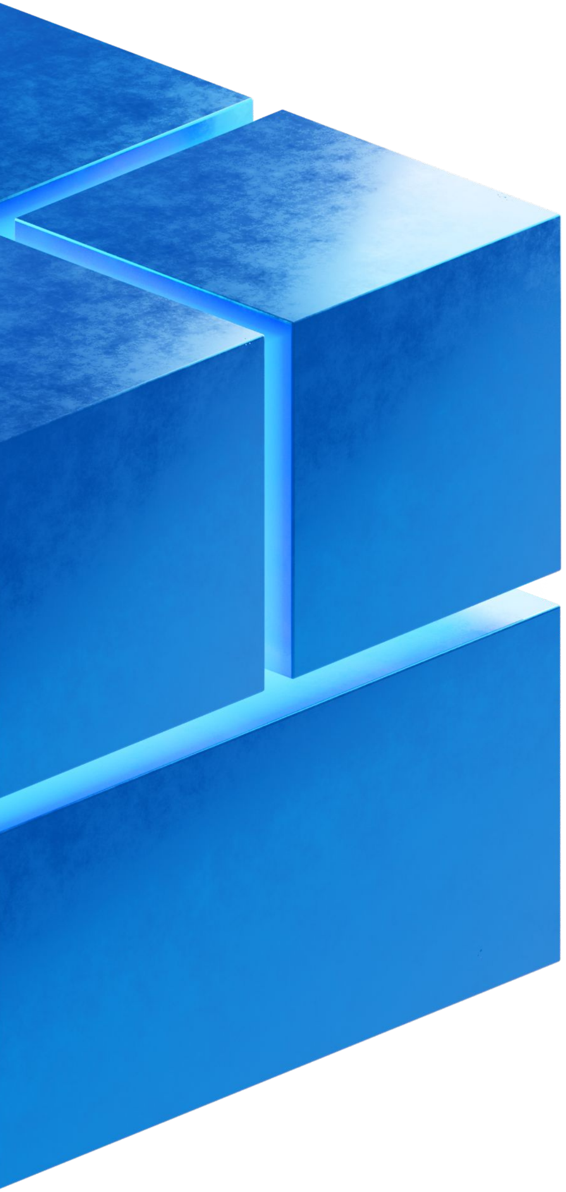


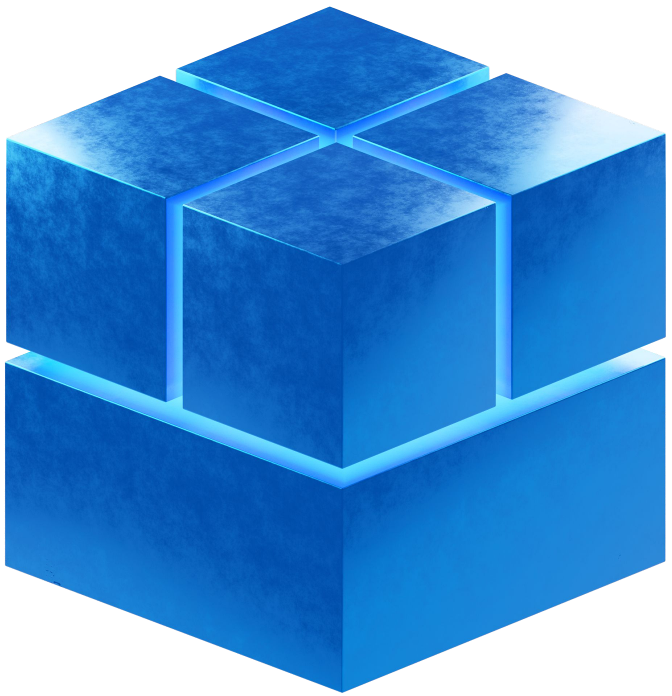
A Formal Specification of the NOVA Microhypervisor

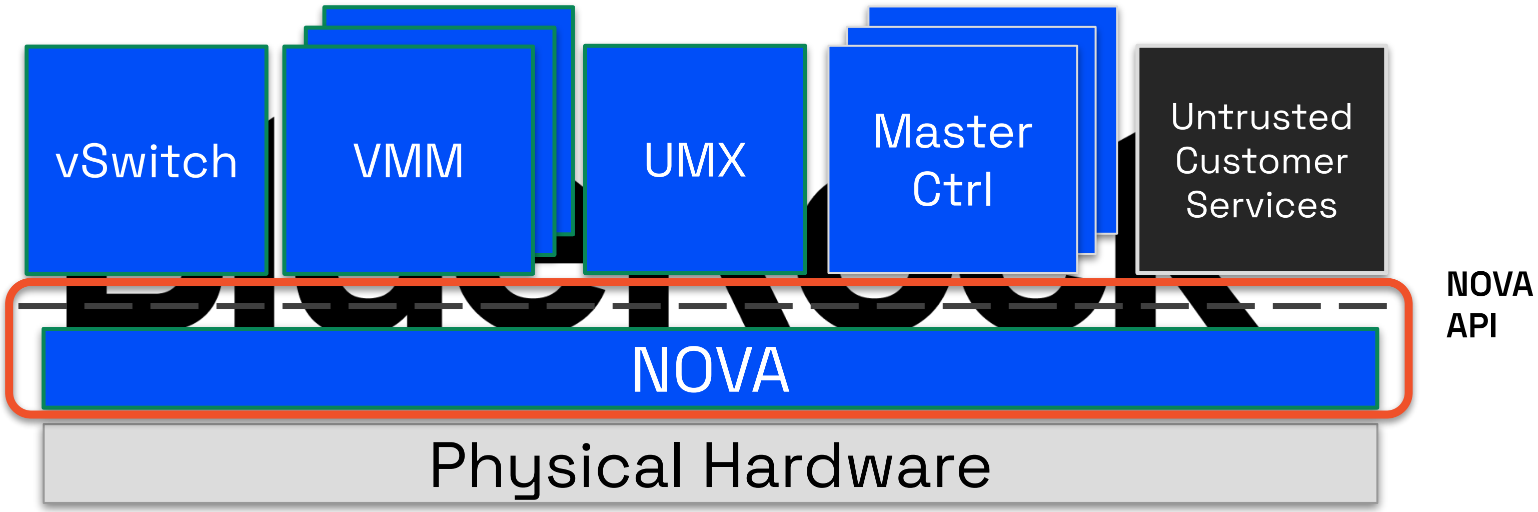
Hai Dang, David Swasey, Paolo G. Giarrusso, Gregory Malecha

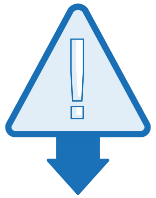
BlueRock Security, Inc

Microkernel and Component-Based OS DevRoom

FOSDEM, 1 February 2025







BIG GOAL

Verify the whole stack against a strong user-space specification. e.g. the “bare-metal property”.

This talk

Modular formal specification and verification of the NOVA microkernel within the BlueRock virtualization stack

Modular, Formal Specification and Verification

- **Formal:** unambiguous, thorough, with greater confidence
    - Properties are stated (specified) and proven (verified) ***rigorously in a logic***.
- **Modular:** independent, loosely coupled development
    - Specification and verification aligns with modularity of implementation.
    - ***Separation logic***: separation of resources with extensible abstractions.
- **Machine-checked:** more automated
    - Proofs are checked algorithmically (in the Rocq proof assistant, formly Coq).
    - ***Powerful proof automation for C++*** to reduce manual proofs.

Separation Logic (SL): separation of resources

Hoare logic describes how the code updates the state from pre- to post-condition, *sequentially*.

