Detecting Inconsistencies in Arm CCA's Formally Verified Specification

Changho Choi
Samsung Research
Seoul, Republic of Korea
ch754.choi@samsung.com

Xiang Cheng
Georgia Institute of Technology
Atlanta, GA, USA
cxworks@gatech.edu

Bokdeuk Jeong
Samsung Research
Seoul, Republic of Korea
bd.jeong@samsung.com

Taesoo Kim*
Samsung Research
Seoul, Republic of Korea
Georgia Institute of Technology
Atlanta, GA, USA
taesoo@gatech.edu

Abstract

Formal verification offers strong guarantees of correctness, robustness, and security. However, these guarantees depend on specification correctness, and even minor flaws can invalidate proofs and introduce critical vulnerabilities. We present Scope, an automated system that identifies specification inconsistencies by combining formal modeling with rule-based consistency checking. Unlike traditional approaches that rely on implementations, Scope treats the specification as the sole ground truth. It translates the specification into a machine-verifiable model using Verus and SMT solvers, then detects inconsistencies in success/failure conditions, dependency rules, and state transitions. We apply Scope to the Realm Management Monitor (RMM) specifications for Arm's Confidential Compute Architecture (CCA), uncovering 35 previously unknown bugs—including security-critical flaws in ABI semantics and missing state transitions—all confirmed by Arm. Compared to modern LLM-based tools, Scope improves inconsistency-detection precision by 7× over GPT-o1 and up to 40× over leading chat models (LLaMA 3.1, GPT-4o, Claude 3.7).

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Consistency**; • **Security and privacy → Trusted computing**.

Keywords: Specification validation; Formal methods; Automated reasoning; Confidential computing; Arm CCA

*Corresponding author

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
ASPLOS '26, Pittsburgh, PA, USA.
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790152

ACM Reference Format:
Changho Choi, Xiang Cheng, Bokdeuk Jeong, and Taesoo Kim. 2026. Detecting Inconsistencies in Arm CCA's Formally Verified Specification. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 21–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3779212.3790152

1 Introduction

Formal verification has gained traction as a key enabler for providing correctness, robustness, and security for decades. Owing to its mathematical rigor, it is sometimes regarded as a silver bullet for building bug-free systems [37]. Numerous systems have adopted this technique, including OS kernels [20, 36, 40, 54, 77], file systems [10, 16–18, 70], network systems [79–81], hypervisors [47, 48, 72–74], and compilers [45]. Advances in computer-aided verification now enable its application to systems once deemed too complex or large to verify.

Strong guarantees of formal verification, however, come at the cost of heavy reliance on the specification. A bug in the specification can invalidate a proof and render an implementation vulnerable, even when it satisfies the specification, while giving the illusion of a secure system [35]. For example, a formally verified smart contract paid $1 million bounty due to a critical vulnerability [75] rooted in a vacuous invariant [39]. Although prior work has sought to enhance the trustworthiness of specifications by formally proving their properties [22, 27, 32, 49, 53, 55, 65, 69, 71] or testing them against implementations [12, 21, 26, 30, 31, 64, 66], the former is difficult to maintain under frequent design changes, and the latter requires an implementation that may not yet exist. As specifications continue to grow in size, ensuring their trustworthiness becomes increasingly challenging.

In this work, we propose Scope, an automated approach for detecting inconsistencies in a specification. Scope exploits a fundamental property of specifications: if two related

statements in the same document contradict, at least one is incorrect. Since each statement is intended to be accurate and consistent, combining them must not yield an inconsistency. Specifically, Scope utilizes how the architects view the system [35, 65], which is expressed through summarized tables or figures in the specification, and validates whether the rest of the specification aligns with it.

We apply Scope to the RMM specification [8]. RMM is the firmware that manages resources and enforces security policies in Arm's CCA. We selected the RMM specification as our target because it shares a similar structure with other specification documents (e.g., using the Architecture Specification Language (ASL) [9]) and several formal verification efforts have already been conducted [32, 49]. In particular, validating the correctness of the RMM specification is critically important, as it is the Trusted Computing Base (TCB) underpinning the security guarantees of the entire CCA system which will cope with rising computations in data centers [14]. However, effectively validating this specification presents several key challenges:

- **C1. Absence of implementation.** Arm's design review process mandates specification development prior to implementation. Consequently, validating these specifications is challenging, as reference implementations—typically required for comparison—are not yet available.
- **C2. Frequent ABI updates.** Arm regularly updates the specification with new features and Application Binary Interfaces (ABIs). As a result, an automated validation technique is necessary to accommodate frequent changes.
- **C3. Lack of predefined validation rules.** Given the complexity of the CCA threat model and its security requirements [5], there are no clear rules or heuristics for systematically detecting and validating bugs in the specification that spans hundreds of pages.

Scope addresses these challenges by using the specification as the sole authoritative source (**C1**). It adapts the Machine Readable Specification (MRS) [32, 64] to automatically parse and analyze the RMM specification, ensuring it remains effective as the specification evolves (**C2**). Although Arm's internal MRS resources (such as YAML files and CCA-specific tools) are not publicly available, Scope uses a PDF parser to extract relevant ASL content from the RMM specification. It then converts these ASL representations into a form suitable for Verus [42, 43], a formal verification framework for reasoning about large-scale systems. By leveraging Verus and SMT (Satisfiability Modulo Theories) solvers [24], Scope systematically detects inconsistencies between ABI semantics and their summarized views. Additionally, Scope applies heuristic rules tailored to the unique characteristics of the RMM specification to further validate extracted components (**C3**). To our knowledge, Scope is the first automated validation process that enables external CCA users to reconstruct a formal RMM model using uninterpreted functions [15]—a capability previously unavailable outside Arm.

We evaluated Scope on multiple versions of the RMM specification, discovering *35 previously unknown* inconsistencies, some persisting for up to 33 months. Scope continues to validate ongoing updates of the RMM specification, demonstrating its benefits in automating frequent, incremental changes. Compared with existing formal verification approaches (e.g., VIA [49], Arm's model checking [32]) and state-of-the-art Large Language Models (LLMs) (GPT-4o, Claude-3.7, GPT-o1, Deepseek-R1), Scope achieves broader coverage—72.8% of ABIs across versions—and substantially higher inconsistency-detection precision, improving from 8.00% (best LLM) to 61.90%. These results confirm Scope's effectiveness in reducing engineering effort and strengthening the trustworthiness of critical specifications.

In summary, our contributions are:

- **End-to-end, automated tool.** We present Scope, an automated framework for detecting inconsistencies in the RMM specification, identifying 35 inconsistencies across multiple versions. Compared to the latest LLM agents, Scope improves precision by over 7×, from 8.00% to 61.90%.
- **New methodology.** Our approach minimizes testing dependencies by operating solely on specification documents. Scope detects inconsistencies without relying on implementations and efficiently handles frequent updates to the RMM specification, including new ABIs.
- **Open source.** The source code for Scope is publicly available in https://github.com/islet-project/scope.

2 Background

2.1 RMM specification for Arm CCA

CCA is a hardware-software architecture introduced in Armv9-A that enables hypervisor-based confidential computing. At its core, the Realm Management Extension (RME) is a hardware feature that enforces strong isolation through a new security state: the Realm. The RMM, a crucial software component of CCA, operates at the highest level within the Realm security state. It is responsible for managing Realms running on it by utilizing the hardware capabilities provided by RME. The RMM specification [8] describes its ABIs including the Realm Management Interface (RMI) for the Host, Realm Services Interface (RSI) for Realms, and Power State Control Interface (PSCI), also for Realms. The specification is structured into four main parts.

Part A: Architecture. The Architecture part introduces the core concepts of the architecture, such as Realms, granules, and Realm Excution Context (REC)s, along with their valid state transitions, which are triggered by specific ABI calls. Among these concepts, a granule is the smallest unit of memory that can be delegated from the NS Host to the Realm world. Another important concept is the definition of

Realm IPA State (RIPAS) and Host IPA State (HIPAS), which represent the memory access states from the perspectives of the Realm and the Host, respectively.

The Realm's Intermediate Physical Address (IPA) space is divided into Protected and Unprotected regions, with RIPAS values applicable only to Protected IPAs. Common RIPAS values include: (1) `EMPTY`, indicating that the IPA is not mapped; (2) `RAM`, meaning the IPA is accessible as Realm RAM; and (3) `DESTROYED`, where the IPA has been reclaimed by the Host and is no longer accessible to the Realm. In contrast, HIPAS reflects the Host's view of memory, with typical values such as: (1) `ASSIGNED(_NS)`, where the IPA is associated with a granule and assigned to either the Realm or the Host; and (2) `UNASSIGNED(_NS)`, indicating that the IPA is not associated with any granule. RMM uses the combination of RIPAS and HIPAS to securely manage memory ownership and transitions between the Realm and the Host.

Part B: Interface. The Interface part defines each RMM command in terms of the following elements: (1) a function identifier (FID), (2) input/output values, (3) context values derived from the inputs and outputs, (4) success and failure conditions, and (5) a command footprint specifying the parts of the system state affected by the command (Table 4). This part also introduces command condition functions, which are logical expressions used for evaluation.

Part C: Types. The Types section provides detailed definitions for all data types used throughout the architecture and interface.

Part D: Usage. The Usage part illustrates how the RMM architecture can be utilized by both the Host and Realms, using flow diagrams that depict sequences of RMM ABIs.

The Interface and Types parts define logical elements such as data types, command condition functions, and commands. In contrast, the Architecture and Usage parts are written primarily in human-readable text, and make use of tables, state transition diagrams, and sequence diagrams.

2.2 Verus: a formal verification tool

Verus is an SMT-based formal verification tool originally designed for low-level system software written in Rust.

It allows developers to write specifications and machine-checked proofs directly through lightweight annotations and run proofs in SMT solvers to identify any violations. Verus offers a set of essential language constructs for expressing program properties, including preconditions (`requires`), postconditions (`ensures`), spec functions (`spec`) for defining desired behavior, and proof functions (`proof`) for verifying those properties. Additional annotations such as assertions (`assert`) support intermediate reasoning within proofs.

All such annotations are treated as ghost code, used exclusively for verification and erased at compile time, allowing seamless integration with the existing codebase.

