arm

Verification of the Realm Management Monitor ABI

AI-generated image

OVERVIEW

- ARM CCA Overview
- RMM Specification
- RMM Verification
- Conclusion

Confidential Computing

Motivation

- Applications processing sensitive data increasingly run on cloud providers.
- Applications must trust supervisor software, e.g. OS kernels and hypervisors. Those components can contain vulnerabilities that risk data confidentiality and integrity.

Solution

- Confidential computing removes the supervisor's right to access (read/modify) the resources used by the application, while retaining the right to manage them.

ARM CCA Overview

Arm CCA architecture overview

- **Arm CCA** is Arm's new trusted execution environment (TEE) solution for confidential compute. It introduces a new kind of environment called Realm.

Hardware Component

RME

Comprises of four worlds of different security domains. (Secure, Non-secure, **Realm**, Root)

Software Component

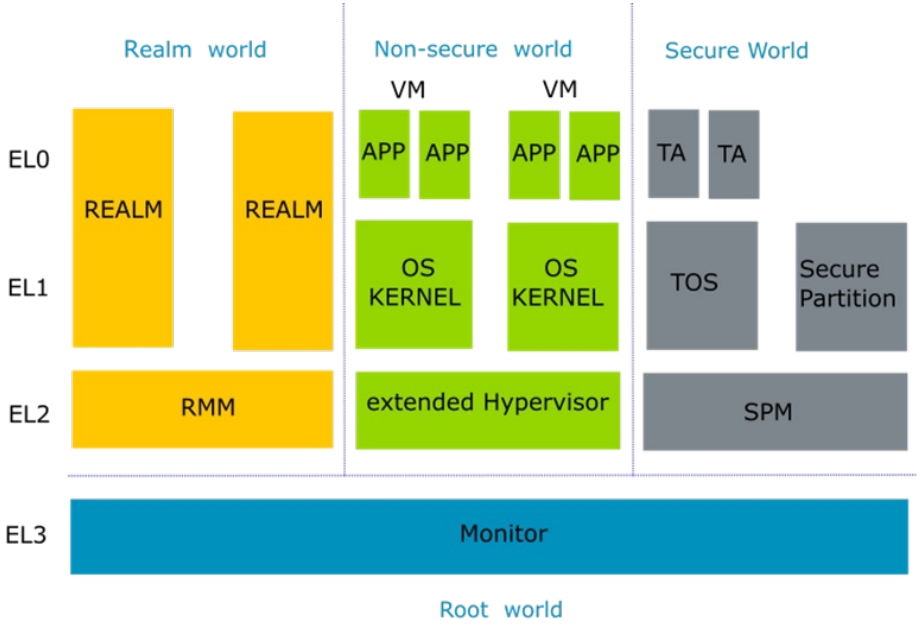
RMM

Constitutes the Realm world firmware of Arm CCA.

Arm CCA architecture overview

The **architecture** comprises of **four** worlds:

- **Non Secure world**: Hosts feature rich operating systems and hypervisors.
- **Realm World**: Offer Realms, which host protected virtual machines.
  - Realms can be destroyed and created dynamically.
  - Are used and provided by software executing in the *non-secure* world.
  - Provide an environment for confidential computing.
  - Realm owners need not trust the (non secure) software components that manage the Realm's resources.
- **Root World**: Hosts the monitor, which is responsible for transitioning between worlds.
- **Secure World**: Hosts trusted applications that run under a trusted operating system.
  - Constrained and not dynamically-resized memory.
  - Configured and provisioned by the platform manufacturer.



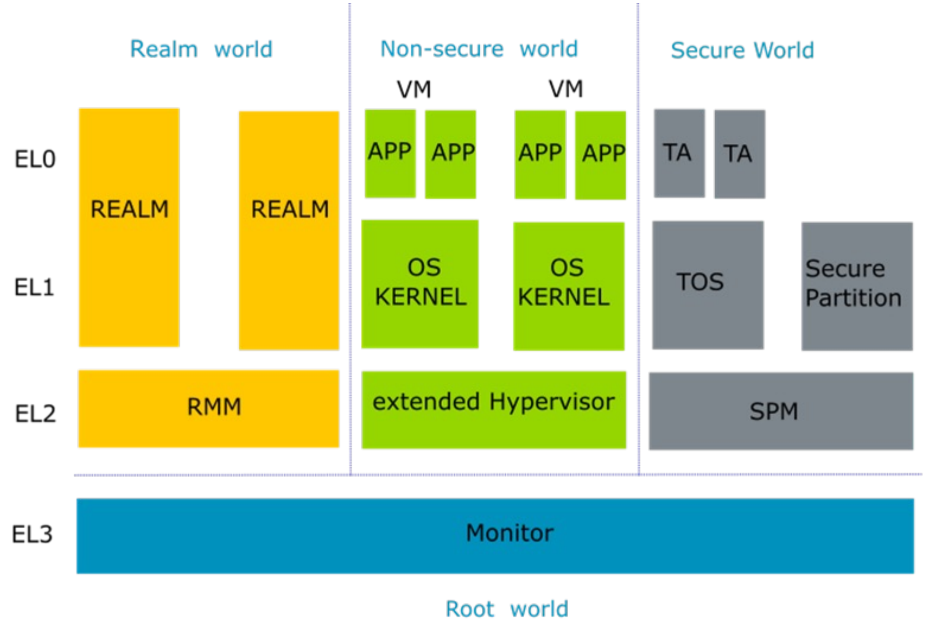
Arm CCA trust model

Realms **trust**

- Arm CCA hardware (RME)
- Own OS (R-EL0 trusts R-EL1)
- Firmware: RMM and monitor code (TF-A)

