm's design team have closely consulted with verification engineers prior to making significant changes. There have been cases where improvements to the specification have made HOL4 proofs harder. In such cases, proof engineers have provided assessments of proof effort, with a mind to always accommodating legitimate improvements that would not render proofs wholly intractable. Our priority is the primary audience for the specification: software engineers implementing an RMM, a Host hypervisor, or Realm software. We paid attention to keeping the HOL4 specification close to the source MRS. In particular, the HOL4 specification refrains from using any *shadow state* (such as extra fields to track granule *ownership*) for the purposes of easing the proof effort. This adds to the proof complexity but does minimize the risk of the HOL4 formalization unintentionally diverging from the MRS.

An initial HOL4 formalization was produced when the RMM documentation was at an early stage of development. The RMM ABI and MRS have since evolved at a regular cadence, and we have successfully maintained the HOL4 model and proofs to track this evolution, prioritizing swift adjustments in order to provide feedback in a timely manner. Minor changes have been relatively

easy to accommodate, often taking little more than a day to complete. However, some major changes have taken weeks to complete (some examples are provided in the supplementary material). To date, the most significant change has been switching from a fixed 48-bit IPA address space to supporting any IPA width from 16 bits to 52 bits (with configurable start levels and base RTT concatenation).

Our work shows that, under the right circumstances, it is possible to successfully deploy theorem proving in an active product development environment, without restricting design choices. We have taken care to avoid overreach, which has been important in avoiding overly long periods in which proofs are out of date or broken.

4 RMM IMPLEMENTATION VERIFICATION

In this section we describe how we generate verification harnesses for the implementation of the RMM from the MRS. Arm is implementing a prototype of the RMM written in C with some inlined AArch64 assembly. Validating that the implementation satisfies the specification means that each of the RMM commands implemented must satisfy the pre/postcondition contracts. Formally, we establish a refinement argument [Abadi and Lamport 1991]. We use the CBMC model checker to validate the adequacy of the implementation with the specification.

The question we ask here is: how can we ensure that the pre/postconditions that we use to *verify* the commands indeed correspond to the intent in the specification? We address this question by synthesizing verification harnesses for each command directly from the MRS used to define the specification. Below is a simple example to demonstrate how we go from the MRS, written in YAML format, to the resulting PDF specification and to the verification harness as a C program exercising the appropriate command in the implementation.[9]

Verification Harnesses. Since the specification is written as a sequence of pre/postcondition pairs, using predicates that constrain the inputs and outputs of the command, we produce code harnesses that, upon *assuming* the precondition, *assert* that the corresponding case of the postcondition is valid. Those harnesses are generated from the MRS. In the example in Listing 3 we can see that the predicate `AddrIsGranuleAligned` at line 8 is directly transcribed from the MRS in Listing 1 into the verification harness. While this predicate is used to describe the abstract state of the RMM in the specification, in the implementation it has to be realized through the implementation representation. To that end, for each of the predicates in the specification we need to provide an equivalent predicate to connect it to the implementation. The function implementing the predicate `AddrIsGranuleAligned` is shown in Listing 2, where we use the RMM implementation's C macro `GRANULE_ALIGNED` in Listing 4 to define the predicate.

In this case the implementation is relatively trivial, while other predicates require substantial understanding of the way that implementation state models the specification state, and their connection needs to be carefully aligned. An example of complicated predicates are preconditions or postconditions establishing the result of page table walks starting from a Realm's base table.

As can be seen from the generated verification harness, all the inputs to the command, stored in the `__tb_regs` data structure at line 1, are initialized to non-deterministic 64-bit unsigned integer values. In code omitted for readability, we also name the value of these registers with the names in the comments next to them (see the supplementary material for the full example). This is a sound approach, because all commands are totally specified. That is, the specification accepts no undefined behavior, and commands define a result for all input combinations, as mentioned in Sec. 3.4.

However, the correct execution of commands in the implementation not only depends on the input, but also on the *initial* state of the RMM. The states reachable for the RMM satisfy the invariant that we discussed in Sec. 3.3.1. In other words, if we start in an initial state satisfying

[9]The snippets contain only relevant portions to explain the example.

```
!commands.Command
name: RMI_DATA_CREATE
description: Creates a Data Granule, copying and
     measuring contents from a Non-secure ...
...
failure:
- !commands.CommandFailureCondition
  id: src_align
  pre: '!AddrIsGranuleAligned(src)'
  post: ResultEqual(result, RMI_ERROR_INPUT)
```

Listing 1. MRS of the src granule alignment case.

```
scr_align  pre:   !AddrIsGranuleAligned(src)
           post:  ResultEqual(result, RMI_ERROR_INPUT)
```

Fig. 7. PDF corresponding to Listing 1.

```
bool AddrIsGranuleAligned(ulong addr) {
        return GRANULE_ALIGNED(addr);
}
```

Listing 2. Linking the abstract and concrete state.

```
1  struct tb_regs __tb_regs = __tb_arb_regs();
2  __tb_regs.X0 = SMC_RMM_DATA_CREATE;
3  __tb_regs.X1 = nondet_uint64_t(); // data
4  __tb_regs.X2 = nondet_uint64_t(); // rd
5  __tb_regs.X3 = nondet_uint64_t(); // map_addr
6  __tb_regs.X4 = nondet_uint64_t(); // src
7  __init_global_state(__tb_regs.X0);  // Generate non-deterministic state
8  bool failure_src_align_pre = !AddrIsGranuleAligned(src);  // Evaluate precondition
9  uint64_t result = tb_handle_smc(&__tb_regs);  // Execute command
10 bool failure_src_align_post = ResultEqual(result, RMI_ERROR_INPUT); // Evaluate
       postcondition
11
12 // Failure condition assertions (excerpt)
13 bool prop_failure_src_align_ante = failure_src_align_pre;
14 __COVER(prop_failure_src_align_ante);
15 if (prop_failure_src_align_ante) {
16     bool prop_failure_src_align_cons = failure_src_align_post;
17     __COVER(prop_failure_src_align_cons);
18     __ASSERT(prop_failure_src_align_cons, "prop_failure_src_align_cons"); }
```