RMI Command	RIPAS Dep.	HIPAS Dep.	New RIPAS	New HIPAS
DATA_DESTROY	EMPTY	ASSIGNED	Unchanged	UNASSIGNED
DATA_DESTROY	RAM	ASSIGNED	DESTROYED	UNASSIGNED
RTT_INIT_RIPAS	EMPTY	UNASSIGNED	RAM	Unchanged

Table 1. Dependency of RMI command execution on RIPAS and HIPAS values excerpted from the RMM specification (1.0-eac5).

B4.3.18.2 Failure conditions

ID	Condition
rtte_state	`pre: walk.rtte.state != UNASSIGNED` `post: ResultEqual(result, RMI_ERROR_RTT, walk.level)`
top_gran	`pre: !AddrIsGranuleAligned(top)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

Table 2. Failure conditions of RMI_RTT_INIT_RIPAS excerpted from the RMM specification (1.0-eac5). The shaded area highlights the failure condition associated with a HIPAS value. The full table is shown in Appendix A.

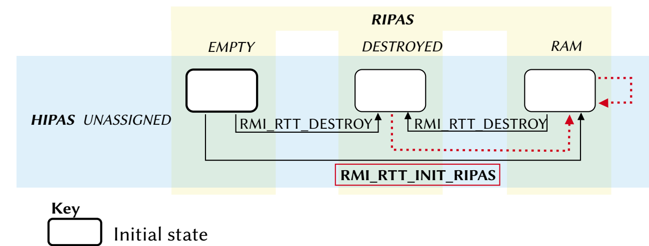

Figure 1. Summary of HIPAS and RIPAS changes at a Protected IPA, excerpted from the RMM specification (1.0-eac5). Red dotted lines indicate missing flows for RMI_RTT_INIT_RIPAS.

2.3 Contradiction in propositional logic

Detecting contradictions by identifying unsatisfiable logical conditions in propositional logic forms the foundation of Scope's formal reasoning approach. To explain this concept, we begin with a simple example. In propositional logic, a formula is composed of atoms, negation ($\neg$), and logical connectives ($\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$) where each atom represents a propositional assertion that is either TRUE or FALSE [41]. Consider the two formulas: a, $\neg a \wedge b$. Each can be satisfied independently by assigning values { a = TRUE } and { a = FALSE, b = TRUE }, respectively. However, when these formulas are combined as a single formula ($a \wedge \neg a \wedge b$), no interpretation can satisfy the result. The contradiction arises from the presence of both a and $\neg a$ connected by conjunction ($\wedge$), making the overall formula unsatisfiable.

3 Motivating Examples

We begin with a motivating example showing how inconsistencies can arise from summarized tables or figures. Table 1 outlines the semantics of each command with respect to the

RIPAS and HIPAS values, capturing its preconditions (*Dependency* columns) and postconditions (*New* columns). According to Table 1, successful execution of `RMI_RTT_INIT_RIPAS` requires RIPAS to be in the `EMPTY` state and HIPAS in the `UNASSIGNED` state. Upon completion, RIPAS transitions to `RAM`, while HIPAS remains unchanged. However, a discrepancy emerges when the failure conditions in Table 2 are considered: the only failure condition listed for the same command concerns HIPAS, with no reference to RIPAS. [1] This omission implies that RIPAS being `EMPTY` is not a required precondition, directly contradicting the dependency specified in Table 1.

A subsequent consultation with an Arm architect confirmed that Table 1 was incorrect: the precondition requiring RIPAS to be `EMPTY` was wrong. It has since been corrected to `None` and documented under FENIMORE-864. This clarification affected downstream components that depended on the original specification. Another summarized view, shown in Figure 1, was based on the previous (incorrect) version of the table and was subsequently updated. The earlier figure depicted the command as permitted only when RIPAS was in the `EMPTY` state. With the corrected specification, Scope revealed additional valid flows, indicated by the red dotted lines in Figure 1.

This motivating example illustrates how inconsistencies in the specification can propagate, leading to broader issues in the design, derivative specifications, and implementation. For CCA developers, the specification serves as the authoritative reference for RMM's behavior. Consequently, any ambiguities or inconsistencies can result in ABI misinterpretations and severe security risks. For example, if a developer relies on an incorrect description of RIPAS, a legitimate call to `RMI_RTT_INIT_RIPAS` might be incorrectly rejected, potentially causing system crashes or undefined behavior.

Moreover, we argue that automatically detecting specification inconsistencies and their cascading effects poses a significant technical challenge. First, the specification is authored and maintained by a team of engineers, and inconsistencies often arise from divergent interpretations or human error. Second, the specification contains internal dependencies that are often difficult to trace and manage.

4 Overview

To detect inconsistencies in the RMM specification, we propose Scope, an automated tool combining formal reasoning and rule-based checking (Figure 2).

Formal reasoning. This approach detects inconsistencies using Verus and an SMT solver. It parses the RMM specification to extract logical components—data types, command condition functions, and commands (①). These components

[1] In the RMM specification, walk.rtte.state contains a HIPAS value of the Realm Translation Table Entry (RTTE) obtained from a page table walking of a given IPA specified in the command's context values, while walk.rtte.ripas a RIPAS value of that.

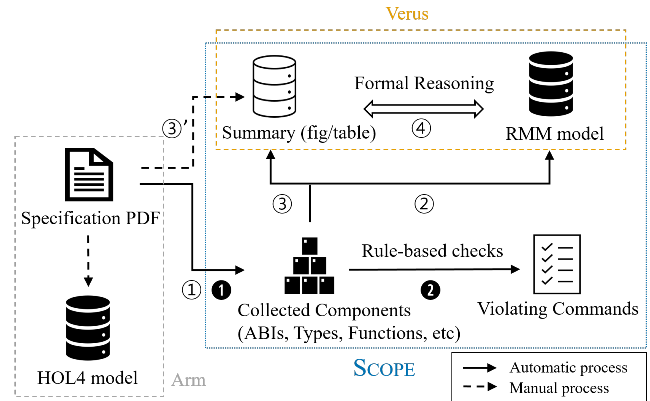

Figure 2. The workflow of Scope. ①–④ are the steps for formal reasoning, while ❶, ❷ are the ones for rule-based consistency checking. ③' is the only one that involves a manual process.

are used to construct the RMM model (②), which encodes the RMM's ABIs and constraints as Verus spec functions. Scope then generates proof queries from summarized tables (③). For non-tabular content, such as diagrams, minimal manual effort converts it into proof queries (③'). The SMT solver checks these queries to determine whether their assertions hold in the RMM model (④). Any failed assertion is reported as an inconsistency.

Rule-based checking. This approach detects violations of predefined inconsistency rules. After parsing the RMM specification and extracting logical components using the same parsing process (❶), Scope analyzes the components, particularly ABI commands, against predefined heuristic rules: footprint check and dangling output check (❷). Any violation is reported as a potential inconsistency.

The process for collecting logical components (①, ❶) is described in §6, while the subsequent workflow steps (②–④, ❷) are detailed in §5.

5 Design

5.1 Reconstructing an RMM model

The RMM specification contains formal definitions spread across multiple sections. Scope consolidates these fragments into coherent entities for formal reasoning. We denote this reconstructed specification as the *RMM model*.[2] This section describes how the RMM model is built (②) by converting ASL, commands, types, and command condition functions.

Conversion from ASL to Verus. The logical components in the RMM specification are originally written in ASL, an internal language used by Arm. To improve accessibility and enable formal analysis, Scope translates these components into Verus.

[2] Note that our RMM model differs from Arm's RMM model [32], which was manually developed for the HOL4 proof assistant (referred to as the HOL4 model in Figure 2). Its primary purpose is to ensure realm memory isolation and prevent undefined behaviors in the model.

ASL representation	Verus representation
array [count] of data_type	[data_type; count]
data_type[count]	[data_type; count]
UInt(x)	x
integer	int
boolean	bool
TRUE	true
FALSE	false
Zeros()	0
2^x	pow2(x)

Table 3. Representation mappings between ASL and Verus.

B4.3.3.1 Interface

B4.3.3.1.1 Input values

Name	Reg.	Bits	Type
fid	X0	63:0	UInt64
rd	X1	63:0	Address
ipa	X2	63:0	Address

B4.3.3.1.3 Output values

Name	Reg.	Bits	Type
result	X0	63:0	CommandReturnCode
data	X1	63:0	Address
top	X2	63:0	Address

B4.3.3.1.2 Context

Name	Type	Value	Before
walk	RttWalkResult	`RttWalk(rd, ipa, RMM_RTT_PAGE_LEVEL)`	false
walk_top	Address	`RttSkipNonLiveEntries( Rtt(walk.rtt_addr), walk.level, ipa)`	false

B4.3.3.2 Failure conditions

ID	Condition
rd_align	pre: `!AddrIsGranuleAligned(rd)` post: `ResultEqual(result, RMI_ERROR_INPUT)`
rtt_state	pre: `walk.rtte.state != ASSIGNED` post: `(ResultEqual(result, RMI_ERROR_RTT, walk.level) && (top == walk_top))`

B4.3.3.3 Success conditions

ID	Condition
rtte_state	`walk.rtte.state == UNASSIGNED`
ripas_ram	pre: `walk.rtte.ripas == RAM` post: `walk.rtte.ripas == DESTROYED`

B4.3.3.4 Footprint

ID	Value
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

Table 4. RMI_DATA_DESTROY command excerpted from the RMM specification (1.0-eac5). The full table is shown in Appendix B.