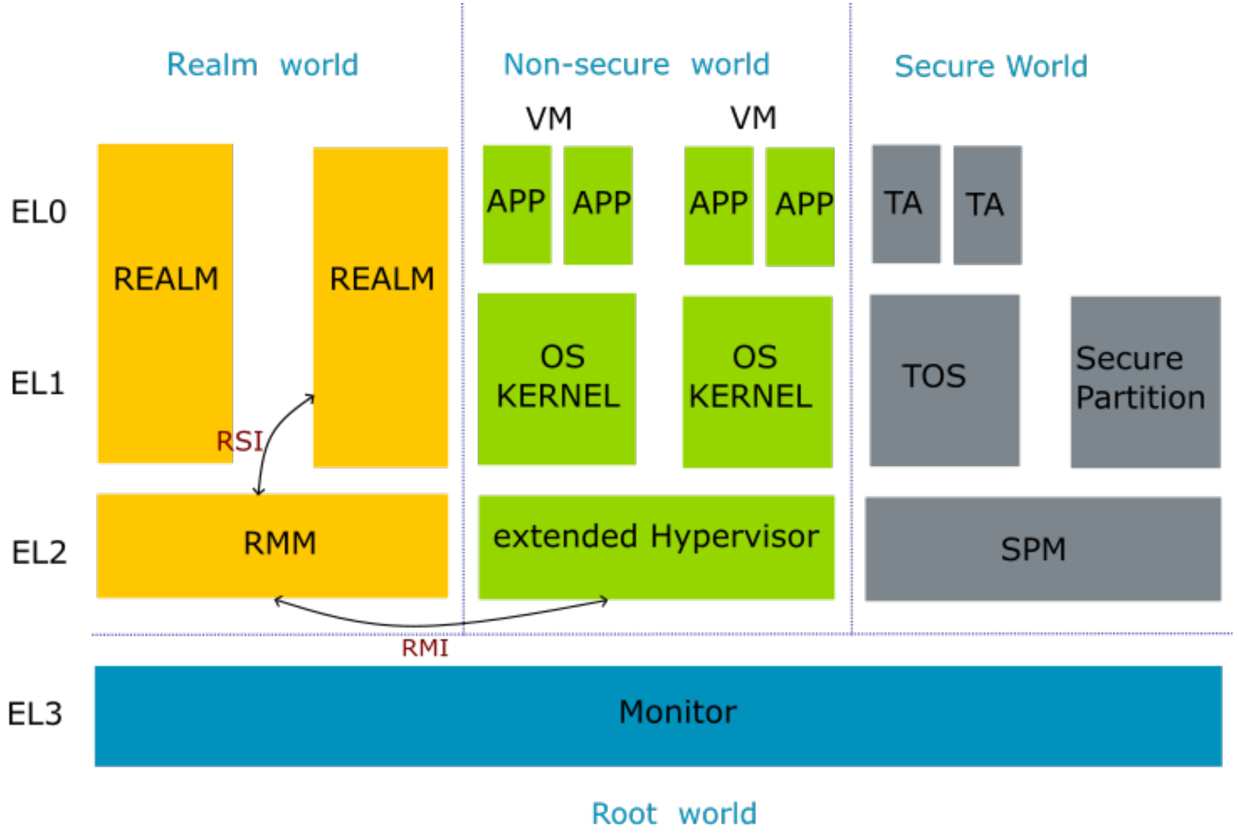
Realms **do not trust**

- Non-secure host code (hypervisor and NS OS)
- Secure world



RMM interface overview

- RMM is responsible for protecting the confidentiality and integrity of the Realm.
- Provides cryptographic services that allow a Realm to protect assets which are associated with it.
- Provides a Realm Management Interface (**RMI**) which allows the host (extended hypervisor) to manage the life cycle of the realm indirectly.
- Provides Realm Services Interface (**RSI**) for attestation and other realm services.



Realm Memory Management

- A key element of confidential compute is ***memory management***

- Physical memory split into 4K granules (pages)

- A **granule protection** mechanism controls **cross-world** access to granules
  - Protection tables are managed by monitor at EL3

- RMM maintains **stage-2 page tables**
  - Provides cross-Realm memory protection
  - Stage 1 tables map virtual addresses to intermediate physical addresses (IPAs)
  - Stage 2 tables map IPA to physical addresses (PAs)

RMM Specification and Verification

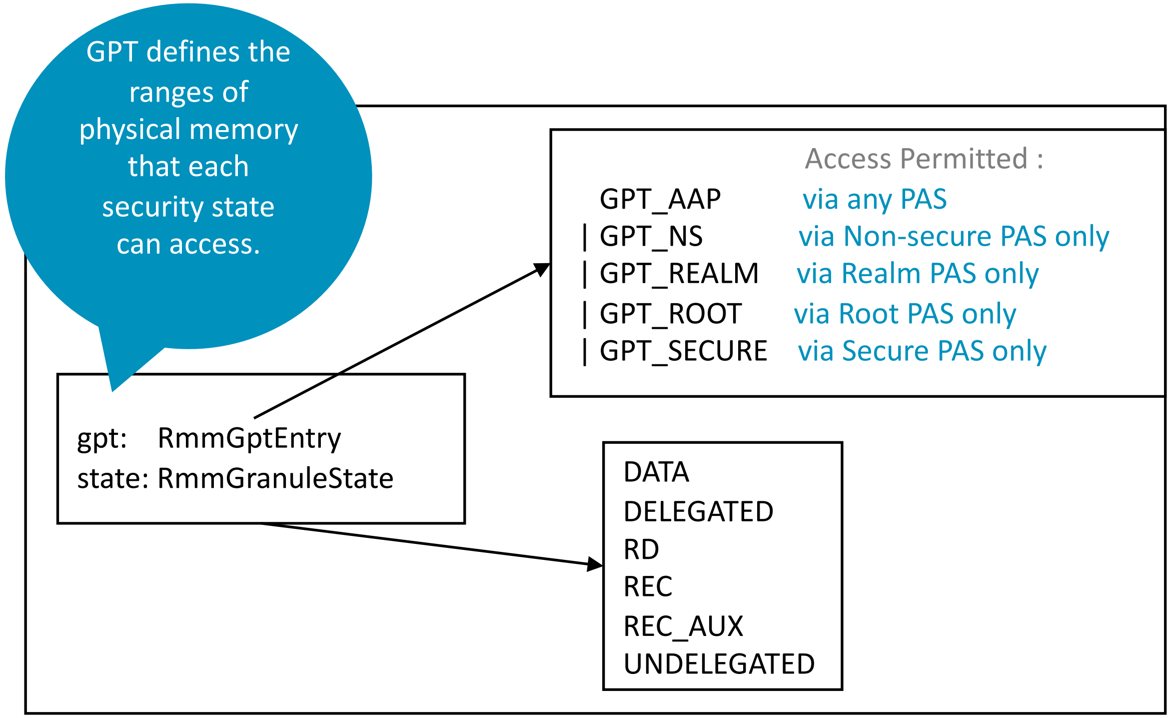
- We have verified the RMM *specification* (using HOL4)
- We have also verified a C code *implementation* (using CBMC)
- The specification is presented in a PDF
- We translate this into HOL4 code
  - We validate the model
  - We verify an invariant

RMM SPECIFICATIONION

The published version of RMM specification is v1.0 however many silicon vendors target v1.1 which is currently in alpha stage.

RMM state

- The RMM ABI is described in the document (DEN0137_1.0-EAC5)
- The specification presents several abstract types for representing RMM objects.
- RMM objects are described by granules + metadata associated with them.



Granule State

State	Granule Object
UNDELEGATED	Not realm world granule
DELEGATED	Realm world granule
RD	Realm Descriptor
DATA	Protected Data
RTT	Realm Translation Table
REC	Realm Execution Context