Each points-to assertion describes only the state *fragment* one cares about

{ sel ↦ (?, obj) }
cap_update(sel, perms)
{ sel ↦ (perms,obj) }

{ sel ↦ (perms,obj) }
cap_validate (sel, perm)
{ ret perm ∈ perms. sel ↦ (perms,obj)
}

Composing modular specs and proofs

The separating conjunction ★ joins separated resources

{ sel1↦ (?, sm) ★ sel2 ↦ ({CTRL},sc)}

{ sel1↦ (?, sm) }	{ sel2 ↦ ({CTRL},sc) }
cap_update(sel, UP);	r2 = cap_validate (sel2, CTRL);
{ sel1↦ ({UP}, sm) }	{ r2 = true ★ sel2 ↦ ({CTRL},sc) }

{ sel1↦ ({UP}, sm) ★ r2 = true ★ sel2 ↦ ({CTRL},sc) }

Capability Table	
sel1	({UP, DOWN}, sm)
…	
sel2	({CTRL}, sc)
…	

★

BlueRock

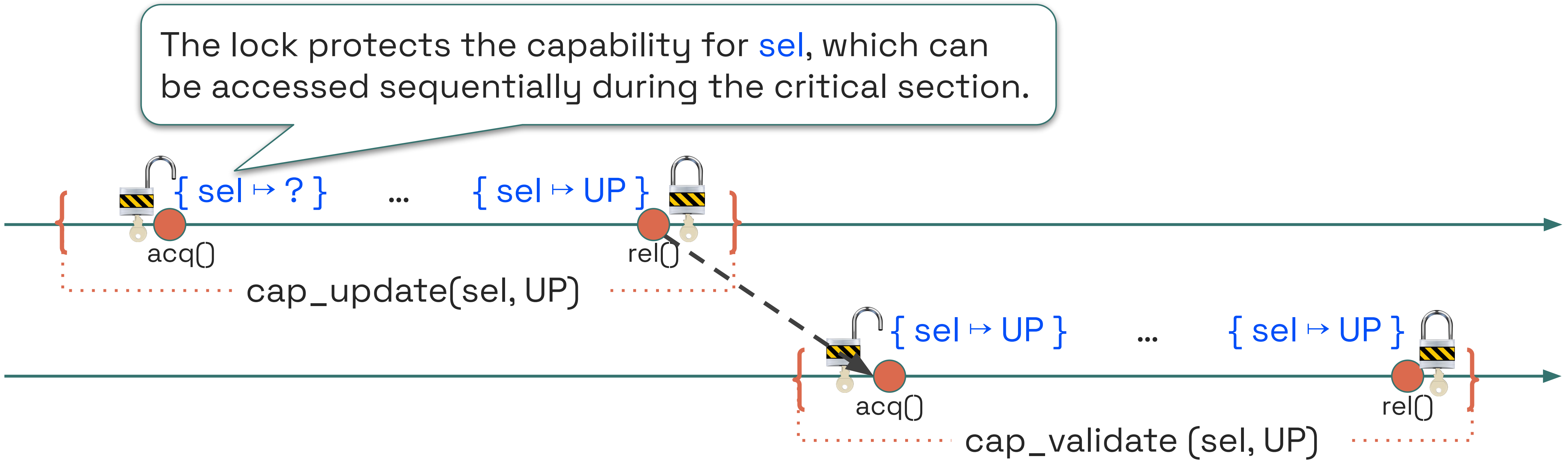
Separation as the basis for modularity

- *Small-footprint* assertions (e.g. sel ↦ (perms,obj)) capture the separation of resources in SL.
  - NOVA state is decomposed into logically disjoint kernel objects, each with its own state.
- ***Spatial separation*** is basic.
- ***Temporal separation*** (concurrently accessed resources) requires state-of-the-art logical constructs.

BlueRock

Advanced Concurrent Separation Logic (CSL)

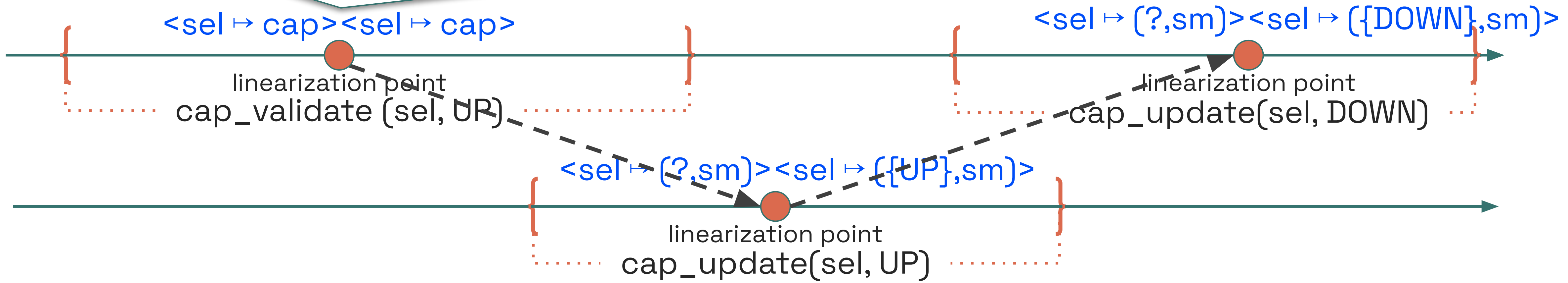
Lock-based concurrency: the resource is lock-protected



Advanced CSL: Logical Atomicity for Linearization

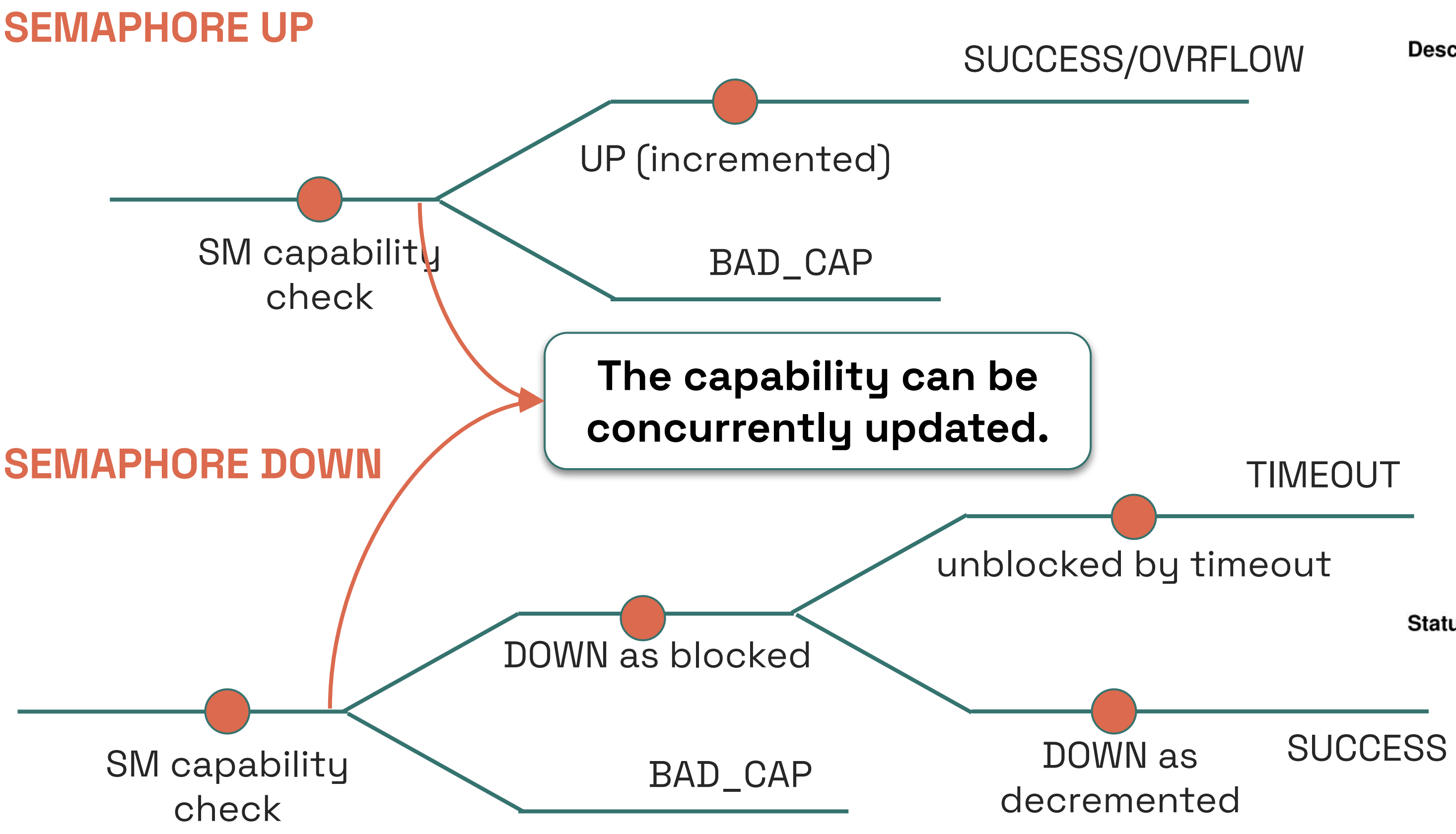
Fine-grained concurrency: the resource is accessed atomically