Listing 3. CBMC verification harness corresponding to the specification in Listing 1.

```
#define ALIGNED(_size, _alignment) (((unsigned long)(_size) % (_alignment)) == 0)
#define GRANULE_ALIGNED(_addr) ALIGNED(_addr, GRANULE_SIZE)
```

Listing 4. GRANULE_ALIGNED Macro.

the invariant explained in Sec. 3.3, and only execute RMM commands (successful or not), we will only arrive at states that also satisfy that invariant. Therefore, in the verification harnesses for our commands, we assume a non-deterministic state that satisfies (or more generally approximates) the invariant of Sec. 3.3.1. This is shown in Listing 3 as `__init_global_state(__tb_regs.X0)` at line 7 in Listing 3. In our experiments, generating precise non-deterministic states that satisfy the invariant limits the scalability of model checking. This is because precisely initializing the initial state according to the invariant, requires non-deterministically populating and linking many data structures which may not be used by the command under check. Since this initialization is performed using C code, this increases the model checking complexity, even if unused by the command. Hence, instead of initializing the whole state, we initialize a carve-out of the state which is sufficient to prove each of the commands while leaving the rest under-constrained. This is why the state initialization function takes the value of register `X0` as input the command identifier. Knowing what portion of the state needs to be populated according to the invariants is not trivial and we have not found means to automate it. Fortunately, in general, commands footprints are a good indications of what portion of the state needs to be constrained, and it only needs changing on specification changes, which are more infrequent than implementation changes.

In Listing 3 we only show the verification of the pre/postcondition pair corresponding to the failure of alignment of the source granule `src`. Ignoring the `__COVER` calls for now, we can see that the verification harness simply evaluates the precondition in the condition of an if-statement at

line 15, and verifies that the result of the command in that case matches the expected result in the specification at line 18.[10] While this is quite a simple command, other commands follow a similar pattern, albeit adding many more clauses in the precondition, and validating additional properties of the state in the case success case of the command.

Validating the Verification Harnesses. As mentioned, to improve the time taken by the model checker to verify a command, we provide specific non-deterministic initial states for each command. However, can we be assured that we have not over or under-constrained the initial state to a command? Answering that question is the purpose of the `__COVER` clauses (e.g., lines 14 and 17 in Listing 3) in the verification harness. These are defined as the `__CPROVER_cover` command of the CBMC model checker, which validates whether the program point, as well as the predicate provided as input, can be reached in some program execution by CBMC. Hence, if we over-constrain the initial state to a command, making any of its pre/postcondition pairs unreachable, we get a `__COVER` violation, meaning that either the initial state is over-constrained, or the pre/postcondition pair is vacuous. The former case is a bug in the verification harness, the latter is indicative of a problem in the specification or implementation.[11]

From Counter-examples to Debugging. When an error in the postcondition checks is found, CBMC provides a counter-example instantiating all the non-deterministic values. Moreover, we get a full trace leading to the error. Given the verbosity of the output obtained from CBMC, we have found that summarizing these calls into embedded call/return pair sub-traces is usually sufficient for engineers to accurately identify and fix the bug. To that end, we have implemented a simple Python script that sanitizes the output of CBMC on postcondition failure, and we use that as a first approximation to repair the error. Alternatively, we have used the `cbmc-viewer` tool provided by AWS [Amazon Inc. 2022], but we have found the simplified call/return terminal output more effective in quickly identifying the source of the problem.

An important remark here is that this is an important qualitative advantage in debugging when compared to the traditional unit testing approach common in testing and debugging workflows.

Inline AArch64 assembly. An important aspect of using CBMC is that the RMM implementation contains a small number of low-level code in AArch64 assembly. Unfortunately, CBMC cannot deal with these code. To solve this problem we provide alternative *shadow* implementations in equivalent C code. Importantly, most of these AArch64 snippets use specific Armv8-A instructions related to memory coherence and ordering constraints. Examples of these are the `stlr` instruction, which is a *store release* [Arm Ltd. 2022a]. Since we do not perform concurrent verification using CBMC, this is equivalent to a normal store `str` instruction, or a simple assignment in C:

```
void write_release(uint64_t *p, uint64_t v) {
  asm volatile(
  "␣␣␣␣stlr␣%[v],␣%[p]\n"
  : [p] "=Q" (*p)
  : [v] "r" (v)
  );
} // The original assembly version
```

```
void write_release(uint64_t *p, uint64_t v) {

  *p = v;

} // The C-equivalent version
```

As in this typical case, although most manual transformations must be trusted, they are trivial liftings of concurrency-aware AArch64 instructions into C.[12] As such, the presence of a small amount of standard assembly code, purely implementing hardware interface primitives, is of limited

[10] There are many alternative ways of encoding this, but for large commands we found this template to be quite effective.
[11] In general, this could also be the result of an implementation refining the specification to make some cases impossible.
[12] Though soundness might become a problem here, we have high confidence on the transformations, since they are done by the knowledgeable engineers in the production team.

concern regarding our primary verification goal for the implementation, which is to catch as many coding bugs as possible.

Bounds and scalability. Fortunately, due to the nature of the firmware, most loops contained in programs are of fixed, and generally small, size. The maximum level of any page table is bounded by 3 (potentially starting from level -1), as shown in Figure 5. This helps alleviate issues with under-approximation of bounds in CBMC. An important exception to this rule is the implementation of spin-locks, which will be discussed in more detail in Sec. 5. However, since in the verification of specification refinement we are not interested in concurrent executions, we assume that initially all locks are free, and therefore spin-locks reduce to a single iteration.