Non-Secure/Secure/Root PAS

Realm PAS

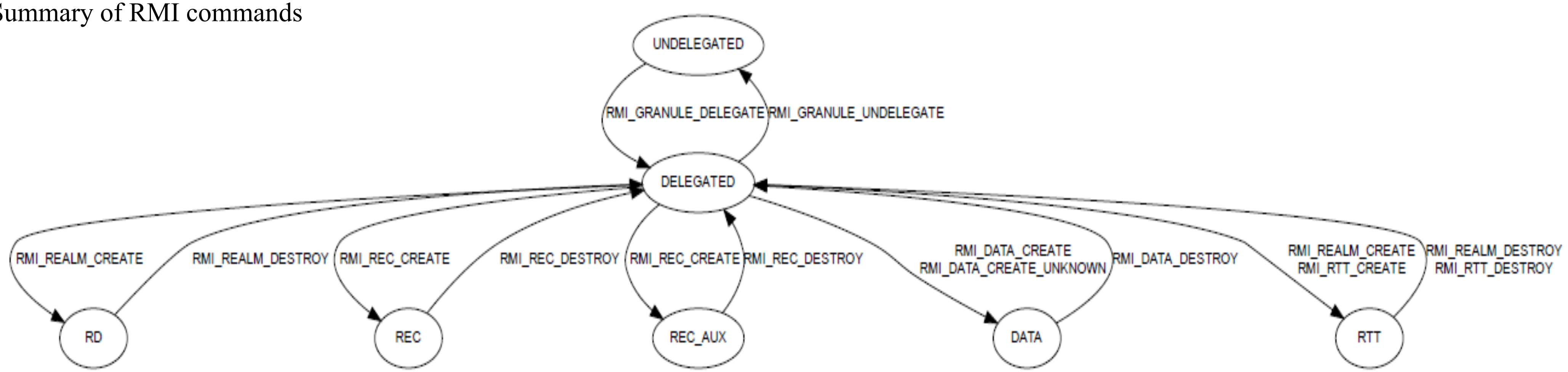
State Object association

RMM commands

RMI commands change the state and ownership of granules

Controlling realm life cycle	Transitioning between Realm and NS world	Controlling Realm Executions Context	Controlling page tables	Controlling data life cycle
RMI_REALM_CREATE RMI_REALM_ACTIVATE RMI_REALM_RUN RMI_REALM_DESTROY	RMI_GRANULE_DELEGATE RMI_GRANULE_UNDELEGATE	RMI_REC_CREATE RMI_REC_ENTER RMI_REC_AUX_COUNT RMI_REC_DESTROY	RMI_RTT_CREATE RMI_RTT_DESTROY RMI_RTT_FOLD RMI_RTT_INIT_RIPAS RMI_RTT_READ_ENTRY RMI_RTT_SET_RIPAS RMI_RTT_MAP_UNPROTECTED RMI_RTT_UNMAP_UNPROTECTED	RMI_DATA_CREATE RMI_DATA_CREATE_UNKNOWN RMI_DATA_DESTROY

Summary of RMI commands



Commands in RMM specification

The RMM command definition consist of:

- A function identifier (FID): A value which identifies a particular RMM command.
- A set of input values: Values read by the RMM command from general-purpose registers.
- A set of output values: Values written by the RMM command to general-purpose registers.
- Failure conditions: Pairs of pre (triggers the failure) and post (constrains post-state after failure) conditions.
- A success condition: Constrains the post state after successful execution of the RMM command.
- Footprint items: Lists of state attributes that **can** be updated on successful execution of the command.

Example of RMM command: RMI.RTT.Destroy

B4.3.16.2 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)`
level_bound	pre: `(!RttLevelIsValid(rd, level) || RttLevelIsStarting(rd, level))` post: `ResultEqual(result, RMI_ERROR_INPUT)`
ipa_align	pre: `!AddrIsRttLevelAligned(ipa, level - 1)` post: `ResultEqual(result, RMI_ERROR_INPUT)`
ipa_bound	pre: `UInt(ipa) >= (2 ^ Realm(rd).ipa_width)` post: `ResultEqual(result, RMI_ERROR_INPUT)`
rtt_walk	pre: `walk.level < level - 1` post: `(ResultEqual(result, RMI_ERROR_RTT, walk.level) && (top == walk_top))`
rtte_state	pre: `walk.rtte.state != TABLE` post: `(ResultEqual(result, RMI_ERROR_RTT, walk.level) && (top == walk_top))`
rtt_live	pre: `RttIsLive(Rtt(walk.rtte.addr))` post: `(ResultEqual(result, RMI_ERROR_RTT, level) && (top == ipa))`

B4.3.16.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC400015E`
rd	X1	63:0	Address	PA of the RD for the target Realm
ipa	X2	63:0	Address	Base of the IPA range described by the RTT
level	X3	63:0	Int64	RTT level

B4.3.16.3 Success conditions

ID	Condition
rtte_state	`walk.rtte.state == UNASSIGNED`
ripas	`walk.rtte.ripas == DESTROYED`
rtt_state	`Granule(walk.rtte.addr).state == DELEGATED`
rtt	`rtt == walk.rtte.addr`
top	`top == walk_top`

B4.3.16.1.3 Output values

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

B4.3.16.4 Footprint

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

Page table entries within the RMM

Host view (HIPAS value)

ASSIGNED: Address associated with **DATA** Granule.

UNASSIGNED: Address associated with any Granule.

Realm view (RIPAS value)

EMPTY: Address where no Realm resources are mapped.

RAM: Address where private code or data owned by the Realm is mapped.

DESTROYED: Address which is inaccessible to the Realm due to an action taken by the Host.

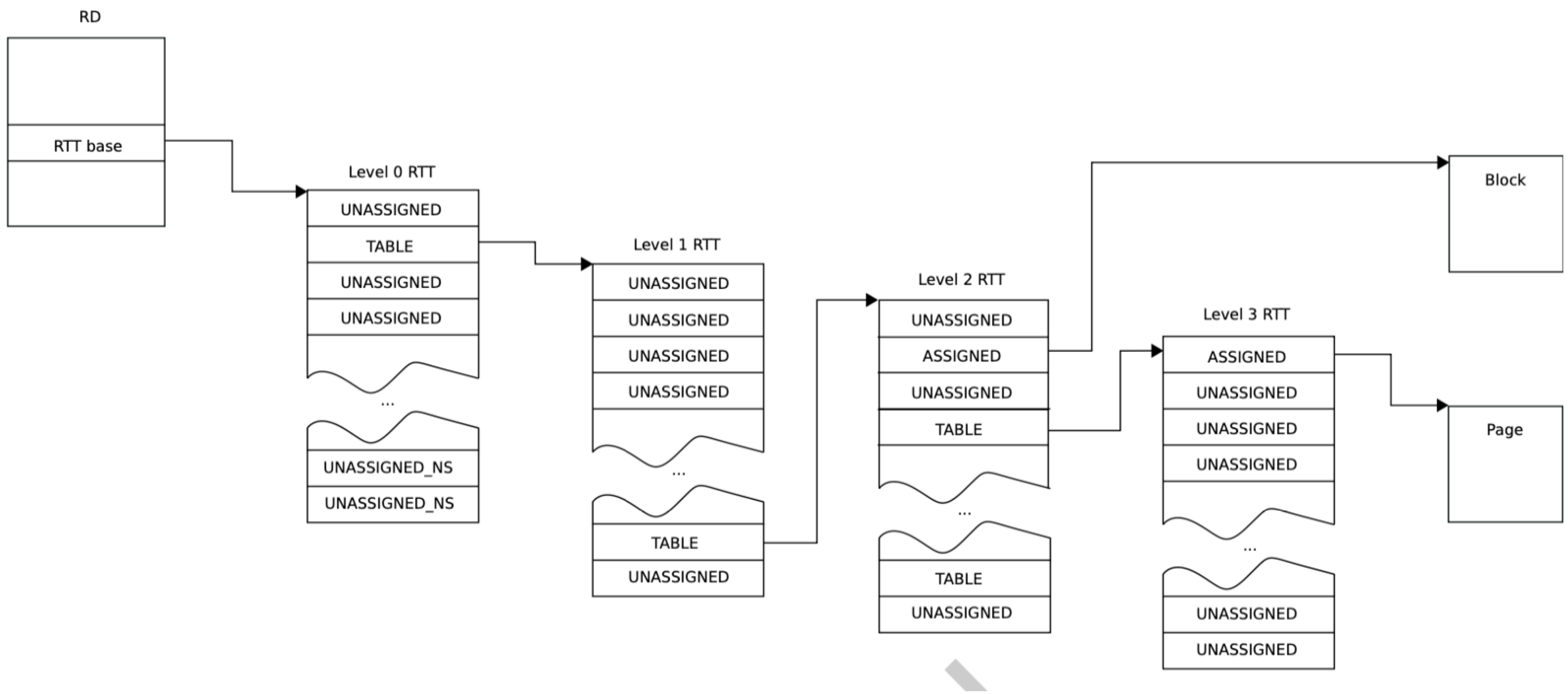
Protected IPA

Unprotected IPA

ASSIGNED_NS: Address associated with an NS granule.