The conversion requires an understanding of the representations in both ASL and Verus, as summarized in Table 3. For example, ASL expresses array types whose elements are Address in two forms: `array [16] of Address` or `Address[16]`. These are converted to the Verus array format `[Address; 16]`. ASL uses the `^` operator for exponentiation (e.g., `2 ^ Realm(rd). ipa_width`), whereas Verus uses the `^` operator for logical XOR. Such expressions are therefore rewritten using Verus's `pow2` function, e.g., `pow2(Realm(rd).ipa_width)`.

```
pub open spec fn rmi_data_destroy_spec(rd: Address,
 ipa: Address, res: Result<(), RmiStatusCode>,
 data: Address, top: Address, old_s: S, new_s: S) -> bool
{
  // Failure conditions
  (!AddrIsGranuleAligned(old_s, rd)
    ==> ResultEqual(res, RMI_ERROR_INPUT))
  ...
  && (old_walk.rtte.state != ASSIGNED
    ==> (ResultEqual(res, RMI_ERROR_RTT(new_walk.level))
      && (top == RttSkipNonLiveEntries(new_s, Rtt(new_s,
          new_walk.rtt_addr), new_walk.level, ipa))))
  // Success conditions
  ...
  && (res.is_Ok()
    ==> new_walk.rtte.state == UNASSIGNED)
  && (res.is_Ok() && old_walk.rtte.ripas == RAM
    ==> new_walk.rtte.ripas == DESTROYED)
  ...
  // A deduced condition from failure conditions
  && ((AddrIsGranuleAligned(old_s, rd) &&
       ...
       !(old_walk.rtte.state != ASSIGNED))
    ==> res.is_Ok())
  // Deduced conditions from success conditions
  ...
  && (res.is_Err()
      ==> new_walk.rtte.state == old_walk.rtte.state)
  && (!(res.is_Ok() && (old_walk.rtte.ripas == RAM))
      ==> new_walk.rtte.ripas == old_walk.rtte.ripas)
}
```

Listing 1. An example of command conversion. The abbreviated forms `old_walk` and `new_walk` are defined in Listing 2.

Command conversion. Each command is converted into a Boolean function. The conversion has two parts: defining the function signature and formulating the function body. These are illustrated using the `RMI_DATA_DESTROY` example in Listing 1, derived from Table 4.

The function signature (lines 1–3 in Listing 1) is constructed from the interface subsection of the specification (B4.3.3.1 in Table 4). All input and output values are included as function parameters, excluding `fid`, which does not affect the command's behavior. The return type follows Rust conventions: `Result<(), RmiStatusCode>`. In addition to the input and output parameters, the signature includes the old and new system states (`old_s` and `new_s`), representing the model's state transition.

The function body encodes command behavior by translating the specification's failure and success conditions (B4.3.3.2 and B4.3.3.3 in Table 4). These conditions are combined as distinct logical units using a conjunction (`&&`) to form the final expression.

First, failure conditions are modeled as logical implications (`==>`) between preconditions and postconditions, combined conjunctively (lines 6-12 in Listing 1). If none are satisfied,

the command is considered successful. Second, success conditions specify the resulting output behaviors, each represented by a proposition with `res.is_Ok()` as the antecedent (lines 15-18 in Listing 1). Verus's built-in predicates `is_Ok()` and `is_Err()` express success and failure concisely.

Third, the deduced conditions capture the implicit behavior of commands. These conditions serve two purposes: (1) asserting that a command is successful when none of its failure preconditions hold, following rule R_VHFHD [8], and (2) specifying which parts of the RMM state remain unchanged under both failure and success scenarios. The former is encoded as the conjunction of the negations of all failure preconditions (lines 21-24 in Listing 1). The latter is encoded by comparing the old and new states of data items in the success conditions to ensure that they remain unchanged (lines 27-30 in Listing 1).

To this end, the targets of unmodified state items are selected based on assignment operations: whenever an assignment occurs, it indicates that the assignee has been modified. We therefore ensure that the assignee does not change under the negation of the assignment's antecedent. This design choice is based on two observations: (1) if an object is updated under some antecedent in a given command, that antecedent is the sole condition governing updates to the object within the command; and (2) failure conditions do not modify state objects except for output values.

In addition, when an update in the success conditions modifies a subset of the fields of a predefined structure type (e.g., `RmmRttEntry`), we add a deduced condition stating that the remaining fields of the structure remain unmodified in both success and failure cases.

The design distinguishes pre- and post-execution state values by passing `old_s` and `new_s`, respectively. Without this distinction, a pair of failure and success conditions involving the same object can contradict each other. For example, a failure condition may state `walk.rtte.state == ASSIGNED` (the contrapositive of lines 9–10 in Listing 1), while a success condition states `walk.rtte.state == UNASSIGNED` (lines 15–16). In preconditions, `old_s` is always passed to command condition functions (line 6 in Listing 1). In postconditions, `new_s` is passed by default (line 11 in Listing 1).

Type conversion. Data types in the RMM specification are classified into four categories: structure, enumeration, fieldset, and generic types. Structure and enumeration types are mapped to Verus `struct` and `enum` types, respectively. Fieldset types, which contain bitfields, are treated as structures with bit position information discarded, since this information is difficult to encode and has minimal impact on inconsistency detection. Generic types in the specification include `Address`, `BitsN`, `IntN`, and `UintN`, where `N` represents the bit width. These types are mapped to Rust type aliases using semantically equivalent or closely related types. For instance, `Bits512` is represented as `[u8; 64]`, and `UInt15` is mapped to `u16` by rounding up the bit width.

Command condition function conversion. The specification defines a comprehensive set of command condition functions, with 82 in version 1.0-eac5 and 165 in version 1.1-alp14. Functions such as `ResultEqual()`, `RttWalk()`, and `RttSkipNonLiveEntries()` in Table 4 are examples of these functions.

These functions are rewritten for reuse as building blocks for verifiable assertion conditions. While some are defined as pseudocode and others appear as short textual descriptions, Scope treats most as uninterpreted functions [15]. This is because the majority are already presented as uninterpreted, without pseudocode (e.g., 61 out of the 82 functions in version 1.0-eac5). Moreover, uninterpreted functions offer abstraction and reasoning efficiency. For example, `AddrIsGranuleAligned(addr)` could be explicitly defined as `{addr % GRANULE_SIZE == 0}`, but treating it as uninterpreted abstracts away the semantics of `%` and `==`, reducing potential scalability issues in SMT-based reasoning. This follows the common principle of abstract model design, omitting low-level behaviors and execution details [40, 71]. Defining precise semantics from short textual descriptions is beyond the scope of this paper but may be explored in future work.

An exception to the use of uninterpreted functions is `ResultEqual()`, a polymorphic function with two or three parameters. Its usage is unified in the two-parameter form by encoding the third parameter into an `RmiStatusCode` enum variant, as in `RMI_ERROR_RTT(walk.level)` on line 10 of Listing 1. The function checks whether `res` is an `Err` and whether the error code matches the provided status via its second parameter. Other polymorphic functions are disambiguated by adding suffixes (e.g., `RttWalk_()` rather than `RttWalk()`), because Verus does not support polymorphism.

An extra parameter is added to each function to pass state information, allowing it to distinguish between old and new states (e.g., the first argument in lines 7-10 in Listing 2). This extension is not applied to predefined functions such as `ResultEqual()`, whose semantics are independent of state.

Context substitution. In the RMM specification, each RMM command defines a list of context values associated with that command. These values, derived from the input or output values, serve to represent long logical conditions in a more concise manner. For example, the context value `walk` in Table 4 stands for `RttWalk(rd, ipa, RMM_RTT_PAGE_LEVEL)`.

To restore the full conditions from these abbreviations when they are used, the following strategies are applied: First, each context name is replaced with its corresponding value, not only within the context entries but also in the success and failure conditions of the command. In Listing 1 (lines 11-12), the context name `walk_top` is expanded to its full value, which incorporates the substitution of `walk` with its corresponding value. Second, substitution does not apply to structure fields, even if their names match defined context names. For instance, in the failure conditions of `RMI_RTT_MAP_UNPROTECTED`, a pre-condition with ID `rtte_state`

```
pub proof fn rmi_data_destroy_rule (rd: Address,
  ipa: Address, res: Result<(), RmiStatusCode>,
  data: Address, top: Address, old_s: S, new_s: S)
    requires rmi_data_destroy_spec(rd, ipa, res,
                                   data, top, old_s, new_s),
{
  let old_walk = RttWalk_(old_s, rd, ipa,
                                  RMM_RTT_PAGE_LEVEL);
  let new_walk = RttWalk_(new_s, rd, ipa,
                                  RMM_RTT_PAGE_LEVEL);
  // Dependency on RIPAS (If RIPAS is EMPTY or RAM)
  // The below would trigger an assertion violation
  assert(res.is_Ok()
      ==> (old_walk.rtte.ripas == EMPTY ||
           old_walk.rtte.ripas == RAM));
  // Dependency on HIPAS (HIPAS is ASSIGNED)
  assert(res.is_Ok()
      ==> old_walk.rtte.state == ASSIGNED);
  // New RIPAS (If RIPAS is EMPTY)
  assert(res.is_Ok() && old_walk.rtte.ripas == EMPTY
      ==> old_walk.rtte.ripas == new_walk.rtte.ripas);
  // New RIPAS (If RIPAS is RAM)
  assert(res.is_Ok() && old_walk.rtte.ripas == RAM
      ==> new_walk.rtte.ripas == DESTROYED);
  // New HIPAS
  assert(res.is_Ok()
      ==> new_walk.rtte.state == UNASSIGNED);
}
```

Listing 2. An example of proof query.

is defined as `walk.rtte.state != UNASSIGNED_NS`. In this case, although both `walk` and `rtte` are defined as context values, only `walk` is substituted. The `rtte` remains unchanged because it represents a member of the `RmmRttWalkResult` structure.

5.2 Using the RMM model as an oracle

This section presents the use of the RMM model in formal reasoning (③, ④) and in the manual generation of two proof codes (③'): summarized view validation and implicit invariant extraction.

Formal reasoning. After the RMM model is reconstructed via the conversion process, the model is used to answer queries on the RMM specification. Since both the reconstructed model and the summarized tables of required RMM properties are derived from the same specification, they should contain valid and consistent information. Consequently, validity checks of the summarized tables can be expressed as contradiction checks. Scope uses Verus proof functions to evaluate these queries. Listing 2 presents a query constructed by Scope that reflects the summarized semantics of `RMI_DATA_DESTROY`, based on Table 1.

The proof function is constructed as follows: First, it adopts the same signature as the corresponding spec function, but with a different name. To leverage the RMM model as an oracle, Verus `requires` primitive is used, treating the given predicate as a valid precondition. In this case, the function `rmi_data_destroy_spec()` is passed as the precondition (lines 4–5 in Listing 2). Second, common local variables are declared from the command's context values (lines 7–10 in Listing 2). Third, assertions are generated based on the dependency rules in Table 1. These assertions evaluate whether the conditions listed as dependencies are sufficient (lines 13–18 in Listing 2), and whether the specified postconditions match the expected results (lines 20–27 in Listing 2). A violation indicates an inconsistency between the model and the table.