An atomic update captures how an operation logically atomically consumes the atomic pre-condition provided by the client, potentially updates it, and returns the atomic post-condition.



Chaining atomic updates <P1><Q1> ; <P2><Q2> ; <P3><Q3> to specify operations with more than one linearization point.

Specifying Fair Semaphores with Timeout



* order is specified formally, but not informally.

5.4.5 Control Semaphore

Parameters:

```
status = ctrl_sm (SEL_OBJ sm,          // Semaphore
                  UINT    stc)         // Absolute Timeout
```

Flags:

3	2	1	0
0	0	Z	D

Description:

Prior to the hypercall:

- If **D=0 (Semaphore Up)**:
  - $SPC_{OBJ_{CURRENT}}$[sm] must refer to an SM Capability ($CAP_{OBJ_{SM}}$) with permission $CTRL_{UP}$.
- If **D=1 (Semaphore Down)**:
  - $SPC_{OBJ_{CURRENT}}$[sm] must refer to an SM Capability ($CAP_{OBJ_{SM}}$) with permission $CTRL_{DN}$.

If the hypercall completed successfully:

- If **D=0 (Semaphore Up)**:
  - If there were ECs blocked on the semaphore, then the microhypervisor has released one of those blocked ECs. Otherwise, the microhypervisor has incremented the semaphore counter. The timeout value and the Z-flag were ignored.
- If **D=1 (Semaphore Down)**:
  - If the semaphore counter was larger than zero, then the microhypervisor has decremented the semaphore counter (**Z=0**) or set it to zero (**Z=1**). Otherwise, the microhypervisor has blocked $EC_{CURRENT}$ on the semaphore. If the timeout value was non-zero, $EC_{CURRENT}$ unblocks with a timeout status when the System Time Counter (STC) reaches or exceeds the specified value.

Blocking and releasing of ECs on a semaphore uses the FIFO queueing discipline.

Status:

SUCCESS

- The hypercall completed successfully.

TIMEOUT

- If **D=1**: Down operation aborted when the timeout triggered.

OVRFLOW

- If **D=0**: Up operation aborted because the semaphore counter would overflow.

BAD_CAP

- $SPC_{OBJ_{CURRENT}}$[sm] did not refer to an SM Capability ($CAP_{OBJ_{SM}}$) or that capability had insufficient permissions.

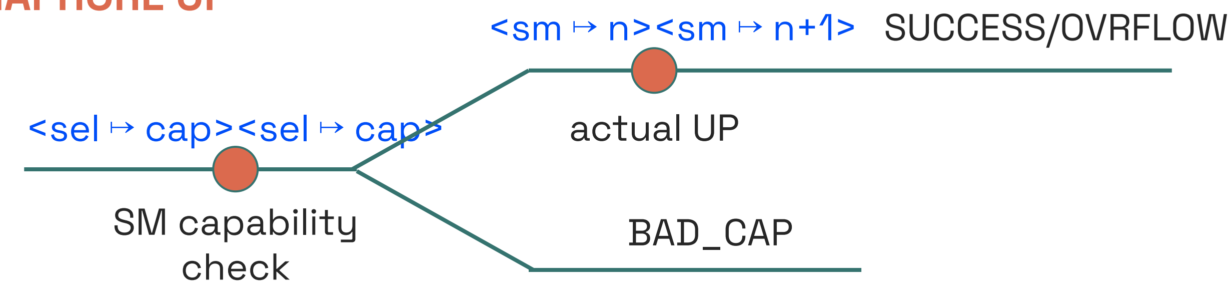
BAD_CPU

- If **D=1** on an interrupt semaphore: Attempt to wait for the interrupt on a different CPU than the CPU to which that interrupt has been routed via `assign_int`.

BlueRock

Specifying Semaphore Up

SEMAPHORE UP



Small-footprint assertions needed for Semaphores

- (capability) sel ↦ (perms, sm)
- (counter value) sm ↦ n
- (blocked ECs) sm ↦ ecs

UP SPEC

<sel ↦ cap><sel ↦ cap> ;
if cap is (perms,sm) ∧ UP ∈ perms then
    <sm ↦ n><sm ↦ (if n < MAX then n+1 else n)> ;
    Q(if n < MAX then SUCCESS else OVRFLOW)
else Q(BAD_CAP)

SM capability check

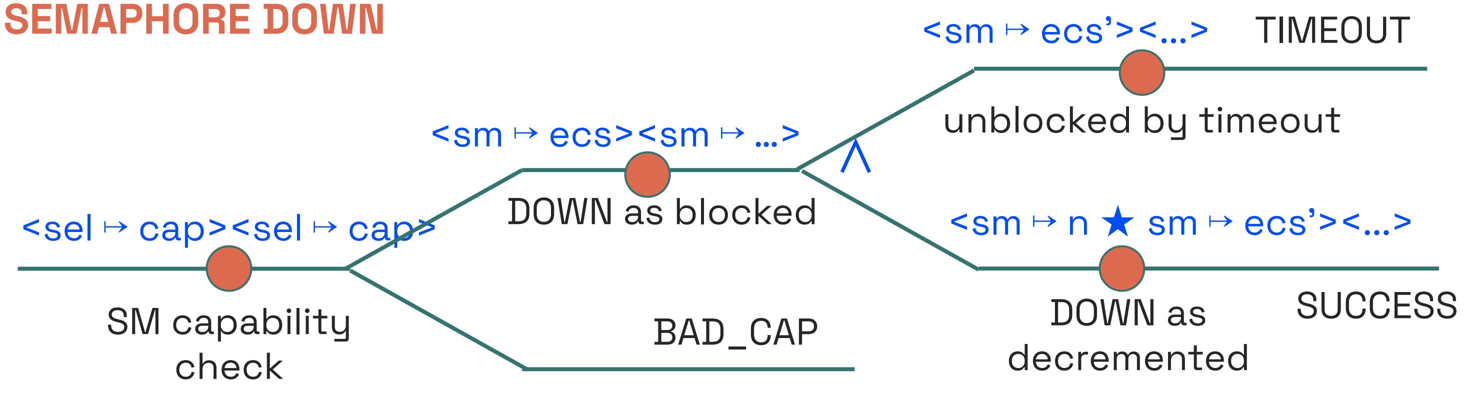
incremented (released)