UNASSIGNED_NS: Address **not** associated with any Granule

Page Table Walks



$walk \ = \ RttWalk\,(rd, ipa, level)$

Constrains the walk to be the result of attempting a page table walk under

- Realm = rd
- IPA (Intermediate Physical Address) = ipa
- Requested depth of walk = $level$

Walk in RMM specification

- HIPAS and RIPAS values are stored in Realm Translation Tables (RTTs).
- The RTT starting level (RTT level of the root of an RTT tree) is set when a Realm is created and the address of the first starting level RTT is stored in the rtt_base attribute of the owning Realm.
- An RTT entry contains an output address which can point to one of the following:
  - Another RTT
  - A DATA Granule which is owned by the Realm
  - Non-Secure memory which is accessible to both the Realm and the Host

RMM VERIFICATION

RMM verification

Verification efforts

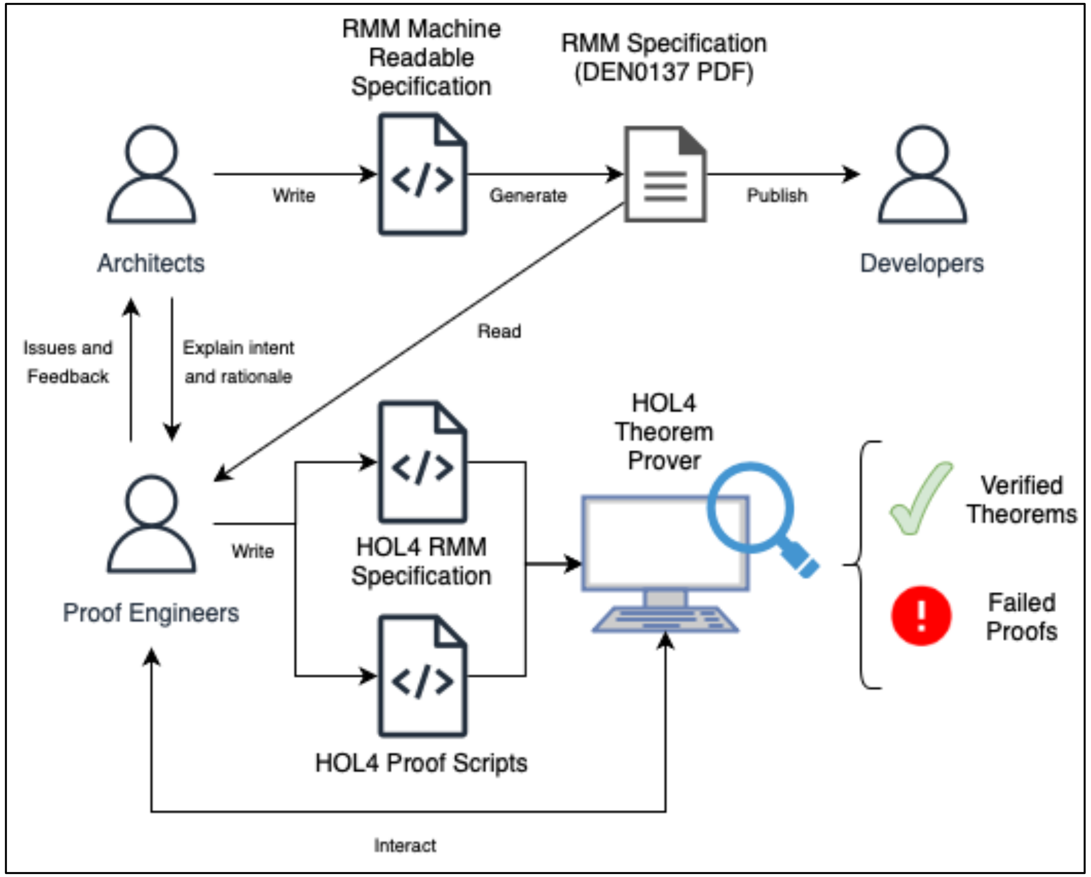
- **CBMC Verification of RMM implementation**: Verifies that the reference implementation satisfies the specification.

- **HOL4 Verification of RMM specification**: Validates the coherence of the specification by proving invariants and properties.

HOL4 model and proofs

Objective

- Identify as early as possible parts of the source specification that are unclear or ambiguous.
- Prove that important invariants as well as desirable security properties for Realms are enforced by the specification.



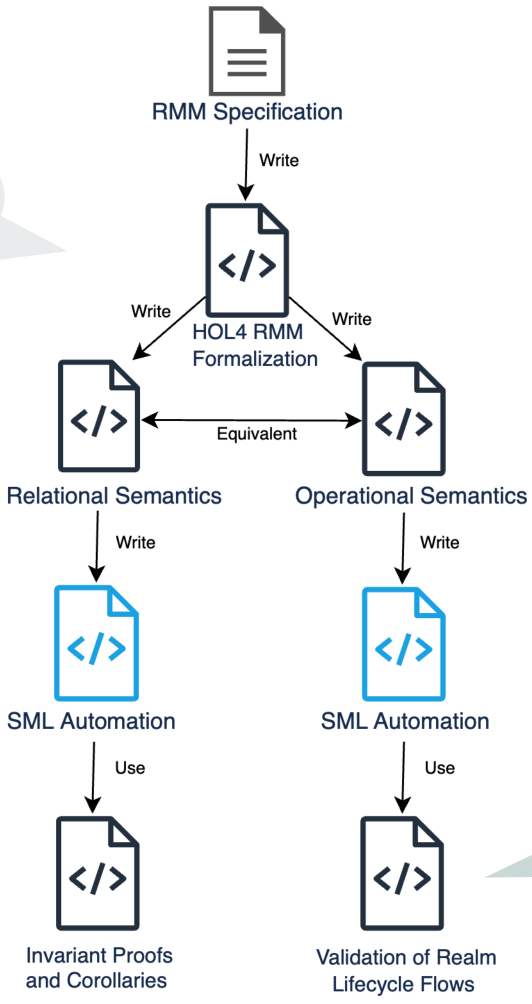
HOL4 verification workflow

HOL4 model and proofs

- Are the spec. definitions clear and accurate?
- Does the text describe the definitions, adequately convey their intended meaning?
- Are any definitions missing?

- Missing inputs/outputs?
- Are the commands sound?
- Missing pre/post conditions?
- Unsatisfiable pre/post conditions?

- What security properties must be maintained during execution?
- Does the spec. adhere to these properties?