To be specific, when a target command in the dependency table consists of a single row (e.g., `RTT_INIT_RIPAS`), we check that (1) `old_walk.rtte.ripas` (or `state`) equals the value in the *Dependency* columns, and (2) `new_walk.rtte.ripas` (or `state`) equals the value in the *New* columns. If the *New* column specifies an `Unchanged` case, `new_walk.rtte.ripas` (or `state`) is compared against `old_walk.rtte.ripas` (or `state`). If the *Dependency* column specifies `None`, the check trivially evaluates to `true`, as no restriction is imposed.

In contrast, when a target command in the dependency table involves mulitple rows (e.g., `DATA_DESTROY`), we check that `old_walk.rtte.ripas` (or `state`) equals one of the values listed in the *Dependency* columns, combined using logical OR (`||`) (lines 13–18 in Listing 2). If all entries in a given *New* column are identical, we check that `new_walk.rtte.ripas` (or `state`) equals that value (lines 26–27 in Listing 2). Otherwise, when a *New* column contains different values, we check that `new_walk.rtte.ripas` (or `state`) equals each value under its corresponding condition specified in the *Dependency* column (lines 20–24 in Listing 2).

Using this proof function, Scope identified an inconsistency in this example. Specifically, the first assertion fails because the condition for RIPAS—being `EMPTY` or `RAM`—is not sufficient. According to the RMM model, the command can also succeed when RIPAS is `DESTROYED`. This inconsistency was reported, and the condition in Table 1 has been revised from `EMPTY` to `not RAM` to reflect the model's actual behavior.

Validating summarized views. Scope formulates formal queries through manual translation of diagrams and descriptive text from the *Usage* part of the specification. Although these summarized views provide developers with a system-wide perspective, they are often dispersed throughout the document, leading to potential inconsistencies due to fragmentation. For example, when a Realm issues an `RSI_IPA_STATE_SET` call, the Usage part specifies that the RMM updates several fields of the REC structure, including `exit_reason`, `ripas_addr`, `ripas_top`, and `ripas_value`. However, the definition of the REC attributes does not include the field `exit_reason`, thereby revealing an inconsistency.

Extracting implicit invariants. Identifying invariants in the target system is essential for verifying security-critical conditions. Scope derives them primarily from data type definitions. For example, the Realm IPA state includes the statement: *A Protected IPA has an associated Realm IPA state (RIPAS).* From this, an invariant is formulated as: *For a given*

```
pub open spec fn is_protected_state(
  state: RmmRttEntryState) -> bool {
    state == RmmRttEntryState::ASSIGNED ||
    state == RmmRttEntryState::UNASSIGNED ||
    state == RmmRttEntryState::TABLE
}
pub open spec fn is_unprotected_state(
  state: RmmRttEntryState) -> bool {
    state == RmmRttEntryState::ASSIGNED_NS ||
    state == RmmRttEntryState::UNASSIGNED_NS ||
    state == RmmRttEntryState::TABLE
}
```

Listing 3. State checking invariants.

IPA, a RIPAS value exists if and only if the IPA is Protected.[3] As another example, the `RmmRttEntryState` type represents the state of a Realm Translation Table Entry (RTTE). According to the specification, states `ASSIGNED` and `UNASSIGNED` are associated with Protected IPAs, while `ASSIGNED_NS` and `UNASSIGNED_NS` are associated with Unprotected IPAs. From these definitions, two predicates are defined: `is_protected_state()` and `is_unprotected_state()`, as shown in Listing 3. These predicates are then used to validate the following invariant: *For a given IPA, if it is Protected or Unprotected, it must be in the corresponding protected or unprotected state.*

5.3 Rule-based checks

This section describes how heuristic rules are applied (❷). These checks rely on two elements: the command footprint and its output values.

Footprint checks. Scope detects inconsistencies by comparing the footprint and success conditions. For each command defined in the RMM specification, the footprint lists all components of the system state that are modified, excluding output values written to registers. The success conditions describe the observable effects of a successful command execution. Because these two are complementary, there should be a strong correlation between them. If a state modification occurs outside the output values but is not reflected in the footprint, the discrepancy likely indicates an inconsistency.

For example, Table 5 shows the earlier version of the footprint and success conditions for the `RMI_RTT_SET_S2AP` command. When the command is received, the RMM updates `rec.s2ap_addr`, which stores the address of the stage 2 access permission. However, this modified component was initially omitted from the footprint, even though it is unrelated to output values. This omission caused confusion, as the success conditions could be interpreted to mean that only `out_top` is updated. To prevent this type of error, Scope first excludes all output values from the assignees in the success conditions

[3]RIPAS is created and assigned only for Protected IPAs in `RMI_REALM_CREATE` (B3.68 `RttsAllProtectedEntriesRipas`, 1.0-eac5). Its value can change only for Protected IPAs (I_HLHZS, 1.0-eac5). From these, the inverse direction of the *if and only if* statement was inferred. To facilitate the proof, we also added a `NONE` field to the `RmmRipas` enumeration type to distinguish the protected RIPAS condition from others.

B4.3.50.1.3 Output values

Name	Reg.	Bits	Type	Description
out_top	X1	63:0	Address	Top IPA of range

B4.3.50.3 Success conditions

ID	Condition
s2ap_addr	`rec.s2ap_addr == out_top`

B4.3.50.4 Footprint

The RMI_RTT_SET_S2AP command does not have any footprint.

Table 5. Buggy footprint of RMI_RTT_SET_S2AP excerpted from the RMM specification (1.1-alp12). `rec.s2ap_addr` is modified upon the successful execution. However, it is missing from the footprint, which can lead to a misunderstanding of having no modifications.

B4.3.20.1.3 Output values

Name	Reg	Bits	Type	Description
result	X0	63:0	CommandReturnCode	Return status
walk_level	X1	63:0	UInt64	RTT level
state	X2	7:0	RmiRttEntryState	State of RTTE

B4.3.20.3 Success conditions

Condition for `walk_level` is missing.

Table 6. Buggy success conditions of RMI_RTT_READ_ENTRY excerpted from the RMM specification (1.0-eac5). A condition for `walk_level` is missing which makes X1 register uninitialized.

and then verifies that all remaining components are included in the footprint.

Dangling output checks. The relationship between output values and postconditions in both success and failure cases is analyzed. Each output value must appear in at least one postcondition, which specifies its expected state after execution. If an output value—excluding the result value—is absent from all postconditions, it is flagged as a potential inconsistency. Table 6 shows such a case in `RMI_RTT_READ_ENTRY`, where the output `walk_level` (assigned to the X1 register) was omitted from the success conditions.

6 Implementation

Scope is implemented in Python and comprises 2,400 lines of code. It begins by converting the RMM specification PDF to a text file using the `pdftotext` tool. The script preprocesses the text by removing nonessential elements such as the table of contents, headers, footnotes, and draft markings to simplify parsing.

To collect logical components, the script employs a top-down approach by dividing the document by Part and analyzing each subsection. Parsing primarily relies on regular expressions. When this approach is insufficient, particularly for multi-line table entries, the script uses indentation information and heuristics extracted from various versions of the RMM specification.

After parsing, the collected components are transformed as described in §5. During model reconstruction, the script generates a preamble and postamble that define data types, type aliases, constants, and Verus-specific keywords. For polymorphic functions, it identifies parameter variations by counting commas and applying keyword matching. To avoid type errors, the script performs type coercion using the as keyword in Verus where necessary.

7 Evaluation

We evaluate Scope through the following research questions:

- **RQ1:** How effectively does Scope detect inconsistencies in the RMM specification?
- **RQ2:** How does Scope perform relative to existing approaches?
- **RQ3:** How robust is Scope across different versions of the specification?
- **RQ4:** What security risks arise from inconsistencies?

Environment Settings. All experiments were conducted using Verus v0.2025.01.11 and Z3 v4.12.5. To address **RQ1** and **RQ3**, we used multiple versions of the RMM specification, while **RQ2** was evaluated on version 1.0-rel0. The running time for processing a single version was approximately 1–2 seconds on a desktop with an Intel Core i7-14700K processor and 32 GB RAM.

7.1 RQ1: Effectiveness of Scope

A total of 38 potential inconsistencies were reported, of which 35 were confirmed as valid by Arm, as summarized in Table 7. Several have been officially labeled and fixed in later specification releases; the rest are scheduled for future updates. Three were rejected due to misinterpretations of the specification. Notably, 13 inconsistencies were found in newly introduced commands (marked with an asterisk), demonstrating Scope's effectiveness in handling frequent updates of new ABIs.

Experiment setup. We evaluated the detection performance of our proposed automated approaches—formal reasoning and rule-based checking—on versions 1.0-eac5 and 1.0-rel0 of the RMM specification. For formal reasoning, we used the original entries in the dependency table (Table 1), which contains 36 cells (4 condition columns × 9 command rows). Our parser successfully analyzed 31 of these cells; the remaining 5 were excluded as they deviated from standard description patterns. For rule-based checking, we evaluated all commands as units, treating each violating command as one case. An exception was made for `RMI_RTT_READ_ENTRY`, where each output value was evaluated independently, as there was a mix of a false positive and a true positive.

Results. Overall, Scope demonstrated high effectiveness and precision in detecting inconsistencies in the RMM specification, achieving precision scores of 63.64% and 61.90% across the two approaches. For formal reasoning, precision scores of 33.33% and 25% were reported. The tool reported 9 and 8 assertion violations in the respective versions, with 3 and 2 true positives and 6 false positives in each case. Among the false positives, 3 were caused by interpretation issues, while the other 3 stemmed from underspecification of the `RttEntriesInRangeRipas()` function, which could be resolved with user input. For rule-based checking, a precision of 84.62% was measured for both versions. This method reported a total of 13 violations, including 11 true positives and 2 false positives. The false positives were due to non-standard usage of the `desc` value and ambiguous semantics related to state modification.