To tackle the scalability of CBMC we need to consider the size of the state space. While the actual implementations must keep track of all *delegable* granules, each command only ever uses a handful of them. Since our verification harnesses only checks one command at a time, it is safe to only enforce the *well-formedness* invariants on this small subset, since any other granule would fail on one or more of the precondition checks of the command. Based on this observation, we conclude that, due to the non-deterministic nature of the inputs to the commands, it is enough to restrict the total size of granules to a number larger than the maximum number used by any of the commands (which will only use them in the successful case, or in checking the most complex failure case). Hence, we provide the verification script, an additional argument to limit the number of granules to be considered for meta-data. With a small number of granules, all of our verification harnesses explore all the failure and success cases, and the complete verification of pre/postconditions for all commands finish in hours.

Similarly as above, Arm's RMM prototype implementation requires that certain data structures have allocated state for each PE. Hence, more PEs increase the state space. Since we only consider one command at a time, we assume that there is only one PE, further reducing the state space.

Results. Our verification harnesses were used for the initial RMM prototype, and are currently being used by the product team developing the Arm's reference RMM implementation. In fact, We prototyped the CBMC verification flow, mostly through auto-generation, with the goal that the production team can maintain and modify without help from verification engineers. Upon the release of the reference implementation, our tool will be made available. At the time of writing we have found over 30 bugs documented in the tracking systems of the prototype implementation—development teams use a different tracking system ranging from memory safety issues to violation of pre/postconditions. Some of these bugs were uncovered by our CBMC runs, and others were found while matching the abstract and implementation representations with each other by reading the specification. Also, some bug reports encompass a suite of related errors, so this number is an approximation of the number of defects that our verification captures.

5 RMM CONCURRENCY: DEADLOCK FREEDOM

We use a combination of formal modeling and interactive theorem proving in HOL4, alongside model checking with CBMC, to ensure the absence of deadlocks in the RMM implementation.

The RMM implementation uses fine-grained locking to mediate concurrent accesses to granules containing metadata about Realms, and stored in Realm PAS, in a performant way. This meta-data stores information about the state of information stored in granules. Moreover, concurrent interactions of the RMM must operate on the same data structures storing this metadata and on granules dedicated to the maintenance and management of Realms. Consequently, the RMM implementation needs to serialize access to these shared data structures. Correct synchronization of RMM commands is not only critical for the safety of the commands, but is also critical to the overall security properties offered by the RMM. This is because the invariants discussed in Sec. 3.3.1

are only valid if we can guarantee that, when operating on a Realm's metadata, that metadata cannot be altered by another RMM command executing concurrently on a different core. Note that this is a legitimate threat, since a malicious untrusted OS, executing on the Non-secure Host, is a threat actor within the Arm CCA threat model. As an example, consider the following two calls issued by the Non-secure Host in two different cores:

```
smc_rmi_realm_create(rd, ...) || smc_rmi_granule_undelegate(rd)
```

Here, the Host tries to create a Realm using the previously delegated granule with physical address `rd` as its realm descriptor metadata, and concurrently, the Host also attempts to move `rd` from the `REALM` PAS to the `NS` PAS. Without locking, commands are susceptible to a *time of check to time of use* (TOCTOU) vulnerability: the left command validates the preconditions on `rd`—ensuring that it is indeed a delegated physical address—and the right command modifies the PAS to `NS`—making the metadata contained in `rd` accessible to the Host—before the left can appropriately update the granule state.[13]

Locking mechanism. The RMM firmware executes at `EL2`, so the choice of synchronization mechanism must be simple, with the RMM implementation adopting spin-locks as the lock of choice. For a given granule metadata `g`, the RMM locks the granule if it finds it free, or it spins (using the *wait for event* instruction, `wfe`, in a loop to release the PE resource) until the lock is free. Upon obtaining the lock, it checks the metadata of the granule has the state expected by the command, in which case the lock operation exits successfully, or it releases the lock and exits reporting a precondition violation if the granule state is not the expected one. This pattern is captured by the code snippet for the `granule_try_lock` function (ignore lines 2, 5 and 8) in Listing 5.[14]

Observe that conservatively acquiring the lock and releasing it immediately if the granule state is not the one expected cannot lead to a new deadlock. That is: if there is a deadlock hazard, this is independent of the granule state of the granule.

Deadlock hazards. While this simple lock enforces mutual exclusion, there is a potential deadlock hazard. Consider the concurrent execution of two commands:

```
smc_rmi_rec_create(addr1, addr2, ...) || smc_rmi_rec_create(addr2, addr1, ...)
```

Ignoring the semantics of these commands, observe that their only difference is the order in which the granules at addresses `addr1` and `addr2` are passed as arguments. Whilst calling these two commands concurrently is bound to fail (since the preconditions that apply to the former granule are inconsistent with the ones that apply to the latter), the granules must be locked by the RMM to check their statuses. Importantly, as granules are provided as inputs by an untrusted Host, this is a route through which a Host may induce a *deadlock* in the RMM, for example, if the implementation naively locks the granules following the order of parameters. Whilst Arm CCA offers no availability guarantee to *Realms*, we wish to avoid obvious sources of deadlock in the RMM, if only to defend against buggy Hosts inadvertently inducing them.

One widely-known mechanism to prevent deadlocks is to acquire locks in a globally defined total order, with an ordering on the physical addresses of granules being an obvious candidate. Unfortunately, this is too simplistic, in our case: some granules, i.e., RTT and Data granules, that must be locked by RMM commands are only *discovered* by dereferencing the content of other granules. Updates to a Realm stage 2 page table by the RMM are the most common example of

[13]This is merely illustrative: in the implementation, granules in `REALM` PAS are *scrubbed* before transitioning to `NS` PAS.
[14]The implementations of `spinlock_acquire` and `granule_spinlock_release` is written in AArch64 assembly.

```
bool granule_try_lock(struct granule *g, enum
     granule_state expected) {
  cbmc_check_locking_discipline(g, expected);
  // Checks discipline via ghost state
  spinlock_acquire(&g->lock);
  cbmc_lock_granule(g, expected); // Update ghost
       state
  if(granule_get_state(g) != expected) {
    spin_lock_release(&g->lock);
    cbmc_unlock_granule(g); // Update ghost state
    return false;
  }
  return true;
}
```