Q: Client-picked obligation describing the continuation

BlueRock

Specifying Semaphore Down

SEMAPHORE DOWN



Small-footprint assertions needed for Semaphores

- (capability) sel ↦ (perms, sm)
- (counter value) sm ↦ n
- (blocked ECs) sm ↦ ecs

DOWN SPEC

<sel ↦ cap><sel ↦ cap> ;
if cap is (perms,sm) ∧ DOWN ∈ perms then
    <sm ↦ ecs><sm ↦ ecs ++ [ec]> ;
      (<sm ↦ ecs'><sm ↦ ecs' \ [ec]> ; Q (TIMEOUT))
    ∧ (<sm ↦ n ★ sm ↦ ecs'> <sm ↦ n-1 ★ sm ↦ ecs" ★ ecs' = [ec]++ecs" > ;
      Q(SUCCESS))
else Q(BAD_CAP)

ec is blocked

ec is unblocked due to timeout

Classical conjunction (∧): Clients have to handle both possibilities

decremented (acquired) and unblocked

BlueRock

Separation Logic as the Specification Language

- **Small footprint**: for every atomic update, the client of NOVA only needs to consider the minimal resources for each NOVA's functionality.
- **Fine-grained concurrency**:
  - resources need not be available all the time.
  - interleavings of atomic updates are visible.
- **Client flexibility**: client can choose to reduce concurrency (the number of interleavings), e.g. but adding locks if desired.
- **Robustness**: the specs cover all cases, NOVA either provides proper functionalities, or reports errors gracefully.

UP SPEC

```
<sel ↦ cap><sel ↦ cap> ;
if cap is (perms,sm) ∧ UP ∈ perms then
        <sm ↦ n><sm ↦ (if n < MAX then n+1 else n)> ;
        Q (if n < MAX then SUCCESS else OVRFLOW)
else Q(BAD_CAP)
```

DOWN SPEC

```
<sel ↦ cap><sel ↦ cap> ;
if cap is (perms,sm) ∧ DOWN ∈ perms then
   <sm ↦ ecs><sm ↦ ecs ++ [ec]> ;
       (<sm ↦ ecs'><sm ↦ ecs' \ [ec]> ; Q (TIMEOUT))
    ∧  (<sm ↦ n ★ sm ↦ ecs'> <sm ↦ n-1 ★ sm ↦ ecs'' ★ ecs' = [ec]++ecs'' >;
        Q(SUCCESS))
else Q(BAD_CAP)
```

NOVA verification example: Semaphore Down

C++ proof automation

Lock-protected internal resources

unscheduled

```
if (!csm.validate (Capability::Perm_sm::CTRL_DN))
    self->sys_finish_status (Status::BAD_CAP);
dn (Ec *const self, bool zero, uint64_t t)
{       { this ↦ sm.R g q }
    {   Lock guard <Spinlock> guard { lock };
            { this→cnt ↦ n ★ this ↦ blocked ecs ★ … }
        if (cnt) {
            cnt = zero ? 0 : cnt - 1;
            return;  { this→cnt ↦ (0 or n-1) ★ … }
        }
        self->block()
        enqueue_tail (self);  { this→cnt ↦ 0 ★ … }
    }
    if (self->block_sc()) {
        if (t)
            self->set_timeout (t, this);
        …
    }
}

bool up() { … }

void timeout (Ec *const ec) { … }
```

<sel ↦ cap><sel ↦ cap> ;
Q(BAD_CAP)

<sm ↦ ecs><sm ↦ ecs ++ [ec]> ; **(ecs = ecs'' = [] and ecs' = [ec] as n > 0)**
<sm ↦ n ★ sm ↦ ecs'><sm ↦ n-1 ★ sm ↦ ecs'' ★ ecs' = [ec]++ecs'' > ;
Q(SUCCESS)

<sm ↦ ecs><sm ↦ ecs ++ [ec]> ;

Formal verification requires us to write down the complex protocol explicitly.

Q(SUCCESS)

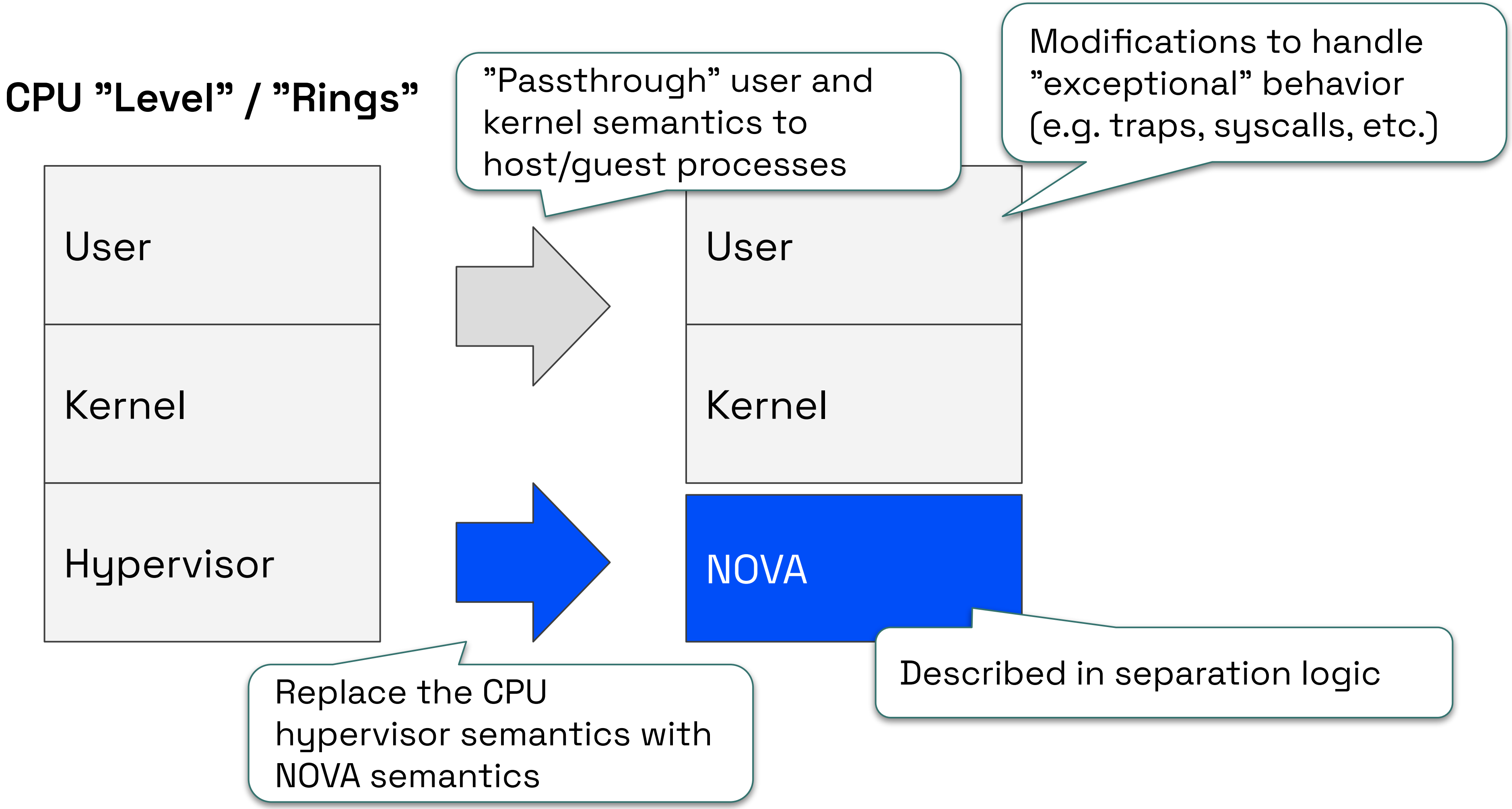
helping

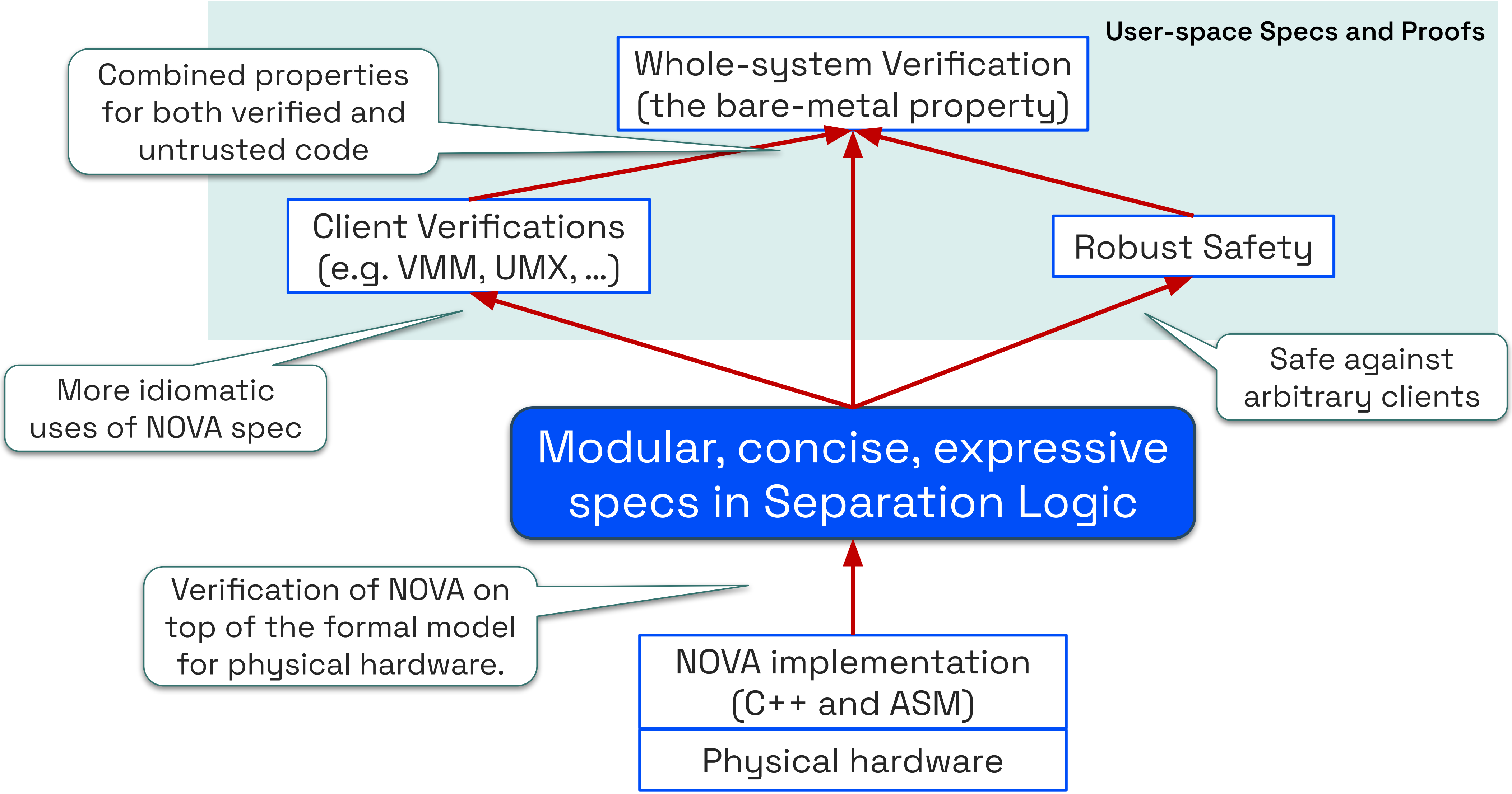
<sm ↦ n><sm ↦ n+1> **(from UP)**
<sm ↦ n+1 ★ sm ↦ ecs'><sm ↦ n ★ sm ↦ ecs'' ★ ecs' = [ec]++ecs'' > ;

<sm ↦ ecs'><sm ↦ ecs' \ [ec]> ; Q (TIMEOUT)

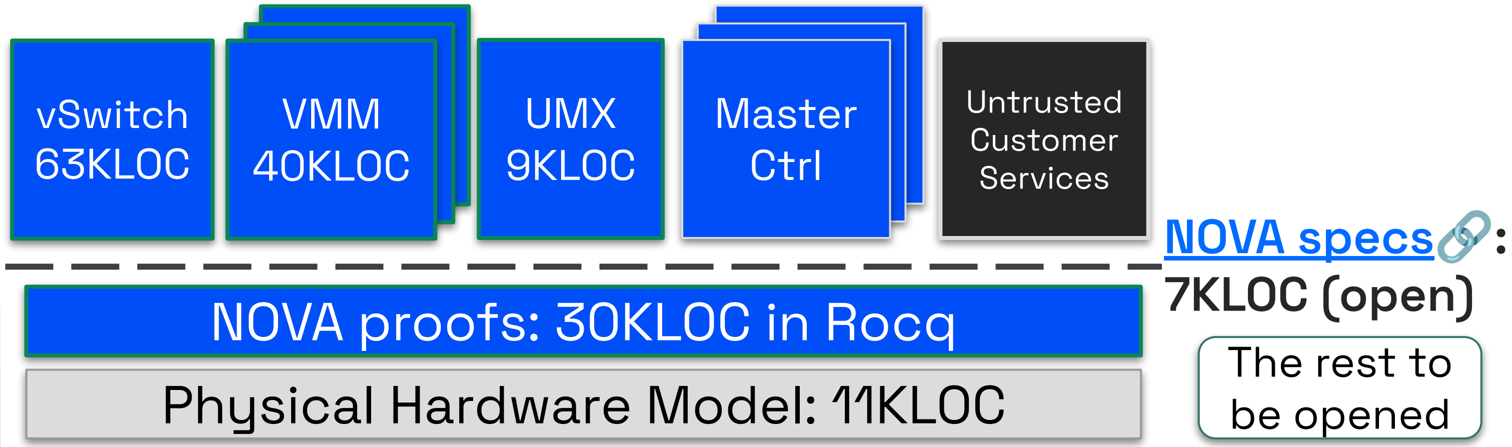
external linearization

NOVA as a Machine





A Formal Specification of the NOVA Microhypervisor



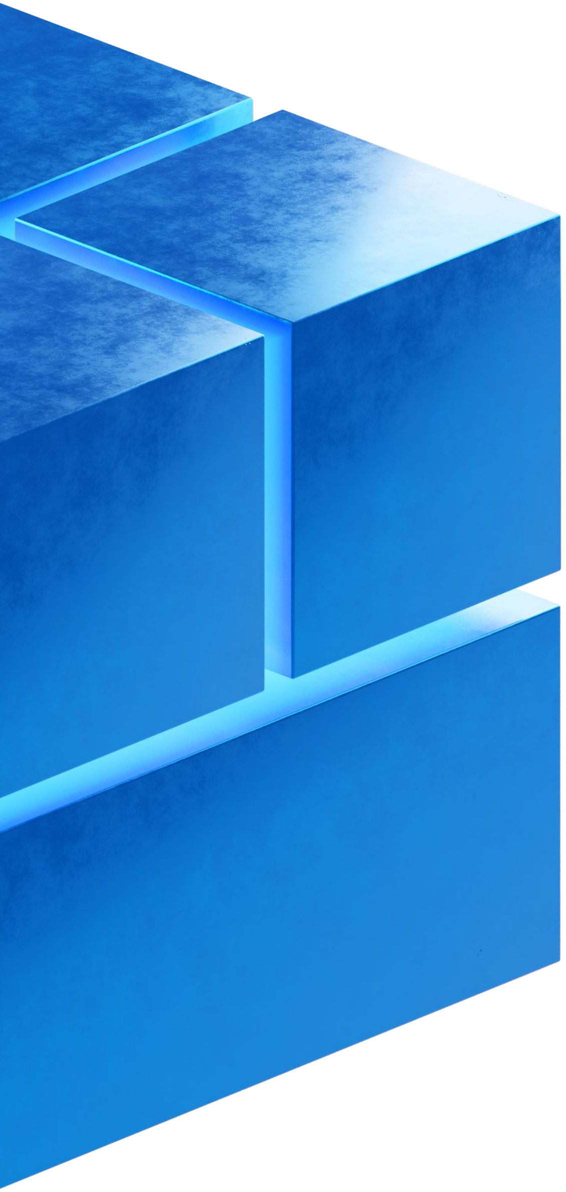
Take-home Messages

- **Formal Specification and Verification:** explicit, unambiguous mathematical modeling provides greater coverage and confidence.
- **Separation and Logical Atomicity** for modular and highly concurrent specification.
- **Expressiveness once-and-for-all:** strong specification supports both disciplined and undisciplined clients, and reduces proof efforts.

💡 More information

- Tech report🔗 for the NOVA formal specification.
- (Open) BriCK🔗 separation logic for C++ semantics.
- (To open) Proof automation for C++ and more languages.

BlueRock



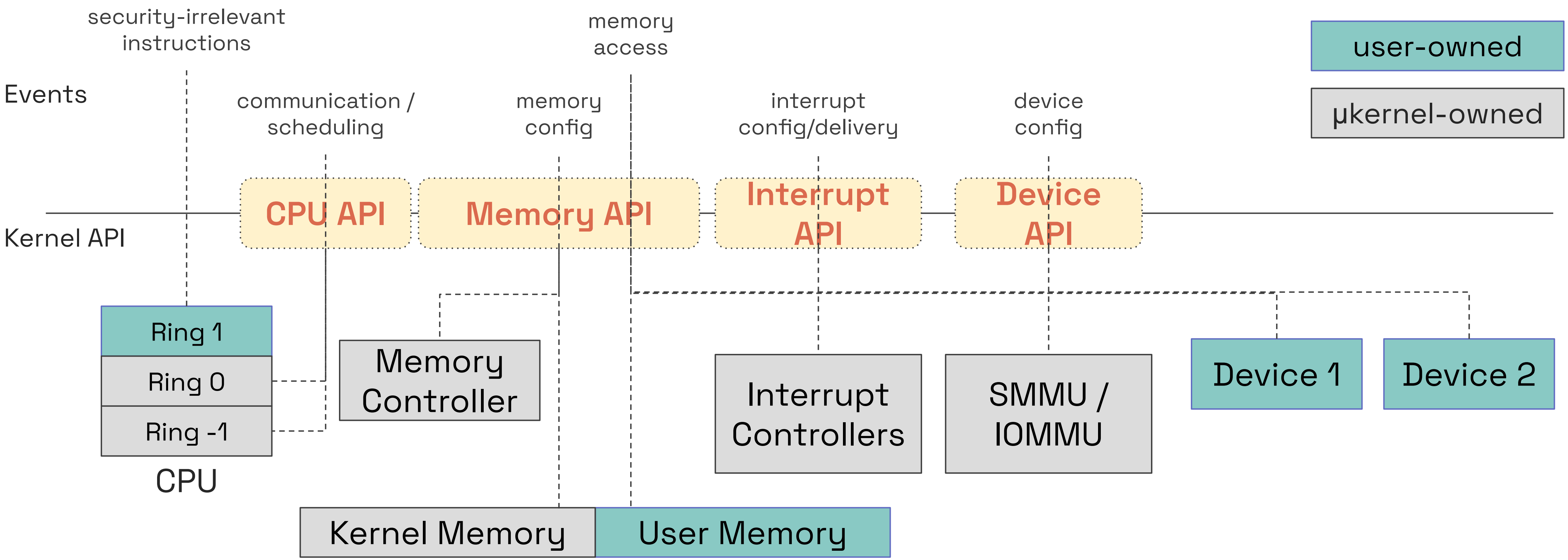
Appendix

Challenges

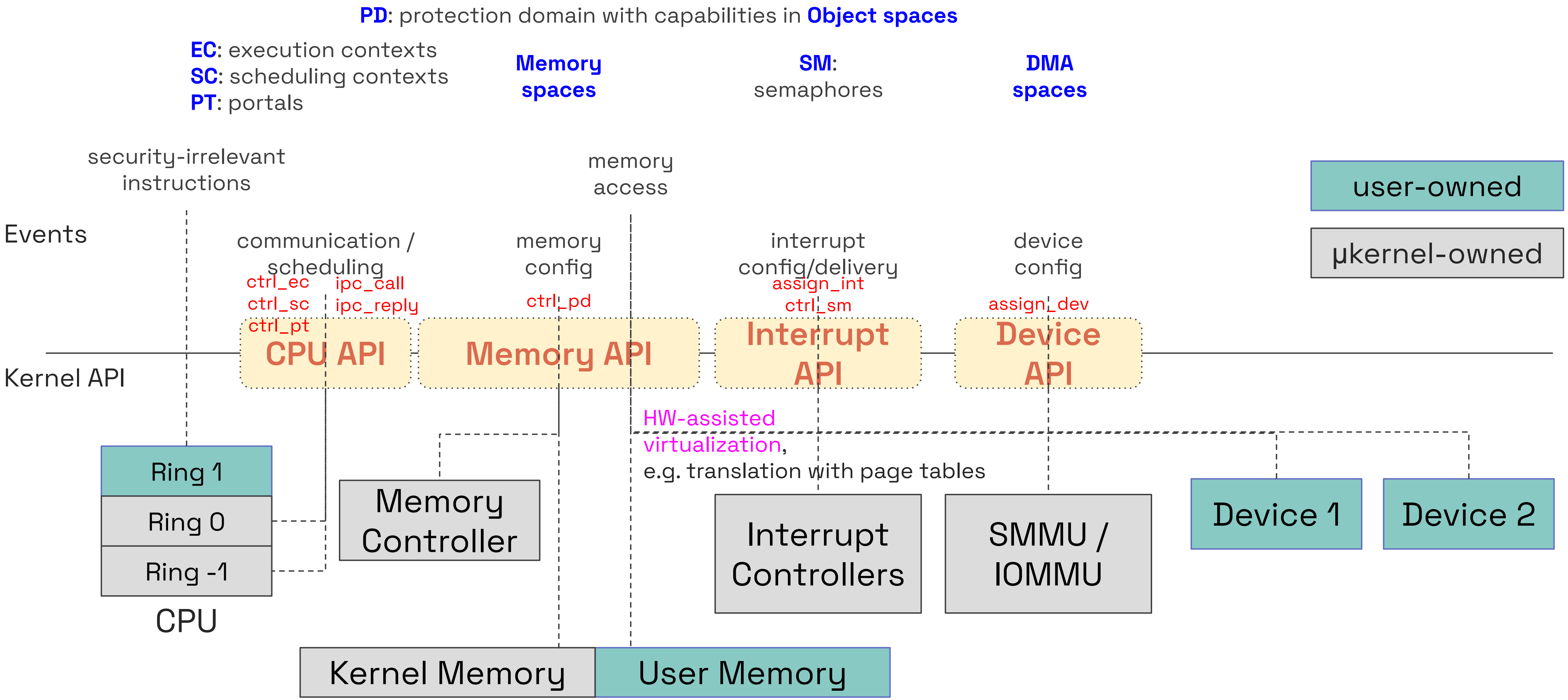
- Hardware modeling
  - semantics decomposition
- ASM verification
- Logic soundness and end-to-end adequacy

BlueRock

Microkernel owns minimum, security-relevant resources



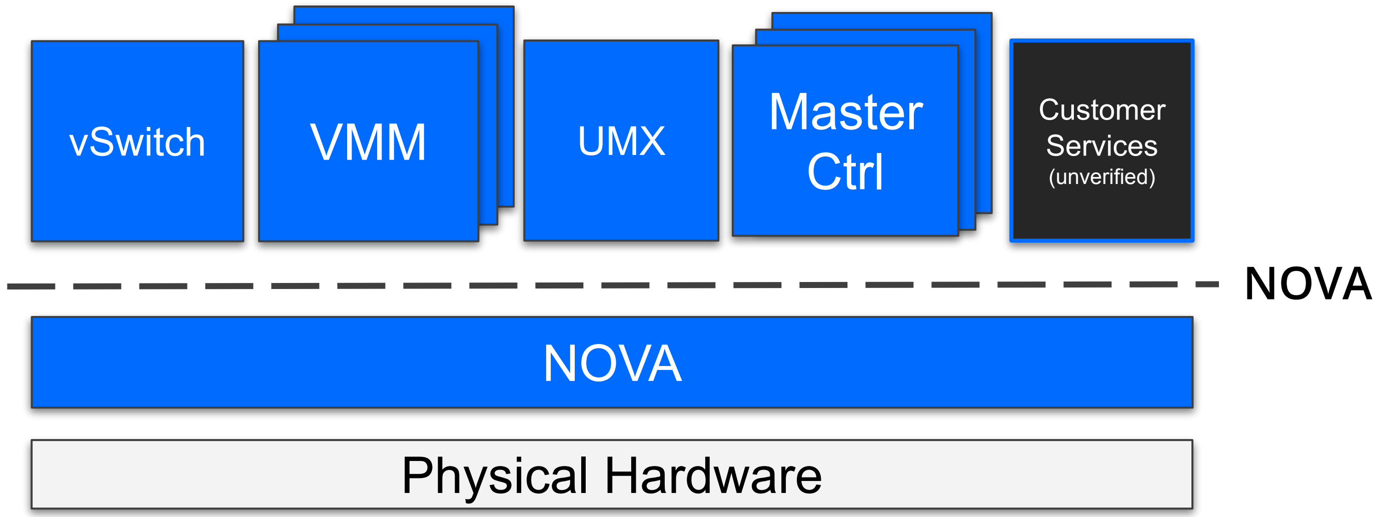
NOVA exposes kernel objects and hypercalls with HW-assisted virtualization



NOVA Specification Requirements

- Support reasoning about applications running on top of NOVA
- Support running untrusted (potentially malicious) applications
- **Support running both trusted and untrusted applications in parallel**

Verify NOVA against *a single specification*



BlueRock

Separation Logic as the Specification Language for NOVA API

Kernel Objects

Exposed as small-footprint SL resources

- Protection Domain and Spaces
  - Object capability **sel ↦ (perms, obj_id)**
  - Memory spaces (host, guest, DMA, …) **va ↦ (perms, pa)**
- Threads
  - Execution context
    - (registers) **ec.r1 ↦{reg} val**
    - (call stack) **ec ↦{callstack} ecs**
    - (UTCB) **ec ↦{utcb} pa**
    - (continuation) **ec ↦{cont} code**
  - Scheduling context **sc ↦ ticks**
- Communication
  - Portals **pt ↦ mtd**
  - Semaphore
    - (counter value) **sm ↦ n**
    - (blocked ECs) **sm ↦ ecs**

Hypercalls

Describe with weakest preconditions and logical atomicity for concurrency

- create_{pd,ec,sc,pt,sm}
- ctrl_{pd,ec,sc,pt,sm}
- ipc_call, ipc_reply
- assign_dev, assign_int, ctrl_pm

User-mode Semantics

"Pass through" CPU behavior

- Behavior of an execution context when it is *not* interacting with NOVA
- "Remaining" behaviors that are parametric to the NOVA separation logic

Specifying User-code Behavior:
Parametric architectural semantics + NOVA logical specs

```
wp_nova_ec ec regs ≈
  ∃ regs k, ec ↦{reg} regs ★ec ↦{cont} k ★ … ★
  (∀ evt regs', cpu.step regs evt regs' ⇒
    ec ↦{reg} regs' ★ … ★ ⇒
    match evt with
    | None => wp_nova_ec regs'
    | Some syscall => wp_hypercall ec syscall ..
    | Some (mem ..) => wp_mem ..
    | …
    end).
```

Weakest-precondition: proof obligation to show that the ec has good behaviors

State before the step

The step semantics that is mostly parametric to the logic

State after the step

No special interaction with NOVA, continue

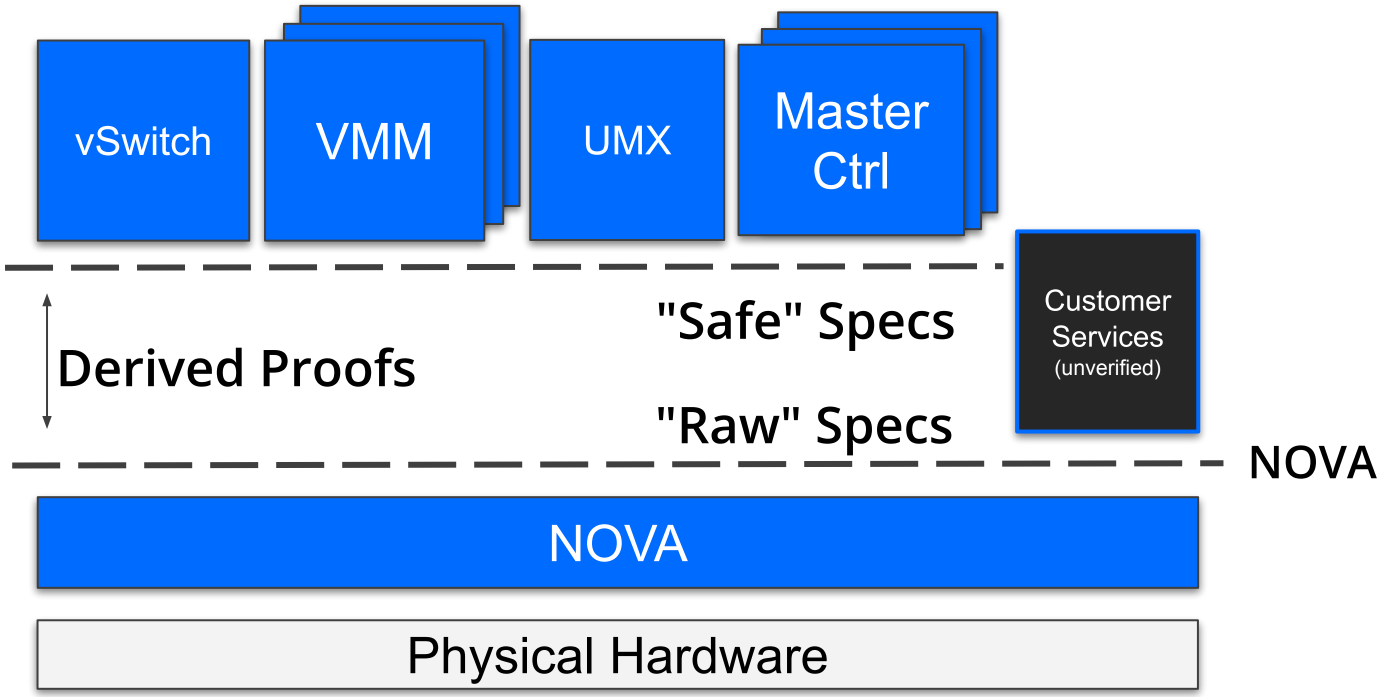
NOVA specs for interacting with NOVA hypercalls

NOVA specs for address translation and memory access

BlueRock

Supporting Verification on top of NOVA

- Prove weaker specifications that are easier to work with when clients are well-behaved
- **Clients choose** which specification they want



BlueRock

Userspace Robust Safety

Support a "data life cycle".