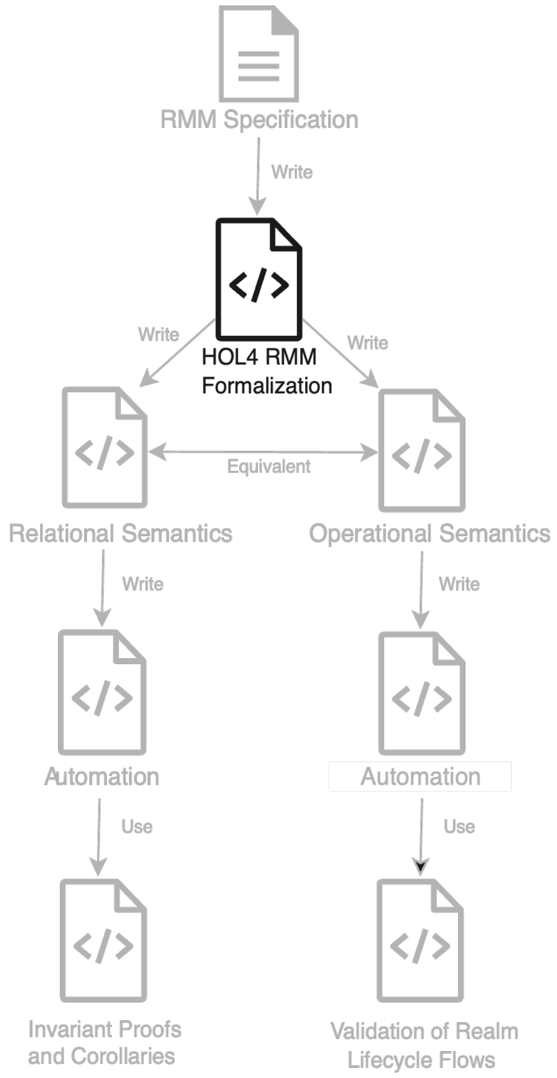


- Do RMM commands behave as expected?
- What is the set of reachable states?

HOL4 model and proofs

HOL4 state

- We translate the RMI and RSI commands into a HOL4 relation manually.
- We try to keep the formalization as close as possible to the specification.
- The formalization process helps us identify parts of specification that are ambiguous.
- Manual translation allows as to handle new features selectively.



HOL4 model and proofs

HOL4 state

```
Datatype: RmmGptEntry =
  | GPT_AAP
  | GPT_NS
  | GPT_REALM
  | GPT_ROOT
  | GPT_SECURE
End

Datatype: RmmGranuleState =
  | DATA
  | DELEGATED
  | RD
  | REC
  | REC_AUX
  | RTT
  | UNDELEGATED
End

Datatype: RmmGranule = <|
    gpt   : RmmGptEntry;
    state : RmmGranuleState
  |>
```

```
Datatype: Object =
  | REC_object   RmmRec
  | Realm_object RmmRealm
  | RTT_object   RmmRtt
  | NS_object    NonSecureObject
  | NULL
End
```

```
Datatype: State = <|
    GPR     : word64[32];
    Granule   Address |-> RmmGranule # Object
  |
End
```

The HOL4 formalization of the state

arm

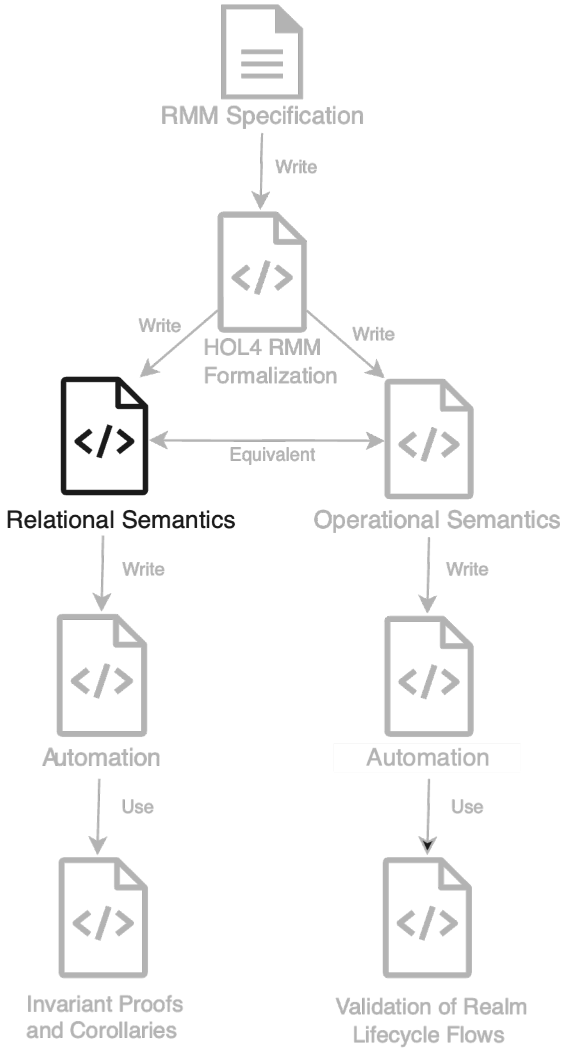
HOL4 model and proofs

Relational Semantics

- The relational semantics for the RMI and RSI commands are modeled with the predicates “RMI_COMMAND *pre post*” and “RSI_COMMAND *pre post*” respectively.

- “RMI_COMMAND *pre post*”: Is true if there is an RMI command that can transition from state “*pre*” to state “*post*”.

- “RSI_COMMAND *pre post*” Is true if there is an RSI command that can transition from state “*pre*” to state “*post*”.



HOL4 model and proofs

Relational Semantics

```
rmm_assertTheory.RMM_ASSERT (DEFINITION)
----------------------------------------

⊢ ∀pre post pre_conditions post_conditions footprint.
    RMM_ASSERT pre post pre_conditions post_conditions footprint ⇔
    ASSERT pre post (WELL_FORMED INSERT pre_conditions)
      ({WELL_FORMED; FOOTPRINT pre footprint} ∪ post_conditions)
```

Spec. failure conditions

Spec. footprint

Spec. success conditions

arm

HOL4 model and proofs

Relational Semantics

```
Inductive RMI_GRANULE_COMMAND:
  (* RMI_GRANULE_DELEGATE failure case *)
  (!pre post.
     RMM_ASSERT pre post
       { X 0 RMI_GRANULE_DELEGATE;
         X 1 addr;
         Granule addr granule_addr;
         GranuleDelegateError (addr, granule_addr) result }
       { X 0 result }
       { GPR 0 } ==>
     RMI_GRANULE_COMMAND pre post)
/\
  (* RMI_GRANULE_DELEGATE success case *)
  (!pre post.
     RMM_ASSERT pre post
       { X 0 RMI_GRANULE_DELEGATE;
         X 1 addr;
         Granule addr granule_addr;
         NoGranuleDelegateError (addr, granule_addr) result }
       { X 0 result;
         Granule addr granule_addr';
         K (granule_addr' = granule_addr with
                                <| state := DELEGATED;
                                   gpt := GPT_REALM |>) }
       { GPR 0; GRAN addr; OBJ addr } ==>
     RMI_GRANULE_COMMAND pre post)
/\
  (* RMI_GRANULE_UNDELEGATE failure case *)
/\
  (* RMI_GRANULE_UNDELEGATE succes case *)
End
```