Listing 5. Code snippet for `granule_try_lock`.

```
typedef struct locked_granule {
  struct granule *gr;
  enum granule_state locked_state;
  bool is_locked;
} g_meta;

struct locking_footprint {
  g_meta *locked[MAX_LOCKS];
  uint8_t current_lock;
  struct granule *rd;
  uint32_t next_rtt_index;
  uint8_t count_data_locks;
  uint8_t count_rtt_locks;
} lock_fpt;
```

Listing 6. Global locking footprint.

```
// Assume the granule g_tbl is locked.
struct granule *find_lock_next_level(struct granule *g_tbl, unsigned long ipa, long level)
{
  const unsigned long idx = s2_addr_to_idx(ipa, level); // Index to the `level` RTT
  locking_fpt.next_rtt_index = idx; // CBMC: update ghost state
  struct granule *g = __find_next_level_idx(g_tbl, idx); // Granule address at `idx`
  if (g) granule_try_lock(g, GRANULE_STATE_RTT);
  return g;
}
```

Listing 7. Code snippet for `find_lock_next_level`.

```
void cbmc_check_locking_discipline(struct granule *g, enum granule_state expected_state) {
  if (expected_state == GRANULE_STATE_RTT) {
    CBMC_ASSERT(lock_fpt.count_data_locks == 0, "rtt_after_data");
    if (lock_fpt.count_rtt_locks == 0) { // We are locking the root rtt
      CBMC_ASSERT(__is_parent(lock_fpt.rd, g), "rtt_root");
    } else { // We already hold an rtt lock, so last rtt locked must be the parent
      struct granule *parent = lock_fpt.locked[lock_fpt.current_lock]->gr;
      unsigned long recorded_index = lock_fpt.next_rtt_index;
      CBMC_ASSERT(__find_next_level_idx(parent, recorded_index) == g, "rtt_tree_structure"
          );
    }
  } else if (expected_state == GRANULE_STATE_DATA) {
    CBMC_ASSERT(lock_fpt.count_data_locks == 0, "data_after_data");
    if (lock_fpt.count_rtt_locks > 0) { // We hold an rtt lock, so the last must be the
        parent
      struct granule *parent = lock_fpt.locked[lock_fpt.current_lock]->gr;
      unsigned long recorded_index = lock_fpt.next_rtt_index;
      CBMC_ASSERT(__find_next_level_idx(parent, recorded_index) == g, "rtt_tree_structure"
          );
    }
  } else {
    CBMC_ASSERT(lock_fpt.count_rtt_locks + lock_fpt.count_data_locks == 0, "phase_change")
        ;
    for (int i = 0; i < lock_fpt.current_lock; i++)
      CBMC_ASSERT( !lock_fpt.locked[i] // i granules have been locked
          || !lock_fpt.locked[i]->is_locked // granule i remains locked
          || granule_addr(lock_fpt.locked[i]->gr) < granule_addr(g), // the address of
              granule @i is lower than the address of g
          "ordering");
} }
```

Listing 8. Code snippet for `check_locking_discipline`.

this. As a result, we instead introduce a locking discipline in two phases (not to be confused with two-phased locking) for commands, which:

(1) locks all granules passed as arguments to the command in physical address (PA) order, i.e., PA-order in Figure 8, and

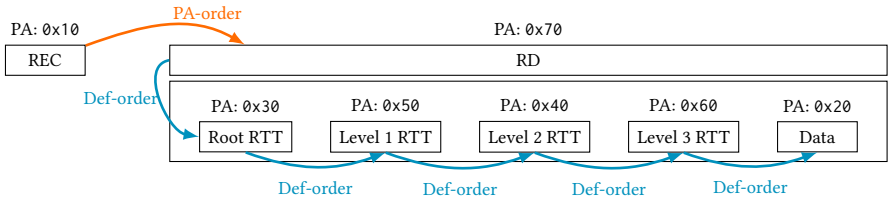

Fig. 8. Lock ordering. PA stands for physical address and Def for dereference.

(2) locks all granules that are referenced from previously locked granules, in the order that they are dereferenced i.e., Def-order in Figure 8.

Sec. 3 establishes an important *shape invariant* of the RMM state: Realm page tables always form a strict tree stemming from the root entry of the Realm descriptor granule metadata. As commands that dereference to-be-locked granules start traversal from a locked Realm descriptor, and its corresponding locked page table *root* granule, we can ensure that pointer chasing through the tree cannot lead to a deadlock (otherwise we have a shape invariant violation). Consequently, the RMM implements page table walks via a fine-grained hand-over-hand locking method (see [Vafeiadis and Parkinson 2007] for a description and an early proof of correctness of this mechanism). The code snippet to traverse a page table is in Listing 7. Ignoring the code only visible to CBMC (line 24), given a parent RTT table `g_tbl` and a source address to be mapped `ipa`, the function `find_lock_next_level` computes the index, and locks and returns the next level RTT table.

Importantly, there is a strong correspondence between a granule's state and whether it should be locked by its address (that is, its address must be known before locking any other granule), or should be locked by pointer chasing from another granule. Granules with a state indicating that they are to be locked by address must be passed as arguments to the command, and no granule that should be locked by dereferencing pointers (that is, granules expected to be in the states `RTT` or `Data`) should be passed as arguments to the command. This property is easy to prove using CBMC.

A Model of the Locking Discipline. To ensure that the discipline above implies deadlock freedom, we formalized an idealized notion of RMM commands in HOL4, generalizing all commands implemented in Arm's RMM. We consider an unbounded number of granules with their corresponding RMM states as described in Sec. 3, and an unbounded number of concurrent RMM commands locking and unlocking granules according to the discipline, enforced by construction in the predicate `RmmStep` in the statement below. We also assume all granule states as well formed, that is: all RTT granules conform to a strict tree (or rather, a forest). Using this model we prove the following theorem, which expresses the property that no deadlock state is reachable, as long as all commands eventually release all acquired locks:

$$\vdash \forall\, init\ state.\ \texttt{InitRmmState}\ init \wedge \texttt{RmmNStep}\ init\ state \Rightarrow \neg\texttt{Deadlock}\ state$$

A deadlocked state is one in which one or more commands are blocked due to a cycle over a *waiting-on-lock* relation.