1. Create state. (high)
2. Configure state. (high)
3. Share permissions with untrusted code? (low)
4. Revoke state. (high)
5. Destroy state. (high)

Entire data lifecycle is provable using the strong specification.

Can share limited permissions, e.g. only up a semaphore, only call a portal, only read a page, etc.

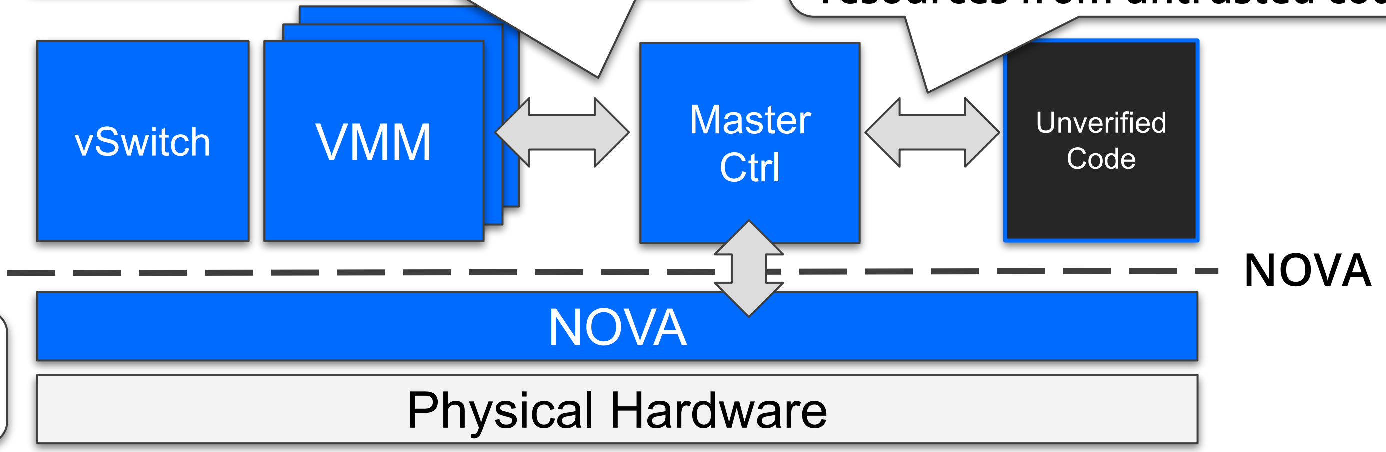
State may change integrity levels throughout the execution of the program.

- High integrity state shared with untrusted code.
- Low integrity state revoked from untrusted code.

Revocation is generally difficult and requires tight reasoning about confinement and visibility.

Share resources with verified code. Requires strong specification.

Create resources, pass to untrusted code, revoke resources from untrusted code.



Precise specifications support endorsement (low -> high) because they decouple state from policy.

Provide mechanisms (assertions), not policies (invariants, quantifiers).

Proving Robust Safety from the Spec

Express the high-low state distinction within separation logic.

- Invariants allow flexible, concurrent sharing.
- Existential quantifiers express low-integrity

Over arbitrary user binaries.

Need to Show

```
Userspace Resources |-- |={T}=> inv rs-inv
inv rs-inv |-- wp_nova_ec boot_ec boot_regs
```

rs-inv

∃ objs, let valid o := o \in objs in
[★list] o \in objs,

∃ v ecs, sm.value γ v
★ sm.queue γ ecs ★ [| Forall valid ecs |]

"Low integrity" invariant for semaphores. Ownership exists, values are minimally constrained.

⋮

Properties for other kernel object types.

★ all schedulable ECs are valid
★ all interrupts bound to valid interrupt SMs

BlueRock

Proving Robust Safety from the Spec

Express the high-low state distinction within separation logic.

- Use an invariant to allow flexible sharing.
- Use existential quantification to express low-integrity

Prove this invariant

- Is constructible from the NOVA boot resources.
- Entails wp_nova_ec (and wp_nova_dev).

Robust safety for NOVA!
This invariant is too weak to support userspace verification.

rs-inv

∃ objs, let valid o := o \in objs in
[★list] o \in objs,

PD
- All legal object/memory selectors map to valid objects/userspace pages (or null)
- Root PD or belong to a valid PD

SM
- Have a value and a wait queue
- All ECs in the wait queue are valid
- Belong to a valid PD

EC
- Have valid user stage, e.g. register file
- Belong to a valid PD
- Bound to a valid CPU

SC
- Bound to a valid global EC or VCPU
- Belong to a valid PD

PT
- Bound to a valid local EC on the same core
- Belong to a valid PD

★ all schedulable ECs are valid
★ all interrupts are bound to valid interrupt SMs

Existential quantifiers capture "low-integrity" state.

∃ v ecs, sm.value γ v
** sm.queue γ ecs
** [| Forall valid ecs |]

Entering an Abstraction
A Generic Pattern

NOVA_boot -* wp_nova_ec ec regs

C++ source code proof

cpp_init nova -* wp_cpp nova

ASM source code proof

cpp_init nova -* wp_cpp nova

C++ compiler correctness

boot -* wp_arm_el2 boot_regs

Starting NOVA – the spec of "`main()`"

Simplified

Parametric over *any* startup program, well-behaved or not.

NOVA machine resources. Given to userspace by NOVA.

```
∀ startup_image,
(∀ root_pd root_ec root_sc,
  pd.mem root_pd startup_image -*
  initial_pd state root_pd root_ec root_sc -*
  memory -* … -*
  wp_nova_ec root_ec (startup_regs …))
⊢ NOVA_loaded -*
 elf startup_image -*
 memory -*
 … -*
 wp_arm_el2 boot_regs
```

Weakest precondition for the root EC.

Raw machine resources. Given to NOVA by the bootloader.

Behavior of the boot CPU expressed as a weakest-precondition.

The "program" is the register state.

System Refinement

Establish properties for the applications that run on top of NOVA.

- Use CaReSL-style techniques to prove refinement using ghost state and invariants.
- Extract the end-to-end proof (independent of SL) using Iris adequacy.

Effectful transitions must be tied to device I/O.

CaReSL-style operational specification of the userspace application.

spec
init ⊢ physical_state -* wp_arm boot

Use adequacy to extract a standard operational refinement proof.

BlueRock

System Refinement

Establish properties for the applications that run on top of NOVA.

- Use CaReSL-style techniques to prove refinement using ghost state and invariants.
- Extract the end-to-end proof (independent of SL) using Iris adequacy.

NOVA

spec init ⊢ nova_state -* elf user.bin -* wp_nova_ec ...

nova_ok

Use the proof of NOVA.

Framing preserves the specification.

spec init ⊢ physical_state -* wp_arm boot

Use adequacy to extract a standard operational refinement proof.

BlueRock

System Refinement

Establish properties for the applications that run on top of NOVA.

- Use CaReSL-style techniques to prove refinement using ghost state and invariants.
- Extract the end-to-end proof (independent of SL) using Iris adequacy.

Verify the userspace binary against `spec init`.

app_ok

spec init ⊢ c++ prog.init -* wp_cpp prog.main

cpp_ok

Verify source code assuming compiler correctness.

NOVA

spec init ⊢ nova_state -* elf user.bin -* wp_nova_ec ...

nova_ok

spec init ⊢ physical_state -* wp_arm boot

Use adequacy to extract a standard operational refinement proof.