7.2 RQ2: Comparison with Existing Approaches

To answer RQ2, we compared Scope's effectiveness against existing tools: LLM-based tools and formal verification tools.

7.2.1 Comparison with LLM-based Tools. Scope improves precision by 7-40× compared to LLM-based approaches (Table 8). We used advanced LLMs from various vendors, including general-purpose chat models—GPT-4o [56], Claude 3.7 [11], and Llama3.1-70B [51]—and reasoning-focused models such as GPT-o1 [57] and Deepseek-R1 [25].[4]

Experiment setup. We developed an automatic context-splitting agent that partitions the document based on its internal hyperlinks and structure. This design decision was made because the RMM specification exceeds 124k tokens, leaving limited space for model-generated outputs and reasoning, as modern LLMs can process up to 128k tokens. Details can be found in Appendix C. The agent produced 321 contextually coherent segments, the largest being approximately 70k tokens. These segmented inputs were then fed into each LLM for evaluation. We randomly sampled approximately 50 outputs across the 321 segments and manually evaluated their accuracy. Due to model-specific behavior and output variability, the number of evaluated outputs varied between models.

Results. GPT-o1 achieved a precision of 8.00%, and Deepseek-R1 reached 1.89%, both significantly lower than Scope's 61.90%. Reasoning models outperformed chat models due to their advanced inference mechanisms; however, they still exhibited low precision and required extensive manual validation. Most false positives produced by the LLMs stemmed from misinterpretation or hallucination, rather than from deficiencies in contextual understanding or reasoning ability. Notably, validating LLM-generated outputs is still a labor-intensive task, requiring manual review of both the model's reasoning and corresponding document statements.

[4] The snapshots we used for evaluation are: gpt-4o-2024-11-20, claude-3.7-sonnet-20250219, llama-3.1-70b-instruct, o1-preview-2024-09-12 and deepseek-r1.

ABI Name	# of report / # of confirmation	Labels	Duration	Categorization	Description
RMI_DATA_CREATE	2/2 (alp11, alp11)	906, 935	18, 25	R, R	inconsistent diagram, missing cond.
RMI_DATA_DESTROY	1/1 (alp10)	899	16	R	inconsistent table
RMI_MEC_SET_PRIVATE*	1/1 (alp14)	1105	8	H (f)	missing footprint
RMI_MEC_SET_SHARED*	1/1 (alp14)	1114	8	H (f)	missing footprint
RMI_P2P_STREAM_ADD*	1/1 (internal)	N/A	N/A	H (f)	missing footprint
RMI_P2P_STREAM_REMOVE*	1/1 (internal)	N/A	N/A	H (f)	missing footprint
RMI_PSMMU_IRQ_NOTIFY*	1/1 (internal)	N/A	N/A	H (d)	missing cond.
RMI_REC_AUX_COUNT	1/1 (alp11)	907	28	R	missing field in realm attributes
RMI_RTT_DESTROY	3/3 (alp10, alp10, alp11)	892, 893, 940	16, 16, 1	R, R, R	inconsistent table, incorrect cond. (2)
RMI_RTT_DEV_MEM_VALIDATE*	1/1 (alp14)	1109	1	H (f)	missing footprint
RMI_RTT_FOLD	1/0 (rejected)			R	
RMI_RTT_INIT_RIPAS	2/2 (rel0, alp14)	864, 1048	15, 22	R, R	inconsistent table, inconsistent diagram
RMI_RTT_READ_ENTRY	1/1 (alp10)	896	27	H (d)	missing cond.
RMI_RTT_SET_S2AP*	2/2 (internal, alp14)	1113	N/A, 4	H (d), H (f)	missing cond., missing footprint
RMI_VERSION	1/1 (alp10)	898	13	H (d)	missing cond.
RSI_ATTESTATION_TOKEN_CONT.	1/1 (alp14)	959	33	H (d)	missing cond.
RSI_ATTESTATION_TOKEN_INIT	1/1 (alp14)	958	18	H (d)	missing cond.
RSI_IPA_STATE_GET	2/2 (alp14, internal)	957	7, N/A	H (d), H (d)	missing cond. (2)
RSI_MEASUREMENT_READ	1/1 (alp14)	944	33	H (d)	missing cond.
RSI_MEM_SET_PERM_INDEX*	2/2 (alp14, alp14)	1058, 1110	8, 8	H (d), H (f)	missing cond., missing footprint
RSI_MEM_SET_PERM_VALUE*	1/1 (internal)	N/A	8	H (f)	missing footprint
RSI_RDEV_GET_INFO*	1/1 (internal)	N/A	N/A	H (f)	missing footprint
RSI_REALM_CONFIG	2/0 (rejected, rejected)			H (f), H (f)	
RSI_VERSION	1/1 (alp10)	898	13	H (d)	missing cond.
RSI_VSMMU_ACTIVATE*	1/1 (alp14)	1059	6	H (d)	missing cond.
PSCI_CPU_ON	1/1 (alp14)	1103	33	H (f)	missing footprint
PSCI_SYSTEM_OFF	1/1 (alp14)	1104	21	H (f)	missing footprint
PSCI_SYSTEM_RESET	1/1 (alp14)	1116	21	H (f)	missing footprint
PSCI_VERSION	1/1 (alp14)	947	33	H (d)	missing cond.
Etc (RIPAS change)	1/1 (alp14)	968	33	R	inconsistent diagram
Total	38/35	28			

Table 7. Summary of bugs discovered by Scope and confirmed by Arm. * marks ABIs newly added in recent RMM specification versions. (version) indicates when the bug was fixed; (internal) means the bug was validated by Arm but remains unpatched. Labels appear in the *Release Information* of the latest specification with the prefix of FENIMORE- and are assigned only after a fix. Duration represents the number of months the bug persisted. R denotes bugs found via formal reasoning; H (d) and H (f) refer to those identified through heuristic methods: dangling output and footprint checks.

Models		Result*	Precision
Chat Model	llama3.1	1/66	1.52%
	gpt-4o	1/54	1.85%
	claude-3.7	2/57	3.51%
Reasoning Model	gpt-o1	4/50	8.00%
	deepseek-r1	1/53	1.89%
Scope		13/21	61.90%

Table 8. Evaluation between Scope and LLM based approaches. Scope effectively improves the precision by 7-40x compared to the LLM-based approaches. *: Result = TP/(TP+FP)

7.2.2 Comparison with Formal Verification Tools.

Compared with two existing formal verification approaches, VIA [49] and Arm's model checking [32], Scope covers up to three times more ABIs and scales to newer versions (Table 9).

Experiment setup. For this comparison, we used publicly available sources. According to VIA, it used an early prototype of the RMM implementation for verification, and the exact version was not disclosed. To align the results, we assumed that this prototype contained a similar number of total commands to version 1.0-eac5, and projected the number of verified commands accordingly.[5] In contrast, the covered commands of Arm's model checking were estimated

[5] While version 1.0-eac5 adds one command over the oldest version (1.0-bet0), VIA may have used a version with fewer commands than the oldest.

Spec Version	1.0-eac5	1.0-rel0	1.1-alp11	1.1-alp12
VIA	22 (54%)	N/A	N/A	N/A
Arm's model checking	8 (20%)	8 (20%)	N/A	N/A
Scope	28 (68%)	28 (68%)	74 (77%)	79 (78%)
Total	41	41	96	101

Table 9. Number of successfully covered ABI commands. N/A indicates no covered rate due to the absence of an implementation in that version.

by counting the number of verification harnesses, each of which targets a single command.

Results. VIA covered 22 commands, resulting in a coverage rate of 54%, with most of the uncovered cases being related to RSI commands. For model checking, 8 commands were covered, yielding a coverage rate of 20%. Although Arm's model checking benefits from MRS [32], its coverage rate remains low. We think that model checking covered fewer ABIs due to scalability limits (e.g., state explosion) and could not benefit from modular verification. Additional verification efforts may have been conducted internally by Arm but not publicly disclosed.

Scope not only achieves higher coverage rates and the number of cases than existing verification approaches, but also supports recent versions of the RMM specification, including 1.1-alp11 and 1.1-alp12. Since Arm's model checking relies on the existence of an implementation, it cannot verify recent versions of the RMM specification until corresponding implementations become available. Furthermore, VIA's heavy reliance on manual proofs is difficult to maintain in the face of frequent updates to the RMM specification. These results suggest that the substantial manual effort involved in formalizing the specification limits the scalability of traditional formal verification tools. As a result, only a limited number of ABIs have been formally verified, potentially leaving security vulnerabilities in the unverified portions. Furthermore, by examining Scope's output, we identified 10 inconsistencies in VIA's verified commands, and 2 inconsistencies in Arm's model checking harnesses. Finding bugs in verified components does not imply the tools are unsound; each has different strengths.

7.3 RQ3: Robustness Evaluation

To evaluate Scope's robustness, we tested it on multiple RMM specification versions, demonstrating its ability to handle specification updates and newly introduced ABIs.

Experiment setup. We counted the number of commands that could be reasoned about as covered commands, and calculated the covered rate by dividing this number by the total number of commands, which includes RMI, RSI, and PSCI commands. While counting the number of covered commands in Scope, we excluded the cases that lacked both success and failure conditions or were incomplete. We regarded the commands with missing conditions or those that were non-executable as incomplete.

B4.3.16.3 Success conditions

ID	Condition
rtte_state	`walk.rtte.state == UNASSIGNED`
ripas	`walk.rtte.ripas == DESTROYED`
state_prot	`pre: AddrIsProtected(ipa, realm)` `post: walk.rtte.state == UNASSIGNED`
ripas	`pre: AddrIsProtected(ipa, realm)` `post: walk.rtte.ripas == DESTROYED`
state_unprot	`pre: !AddrIsProtected(ipa, realm)` `post: walk.rtte.state == UNASSIGNED_NS`

Table 10. Buggy success conditions of RMI_RTT_DESTROY excerpted from the RMM specification (1.0-eac5). The invariants for Protected IPAs and Unprotected IPAs are not considered, leading to a potential security flaw, as shown in the red shaded area. The green shaded area indicates the fixed conditions following our report.