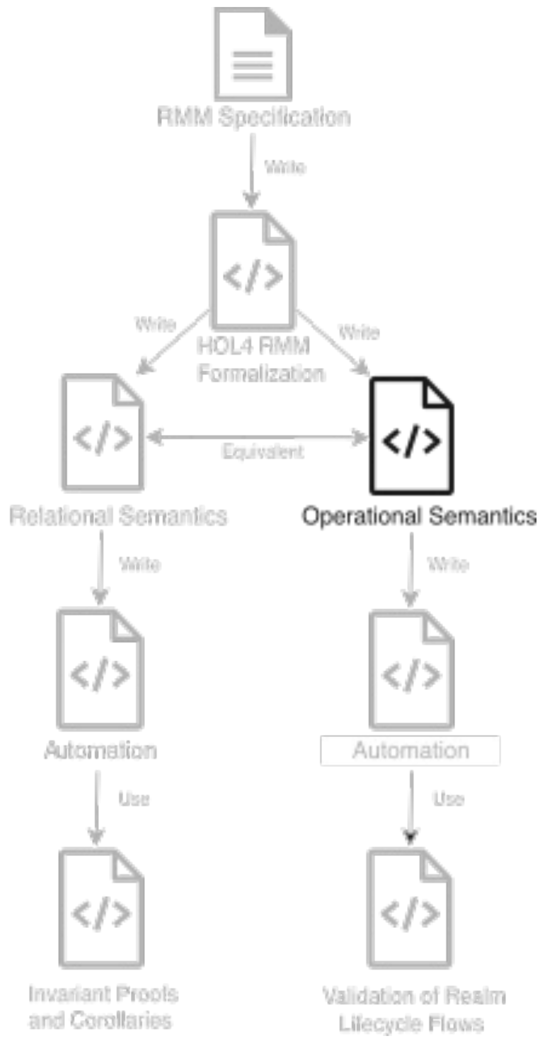
```
Definition RMI_COMMAND:
  RMI_COMMAND pre post <=>
  RMI_DATA_COMMAND pre post \/
  RMI_GRANULE_COMMAND pre post \/
  RMI_REALM_COMMAND pre post \/
  RMI_REC_COMMAND pre post \/
  RMI_RTT_COMMAND pre post \/
  RMI_OTHER_COMMAND pre post \/
  RMI_DA_COMMAND pre post
End
```

Definition of RMI_COMMAND predicate

HOL4 model and proofs

Operational Semantics

- The operational semantics is defined as HOL4 functions that implement RMM commands.
- We prove that the HOL4 function that model each RMI/RSI command satisfy its corresponding axiomatic specification.
- The operational semantics is executable and can be used to run the model.



HOL4 model and proofs

Operational Semantics

```
Definition RmiNext:
  RmiNext state =
  let fid = state.GPR ' 0 in
    if fid = RMI_FEATURES then
      RmiFeaturesNext state
    else if fid = RMI_GRANULE_DELEGATE then
      RmiGranuleDelegateNext state
    else if fid = RMI_GRANULE_UNDELEGATE then
      RmiGranuleUndelegateNext state
    else if fid = RMI_REALM_ACTIVATE then
      RmiRealmActivateNext state
    else if fid = RMI_REALM_CREATE then
      RmiRealmCreateNext state
    else if fid = RMI_REALM_DESTROY then
      RmiRealmDestroyNext state
    else if fid = RMI_REC_AUX_COUNT then
    . . .
  End
```

RmiNext definition

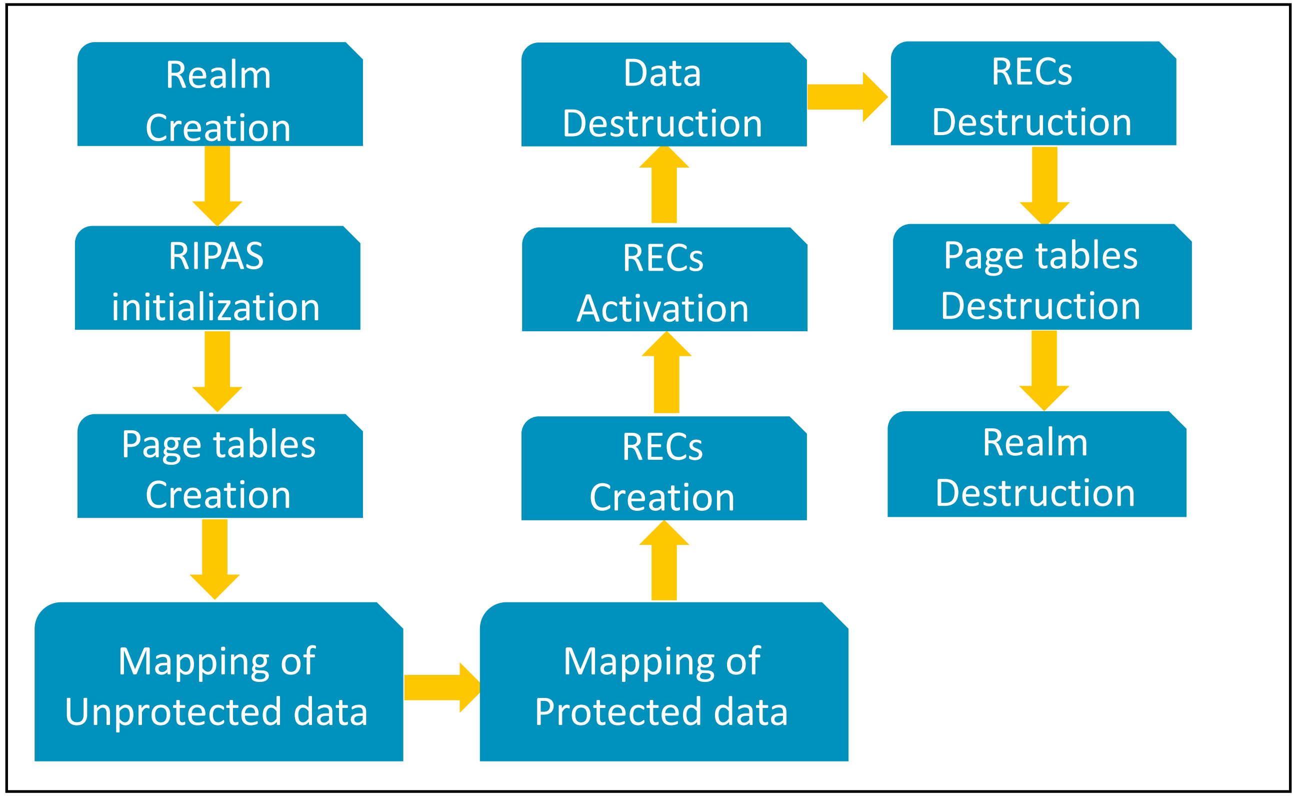
```
val RMI_COMMAND_OPSEM =
   ⊢ ∀state.
       state ∈ WELL_FORMED ∧
       state.GPR ' 0 ∈
       {RMI_FEATURES; RMI_GRANULE_DELEGATE;
        RMI_GRANULE_UNDELEGATE; RMI_REALM_ACTIVATE;
        RMI_REALM_CREATE; RMI_REALM_DESTROY;
        RMI_REC_AUX_COUNT; RMI_REC_CREATE; RMI_REC_DESTROY;
        RMI_REC_ENTER; RMI_VERSION; RMI_DATA_CREATE;
        RMI_DATA_CREATE_UNKNOWN; RMI_RTT_MAP_UNPROTECTED;
        RMI_RTT_UNMAP_UNPROTECTED; RMI_RTT_READ_ENTRY;
        RMI_RTT_INIT_RIPAS; RMI_RTT_SET_RIPAS;
        RMI_RTT_CREATE; RMI_RTT_FOLD; RMI_RTT_DESTROY;
        RMI_DATA_DESTROY} ⇒
       RMI_COMMAND state (RmiNext state): thm
```