The proof proceeds via a simple induction on the number of transitions taken by well-disciplined commands in our idealized model. The key invariant of the proof is that commands only ever attempt to lock granules per the order described above.[15]

[15]Proofs will be provided as additional material for reviewing purposes.

Checking the Locking Discipline. We established that discipline-respecting commands do not run into deadlocks, using HOL4 above. Yet, we must prove that all RMM commands in the implementation follow the locking discipline. Fortunately, since this discipline is on a per-command basis, we do not need to consider concurrent executions when checking that a command satisfies it. To verify this discipline, we use *ghost state* and the *shadowing mechanism*[16] of CBMC to check that whenever a lock is acquired or released by a command, the discipline is respected. The ghost state captures the locking history of a command through its locking footprint, `lock_fpt` in Listing 6, comprising: (1) the locking trace `locked` and the most recently locked granule in the trace indexed by `current_lock`, (2) bookkeeping information, `rd` and `next_rtt_index`, for verifying Def-order, and (3) counters to the discovered data and RTT granules, `count_*`, for checking phases: that is whether we are currently locking by address order, or by dereference. Given the ghost state, one property we verify is the lock ordering via function `check_locking_discipline` in Listing 8. When locking a RTT or Data granule, either the previously locked RD `lock_fpt.rd` is the parent to the about-to-lock root RTT (line 34) or the most recently locked RTT granule is the parent to the next level RTT or Data (lines 38 and 45, respectively). For other granule states, locking must follow the PA-order (line 52). The two counters further check the locking phase: non-discovered granules (line 48) are followed by RTT granules (line 32) and then Data granules. To enable the lock ordering check above, we have to shadow `granule_try_lock` and `find_lock_next_level`. In particularly, the lock ordering check happens at line 2 before the original RMM lock implementation and necessary ghost state updates are introduced at lines 5, 8 and 24. Using the same methodology, we further prove: (1) no unlock operation is performed on a lock that was not acquired before, (2) that in every path, all locks are eventually released, and (3) that no lock is released more than once. These implementation-specific properties proven by CBMC, together with the guarantees established in our HOL4 model[17], imply that the implementation cannot be induced to deadlock through any possible combination of calls to RMM commands by the Host.

6 TRUSTED FIRMWARE EXPLORER (TFX)

Our HOL4 and CBMC verification flows do not consider concurrency beyond deadlock freedom (e.g., interrupt or exception injection), nor architectural state. We address these aspects through system level tests with high *coverage* and *expressivity*. Implementing the necessary level of instrumentation at the system level, without sacrificing the *fidelity* of the test infrastructure, is the primary difficulty here. For example, testing a concurrent interleaving of commands running on multiple PEs could be achieved via manual code instrumentation, but aside from being laborious and not scaling, this would lead to a divergence between the production system and the system under test.

To address this, we developed *Trusted Firmware Explorer* (TFX), a system-level testing, debugging and introspection infrastructure with emphasis on concurrency control. At the lowest level, TFX connects to the system under test via Arm's *Iris Debug Interface* [Arm Ltd. 2022b] and exposes a Python scripting interface to the user, providing a "bird's eye view" of the system under test with the ability to control PE's and inspect their full architectural state. This infrastructure is not specific to Arm CCA, but atop it. TFX provides three Arm CCA-specific support libraries: support for dynamic injection of RMI commands, and common flows such as creating Realms from ELF binaries; support for C-like state inspection of the RMM; and a library for concurrency control.

State inspection. TFX's Python interface also provides C-like access to RMM state, and access to error codes and other architectural constants; the former parses the Arm CCA MRS, and the latter

[16]The shadowing mechanism enables us to provide different versions of lock-related functions that manipulate ghost state.
[17]The implementation properties verified by CBMC are in fact stronger than the theorem in HOL4 model.

```
tfx.run(tfx.rmi(cpuid=0). RMI_Granule_Delegate(addr=foo))
assert tfx.cpus[0].read_register("X0") == tfx.spec.RmiStatusCode.RMI_SUCCESS
```

Listing 9. Injecting an RMI call into the NS host using TFX and checking return code.

```
realm = Realm(tfx, elf_img='payload
    .elf')
rec = REC(realm, mpidr=0)
realm.create(cpuid=0)
rec.create(cpuid=1)
```

Listing 10. High-level flows for creation of Realms and RECs in TFX.

```
realm = Realm(tfx,elf_img='payload.elf'); realm.create
    (cpuid=0)
# Create two RECs concurrently on two CPUs
rec = [REC(realm, mpidr=i) for i in range(2)]
tfx.zip([rec[i].f_create(cpuid=i) for i in range(2))
```

Listing 11. Randomly interleaving the creation of two RECs for the same Realm using TFX.

```
# Construct Realm creation command ...
call = tfx.rmi(cpuid=0).RMI_Realm_Create(rd=g_rd, params_ptr=g_params, may_fail=True)
# ... run until parameter granule is about to be copied from
tfx.until(call, tfx.rmm.is_at("memcpy_ns_read", cpuid=0))
tfx.model.step(count=17) # Run a few steps to get into copying
# Now issue a delegate on CPU1, moving parameter granule to Realm PAS
tfx.run(tfx.rmi(cpuid=1).RMI_Granule_Delegate(addr=g_params))
# NS copy should fault now; check via breakpoint in fault handler
tfx.until(call, tfx.rmm.is_at("handle_rmm_trap", cpuid=0))
tfx.run(call) # Finish call
assert call.result.result_code == tfx.spec.RmiStatusCode.RMI_ERROR_INPUT
```

Listing 12. A failing TFX test: an `NS` parameter granule in `RMI_Realm_Create` is moved to `Realm` PAS whilst being read by the RMM.

parses the RMM binary. Together, with access to system registers provided by Iris, TFX allows the user to specify and dynamically check invariants of the architectural and RMM-internal state.

Concurrency Control. TFX provides fine-grained concurrency control by decoupling the definitions of flows, preemption points, and scheduling decisions, analogous to how multiple logically independent applications are multiplexed in an operating system, based on preemption points given by timers and traps and a scheduling strategy. In TFX, a flow is modeled as a Python *generator*: a resumable function that can `yield` early and continue execution later. Whenever a flow `yield`s, control returns to the parent flow whose scheduler decides which flow to continue next.

In the simplest case, consider two RMI call flows `F` and `G` running on separate PEs, and a parent flow orchestrating them: if `F` and `G` do not `yield` prior to completion, no interesting interleaving is possible, akin to applications finishing work prior to a timer preemptively interrupting them in an operating system. However, when introducing a preemption condition (such as a timeout or breakpoint) to `F` and `G`, they will `yield` potentially many times prior to completion, allowing the parent flow's scheduler to construct nontrivial interleavings. This flexibility allows for both the exploration of *specific* (e.g., Listing 12) as well as *random* (e.g., Listing 11) interleavings, similar to property-based testing tools, like Quickcheck [Claessen and Hughes 2000].

Applications of TFX. Aside from being a valuable exploration vehicle for RMM development, TFX supplements our verification methodologies, empirically validating the correctness of the RMM implementation in situations not easily amenable for verification: interrupt and exception injection (e.g., Listing 12), concurrent execution of attestation flows, and similar. These tests are expanding, and will grow to also include the checking of non-trivial invariants spanning the software-hardware boundary, and also potentially finding application in the testing of other firmware components.

7 RELATED WORK

We focus on works related to OS verification and formal methods for Confidential Computing.

OS verification. Previous work has presented verified operating system kernels [Bevier 1989; Chen et al. 2018; Gu et al. 2011, 2019, 2016; Klein et al. 2010, 2009; Sewell et al. 2011], separation kernels [Dam et al. 2013b], and hypervisors [Dam et al. 2013a; Li et al. 2021a,b; McCoyd et al. 2013; Nemati et al. 2015; Tao et al. 2021], all largely using interactive theorem proving, given the complexity and scale of the task, though we note some use of deductive tools [Blanchard et al. 2015, 2018; Chong and Jacobs 2021; Haque et al. 2020; Leinenbach and Santen 2009; Mangano et al. 2016]. We use model checking for verifying the functional properties of an implementation, reserving HOL4 for reasoning about security and consistency properties of the specification.

In an industrial context, where the codebase being verified is being co-developed concurrently with the verification flow, this approach has advantages. Model checking allows us to reason directly on the source code of the implementation, rather than a model or embedding within a theorem prover, as CBMC's front end is especially liberal, accepting a large fragment of C including constructs which often pose problems for verification tools. This has the advantage of eliminating any translation from our TCB, which consists of only CBMC and support code, though we must trust CBMC's understanding of C semantics. Moreover, model checking with CBMC *seems* simply easier to understand than theorem proving for product engineers—especially in the semiconductor industry, where model checking is commonly used—and our support code can be audited by product engineers reusing their existing C expertise. Model checking is also flexible enough to maintain a working verification flow on a rapidly-changing codebase, easing integration into CI flows. In contrast, the slower-moving specification of the behavior of the RMM can be effectively modeled within an interactive theorem prover. Finally, as explained above, CBMC verification harnesses for Arm's prototype RMM implementation are automatically generated from the MRS, which is also used to generate the specification. Generated verification conditions can be kept separate from the RMM codebase, leading to cleaner code, and a clearer separation of concerns between verification and implementation of the RMM. Doing this with a proof assistant would be much more difficult.

Formal methods and Confidential Computing. Intel established linearizability for SGX instructions using model checking and *Accordion*, a custom DSL [Leslie-Hurd et al. 2015]. Attestation protocols have also been verified [Sardar et al. 2020a, 2021, 2020b]. We ignore attestation in this work.

Komodo [Ferraiuolo et al. 2017] implements a protected execution environment using Arm TrustZone, and implemented in Armv7 machine code. The functional correctness of the security monitor, and the confidentiality and integrity (up-to explicit declassification and endorsement events) are verified in the Dafny deductive verification tool [Leino 2010], using an embedding of the semantics of Arm machine code, representing the closest previous work to our own. However, the RMM is more complex than the Komodo security monitor, as it must expose sufficient functionality to untrusted hypervisors to blindly manager a full virtual machine, supporting a wide range of operating systems, both commercial and open-source. Komodo, on the other hand, implements Intel SGX-style protected libraries, which requires a simpler implementation, but are more limited.

Serval [Nelson et al. 2019] was applied to the implementations of the security monitors for Komodo and RISC-V Keystone, finding, in the latter, a range of implementation and specification bugs, with fixes adopted by the Keystone project. This verification effort targeted a point-release of Keystone, with no continuous re-verification of the implementation as the codebase changes, and before verifying code, a specification for the Keystone security monitor had to be written.

An important related work [Li et al. 2022] verifies an early snapshot of the RMM prototype using the CertikOS [Gu et al. 2018] toolchain for Coq. There are several important differences compared

to our work. (1) Li et al. [2022] did not formalize the pre/postconditions of the RMM ABI [Arm Ltd. 2022c]. Instead, functional correctness was established with respect to an idealized RMM that Li et al. [2022] created directly from the C code itself. A major advantage of our work is that we have verified the RMM implementation against the official specification [Arm Ltd. 2022c]. Furthermore, our HOL4 results provide security guarantees that extend to *all* RMM implementations. (2) Li et al. [2022] was a one off evaluation of formal methods for Arm CCA, and changes to the RMM were not tracked during their verification effort. As such, the verified code quickly became outdated: it implements an early version of the ABI, which has subsequently undergone numerous changes. (The ABI now provides a different set of commands and more features.) By contrast, our work has tracked changes and our results will apply to the production RMM. (3) Our work takes a more practical and holistic approach, and has yielded better results than Li et al. [2022] in terms of identifying issues and supporting the active development of the RMM codebase. We support rapid verification of the evolving C code, as well as debugging at the hardware/software interface. In summary, Li et al. [2022] presents initial results about a certified RMM point-release, but says nothing about the final RMM product, with goals largely orthogonal to ours.