Results. Scope attained an average covered rate of 73% (Table 9). For 1.0-eac5 and 1.0-rel0, 28 commands were covered and 13 were uncovered. Out of 13 uncovered commands, 10 were closely related to the document's issues; 5 were empty commands, and 5 had prosaic descriptions instead of logical conditions. The remaining 3 were excluded because they contained ASL syntaxes that we did not support, leading to a compile error. For example, we translated the `RmmRealmMeasurement` type of 512 bits as `[u8; 64]`, and it could not be directly compared to the natural number `0`, which was the translation result of `Zeros()`. For 1.1-alp11 and 1.1-alp12, 22 were uncovered while 74 and 79 were covered. Among the 22 uncovered commands, 3 were empty commands, 11 contained prosaic descriptions, and 8 were related to unsupported syntaxes. 6 of the unsupported syntaxes were all caused by polymorphic functions of `Equal()`; these are not fundamental limitations, but supporting them would require substantial engineering effort, as there are a large number of variants of `Equal()`—35 in 1.1-alp12 alone.

7.4 RQ4: Security Implication

To address RQ4, we present two case studies that illustrate the potential security risks arising from inconsistencies.

7.4.1 RMI_RTT_DESTROY [FENIMORE-893, 940]. The incorrect success condition in the `RMI_RTT_DESTROY` ABI would cause a risk of unauthorized access to protected memory regions. Table 10 highlights the initial success conditions of this ABI before our report, along with the subsequent corrections. Originally, both Protected and Unprotected IPAs were incorrectly considered capable of meeting the ABI's success conditions, contradicting key invariants of RIPAS and unprotected states. Specifically, the proper handling of Unprotected IPAs was neglected. Implementations based on these incorrect conditions could mistakenly assign the value

B4.3.1.1.1 Input values

Name	Register	Bits	Type	Description
src	X4	63:0	Address	PA of the source Granule

B4.3.1.3 Success conditions

ID	Condition
data_state	`Granule(data).state == DATA`
rtte_state	`walk.rtte.state == ASSIGNED`
rtte_ripas	`walk.rtte.ripas == RAM`
rtte_addr	`walk.rtte.addr == data`
rim	`Realm(rd).measurements[0] == RimExtendData( realm, ipa, data, flags)`

Condition for the contents copied from `src` is missing.

Table 11. Buggy success conditions of RMI_DATA_CREATE excerpted from the RMM specification (1.0-eac5). A condition for content copy from the Host is missing from the Success conditions, which can lead to a failure to deploy the contents in the Realm.

`UNASSIGNED` instead of `UNASSIGNED_NS` for Unprotected IPAs, particularly in cases where the `NS [55]` field in the page table descriptor is cleared. Since the `NS` field determines access permissions for the owning Realm (`PAS=Realm` when the bit is `0` and `PAS=NS` when it is `1`), such misconfigurations could introduce serious security vulnerabilities—potentially enabling Unprotected IPAs to access protected memory of the Realm without triggering a Granule Protection Fault (GPF).

7.4.2 RMI_DATA_CREATE [FENIMORE-935].

The original success conditions of the `RMI_DATA_CREATE` ABI omitted an essential requirement, leading to potential inconsistencies and verification failures (shown in Table 11). According to the Realm lifecycle, the Host uses this ABI to add pages to a Realm by specifying a source Granule whose contents should subsequently be copied into a destination Granule. However, the previous success conditions failed to explicitly state this necessary content-copying step. If an implementation strictly adheres to these incomplete conditions, neither code nor data could be properly instantiated within the Realm's memory region. Moreover, formal verification efforts based on these flawed ABI semantics—particularly those targeting critical properties such as Realm Memory Isolation—would be fundamentally unsound.

8 Discussion

Assumptions. For automatic processing, Scope treats uninterpreted functions as-is, which constitute most command condition functions in the specification, as described in §5.1. This design choice assumes that any side effects arising from uninterpreted functions are acceptable. While this may introduce false positives due to underspecification (e.g., `RTTEntries`-InRangeRipas, §7.1), it enables efficient detection of inconsistencies within the specification.

During proof query generation, Scope assumes that the *Dependency* on RIPAS (or HIPAS) and the *New* RIPAS (or HIPAS) refer to the same object (e.g., an RTT entry at the same RTT walk level). This assumption generally holds, but it can lead to false positives when violated. For example, in `RMI_RTT_CREATE`, the former refers to entries in the parent table, whereas the latter refers to entries in the child table.

We further assume the correctness of underlying tools, including Verus and SMT solvers used by Scope, as is standard in prior work.

Limitations. Scope may fail to detect bugs that are not reflected in any related statements. However, this limitation is mitigated by the observation that all versions of the specification have undergone rigorous review, making it unlikely for a single statement to convey blatantly incorrect information. Also, our model is neither bit-precise nor byte-precise, and inconsistencies that involve bit- or byte-level semantics would not be detected if present. Additionally, we do not extract the ordering information of each command's failure conditions from the RMM specification. As a result, subtle logical inconsistencies involving this ordering may go undetected by Scope. To mitigate scalability issues in reasoning about loops, Scope currently checks only a single entry in the target page table, rather than checking all entries. In addition, the added deduced conditions represent only a subset of the unmodified state items, as the conversion process prioritizes automation over completeness. Finally, the use of uninterpreted functions may lead to false positives when the semantics of one function are subsumed by those of another. We believe that Scope can be extended to address these limitations in future work.

Manual effort. In version 1.0-eac5, formal definitions (Interface and Types) occupy 166 of 281 pages (59%), which Scope can process automatically. The remaining content requires manual handling and the amount of effort differs depending on the user type. For passive users, model generation requires 2–10 minutes per supported version to fix type errors. For active users, modifying or inserting a condition takes 10–60 minutes depending on the target. Translating figures into queries typically takes 2–20 minutes each. When ASL syntax, Verus representation, or document structure changes, the parser must be adjusted, requiring hours of engineering effort. Also, discovering new rules or invariants may take several days or weeks. Overall, the effort took about 11 months, mostly due to implementation work.

Applicability to other areas. PDF is not a suitable input format for extension, and reusing our parser beyond the RMM specification would require substantial additional engineering effort. Nevertheless, we argue that the central idea of this work—cross-referencing different sections within the same source to identify inconsistencies—can be extended to other specification types, such as architectural reference manuals, protocol specifications, and network function specifications [79, 80], to check internal consistency. For example, modern specifications (e.g., RISC-V) increasingly include semi-formal definitions, which make them amenable to our

approach. Furthermore, our novel use of a formal model as an oracle for answering proof queries that reflect the intent of the specification's designers can be applied to other formally verified systems and specifications to strengthen confidence in their correctness. Lastly, we demonstrate that PDF documents can be directly parsed, and the extracted data can be sanitized through rule-based checking. This process of automated parsing and validation holds potential for early error detection in other technical documents governed by complex rules.

Alternatives. To facilitate formal reasoning, Verus serves as the initial target for translating the RMM specification. This conversion can be generalized to other formal frameworks, including Dafny [44], Boogie [13], and UCLID5 [61]. Also, once Arm releases the Machine Readable Specification input (YAML files) for the RMM specification, our system can be extended to support it, further improving generalizability through common parsing formats.

9 Related Work

Significant efforts have been made to improve the trustworthiness of specifications. Based on their focus and methods, prior work falls into five main categories.

Testing against implementation. Executing specifications has been widely used to improve their correctness and reliability. Fox and Myreen develop executable formal specifications of Armv6 and Armv7 in HOL, and validate them via testing and proof [30, 31]. ISA-Formal translates Arm manuals into Verilog to enable model checking [66]. Sail provides a reusable infrastructure for ISA semantics, supporting emulator generation and theorem prover integration [12]. Concurrency models for Armv8 are extensively tested by Flur et al. [28]. Executable and validated models have also been developed for x86 [34, 68]. Capability extensions for RISC-V and CHERI-MIPS are verified using translations into Sail or L3 [33, 55]. Similarly, seL4 executes its microkernel specification to support validation [40].

Formal modeling. Proving critical properties of a specification is a common approach to validating its correctness. seL4 strengthens the trustworthiness of its specification by formally proving integrity [69], and information flow security [53]. Enclave execution mechanisms, such as Intel SGX and Sanctum [23], are formalized and verified against security definitions [71]. For Arm CCA, Fox et al. develop a secure, machine-checkable specification and prove core security properties [32]. Remote attestation for confidential virtual machines is also formally modeled and verified [67]. In contrast, our work focuses on identifying internal inconsistencies within the specification itself.

Specification auditing. Auditing specifications often relies on high-level views that abstract key aspects of system behavior [35, 65], requiring substantial manual effort and domain expertise. Kemmerer proposes validating specifications by defining minimal requirements and testing them against these requirements using symbolic execution [38]. Fonseca et al. manually examine the specifications of distributed systems and uncover several bugs [29]. Reid verifies the Arm v8-M architecture specification using architect-defined views and coverage properties [65]. Goldweber et al. generalize specification testing by incorporating developers' intent as an explicit auditing criterion [35]. In contrast, our approach automates inconsistency detection.

Test generation from specifications. Automatic test generation is widely used to support the validation of specifications and hardware by producing executable tests. Several tools [21, 52, 58–60, 62, 63, 78] generate tests from specification documentation or heuristic rules to expose inconsistencies. For example, Sherlock [21] employs machine learning to extract dependency graphs and generate tests that reveal mismatches between hardware and specifications. While effective, these approaches typically rely on hardware or emulators for test execution. In contrast, Scope identifies logical inconsistencies directly within the specification itself, without requiring hardware.

LLM-based inconsistency detection. Recent work has explored the use of LLMs to detect inconsistencies in specifications by leveraging their ability to reason over natural language and code [19, 46, 50, 76, 82]. AutoVerus [76], for example, generates Verus specifications from code and verifies them using SMT solving. In contrast, Scope analyzes a formal RMM model against structured specification elements, such as tables and figures. While LLMs perform well on textual inputs, reasoning over non-textual specification artifacts, including diagrams, remains a significant limitation.

10 Conclusion

In this paper, we present Scope, an automatic approach for processing and analyzing the RMM specification. Scope is based on the assumption that such specifications are internally consistent. It extracts relevant information from the specification and converts it into logical constraints to detect inconsistencies. Unlike prior test generation-based tools, Scope does not rely on existing hardware or software implementations for validation. Its fully automated methodology significantly reduces the manual effort required to maintain formal verification proofs developed by engineers. We evaluated Scope on the RMM specification and demonstrated its effectiveness by identifying 35 inconsistencies, which were subsequently confirmed by the engineers at Arm. Furthermore, our comparison with LLM-based approaches shows that Scope achieves higher precision in inconsistency detection.

Acknowledgments

We thank our shepherd, David Cock, and other anonymous reviewers for their constructive comments and feedback. We thank Gareth Stockwell and Kshitij Wavre at Arm for validating results. We thank Jangseop Shin, Eunsoo Kim, Jun Ho Huh, Yonggon Kim, and the entire Security&Privacy team at Samsung Research for their help. We thank our anonymous artifact reviewers for their patience and suggestions.

A RTT_INIT_RIPAS table in full

We present the complete versions of Table 2 in Table 12.

B4.3.18.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre: !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre: Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`
size_valid	`pre: UInt(top) <= UInt(base)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

top_bound	`pre: !AddrIsProtected(`
	`ToAddress(UInt(top) - RMM_GRANULE_SIZE),`
	`realm)`
	`post: ResultEqual(result, RMI_ERROR_INPUT)`

realm_state	`pre: realm.state != REALM_NEW` `post: ResultEqual(result, RMI_ERROR_REALM)`
base_align	`pre: !AddrIsRttLevelAligned(base, walk.level)` `post: ResultEqual(result, RMI_ERROR_RTT, walk.level)`
rtte_state	`pre: walk.rtte.state != UNASSIGNED` `post: ResultEqual(result, RMI_ERROR_RTT, walk.level)`
top_gran_align	`pre: !AddrIsGranuleAligned(top)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

top_rtt_align	`pre: ((UInt(top) < UInt(RttUpperBound(`
	`base, walk.level, realm.ipa_width)))`
	`&& RttEntryHasRipas(RttEntry(`
	`walk.rtt_addr,`
	`RttEntryIndex(top, walk.level)))`
	`&& !AddrIsRttLevelAligned(top, walk.level))`
	`post: ResultEqual(result, RMI_ERROR_RTT, walk.level)`

Table 12. Failure conditions of RMI_RTT_INIT_RIPAS excerpted from the RMM specification (1.0-eac5). The shaded area highlights the failure condition associated with a HIPAS value.

B RMI_DATA_DESTROY table in full

We present the complete versions of Table 4 in Table 13.

B4.3.3.1 Interface

B4.3.3.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000155`
rd	X1	63:0	Address	PA of the RD which owns the target Data
ipa	X2	63:0	Address	IPA at which the Granule is mapped in the target Realm

B4.3.3.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status
data	X1	63:0	Address	PA of the Data Granule which was destroyed
top	X2	63:0	Address	Top IPA of non-live RTT entries, from entry at which the RTT walk terminated

B4.3.3.1.2 Context

The RMI_DATA_DESTROY command operates on the following context.

Name	Type	Value	Before	Description

walk	RmmRttWalkResult	`RttWalk(`
		`rd, ipa,`
		`RMM_RTT_PAGE_LEVEL)`
			false	RTT walk result

entry_-idx	UInt64	`RttEntryIndex(` `ipa, walk.level)`	false	RTTE index

walk_top	Address	`RttSkipNonLiveEntries(`
		`Rtt(walk.rtt_addr),`
		`walk.level,`
		`ipa)`
			false	Top IPA of non-live RTT entries, from entry at which the RTT walk terminated

B4.3.3.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre: !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre: Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`
ipa_align	`pre: !AddrIsGranuleAligned(ipa)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
ipa_bound	`pre: !AddrIsProtected(ipa, Realm(rd))` `post: ResultEqual(result, RMI_ERROR_INPUT)`

rtt_walk	`pre: walk.level < RMM_RTT_PAGE_LEVEL`
	`post: (ResultEqual(result, RMI_ERROR_RTT, walk.level)`
	`&& (top == walk_top))`

rtt_state	`pre: walk.rtte.state != ASSIGNED`
	`post: (ResultEqual(result, RMI_ERROR_RTT, walk.level)`
	`&& (top == walk_top))`

B4.3.5.3 Success conditions

ID	Condition
data_state	`Granule(addr).state == DELEGATED`
rtte_state	`walk.rtte.state == UNASSIGNED`
ripas_ram	`pre: walk.rtte.ripas == RAM` `post: walk.rtte.ripas == DESTROYED`
data	`data == walk.rtte.addr`
top	`top == walk_top`

B4.3.5.4 Footprint

ID	Value
data_state	`Granule(walk.rtte.addr).state`
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

Table 13. RMI_DATA_DESTROY command excerpted from the RMM specification (1.0-eac5).

C Prompt and Agent Details

We present the detailed prompt we use for LLM-based evaluation and demonstrates our agent's workflow. Our agent works for Arm CCA specification document by trying to parse its index and divided the whole document into small sections as nodes. For each node, the agent goes over the content and identify all the hyperlinks to other sections. We define the 'relevant' section as the nodes whose distance to the target node is at most 1 hop. For each section and its relevant sections, we combined them together and feed to the LLM. The system prompt of agent is used as below:

````
You are an expert in ARM CCA and PDF document.
I need your help to find contradictions in the
long document, usually across tables, sections.
Your input will be a long text extracted from pdf
document, and you are expected to find **all**
the **contradictions** in the input document.


For example, the below example shown:

```
<1-shot-example>
```
````

For each related context, we use following user prompt to ask for answers:

````
This section is about <topic>, the text
extracted from PDF is:

```
<pdfTotext>
```

Find all the inconsistencies inside the
input text, with detailed explanations.
````

D Artifact Appendix

D.1 Abstract

The scripts and step-by-step guide to reproduce the evaluation in this paper are available at https://doi.org/10.5281/zenodo.17946326. It contains a sophisticated Python tool that converts Arm CCA's Realm Management Monitor (RMM) specification documents into Verus verification code for formal analysis and inconsistency detection.

D.2 Artifact check-list (meta-information)

- **Model: Token/Api Key for LLM models to test: gpt-4o, gpt-o1, claude-3.7, deepseek-r1 and llama3.1**
- **Data set: Realm Management Monitor specification**
- **Hardware: CPU (Intel Core i7-14700K)**
- **Metrics: Precision scores are defined as True Positives (TP) / (True Positives (TP) + False Positives (FP)). Covered rates are defined as the number of covered commands / the total number of commands.**
- **How much disk space required (approximately)?: < 2GB (mostly for Verus)**
- **How much time is needed to prepare workflow (approximately)?: 1 hour**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: Apache-2.0 license**
- **Archived (provide DOI)?: https://doi.org/10.5281/zenodo.17946326**

D.3 Description

D.3.1 How to access

- Source code: https://github.com/islet-project/scope or https://doi.org/10.5281/zenodo.17946326.

D.3.2 Software dependencies

- Ubuntu 24.04
- Python 3.12.3
- Verus v0.2025.01.11
- Z3 v4.12.5
- pdftotext 24.02.0

D.3.3 Data sets

The Realm Management Monitor specifications (versions 1.0-eac5, 1.0-rel0, 1.1-alp11, and 1.1-alp12) are provided as PDF files. These files should be downloaded from the links below and placed in the same directory as the scope script.

- https://developer.arm.com/documentation/den0137/1-0eac5/
- https://developer.arm.com/documentation/den0137/1-0rel0/
- https://developer.arm.com/-/cdn-downloads/permalink/Architectures/Armv9/DEN0137_1.1-alp11.zip
- https://developer.arm.com/-/cdn-downloads/permalink/Architectures/Armv9/DEN0137_1.1-alp12.zip

D.4 Installation

We require installing pdftotext, Rust, Verus, and Z3.

```
# Install pdftotext
sudo apt-get install libpoppler-dev
# Install python dependencies
sudo apt install python3-pypdf python3-networkx python3-openai
# Install rust
curl -proto '=https' -tlsv1.2 -sSf https://sh.rustup.rs | sh
# Install verus and z3
git clone https://github.com/verus-lang/verus.git
cd verus
git reset -hard bec74a67d9281a4f51a7e1855760c5d16d8f63ff
cd source
./tools/get-z3.sh
```

```
source ../tools/activate
vargo build -release
```

D.5 Experiment workflow

Reviewers can use §D.5.1 and §D.5.2 to reproduce the results in §7.1. §D.5.3 corresponds to §7.3, and §D.5.4 corresponds to §7.2.1.

Because the RMM specifications are provided as unlabelled documents, identifying true positives (TPs) and false positives (FPs) required manual validation via Arm's feedback system. To facilitate reproducibility, we include pre-labelled data in the form of patch files.

D.5.1 Formal Reasoning

The precision scores of formal reasoning can be reproduced with the following steps.

```
# generate a model
./scope -target {eac5, rel0} -input-type pdf -mode reason >
{eac5, rel0}.rs

# apply a patch to avoid type errors
patch -p0 < ./patch/{eac5, rel0}.patch
cp {eac5, rel0}.rs ~/

# verify
cd ~/verus/source
./target-verus/release/verus ~/{eac5, rel0}.rs
```

D.5.2 Rule-based Checks

The precision scores of rule-based checks can be reproduced with the following steps.

```
# perform heuristic checks
./scope -target {eac5, rel0} -input-type pdf -mode rule >
{eac5, rel0}_rule.txt

# apply a patch for pre-determined labelling
patch -p0 < ./patch/{eac5, rel0}_rule.patch

# see the result
cat {eac5, rel0}_rule.txt
```

D.5.3 Robustness Evaluation

The covered rates can be reproduced with the following steps.

```
# measure covered rates
./scope -target {eac5, rel0, alp11, alp12} -input-type pdf
-mode reason -is-coverage -no-dependency > {eac5, rel0, alp11,
alp12}_coverage.rs

# apply a patch for pre-determined labelling
patch -p0 < ./patch/{eac5, rel0, alp11, alp12}_coverage.patch
cp {eac5, rel0, alp11, alp12}_coverage.rs ~/

# verify
cd ~/verus/source
./target-verus/release/verus ~/{eac5, rel0, alp11, alp12}
_coverage.rs

# see the result
cat ~/{eac5, rel0, alp11, alp12}_coverage.rs
```

D.5.4 LLM Agent Evaluation

To reproduce LLM agent evaluation in the paper, we provide the LLM agent code in the `llm-baseline` folder. The evaluation contains following steps:

1. Split the pdf input into sections based on the table of contents
2. Build the section dependency graph based on the hyperlink inside each sections; the section and its recursivly extended dependencies are contexts
3. Take the whole section and context as input and feed into LLM

Please follow the instructions from the readme file, or run following commands:

```
cp DEN0137_1.0-rel0_rmm-arch_external.pdf cca.pdf
python3 main.py cca.pdf 9 16
```

D.6 Evaluation and expected results

D.6.1 Formal Reasoning

With the execution, 8 assertion violations would be reported for eac5 and rel0. Another assertion violation would be generated by commenting out the first FP `assert` statement of `rmi_rtt_init_ripas_rule` function in eac5.rs. Out of them, 6 are FPs and 3 and 2 are TPs, as indicated in comments and §7.1.

D.6.2 Rule-based Checks

In the text files, 12 violating commands would be detected for eac5 and rel0. Among them, 10 commands are TPs, 1 command is a FP, and 1 command contains both of a TP and a FP, as indicated in comments and §7.1.

D.6.3 Robustness Evaluation

For all cases, verification would be successful with lots of camel-related warnings. Counting the numbers of `: COVERED` and `: UNCOVERED` would result in 28, 28, 74, 79 and 13, 13, 22, 22 for eac5, rel0, alp11, alp12. `[EXCLUDED]` diagnoses problematic places of uncovered commands: emptiness, unsupported syntaxes, and prosaic descriptions, as described in §7.3.

D.6.4 LLM Agent Evaluation

Due to the large volume output of LLMs, we only sample 50 cases for each model and manually checked the correctness. The full output of the LLM result is attached in the `llm-output.txt`.

References

[1] 2013. *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[2] 2016. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA.

[3] 2016. *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX.

[4] 2017. *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China.

[5] Arm 2021. *DEN0096: Arm CCA Security Model*. Arm. Accessed: 2025-08-08.

[6] 2022. *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA.

[7] 2023. *Proceedings of the 38th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Cascais, Portugal.

[8] Arm 2024. *DEN0137: Realm Management Monitor specification*. Arm. Accessed: 2025-08-08.

[9] Arm 2025. *DDI0626: Architecture Specification Language Reference*. Arm. Accessed: 2025-08-05.

[10] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA.

[11] Anthropic. 2025. Claude 3.7 Sonnet. https://www.anthropic.com/news/claude-3-7-sonnet. Introduced extended thinking mode for advanced reasoning and problem-solving, with a context window of up to 128,000 tokens. Knowledge cutoff: October 2024..

[12] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*. Cascais, Portugal.

[13] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: a modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO)* (Amsterdam, The Netherlands). Berlin, Heidelberg.

[14] Aditya Bhuyan. 2024. How Arm's Success in Data Centers is Shaping the Future of Chip Technology. https://aditya-sunjava.medium.com/how-arms-success-in-data-centers-is-shaping-the-future-of-chip-technology-50a7748861f7. [Online; accessed: 7-Aug-2025].

[15] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. 2002. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Copenhagen, Denmark.

[16] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning, See [6].

[17] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification, See [4].

[18] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[19] Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, et al. 2024. Automated Proof Generation for Rust Code via Self-Evolution. *arXiv preprint arXiv:2410.15756* (2024).

[20] Xiangdong Chen, Zhaofeng Li, Jerry Zhang, Vikram Narayanan, and Anton Burtsev. 2025. Atmosphere: Practical Verified Kernels with Rust and Verus. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. New York, NY, 16 pages.

[21] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Baoxu Liu. 2023. Sherlock on Specs: Building LTE Conformance Tests through Automated Reasoning. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA.

[22] David Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure microkernels, state monads and scalable refinement. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 167–182.

[23] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: minimal hardware extensions for strong software isolation, See [3].

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Budapest, Hungary.

[25] DeepSeek AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. https://arxiv.org/abs/2501.12948. Released in January 2025. Achieves performance comparable to OpenAI-o1-1217 on reasoning tasks, leveraging multi-stage training and cold-start data..

[26] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. 2006. Running the manual: an approach to high-assurance microkernel development. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell (Haskell '06)*. New York, NY, 60–71.

[27] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software, See [4].

[28] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL.

[29] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*. Belgrade, RS.

[30] Anthony Fox. 2003. Formal Specification and Verification of ARM6. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, David Basin and Burkhart Wolff (Eds.). Rome, Italy.

[31] Anthony Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP)*. Edinburgh, UK.

[32] 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, See [7].

[33] Dapeng Gao and Tom Melham. 2021. End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers. In *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Virtual Event, CT.

[34] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and formal verification of x86 machine-code programs that make system calls. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Lausanne, Switzerland.

[35] Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos. 2024. IronSpec: Increasing the Reliability of Formal Specifications. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Santa Clara, CA.

[36] 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, See [2].

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

[38] R.A. Kemmerer. 1985. Testing Formal Specifications to Detect Design Errors. *IEEE Transactions on Software Engineering* SE-11, 1 (1985), 32–43. doi:10.1109/TSE.1985.231535

[39] Uri Kirstein. 2022. Post-Mortem Analysis of the Notional Finance Vulnerability — A Tautological Invariant. https://medium.com/certora/post-mortem-analysis-of-the-notional-finance-vulnerability-a-tautological-invariant-574d02d6ac15. [Online; accessed: 7-Aug-2025].

[40] 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 (SOSP)*. Big Sky, MT.

[41] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated.

[42] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jay Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*. Austin, TX.

[43] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types, See [7].

[44] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Berlin, Heidelberg, 348–370.

[45] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814

[46] Mengming Li, Wenji Fang, Qijun Zhang, and Zhiyao Xie. 2024. SpecLLM: Exploring Generation and Review of VLSI Design Specification with Large Language Model. *arXiv preprint arXiv:2401.13266* (2024).

[47] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Event, Canada.

[48] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. Virtual Event, CA.

[49] 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, See [6].

[50] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. *arXiv preprint arXiv:2401.08807* (2024).

[51] Meta AI. 2024. Llama 3.1 70B. https://ai.meta.com/llama/. Released on July 23, 2024.

[52] Kazi Samin Mubasshir, Imtiaz Karim, and Elisa Bertino. 2024. FBSDetector: Fake Base Station and Multi Step Attack Detection in Cellular Networks using Machine Learning. *arXiv preprint arXiv:2401.04958* (2024).

[53] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement, See [1].

[54] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel, See [4].

[55] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[56] OpenAI. 2024. GPT-4o. https://openai.com/gpt-4o. Released on May 13, 2024. Multimodal AI model integrating text, voice, and vision capabilities..

[57] OpenAI. 2024. GPT-o1. https://openai.com/index/openai-o1-system-card/. Released on December 5, 2024. Reflective generative pre-trained transformer with enhanced reasoning capabilities in STEM, programming, and complex tasks..

[58] Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically deriving JavaScript static analyzers from specifications using meta-level static analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1022–1034.

[59] Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu. 2021. JSTAR: JavaScript specification type analyzer using refinement. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 606–616.

[60] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. Jest: N+ 1-version differential testing of both javascript engines and specification. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 13–24.

[61] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. 2022. UCLID5: Multi-Modal Formal Modeling, Verification, and Synthesis. In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV)*. Haifa, Israel.

[62] Mirza Masfiqur Rahman, Imtiaz Karim, and Elisa Bertino. 2024. CellularLint: A Systematic Approach to Identify Inconsistent Behavior in Cellular Network Specifications. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5215–5232.

[63] Syed Md Mukit Rashid, Tianwei Wu, Kai Tu, Abdullah Al Ishtiaq, Ridwanul Hasan Tanvir, Yilu Dong, Omar Chowdhury, and Syed Rafiul Hussain. 2024. State Machine Mutation-based Testing Framework for Wireless Communication Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2102–2116.

[64] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Mountain View, CA.

[65] Alastair Reid. 2017. Who guards the guards? formal validation of the Arm v8-m architecture specification. In *Proceedings of the 32nd Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Vancouver, Canada.

[66] 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 *Proceedings of the 28th International Conference on Computer Aided*

Verification (CAV). Toronto, Canada.

[67] Muhammad Usama Sardar, Thomas Fossati, Simon Frost, and Shale Xiong. 2023. Formal specification and verification of architecturally-defined attestation mechanisms in arm cca and intel tdx. *IEEE Access* 12 (2023), 361–381.

[68] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*. Savannah, GA.

[69] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. 2011. seL4 enforces integrity. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP)*. Nijmegen, The Netherlands.

[70] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement, See [2].

[71] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[72] 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 *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*. Virtual Event, Germany.

[73] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2013. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework, See [1].

[74] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. 2016. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor, See [3].

[75] Jeff Wu. 2022. Critical Bug Payout Report. https://blog.notional.finance/critical-bug-payout-report. [Online; accessed: 7-Aug-2025].

[76] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R Lorch, Shuai Lu, et al. 2024. AutoVerus: Automated proof generation for Rust code. *arXiv preprint arXiv:2409.13082* (2024).

[77] Jean Yang and Chris Hawblitzel. 2010. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Canada.

[78] Dongjun Youn, Wonho Shin, and Sukyoung Ryu. 2025. WEST: Specification-Based Test Generation for WebAssembly. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM.

[79] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Ontario, Canada.

[80] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of the 2017 ACM SIGCOMM*. Los Angeles, CA.

[81] Naiqian Zheng, Mengqi Liu, Yuxing Xiang, Linjian Song, Dong Li, Feng Han, Nan Wang, Yong Ma, Zhuo Liang, Dennis Cai, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2023. Automated Verification of an In-Production DNS Authoritative Engine. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*. Koblenz, Germany.

[82] Sicheng Zhong, Jiading Zhu, Yifang Tian, and Xujie Si. 2025. RAG-Verus: Repository-Level Program Verification with LLMs using Retrieval Augmented Generation. *arXiv preprint arXiv:2502.05344* (2025).