Theorem for soundness of operational semantics

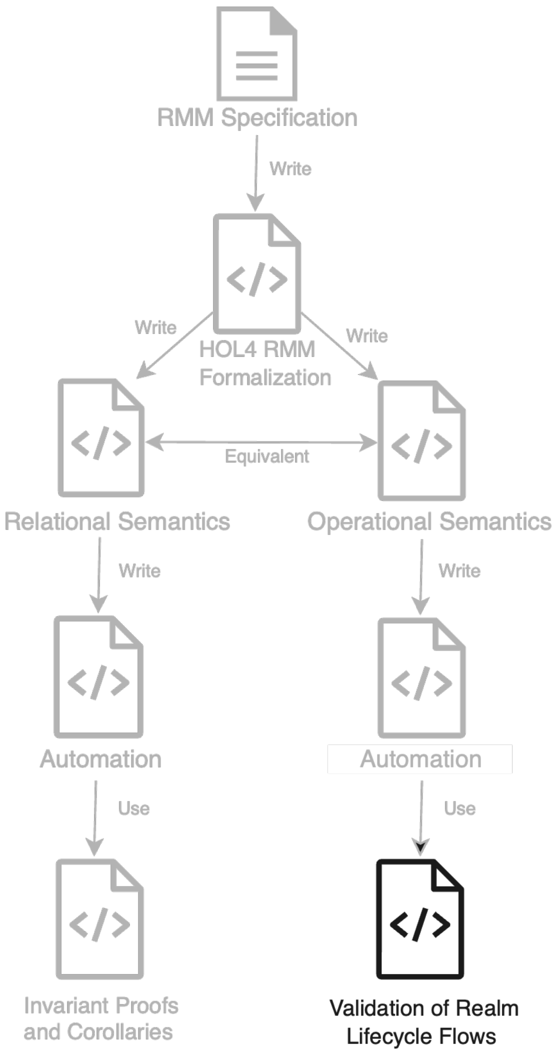
HOL4 model and proofs

Exploring and validating realm lifecycle flows

- We use the verified operational semantics and HOL4 automation to to run the specification.



Main realm lifecycle flow



HOL4 model and proofs

Exploring and validating realm lifecycle flows

```
Datatype: Command =
   | RMI
   | RSI Address Address
   | Move num word64
   | Store Address (NonSecureObject option)
End
```

Command Datatype

```
Definition COMMANDS:
  (COMMANDS pre [] post = (pre = post)) /\
  (COMMANDS pre (command::commands) post =
     ?next. COMMAND pre command next /\ COMMANDS next commands
post)
End
```

COMMANDS Definition

- The automation takes a theorem

  $\vdash$ $COMMANDS\ pre\ c\ post$ and

  a sequence of commands s and returns

  $\vdash$ $COMMANDS\ pre\ (c\ ++\ s)\ post'$

HOL4 model and proofs

Exploring and validating realm lifecycle flows

```
val rd = ``0w: Address``
val rtt_base = ``0x4000w: Address``
val params_ptr = ``0x1000w: Address``

val params =
  ``RealmParams_object
    <| s2sz := 48w;
       rtt_level_start := 0;
       rtt_num_start := 1;
       rtt_base := ^rtt_base;
       hash_algo := RMI_HASH_SHA_256;
       rpv := 0w;
       vmid := 0w
    |>``;

val RealmCreate = commands0 ``initial_state`` |> commands`
  Move 0 RMI_GRANULE_DELEGATE,
  Move 1 ^rd,
  RMI,
  Move 0 RMI_GRANULE_DELEGATE,
  Move 1 ^rtt_base,
  RMI,
  Store ^params_ptr (SOME ^params1),
  Move 0 RMI_REALM_CREATE,
  Move 1 ^rd,
  Move 2 ^params_ptr,
  RMI,
  Store ^params_ptr NONE`
```

Realm Creation example

B4.3.9 RMI_REALM_CREATE command

Creates a Realm.

See also:

- A2.1 *Realm*
- A2.1.6 *Realm parameters*
- B4.3.10 *RMI_REALM_DESTROY command*
- D1.2.1 *Realm creation flow*

B4.3.9.1 Interface

B4.3.9.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000158`
rd	X1	63:0	Address	PA of the RD
params_ptr	X2	63:0	Address	PA of Realm parameters

B4.3.9.1.2 Context

The RMI_REALM_CREATE command operates on the following context.

Name	Type	Value	Before	Description
params	RmiRealmParams	`RealmParams(params_ptr)`	false	Realm parameters

B4.3.9.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B4.3.9.2 Failure conditions

ID	Condition
params_align	`pre: !AddrIsGranuleAligned(params_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
params_bound	`pre: !PaIsDelegable(params_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
params_pas	`pre: !GranuleAccessPermitted(params_ptr, PAS_NS)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
params_valid	`pre: !RmiRealmParamsIsValid(params_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
params_supp	`pre: !RealmParamsSupported(params)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

HOL4 model and proofs

Exploring and validating realm lifecycle flows

```
val rd = ``0w: Address``
val rtt_base = ``0x4000w: Address``
val params_ptr = ``0x1000w: Address``

val params =
  ``RealmParams_object
    <| s2sz := 48w;
       rtt_level_start := 0;
       rtt_num_start := 1;
       rtt_base := ^rtt_base;
       hash_algo := RMI_HASH_SHA_256;
       rpv := 0w;
       vmid := 0w
    |>``;

val RealmCreate = commands0 ``initial_state`` |> commands`
  Move 0 RMI_GRANULE_DELEGATE,
  Move 1 ^rd,
  RMI,
  Move 0 RMI_GRANULE_DELEGATE,
  Move 1 ^rtt_base,
  RMI,
  Store ^params_ptr (SOME ^params1),
  Move 0 RMI_REALM_CREATE,
  Move 1 ^rd,
  Move 2 ^params_ptr,
  RMI,
  Store ^params_ptr NONE`
```