8 CONCLUSION AND FUTURE WORK

Arm CCA is a recently-announced extension for the Armv9-A architecture. Given the exacting threat model of Arm CCA, applying varied verification techniques allows us to boost confidence in the specification and implementation. Since both are under active development, we have shown how interactive theorem proving helps validate fundamental properties of the specification, including coherence of the commands, well-formedness invariants, and isolation between Realms. Moreover, we have shown how these properties can be used alongside the specification to generate verification harnesses for the CBMC model checker and thus guarantee that the reference implementation satisfies the specification, establishing a refinement argument.

In addition to HOL4 and CBMC, we also built tools to test the interface between the RMM and RME hardware implementation. TFX (see Sec. 6) allowed us to repeatably test and validate execution flows, with specific or exhaustive coarse-grained interleavings of Host and RMM commands, for property checking, and was useful for validating typical command flows for Arm CCA.

Our work verifying the Arm CCA specification and a prototype RMM is the most extensive use of formal methods for a Confidential Computing implementation with commercial viability. Yet, there are aspects of Arm CCA that we did not consider, and leave open for future work:

Data-race freedom, and atomicity. We mostly considered the sequential verification for commands, with the exception of the deadlock-freedom analysis described in Sec. 5. We aim to prove the absence of data races and the atomicity of RMM commands, cognizant of the AArch64 usage of atomic Armv8-A assembly, using a reduction to sequential CBMC verification. Whilst we were careful to ensure that the verification of RMM command pre/postconditions in CBMC was correct, in a concurrent setting, we would like to apply reduction techniques like CIVL [Kragl and Qadeer 2021] to make commands as atomic as possible, and use rely-guarantee style reasoning [Jones 1983] to ensure the stability of pre/postconditions in the case where commands are not fully atomic.

Composing hardware and software verification. The hardware specification and subsequent implementations by Arm partners is critical to the security of Arm CCA. Arm, however, has substantial existing expertise in validating and verifying hardware specifications and implementations, using a mixture of industrial model checkers, and internal specification and verification tools [Reid 2016; Reid et al. 2016]. However, *composing* the verification of hardware, and software that executes atop it and which relies crucially on security properties that it provides, remains a challenge. We aim to further investigate the composition of our software and hardware verification efforts.

Attestation. Arm CCA specifies an attestation token format, but does not constrain the associated attestation protocol used to authenticate that token by skeptical challengers. Arm's own implementations of Arm CCA will, however, need to specify a particular protocol. We aim to use protocol model checking tools, such as Blanchet [2013] and Meier et al. [2013], to verify this.

Noninterference. The memory isolation result presented in Sec. 3.3.2 falls short of a full proof of confidentiality and integrity-preservation for Arm CCA, both of which are typically formally phrased as hyperproperties like noninterference [Goguen and Meseguer 1982], or one of its many refinements. Useful programs require external communication to retrieve inputs and communicate outputs, and any communication channel precludes an *absolute* statement of confidentiality or integrity, as Realms may *declassify* or *endorse* arbitrary data, and with this, confidentiality and integrity must be relativized against *assumptions* about the software executing inside a Realm. Additionally, there is *implicit* information flow as a Realm executes, for example, through page faults which is outside of the Realm's control. We leave formally establishing stronger security properties for Arm CCA, beyond our memory isolation result, as future work.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Amazon Inc. Last viewed June 2022. AWS CBMC viewer. https://github.com/model-checking/cbmc-viewer.

Arm Ltd. 2008. ARM Security Technology. Building a Secure System using TrustZone Technology. https://documentation-service.arm.com/static/5f212796500e883ab8e74531.

Arm Ltd. 2021. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

Arm Ltd. 2022c. Realm Management Monitor *beta0* specification. https://developer.arm.com/documentation/den0137/a/ Accessed 25th October 2022, final version to be published 2022.

Arm Ltd. Last viewed June 2022a. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest.

Arm Ltd. Last viewed June 2022b. Introducing Iris, the new generation of debug and trace interface in Arm Models. https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/iris-the-new-debug-and-trace-interface-in-arm-models.

William R. Bevier. 1989. Kit: A Study in Operating System Verification. *IEEE Trans. Software Eng.* 15, 11 (1989), 1382–1396. https://doi.org/10.1109/32.41331

Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue. 2015. A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C. In *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings (Lecture Notes in Computer Science, Vol. 9128)*, Manuel Núñez and Matthias Güdemann (Eds.). Springer, 15–30. https://doi.org/10.1007/978-3-319-19458-5_2

Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2018. A Lesson on Verification of IoT Software with Frama-C. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*. IEEE, 21–30. https://doi.org/10.1109/HPCS.2018.00018

Bruno Blanchet. 2013. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 8604)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.). Springer, 54–87. https://doi.org/10.1007/978-3-319-10082-1_3

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2018. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. *J. Autom. Reason.* 61, 1-4 (2018), 141–189. https://doi.org/10.1007/s10817-017-9446-0

Nathan Chong and Bart Jacobs. 2021. Formally Verifying the FreeRTOS IPC Mechanism. In *Embedded World Conference*. 202–211. https://www.amazon.science/publications/formally-verifying-freertos-interprocess-communication-mechanism

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

George Coker, Joshua D. Guttman, Peter A. Loscocco, Amy L. Herzog, Jonathan K. Millen, Brian O'Hanlon, John D. Ramsdell, Ariel Segall, Justin Sheehy, and Brian T. Sniffen. 2011. Principles of remote attestation. *Int. J. Inf. Sec.* 10, 2 (2011), 63–81.

https://doi.org/10.1007/s10207-011-0124-7

Byron Cook, Björn Döbel, Daniel Kroening, Norbert Manthey, Martin Pohlack, Elizabeth Polgreen, Michael Tautschnig, and Pawel Wieczorkiewicz. 2020. Using model checking tools to triage the severity of security bugs in the Xen hypervisor. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 185–193. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_26

Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. 2013b. Formal verification of information flow security for a simple arm-based separation kernel. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 223–234. https://doi.org/10.1145/2508859.2516702

Mads Dam, Roberto Guanciale, and Hamed Nemati. 2013a. Machine code verification of a tiny ARM hypervisor. In *TrustED'13, Proceedings of the 2013 ACM Workshop on Trustworthy Embedded Devices, Co-located with CCS 2013, November 4, 2013, Berlin, Germany*, Ahmad-Reza Sadeghi, Frederik Armknecht, and Jean-Pierre Seifert (Eds.). ACM, 3–12. https://doi.org/10.1145/2517300.2517302

Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 287–305. https://doi.org/10.1145/3132747.3132782

Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 11–20. https://doi.org/10.1109/SP.1982.10014

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: a certified kernel for secure cloud computing. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, Haibo Chen, Zheng Zhang, Sue Moon, and Yuanyuan Zhou (Eds.). ACM, 3. https://doi.org/10.1145/2103799.2103803

Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building certified concurrent OS kernels. *Commun. ACM* 62, 10 (2019), 89–99. https://doi.org/10.1145/3356903

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 646–661. https://doi.org/10.1145/3192366.3192381

Inzemamul Haque, Deepak D'Souza, Habeeb P, Arnab Kundu, and Ganesh Babu. 2020. Verification of a Generative Separation Kernel. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12302)*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer, 305–322. https://doi.org/10.1007/978-3-030-59152-6_17

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619. https://doi.org/10.1145/69575.69577

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. https://doi.org/10.1145/1743546.1743574

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. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 207–220. https://doi.org/10.1145/1629575.1629596

Bernhard Kragl and Shaz Qadeer. 2021. The Civl Verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 143–152. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_23

Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5850)*, Ana Cavalcanti and Dennis Dams (Eds.). Springer, 806–809. https://doi.org/10.1007/978-3-642-05089-3_51

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Rebekah Leslie-Hurd, Dror Caspi, and Matthew Fernandez. 2015. Verifying Linearizability of Intel® Software Guard Extensions. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 144–160. https://doi.org/10.1007/978-3-319-21668-3_9

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021a. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3953–3970. https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021b. A Secure and Formally Verified Linux KVM Hypervisor. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1782–1799. https://doi.org/10.1109/SP40001.2021.00049

Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 465–484. https://www.usenix.org/conference/osdi22/presentation/li

Linaro Ltd. 2022a. The Open Portable Trusted Execution Environment (OP-TEE). https://www.op-tee.org Accessed 23rd June 2022.

Linaro Ltd. 2022b. TrustedFirmware-A. https://www.trustedfirmware.org/projects/tf-a/.

Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. 2016. Formal Verification of a Memory Allocation Module of Contiki with Frama-C: A Case Study. In *Risks and Security of Internet and Systems - 11th International Conference, CRiSIS 2016, Roscoff, France, September 5-7, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10158)*, Frédéric Cuppens, Nora Cuppens, Jean-Louis Lanet, and Axel Legay (Eds.). Springer, 114–120. https://doi.org/10.1007/978-3-319-54876-0_9

Michael McCoyd, Robert Bellarmine Krug, Deepak Goel, Mike Dahlin, and William D. Young. 2013. Building a Hypervisor on a Formally Verifiable Protection Layer. In *46th Hawaii International Conference on System Sciences, HICSS 2013, Wailea, HI, USA, January 7-10, 2013*. IEEE Computer Society, 5069–5078. https://doi.org/10.1109/HICSS.2013.121

Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 225–242. https://doi.org/10.1145/3341301.3359641

Hamed Nemati, Roberto Guanciale, and Mads Dam. 2015. Trustworthy Virtualization of the ARMv7 Memory Subsystem. In *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 8939)*, Giuseppe F. Italiano, Tiziana Margaria-Steffen, Jaroslav Pokorný, Jean-Jacques Quisquater, and Roger Wattenhofer (Eds.). Springer, 578–589. https://doi.org/10.1007/978-3-662-46078-8_48

Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. 2013. A survey of security issues in hardware virtualization. *ACM Comput. Surv.* 45, 3 (2013), 40:1–40:34. https://doi.org/10.1145/2480741.2480757

Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 161–168. https://doi.org/10.1109/FMCAD.2016.7886675

Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Processors with ISA-Formal. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 42–58. https://doi.org/10.1007/978-3-319-41540-6_3

Muhammad Usama Sardar, Rasha Faqeh, and Christof Fetzer. 2020a. Formal Foundations for Intel SGX Data Center Attestation Primitives. In *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12531)*, Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony (Eds.). Springer, 268–283. https://doi.org/10.1007/978-3-030-63406-3_16

Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. 2021. Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification. *IEEE Access* 9 (2021), 83067–83079. https://doi.org/10.1109/ACCESS.2021.3087421

Muhammad Usama Sardar, Do Le Quoc, and Christof Fetzer. 2020b. Towards Formalization of Enhanced Privacy ID (EPID)-based Remote Attestation in Intel SGX. In *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*. IEEE, 604–607. https://doi.org/10.1109/DSD51259.2020.00099

Thomas Sewell, Simon Winwood, Peter Gammie, Toby C. Murray, June Andronick, and Gerwin Klein. 2011. seL4 Enforces Integrity. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6898)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer, 325–340. https://doi.org/10.1007/978-3-642-22863-6_24

Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 866–881. https://doi.org/10.1145/3477132.3483560

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4703)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.). Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

Received 2022-10-28; accepted 2023-02-25