Realm Creation example

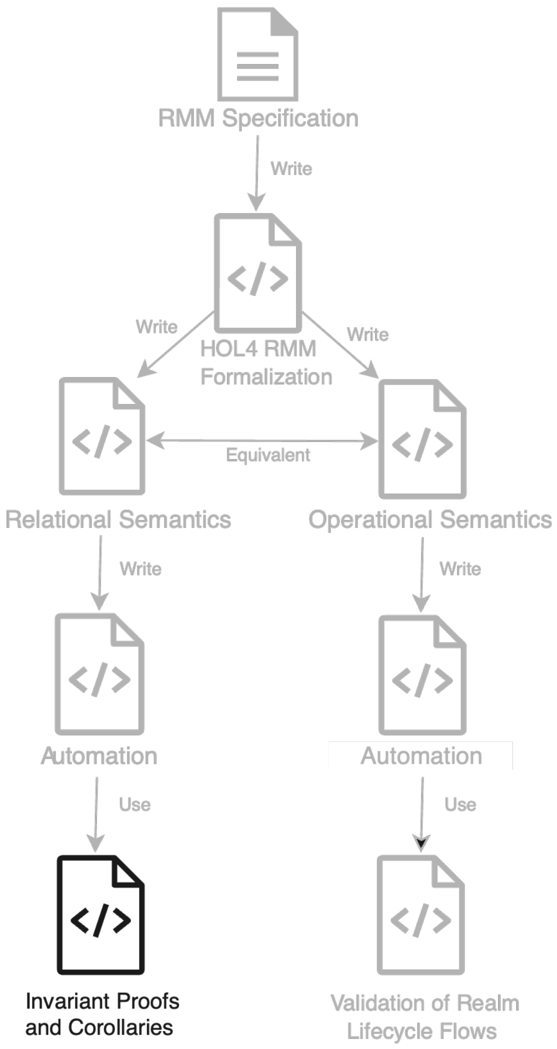

```
val RealmCreate =
   [..]
 ⊢ COMMANDS initial_state
     [
       Move 0 RMI_GRANULE_DELEGATE;
       Move 1 0x0w;
       RMI;
       Move 0 RMI_GRANULE_DELEGATE;
       Move 1 0x4000w;
       RMI;
       Store 0x1000w
         (SOME
            (RealmParams_object
               <|s2sz := 0x30w; hash_algo := RMI_HASH_SHA_256; rpv := 0x0w;
                 vmid := 0x0w; rtt_base := 0x4000w; rtt_level_start := 0;
                 rtt_num_start := 1|>));
       Move 0 RMI_REALM_CREATE;
       Move 1 0x0w;
       Move 2 0x1000w;
       RMI;
       Store 0x1000w NONE
     ]
     <|GPR := (0 :+ 0x0w) ((1 :+ 0x0w) ((2 :+ 0x1000w) ARB.GPR));
       Granule :=
         all_undelegated (
           0x0w ↦
             (<|gpt := GPT_REALM; state := RD|>,
              Realm_object
                <|ipa_width := 48;
                  measurements :=
                    FCP i.
                      if i = 0 then
                        RimInit RMI_HASH_SHA_256
                          <|s2sz := 0x30w; hash_algo := RMI_HASH_SHA_256;
                            rpv := 0x0w; vmid := 0x0w; rtt_base := 0x4000w;
                            rtt_level_start := 0; rtt_num_start := 1|>
                      else 0x0w; hash_algo := RMI_HASH_SHA_256;
                  rec_index := 0; rtt_base := 0x4000w;
                  rtt_level_start := 0; rtt_num_start := 1; state := NEW;
                  vmid := 0x0w; rpv := 0x0w|>);
           0x1000w ↦ (<|gpt := GPT_NS; state := UNDELEGATED|>,NULL);
           0x4000w ↦
             (<|gpt := GPT_REALM; state := RTT|>,
              RTT_object
                (λindex.
                     if w2n index < 256 then UNASSIGNED EMPTY_
                     else UNASSIGNED_NS))
         ) |>: thm
```

HOL4 model and proofs

RMM Invariant

- The invariant expresses important security properties about the components of the state.
- It splits in four main parts

```
Definition OWNERSHIP_INVARIANT:
  OWNERSHIP_INVARIANT =
  BIGINTER { REALM_INVARIANT; REC_INVARIANT; REC_AUX_INVARIANT;
WALK_INVARIANT }
End
```



HOL4 model and proofs

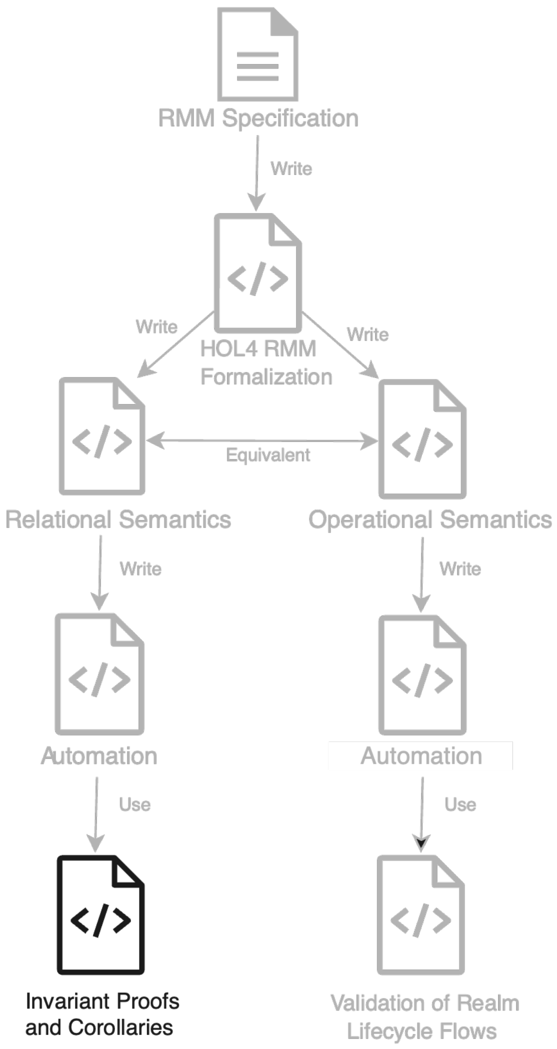
RMM Invariant

Realm invariant

- The attributes $realm.ipa\_width$,$realm.rtt\_level\_start$ and $realm.rtt\_num\_start$ are consistent and valid according to the VMSA.
- $realm.ipa\_width$ is less than or equal to RMM_MAX_S2SZ.
- Each of the root page table granules are delegable and are in state RTT. There are $realm.rtt\_num\_start$ such granules, starting at address $realm.rtt\_base$.

REC invariant

- There is no aliasing of the identifier $rec.mpidr$.
- There exists a Realm object realm, located at $rec.owner$, which is the REC's owner.
- $rec.mpidr$ is lower than $realm.rec\_index$.
- There exists a collection of REC_AUX objects associated with the REC that can be uniquely located via the array rec.aux. [The number of REC_AUX objects is Realm dependent. …



HOL4 model and proofs

Walk entry invariant

```
Definition WALK_ENTRY_INVARIANT:
  WALK_ENTRY_INVARIANT state rd (realm: RmmRealm) ipa walk =
  case walk.rtte of
  | TABLE a => walk.level < RMM_RTT_PAGE_LEVEL /\ WALK_OWNED_RTT state rd a

  | ASSIGNED rtte =>
      RttLevelIsBlockOrPage state rd walk.level /\
      AddrIsRttLevelAligned rtte.addr walk.level /\
      (!a. a IN LevelAddresses walk.level ipa ==> AddrIsProtected a realm) /\
      let addresses = LevelAddresses walk.level rtte.addr in
        WALK_OWNED_DATA state rd addresses /\
        DATA_NO_ALIAS state rd realm ipa walk.level addresses

  | ASSIGNED_NS rtte =>
      RttLevelIsBlockOrPage state rd walk.level /\
      AddrIsRttLevelAligned rtte.addr walk.level /\
      (!a. a IN LevelAddresses walk.level ipa ==> ~AddrIsProtected a realm)
  | _ => walk.level <= RMM_RTT_PAGE_LEVEL
End
```

walk.level is strictly lower than RMM_RTT_PAGE_LEVEL

- walk.level = 1,2,3
- PA is aligned in accordance with the page table level.
- IPA for the walk is a Protected address.
- For every aligned PA at this level there is a granule in state DATA. Data is only mapped once in the Protected IPA space and it can only be reached by its owner.

- walk.level = 1,2,3
- PA is aligned in accordance with the page table level.
- IPA for the walk is a not Protected address.

Current and future work

- Device assignment: RMM is updated to allow a device to be assigned to a Realm (19 new commands)

- Planes: A realm can be divided into mutually isolated execution environments (9 new commands)

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు