seL4 Summit

Splitting the seL4 Specification

Thomas Sewell

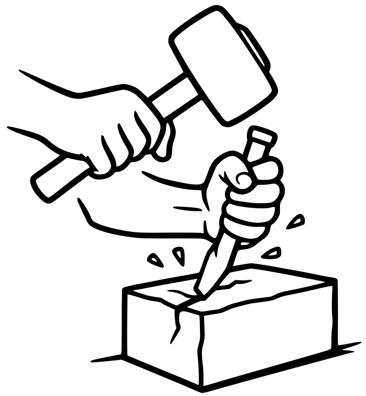
4 September 2025

Splitting transitions in the seL4 Specification

Existing work:

- The seL4 μ-kernel.
- **Verified** functionally correct!

This talk: spec splitting.



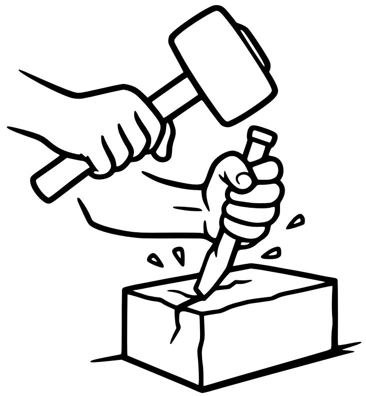
Splitting transitions in the seL4 Specification

Existing work:

- The seL4 μ-kernel.
- **Verified** functionally correct!

This talk: spec splitting.

- What does that mean?
- Why might we do that?
- How?
- When will it be finished?



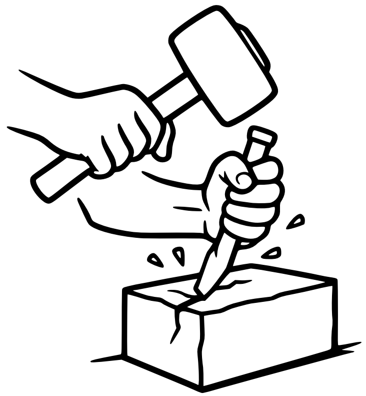
Splitting transitions in the seL4 Specification

Existing work:

- The seL4 μ-kernel.
- **Verified** functionally correct!

This talk: spec splitting.

- What does that mean?
- Why might we do that?
- How?
- When will it be finished?
  - Yeah, dunno.



This talk is also splitting a time slot.

See:

"Verifying Kernel–Userland Integration for LionsOS"

- Dr Rob Sison, UNSW.

This talk is also splitting a time slot.

See:

“Verifying Kernel–Userland Integration for LionsOS”

- Dr Rob Sison, UNSW.

Both talks are about using the seL4 abstract spec to justify a user verification environment.

An Example

In thread *X*:

```
int do_send(void) {
    seL4_MessageInfo_t i;
    ...
    i = mk_send_info();
    i = seL4_Send(cptr, i);
    ...
}
```

In thread *Y*:

```
int do_recv(void) {
    seL4_MessageInfo_t i;
    ...
    ...
    i = seL4_Recv(cptr, NULL);
    ...
}
```

What does this do?

- Under the right conditions, *X* sends a message to *Y*.

An Example

In thread *X*:

```
int do_send(void) {
    seL4_MessageInfo_t i;
    ...
    i = mk_send_info();
    i = seL4_Send(cptr, i);
    ...
}
```

In thread *Y*:

```
int do_recv(void) {
    seL4_MessageInfo_t i;
    ...
    ...
    i = seL4_Recv(cptr, NULL);
    ...
}
```

What does this do?

- Under the right conditions, *X* sends a message to *Y*.

Can we verify that?

An Example (II)

In thread Y:

```
int do_recv(void) {
    seL4_MessageInfo_t i;
    ...
    ...
    i = seL4_Recv(cptr, NULL);
    ...
}
```

What does `seL4_Recv` here *really* do?

An Example (II)

In thread Y:

```
int do_recv(void) {
    seL4_MessageInfo_t i;
    ...
    ...
    i = seL4_Recv(cptr, NULL);
    ...
}
```

What does `seL4_Recv` here *really* do?

- System call, not a function call.
- Triggers a kernel entry.

An Example (II)

In thread *Y*:

```
int do_recv(void) {
    seL4_MessageInfo_t i;
    ...
    ...
    i = seL4_Recv(cptr, NULL);
    ...
}
```

What does `seL4_Recv` here *really* do?

- System call, not a function call.
- Triggers a kernel entry.
  - There is a *specification* for that.

The seL4 Kernel: Event Handlers

seL4 is an *event reactive* kernel.
It consists of a small number of *event handlers*.

The seL4 design takes this to extremes.
Each event handler is an entry point from assembly to C.
Each function terminates, and returns out of C entirely.

Thus we can model each one as a function, *state* $\rightarrow$ *state*.

Functional Correctness of the seL4 Kernel

Each kernel entry is a function *state* $\rightarrow$ *state*.

The functional correctness proof guarantees that these functions implement the `call_kernel` function.

```
definition
    call_kernel :: ''event ⟹ (unit, ...) s_monad'' where
    call_kernel ev ≡ do
        handle_event ev <handle>
            (λ _.  ...)
        schedule;
        activate_thread
    od
```

`seL4_Recv` causes a kernel entry.

```
definition
    call_kernel :: ''event ⟹ (unit, ...) s_monad'' where
    call_kernel ev ≡ do
        handle_event ev <handle>
            (λ _.  ...)
        schedule;
        activate_thread
    od

handle_event (SyscallEvent SysRecv) =
    without_preemption (handle_recv True)
```

∴ `seL4_Recv` ≈ call to `handle_recv`.

`seL4_Recv` causes a kernel entry.

```
definition
    call_kernel ::  ''event ⟹ (unit, ...)  s_monad'' where
    call_kernel ev ≡ do
        handle_event ev <handle>
            (λ _.  ...)
        schedule;
        activate_thread
    od

handle_event (SyscallEvent SysRecv) =
    without_preemption (handle_recv True)
```

∴ `seL4_Recv` ≈ call to `handle_recv`.

- But when do they *finish*?

`seL4_Recv` causes a kernel entry.

```
definition
    call_kernel ::  ''event ⟹ (unit, ...)  s_monad'' where
    call_kernel ev ≡ do
        handle_event ev <handle>
            (λ _.  ...)
        schedule;
        activate_thread
    od

handle_event (SyscallEvent SysRecv) =
    without_preemption (handle_recv True)
```

∴ `seL4_Recv` ≈ call to `handle_recv`.

- But when do they *finish*?
- The `schedule` function might not pick *Y*.

`handle_recv` $\approx$

1. Walk C-space and look up capability.
2. Check that the capability can receive.
   - Is an Endpoint or Notify capability.
   - Has Receive/Read rights.
3. Is there a blocked sender?
   - **No**: block this thread as a receiver.
   - **Yes**:
     `do_ipc_transfer sender endpoint badge grant receiver`

n.b.: This is very simplified and omits a lot of cases.

Recap: Example

In thread *X*:

```
int do_send(void) {
    seL4_MessageInfo_t i;
    ...
    i = mk_send_info();
    i = seL4_Send(cptr, i);
    ...
}
```

In thread *Y*:

```
int do_recv(void) {
    seL4_MessageInfo_t i;
    ...
    ...
    i = seL4_Recv(cptr, NULL);
    ...
}
```

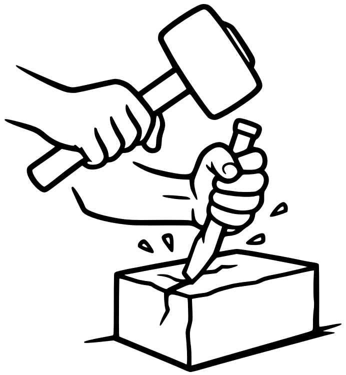
What does this do?

- Both threads enter the kernel, do syscall capability checks.
- Either *X* or *Y* also causes an IPC transfer and completes both syscalls.

Splitting by Example

Can we split the IPC transfer?

`do_ipc_transfer sender endpoint badge grant receiver`

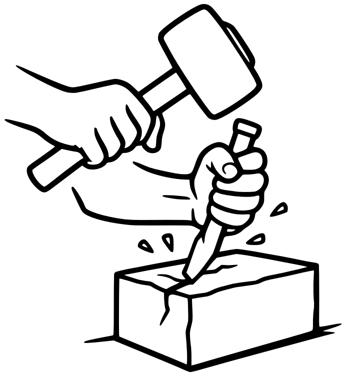


1. *X*'s IPC send action.
    - Part of `seL4_Send`.
2. *Y*'s IPC receive action.
    - Part of `seL4_Recv`.

Splitting by Example

Can we split the IPC transfer?

`do_ipc_transfer sender endpoint badge grant receiver`



1. *X*'s IPC send action.
   - Part of `seL4_Send`.
2. *Y*'s IPC receive action.
   - Part of `seL4_Recv`.

Let's hear about the Kernel/User project.

Splitting the Spec: Why?

Let's come back to:

Why would we want to split the spec?

Client Timelines

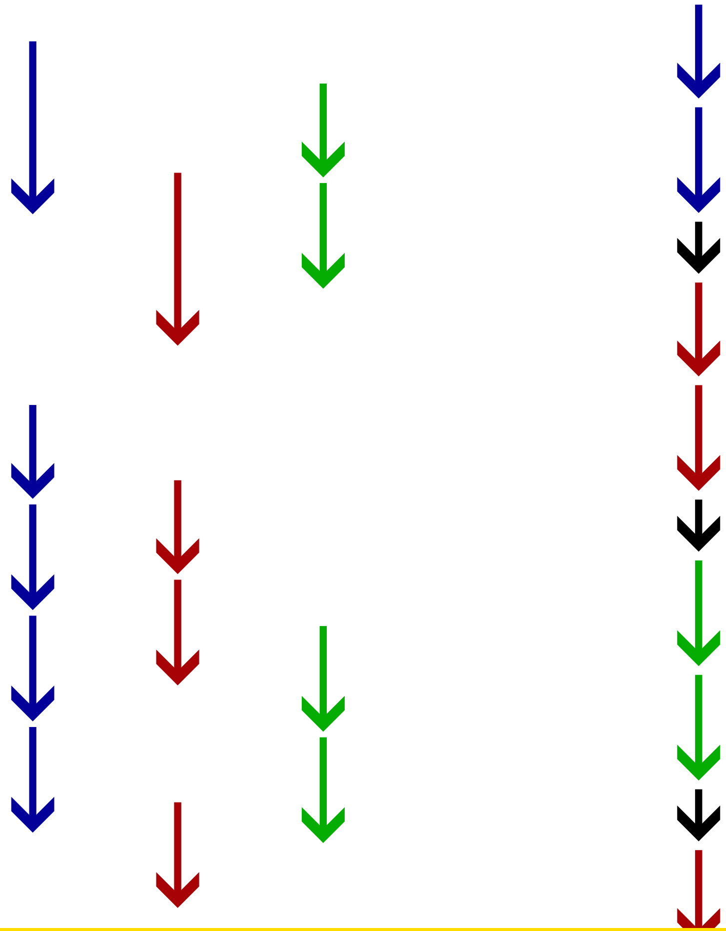
(Recap of OS lecture 2.)

Logical perspective:

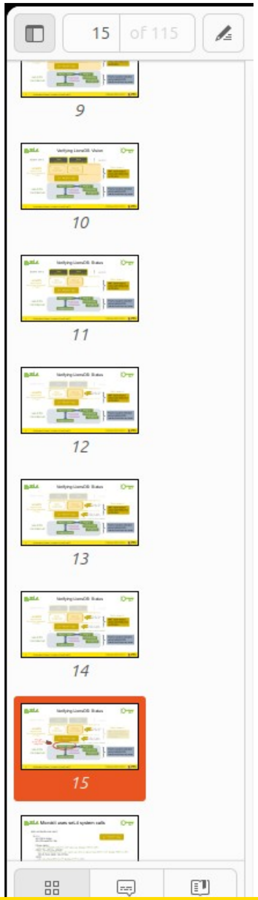
- 3 user threads, X, Y, Z.
- Each has registers.
- Each executes instructions.
- Memory is shared.

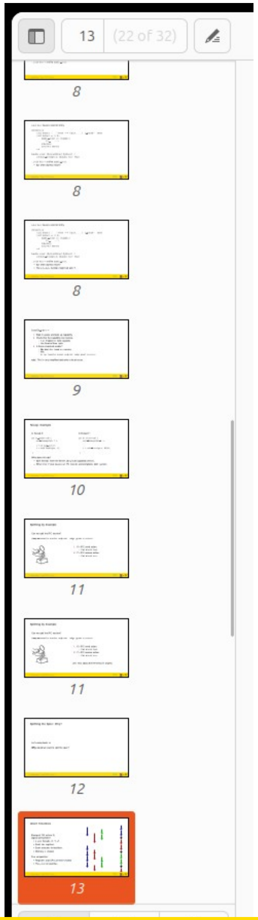
True perspective:

- Registers and CPU are time-shared.
- The μ-kernel switches.

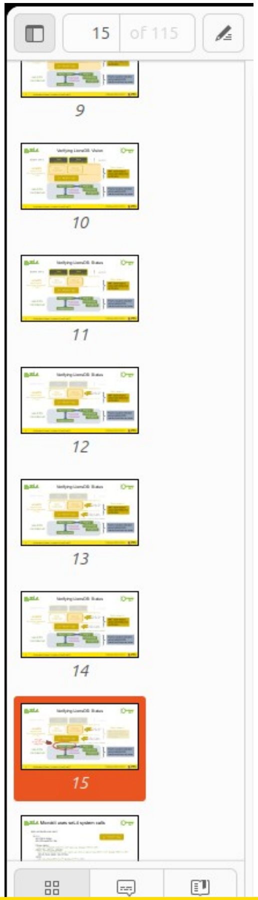


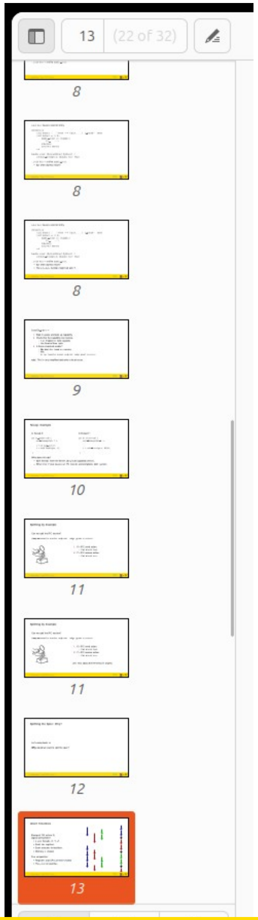
A Metaphor





A Metaphor





Multiple lists of instructions, but just one talk.

Questions about the validity of the talk need to be addressed to one of these timelines.

Verifying Client Timelines

We want to verify a client program.

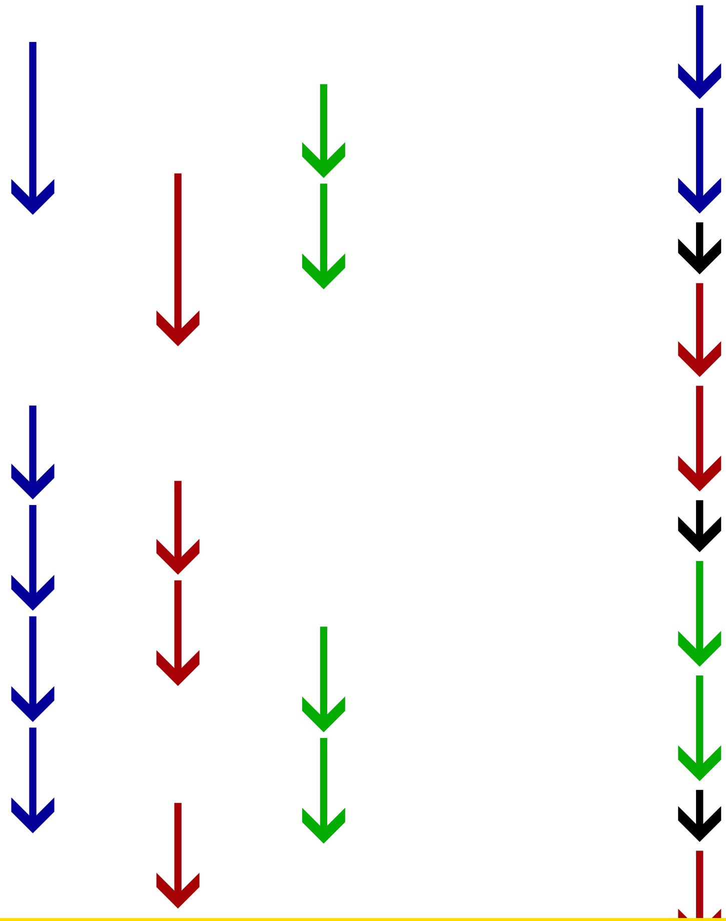
Clients run *within* these logical timelines.

It would help to have a model that exposes these timelines.

Executions of seL4 fit awkwardly with the logical timelines.

Obvious options:

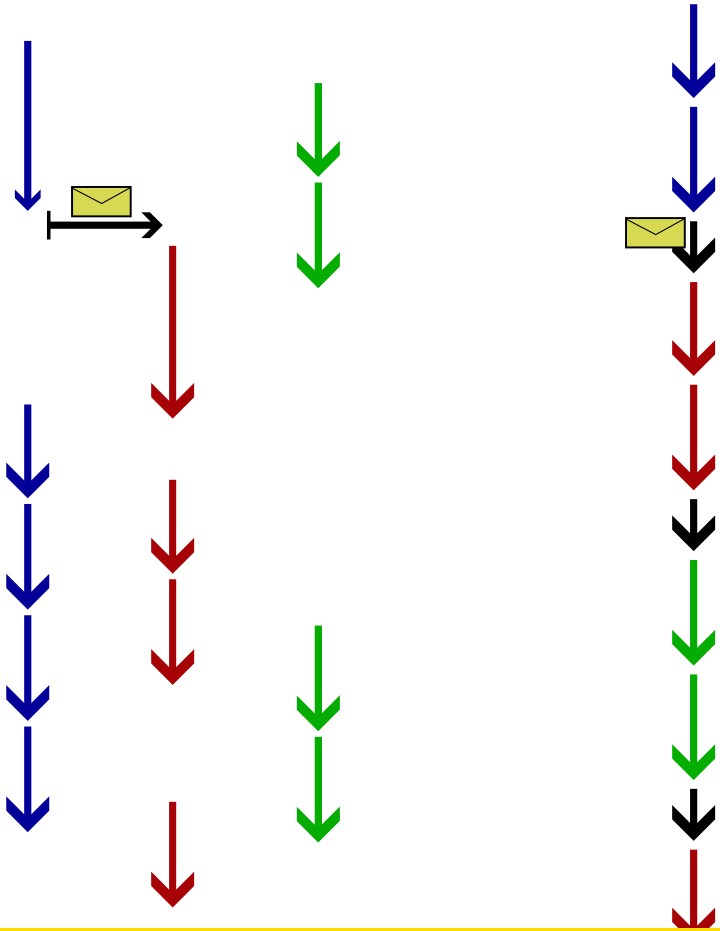
- Put μ-kernel on its own timeline.
- Put system calls in the timeline of the caller.

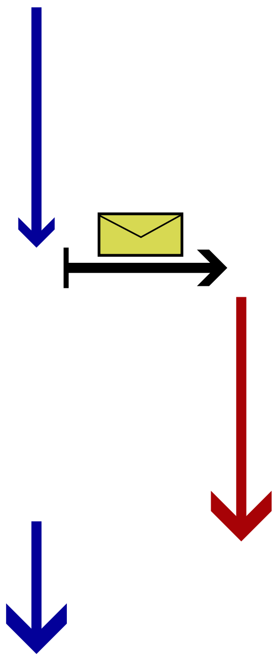


Inter-Timeline Actions

Timeline-crossing events cause additional complications.

- Synchronous seL4 message sends.
- Asynchronous seL4 notifications.





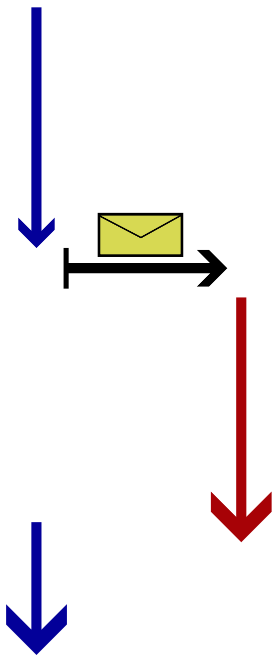
From our example: X makes a synchronous send to Y.

The seL4 “integrity” proof compares X's authority to what it can do:

- Assume that Y was blocked, and X causes the transfer.
- X gets a very conditional right to modify Y.
  - X can save a sent message to Y.

The blocking system call by Y ends *in* an event from X's timeline.

Verifying a simple IPC example is difficult; we are not aware of such a verification.



From our example: X makes a synchronous send to Y.

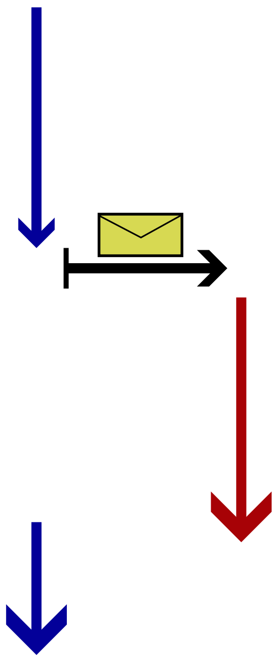
The seL4 “integrity” proof compares X’s authority to what it can do:

- Assume that Y was blocked, and X causes the transfer.
- X gets a very conditional right to modify Y.
    - X can save a sent message to Y.
    - First recognized version of this problem.

The blocking system call by Y ends *in* an event from X’s timeline.

Verifying a simple IPC example is difficult; we are not aware of such a verification.

|



From our example: X makes a synchronous send to Y.

The seL4 “integrity” proof compares X’s authority to what it can do:

- Assume that Y was blocked, and X causes the transfer.
- X gets a very conditional right to modify Y.
  - X can save a sent message to Y.
  - First recognized version of this problem.

The blocking system call by Y ends *in* an event from X’s timeline.

Verifying a simple IPC example is difficult; we are not aware of such a verification.

Proposed diagnosis: these are all *symptoms* of the mismatch between the kernel timeline and the user timelines.

Splitting: How?

```
definition
    call_kernel :: ''event ⟹ (unit, ...)  s_monad'' where
    call_kernel ev ≡ do
        handle_event ev <handle>
            (λ _.  ...)
        schedule;
        activate_thread
    od

definition
    call_kernel_step :: ''event ⟹ (next_step, ...)  s_monad'' where
    call_kernel_step ev ≡ do
        handle_event ev <handle>
            (λ _.  ...)
        return (Next_Step Schedule_Next);
    od
```

Splitting

Desired properties:

- Chaining the transitions yields original spec.
  - Preserves refinement to existing implementation.
- Intermediate states satisfy kernel invariants.
- “Integrity” security property is simpler and stronger.
  - X may only modify structures it controls.
- Selecting events on timeline X makes sense.
  - Yields a model against which program X can be verified.

Splitting the Example

`handle_recv_step` ≈

1. Check *Y*'s syscall arguments and look up endpoint.
2. `Sync_Sender_Unblock`: remove *X* from endpoint.
3. `Sync_Sender_Gen_Message`: fetch message from *X*.
4. `Sync_Receiver_Accept`: save message to *Y*.
5. `Sync_Sender_Cleanup`: possibly clean up *X* Reply caps.
6. `Sync_Receiver_Accept`: possibly update *Y* scheduler state.

Thanks to `ReplyRecv`, there can be more than 10 split steps per kernel entry.

This order is partly chosen to agree with the existing abstract spec.
It is already somewhat re-ordered though.

Re-ordering transfer/thread-set

We do prove that two operations can be re-ordered:

- `do_ipc_transfer sender endpoint badge grant receiver`
- `set_thread_state thread Running`

This is because they access independent parts of the abstract spec state.

Setting a thread state uses:

- The `tcb_state` field of TCB heap objects.
- The current-thread pointer.
- The `scheduler_action` element of the “extended” state.

. . .

Re-ordering transfer/thread-set

Setting a thread state uses:

- The `tcb_state` field of TCB heap objects.
- The current-thread pointer.
- The `scheduler_action` element of the "extended" state.

IPC transfers use:

- Registers.
  - The "machine" state.
- Frame contents.
  - Architecture-specific heap objects.
- Capabilities.
  - CNode (capability container) heap objects.
  - The CDT and is-original global databases.
  - The `cdt_list` field of the "extended" state.
  - Capability slots of TCB heap objects.

Status

Desired properties:

- Chaining the transitions yields original spec.
  - Proven and then broken again.
- Intermediate states satisfy kernel invariants.
  - Mostly done.
- “Integrity” security property is simpler and stronger.
  - To do.
- Divided model makes sense.
  - Requires a re-factor.
- Selecting events on timeline X makes sense.
  - To do.

Cross to: Kernel/User

Let’s cross to Rob’s talk and hear the status of the Kernel/User gap project.

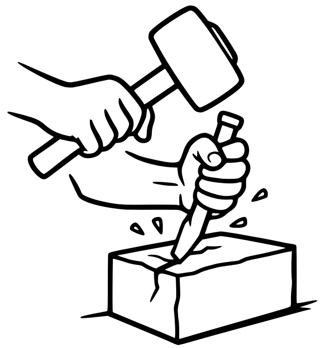
Conclusion

Two projects on connecting the seL4 abstract spec to users.

Spec splitting:

- Problem: mismatch between kernel and user timelines.
- Need user timelines to verify user programs.
- Proposal: subdivide kernel steps.
- Substantial work in progress on subdivision.
- To-do: work with derived user spec.

Also kernel/user proof integration.



School of Computer Science & Engineering

Trustworthy Systems Group

Verifying Kernel–Userland Integration for LionsOS

Dr Rob Sison

Senior Research Associate, UNSW Sydney

r.sison@unsw.edu.au

The verified seL4 OS microkernel

seL4: World's first OS kernel with correctness proof!

But: seL4 is proved functionally correct in some senses, but not others needed

The verified seL4 OS microkernel

seL4: World's first OS kernel with correctness proof!

But: seL4 is proved functionally correct in some senses, but not others needed

Abstract Model

Proofs machine-checked using Isabelle/HOL *interactive theorem prover*

The verified seL4 OS microkernel

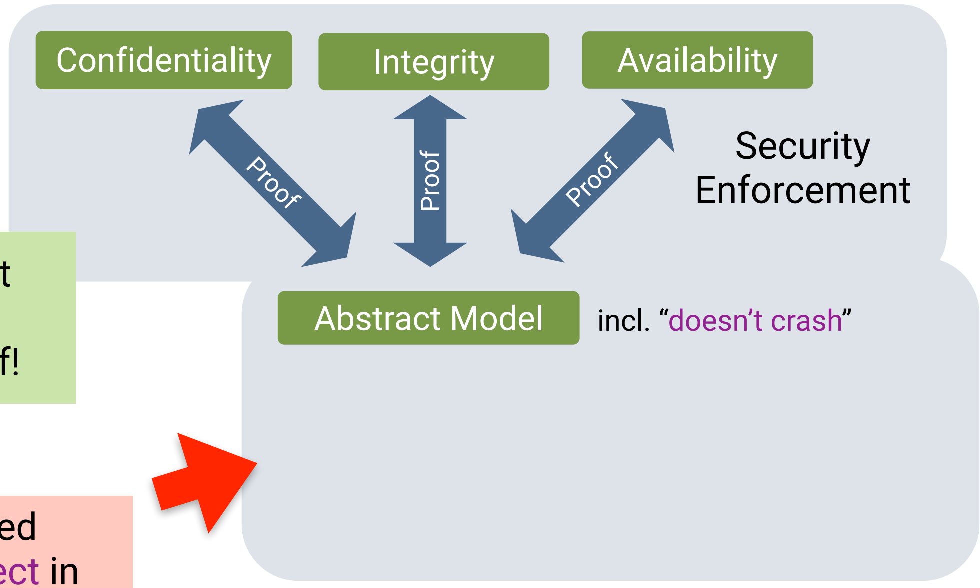
seL4: World's first OS kernel with correctness proof!

Abstract Model incl. "doesn't crash"

Proofs machine-checked using Isabelle/HOL *interactive theorem prover*

But: seL4 is proved functionally correct in some senses, but not others needed

The verified seL4 OS microkernel



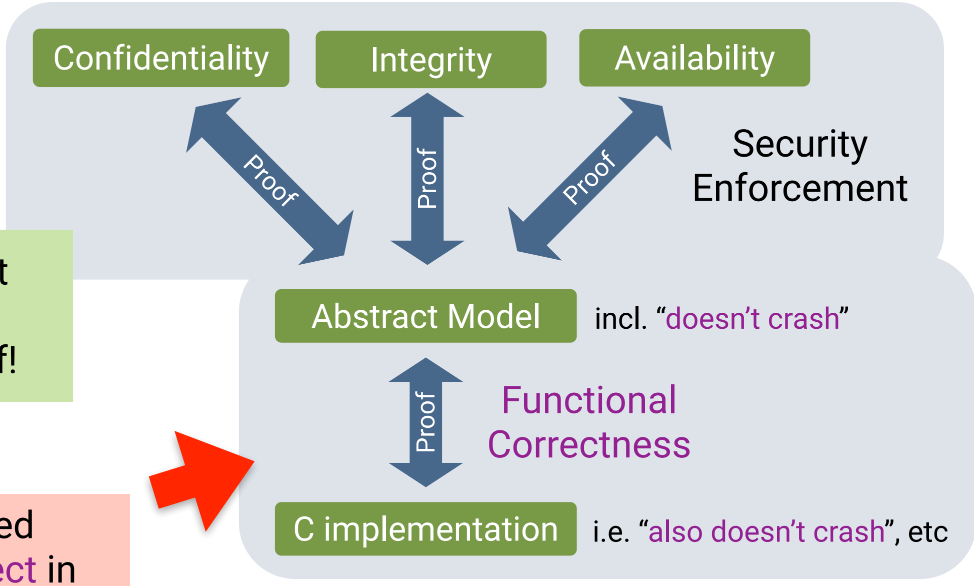
Proofs machine-checked using Isabelle/HOL *interactive theorem prover*

seL4: World's first OS kernel with correctness proof!

But: seL4 is proved functionally correct in some senses, but not others needed

The verified seL4 OS microkernel

seL4: World's first OS kernel with correctness proof!

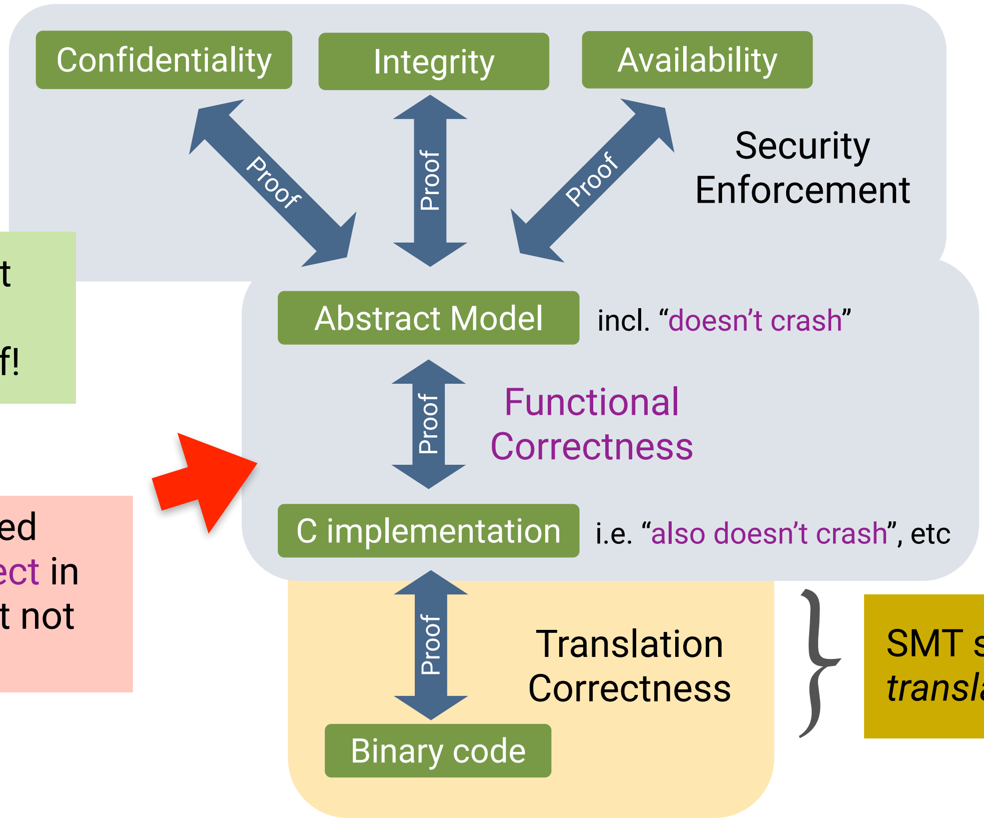


Proofs machine-checked using Isabelle/HOL *interactive theorem prover*

But: seL4 is proved functionally correct in some senses, but not others needed

The verified seL4 OS microkernel

seL4: World's first OS kernel with correctness proof!



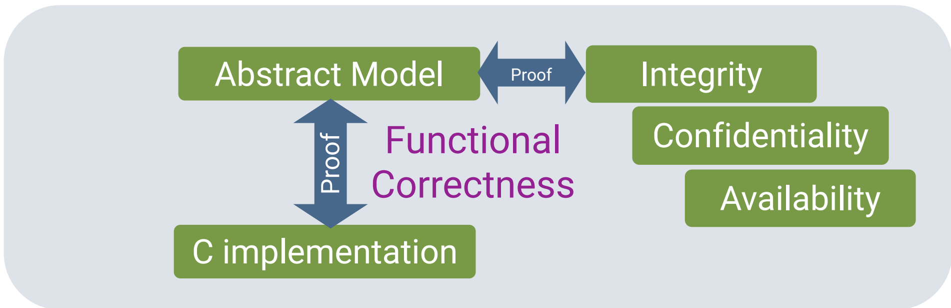
But: seL4 is proved functionally correct in some senses, but not others needed

Proofs machine-checked using Isabelle/HOL *interactive theorem prover*

SMT solver-based *translation validation*

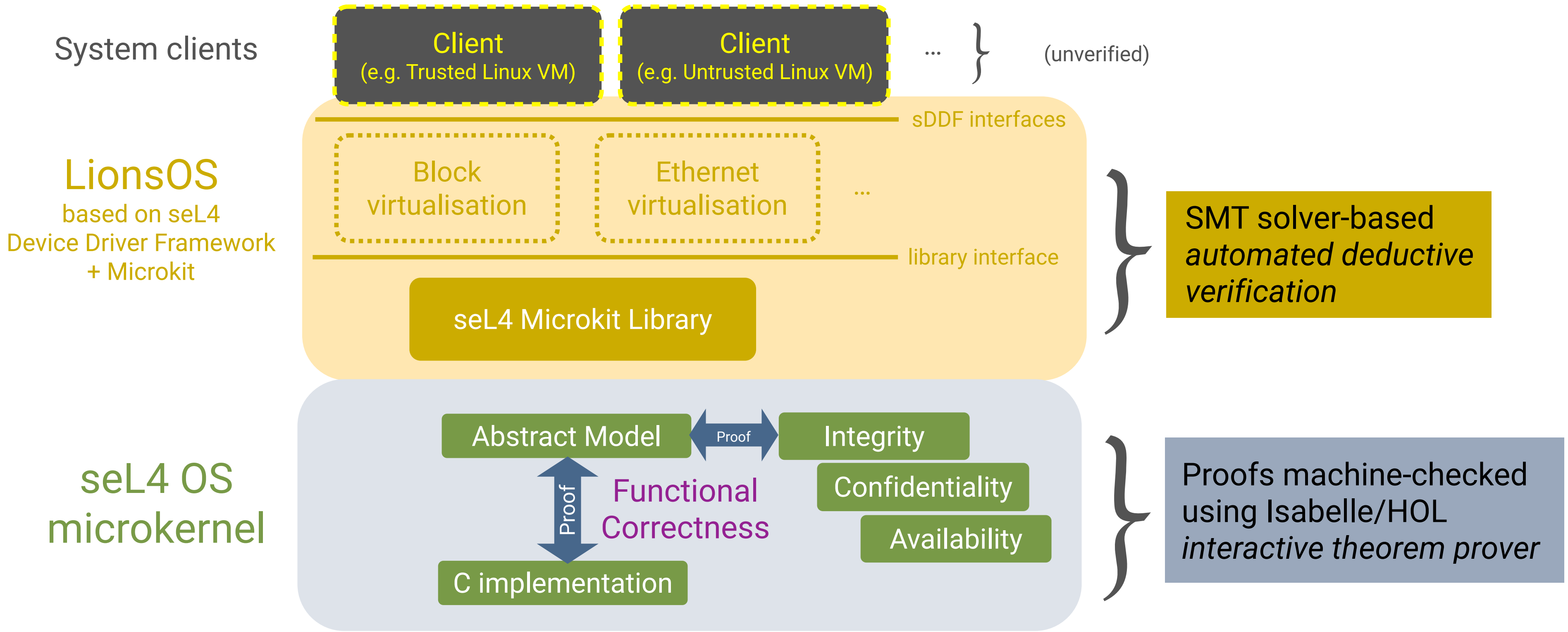
Verifying LionsOS: Vision

seL4 OS
microkernel

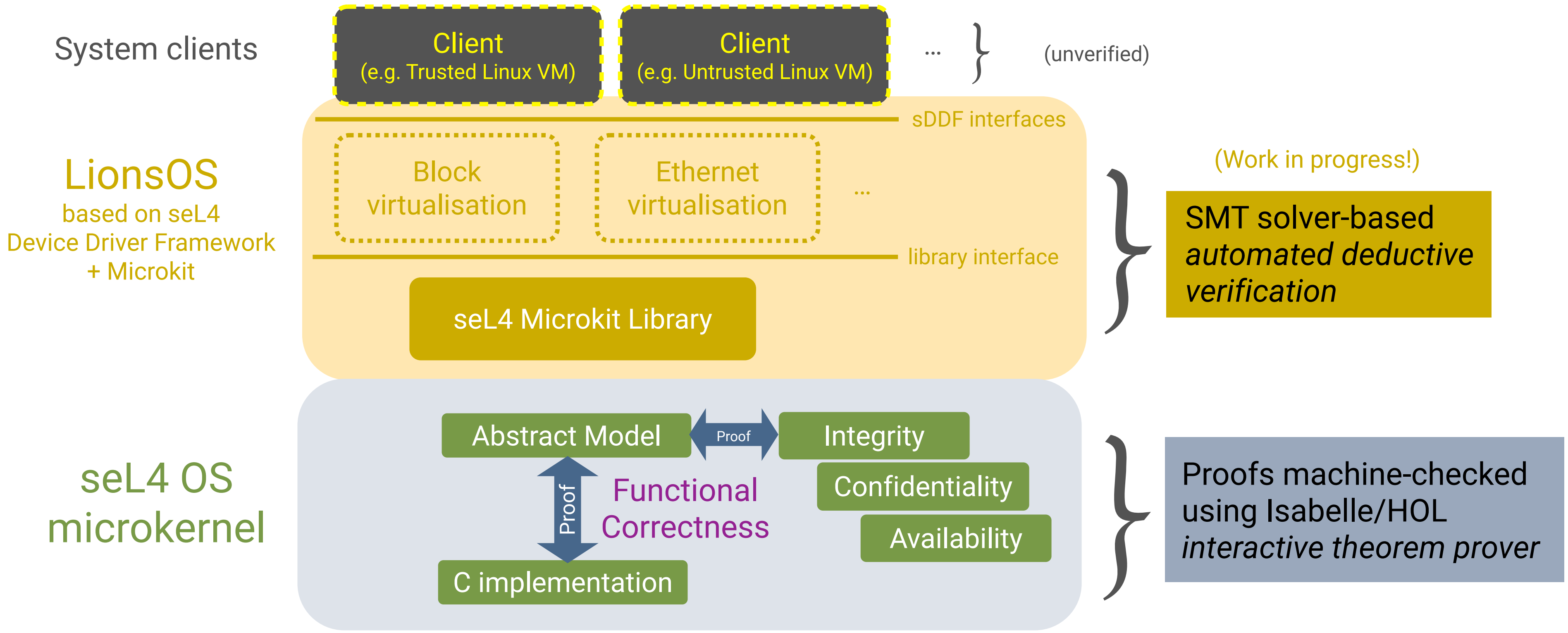


Proofs machine-checked
using Isabelle/HOL
interactive theorem prover

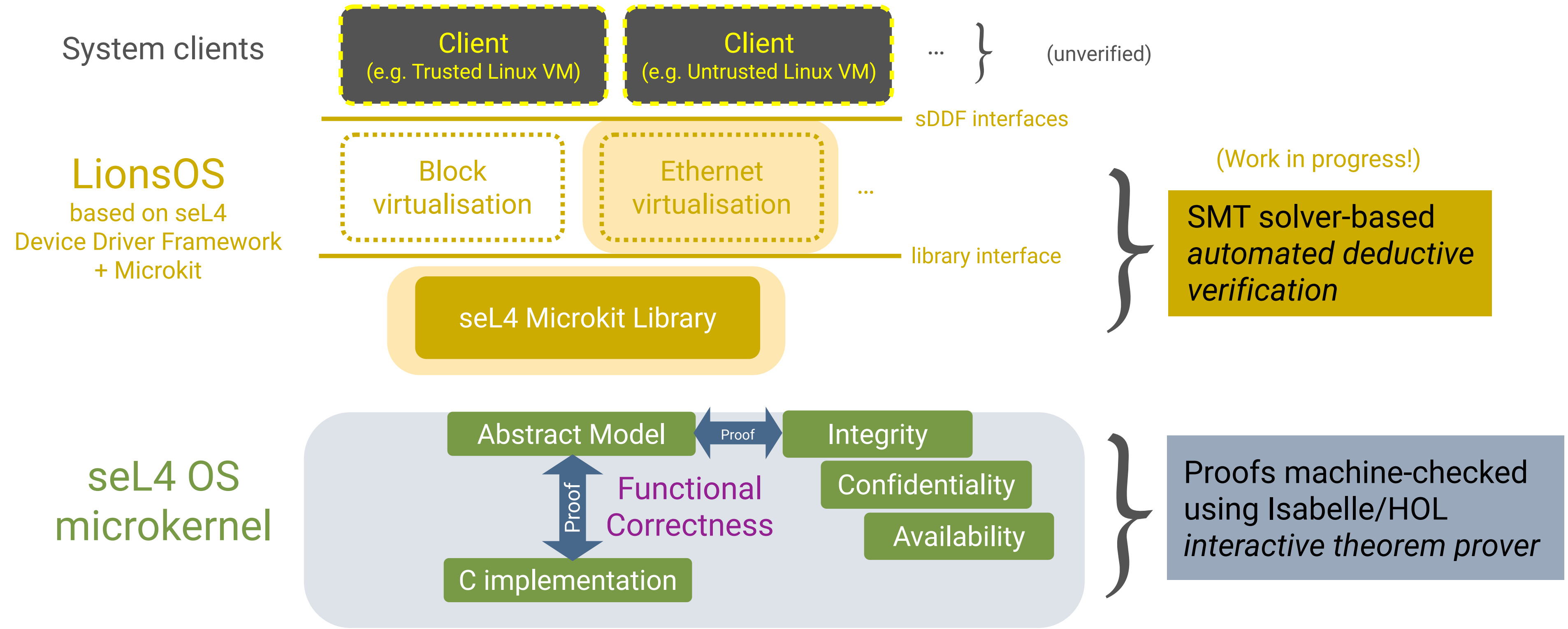
Verifying LionsOS: Vision



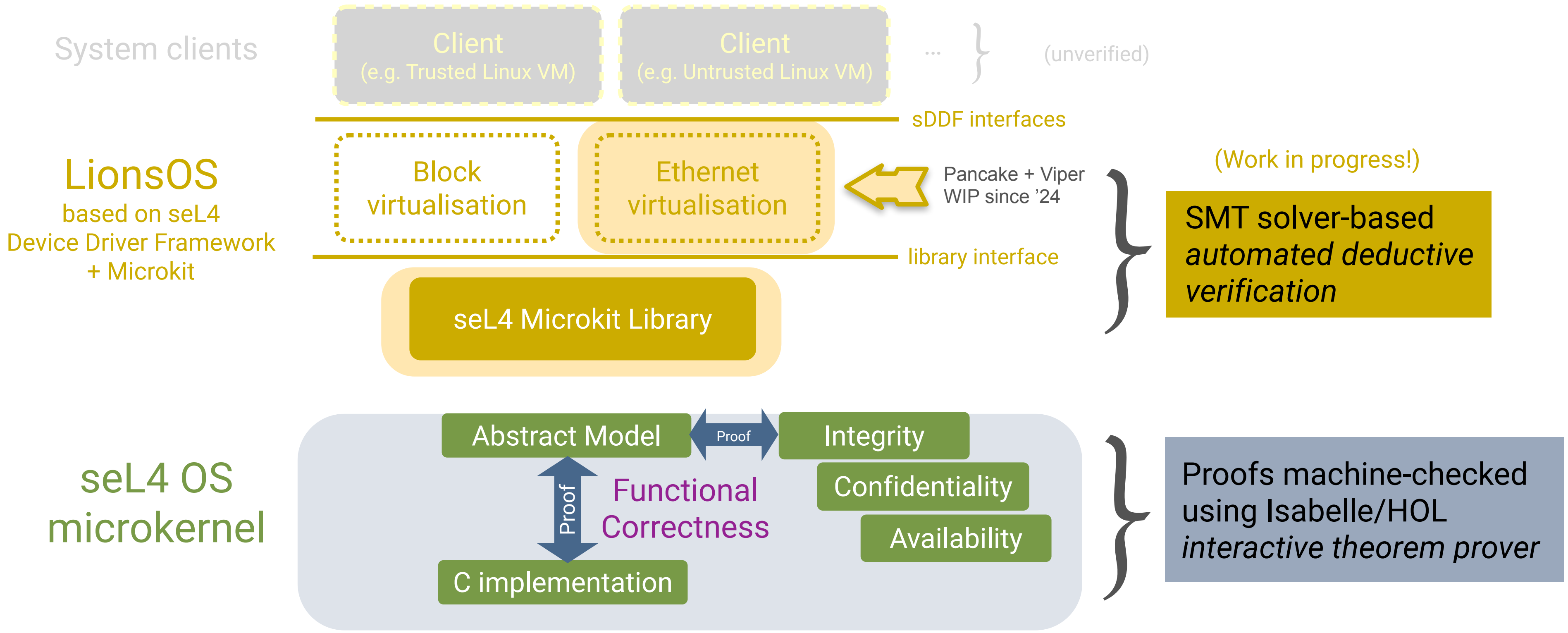
Verifying LionsOS: Vision



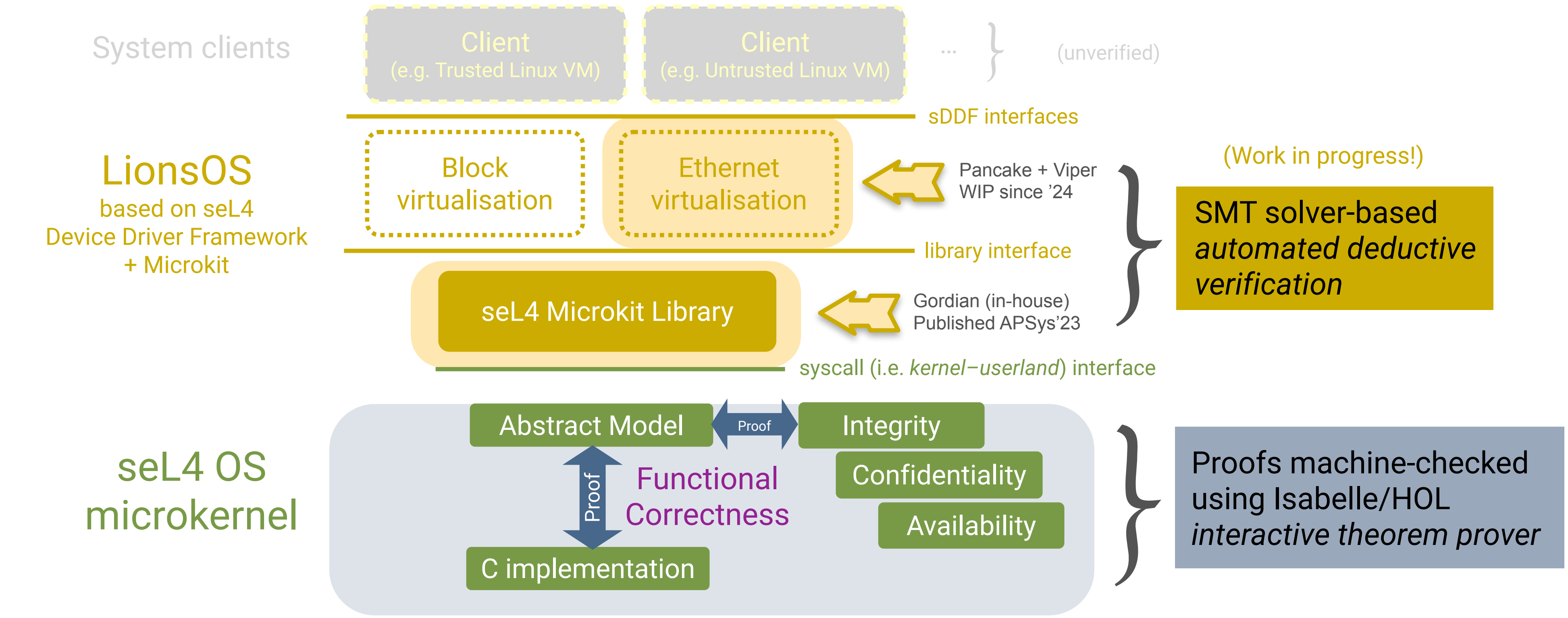
Verifying LionsOS: Status



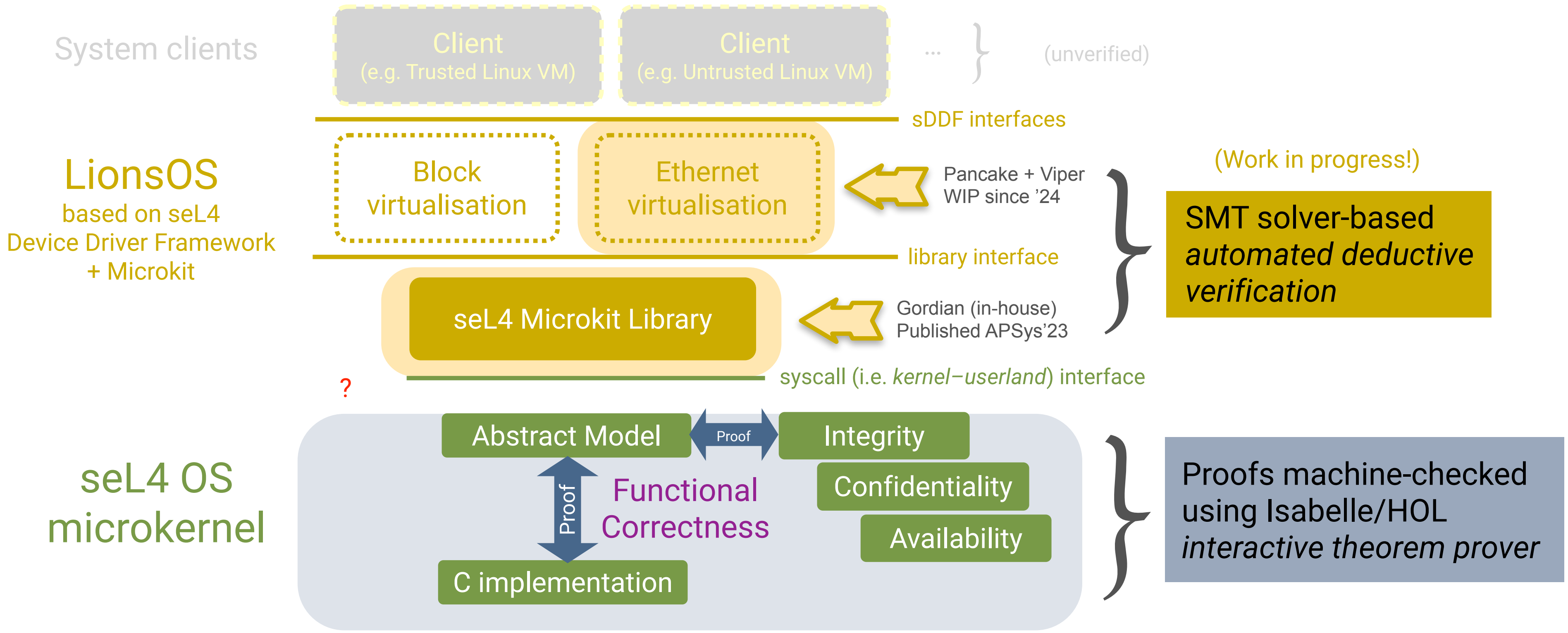
Verifying LionsOS: Status



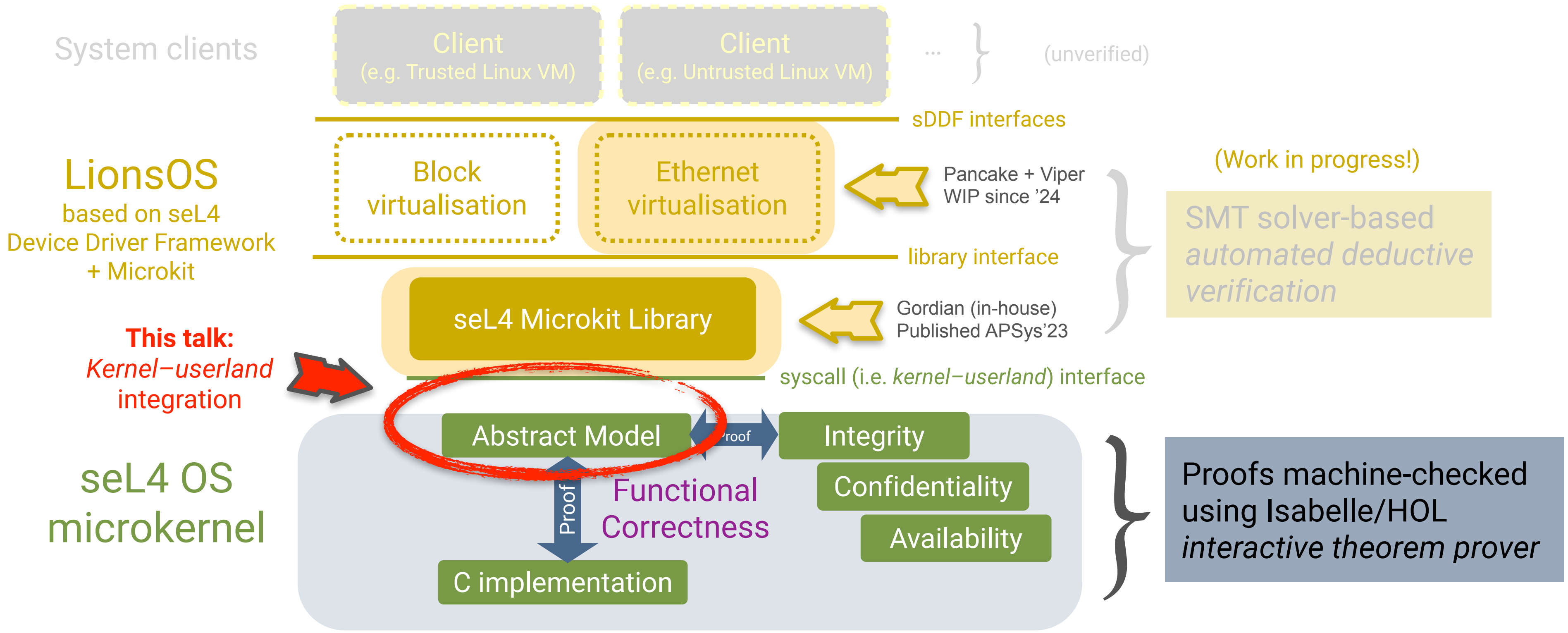
Verifying LionsOS: Status



Verifying LionsOS: Status



Verifying LionsOS: Status



Microkit uses seL4 system calls

```
static void handler_loop(void) {
    …
    for (;;) {
        seL4_Word badge;
        seL4_MessageInfo_t tag;

        if (have_reply) {
            tag = seL4_ReplyRecv(INPUT_CAP, reply_tag, &badge, REPLY_CAP);
        } else if (microkit_have_signal) {
            tag = seL4_NBSendRecv(microkit_signal_cap, microkit_signal_msg, INPUT_CAP, &badge, REPLY_CAP);
            microkit_have_signal = seL4_False;
        } else {
            tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        }
    …
    }
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 Microkit uses seL4 system calls

```
static void handler_loop(void) {
    ...
    for (;;) {
        seL4_Word badge;
        seL4_MessageInfo_t tag;

        if (have_reply) {
            tag = seL4_ReplyRecv(INPUT_CAP, reply_tag, &badge, REPLY_CAP);
        } else if (microkit_have_signal) {
            tag = seL4_NBSendRecv(microkit_signal_cap, microkit_signal_msg, INPUT_CAP, &badge, REPLY_CAP);
            microkit_have_signal = seL4_False;
        } else {
            tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        }
        ...
    }
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    …

        if (have_reply) {
            …
        } else {



            tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);



        }

    …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    …

        if (have_reply) {
            …
        } else {



            tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);



        }

    …
}
```

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    …
     {{ R }}
     if (have_reply) {
         …
     } else {

          tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);

     }
     {{ S }}
    …
}
```

Precondition

Postcondition

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    …
      {{ R }}
      if (have_reply) {
        …
      } else {
        {{ R && ! have_reply && ! microkit_have_signal }}


        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);



      }
      {{ S }}
    …
}
```

Precondition

Postcondition

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    …
     {{ R }}        Precondition
     if (have_reply) {
         …
     } else {
         {{ R && ! have_reply && ! microkit_have_signal }}
         // if R && … => P, assume
         {{ P }}
         tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
         {{ Q }}
     }
     {{ S }}
    …               Postcondition
}
```

Precond.

Postcond.

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

UNSW SYDNEY

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    …
     {{ R }}          Precondition
     if (have_reply) {
         …
     } else {
         {{ R && ! have_reply && ! microkit_have_signal }}
Precond. // if R && … => P, assume
         {{ P }}
         tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
         {{ Q }}
         // if Q => S, conclude
Postcond. {{ S }}
     }
     {{ S }}
    …               Postcondition
}
```

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    …
     {{ R }}
     if (have_reply) {
        {{ R && … }} … {{ S }}
     } else {
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
     }
     {{ S }}
    …
}
```

Precondition

Postcondition

Precond.

Postcond.

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    ...
      {{ R }}        Precondition
      if (have_reply) {
         {{ R && … }} … {{ S }}
      } else {
         {{ R && ! have_reply && ! microkit_have_signal }}
         // if R && … => P, assume
         {{ P }}
         tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
         {{ Q }}
         // if Q => S, conclude
         {{ S }}
      }
      {{ S }} // done
    ...          Postcondition
}
```

Precond.

Postcond.

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

UNSW SYDNEY

Microkit assumes seL4's syscalls work

```
static void handler_loop(void) {
    …
     {{ R }}
     if (have_reply) {
        {{ R && … }} … {{ S }}
     } else {
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
     }
     {{ S }} // done
    …
}
```

Precondition

Precond.

Postcond.

Postcondition

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit assumes seL4's syscalls work

But is it true?

```
static void handler_loop(void) {
    …
      {{ R }}          Precondition
    if (have_reply) {
        {{ R && … }} … {{ S }}
    } else {
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
    }
    {{ S }} // done
    …                  Postcondition
}
```

Precond.

Postcond.

?

Hoare triples:
{{ Pre }} f {{ Post }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

syscall (i.e. *kernel–userland*) interface

seL4 Microkit Library

seL4 OS Microkernel Abstract Model

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

```
definition call_kernel :: "event ⇒ (unit, …) s_monad" where

  call_kernel ev ≡ do
      handle_event ev …;
      schedule;
      activate_thread
  od
```

seL4 OS Microkernel Abstract Model

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

```
definition call_kernel :: "event ⇒ (unit, …) s_monad" where
  {{ P }}
  call_kernel ev ≡ do
      handle_event ev …;
      schedule;
      activate_thread
  od
  {{ Q }} ?
```

seL4 OS Microkernel Abstract Model

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

```
definition call_kernel :: "event ⇒ (unit, …) s_monad" where
  {{ invs }}
  call_kernel ev ≡ do
      handle_event ev …;
      schedule;
      activate_thread
  od
  {{ invs }}   incl. "doesn't crash"
```

seL4 OS Microkernel Abstract Model

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
   …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

```
definition call_kernel :: "event ⇒ (unit, …) s_monad" where
 {{ P }}            {{ invs }}
 call_kernel ev ≡ do
     handle_event ev …;
     schedule;
     activate_thread
 od
 {{ Q }} ?          {{ invs }}  incl. "doesn't crash"
```

seL4 OS Microkernel Abstract Model

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

```
definition call_kernel :: "event ⇒ (unit, …) s_monad" where
 {{ P }}
 call_kernel ev ≡ do
     handle_event ev …;
     schedule;
     activate_thread
 od
 {{ Q }} ? ✗
```

seL4 OS Microkernel
Abstract Model

UNSW SYDNEY

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
   …
}
```

syscall (i.e. *kernel–userland*) interface

```
definition call_kernel :: "event ⇒ (unit, …) s_monad" where
{{ P }} => {{ P' && cap_reg1 == INPUT_CAP && cap_reg2 == REPLY_CAP }}
    call_kernel ev ≡ do
        handle_event ev …;
        schedule;
        activate_thread
    od
{{ Q }} <= {{ Q' && badge_reg == ? }}
```

seL4 Microkit Library

Note: libsel4 wrappers do this register marshalling - not focusing on it for now!

seL4 OS Microkernel Abstract Model

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
   …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Note: libsel4 wrappers do this register marshalling - not focusing on it for now!

```
        definition call_kernel :: "event ⇒ (unit, …) s_monad" where
{{ P }} => {{ P' && cap_reg1 == INPUT_CAP && cap_reg2 == REPLY_CAP }}
          call_kernel ev ≡ do
              handle_event ev …;
              schedule;
              activate_thread
          od
{{ Q }} <= {{ Q' && badge_reg == ? }}
```

seL4 OS Microkernel Abstract Model

Question:

Is proving a Hoare triple over one call_kernel entry always enough?

Microkit's assumptions need verifying!

```
static void handler_loop(void) { …
        {{ R && ! have_reply && ! microkit_have_signal }}
        // if R && … => P, assume
        {{ P }}
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
        {{ Q }}
        // if Q => S, conclude
        {{ S }}
   …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Note: libsel4 wrappers do this register marshalling - not focusing on it for now!

```
      definition call_kernel :: "event ⇒ (unit, …) s_monad" where
{{ P }} => {{ P' && cap_reg1 == INPUT_CAP && cap_reg2 == REPLY_CAP }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread
        od
{{ Q }} ✗ {{ Q' && badge_reg == ? }}
```

seL4 OS Microkernel Abstract Model

Question:

Is proving a Hoare triple over one call_kernel entry always enough?

No.

System calls can block

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ P_A }}
        tag = seL4_Recv(…);
        {{ Q_A }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

{{ P_A }} => {{ P' && … }}

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

{{ Q_A }} ✗ {{ Q' && … }}

seL4 OS Microkernel Abstract Model

System calls can block

Process A

Process B

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ PA }}
        tag = seL4_Recv(…);
        {{ QA }}
        // if Q => S, conclude
        {{ S }}
   …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

```
{{ PA }} => {{ P' && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // B
        od
        {{ Q' && … }}
```

seL4 OS Microkernel Abstract Model

System calls can block

Process A

Process B

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ PA }}
        tag = seL4_Recv(…);
        {{ QA }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

{{ PA }} => {{ P' && … }}

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread // B
od
```

{{ Q' && … }}

Unblocking seL4_Signal call

seL4 OS Microkernel Abstract Model

System calls can block

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ PA }}
        tag = seL4_Recv(…);
        {{ QA }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

Process B

```
static inline void
microkit_notify(microkit_channel ch) { …
    {{ PB }}
    seL4_Signal(… + ch);
    {{ QB }}
}
```

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

```
{{ PA }} => {{ P' && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // B
        od
        {{ Q' && … }}
```

Unblocking seL4_Signal call

seL4 OS Microkernel Abstract Model

seL4 System calls can block

Verifying these is an unsolved problem*

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ PA }}
        tag = seL4_Recv(…);
        {{ QA }}
        // if Q => S, conclude
        {{ S }}
    …
}
```

Process B

```
static inline void
microkit_notify(microkit_channel ch) { …
    {{ PB }}
    seL4_Signal(… + ch);
    {{ QB }}
}
```

seL4 Microkit Library

* to the best of our knowledge, and when they block on other processes (not just I/O!)

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv **call**

```
{{ PA }} => {{ P' && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // B
        od
        {{ Q' && … }}
```

Unblocking seL4_Signal **call**

seL4 OS Microkernel Abstract Model

seL4 System calls can block

Verifying these is an unsolved problem*

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ PA }}
        tag = seL4_Recv(…);
        {{ QA }}
        // if Q => S, conclude
        {{ S }}
  …
}
```

Process B

```
static inline void
microkit_notify(microkit_channel ch) { …
    {{ PB }}
    seL4_Signal(… + ch);
    {{ QB }}
}
```

seL4 Microkit Library

* to the best of our knowledge, and when they block on other processes (not just I/O!)

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

```
{{ PA }} => {{ P' && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // B
        od
        {{ Q' && … }}
```

Unblocking seL4_Signal call

```
{{ PB }} => {{ P" && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread
        od
        {{ Q" && … }}
```

seL4 OS Microkernel Abstract Model

seL4 System calls can block

Verifying these is an unsolved problem*

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ PA }}
        tag = seL4_Recv(…);
        {{ QA }}
        // if Q => S, conclude
        {{ S }}
  …
}
```

Process B

```
static inline void
microkit_notify(microkit_channel ch) { …
    {{ PB }}
    seL4_Signal(… + ch);
    {{ QB }}
}
```

seL4 Microkit Library

* to the best of our knowledge, and when they block on other processes (not just I/O!)

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

```
{{ PA }} => {{ P' && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // B
        od
        {{ Q' && … }}
```

Unblocking seL4_Signal call

```
{{ PB }} => {{ P" && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // A
        od
        {{ Q" && … }}  => {{ QA }}
```

seL4 OS Microkernel Abstract Model

System calls can block

Verifying these is an unsolved problem*

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ PA }}
        tag = seL4_Recv(…);
        {{ QA }}
        // if Q => S, conclude
        {{ S }}
   …
}
```

Process B

```
static inline void
microkit_notify(microkit_channel ch) { …
    {{ PB }}
    seL4_Signal(… + ch);
    {{ QB }}
}
```

seL4 Microkit Library

* to the best of our knowledge, and when they block on other processes (not just I/O!)

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

```
{{ PA }} => {{ P' && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // B
        od
        {{ Q' && … }}
```

Unblocking seL4_Signal call

```
{{ PB }} => {{ P" && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // A
        od
        {{ Q" && … }}  => {{ QA }}
```

seL4 OS Microkernel Abstract Model

System calls can block

Verifying these is an unsolved problem*

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ P_A }}
        tag = seL4_Recv(…);
        {{ Q_A }}
        // if Q => S, conclude
        {{ S }}
  …
}
```

Process B

```
static inline void
microkit_notify(microkit_channel ch) { …
    {{ P_B }}
    seL4_Signal(… + ch);
    {{ Q_B }}
}
```

seL4 Microkit Library

* to the best of our knowledge, and when they block on other processes (not just I/O!)

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

{{ P_A }} => {{ P' && … }}

```
call_kernel ev ≡ do
    handle_event ev …;
?   schedule;
    activate_thread // X
od
```

{{ Q' && … }}

Unblocking seL4_Signal call

{{ P_B }} => {{ P'' && … }}

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread // A
od
```

{{ Q'' && … }} => {{ Q_A }}

seL4 OS Microkernel Abstract Model

System calls can block

Verifying these is an unsolved problem*

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ P_A }}
        tag = seL4_Recv(…);
        {{ Q_A }}
        // if Q => S, conclude
        {{ S }}
   …
}
```

Process B

```
static inline void
microkit_notify(microkit_channel ch) { …
   {{ P_B }}
   seL4_Signal(… + ch);
   {{ Q_B }}
}
```

seL4 Microkit Library

* to the best of our knowledge, and when they block on other processes (not just I/O!)

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

```
{{ P_A }} => {{ P' && … }}
        call_kernel ev ≡ do
           handle_event ev …;
      ?    schedule;
           activate_thread // X
        od
        {{ Q' && … }}
```

X, Y, Z … B

Unblocking seL4_Signal call

```
{{ P_B }} => {{ P" && … }}
        call_kernel ev ≡ do
           handle_event ev …;
           schedule;
           activate_thread // A
        od
        {{ Q" && … }}  => {{ Q_A }}
```

seL4 OS Microkernel Abstract Model

System calls can block

Verifying these is an unsolved problem*

Process A

```
static void handler_loop(void) { …
        {{ R && … }}
        // if R && … => P, assume
        {{ PA }}
        tag = seL4_Recv(…);
        {{ QA }}
        // if Q => S, conclude
        {{ S }}
   …
}
```

Process B

```
static inline void
microkit_notify(microkit_channel ch) { …
    {{ PB }}
    seL4_Signal(… + ch);
    {{ QB }}
}
```

seL4 Microkit Library

* to the best of our knowledge, and when they block on other processes (not just I/O!)

syscall (i.e. *kernel–userland*) interface

Blocking seL4_Recv call

```
{{ PA }} => {{ P' && … }}
        call_kernel ev ≡ do
            handle_event ev …;
      ?     schedule;
            activate_thread // X
        od
        {{ Q' && … }}
```

X, Y, Z … B

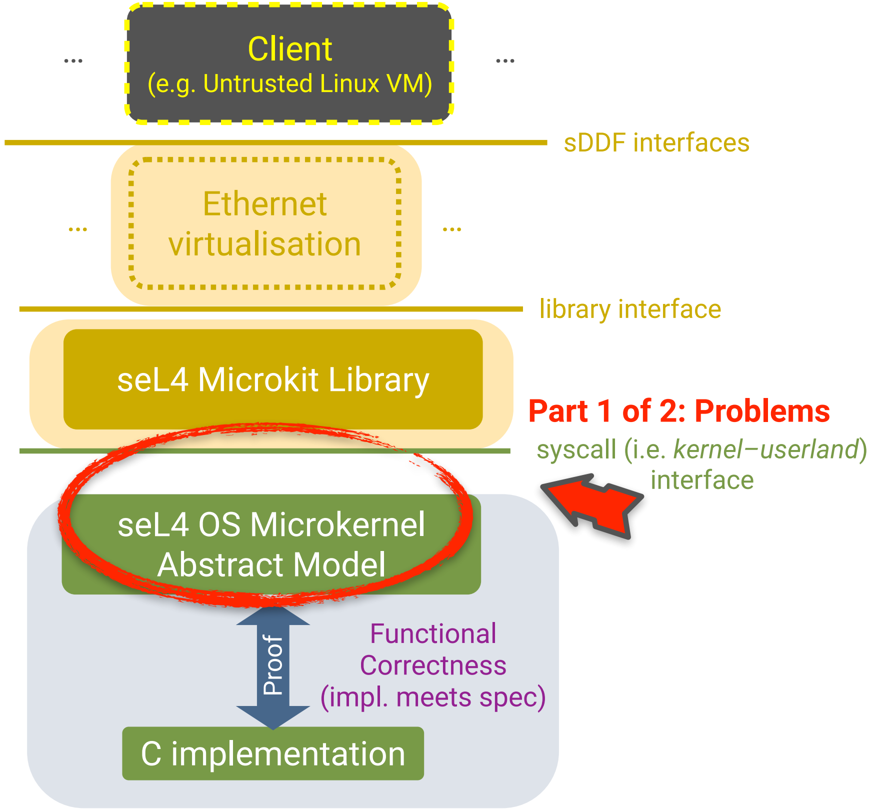
Irrelevant calls (IRQs too!)

Unblocking seL4_Signal call

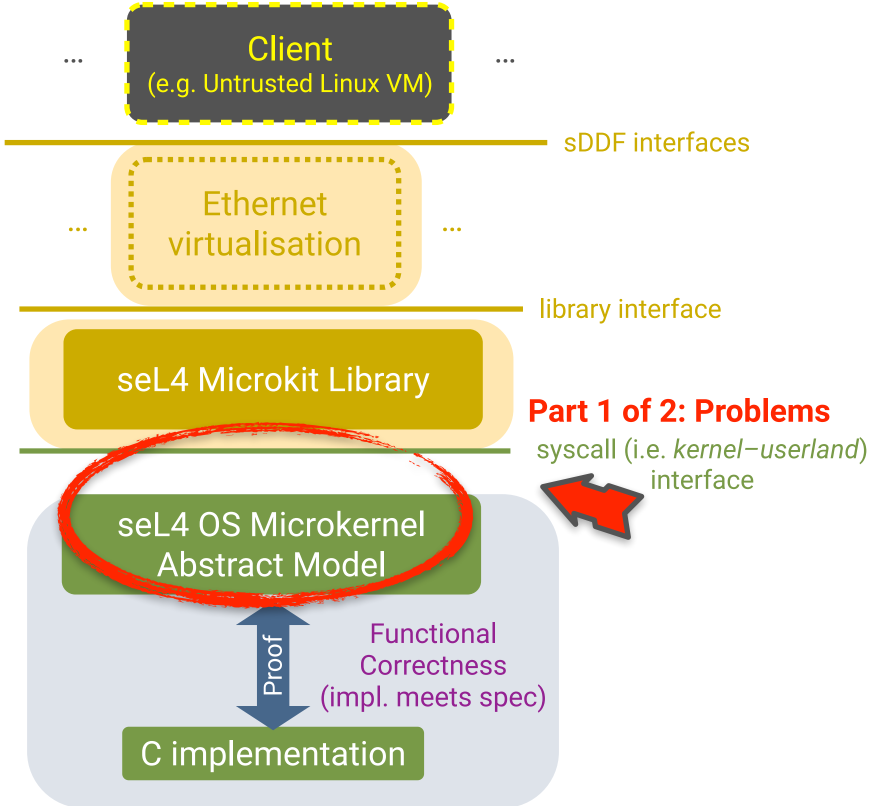
```
{{ PB }} => {{ P" && … }}
        call_kernel ev ≡ do
            handle_event ev …;
            schedule;
            activate_thread // A
        od
        {{ Q" && … }}  => {{ QA }}
```

seL4 OS Microkernel Abstract Model

Summary: Kernel–userland gap so far

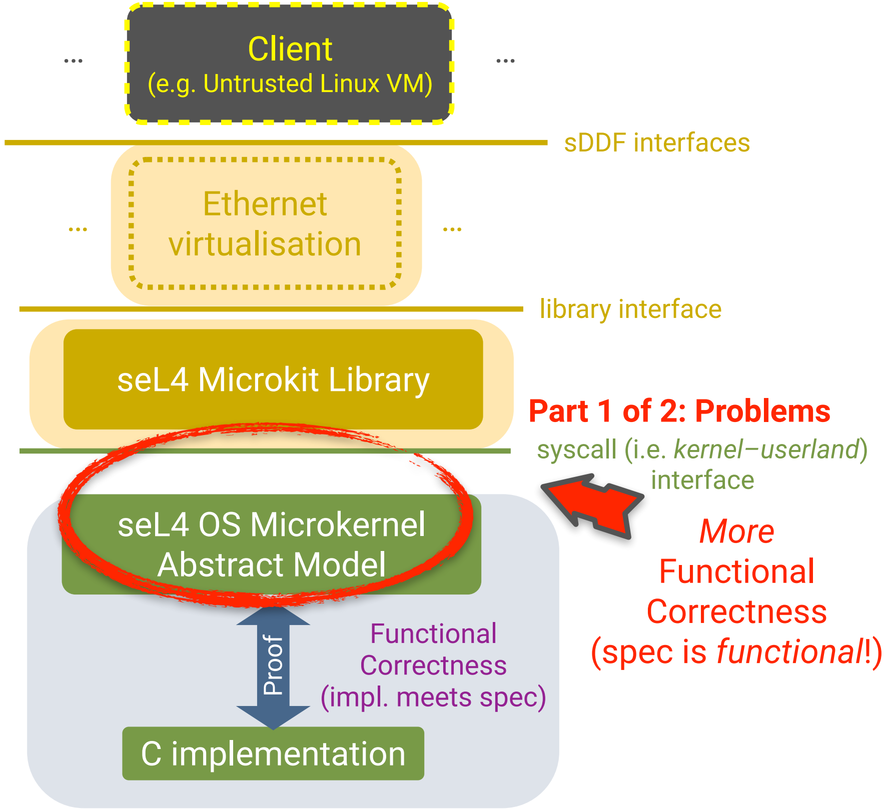


Summary: Kernel–userland gap so far



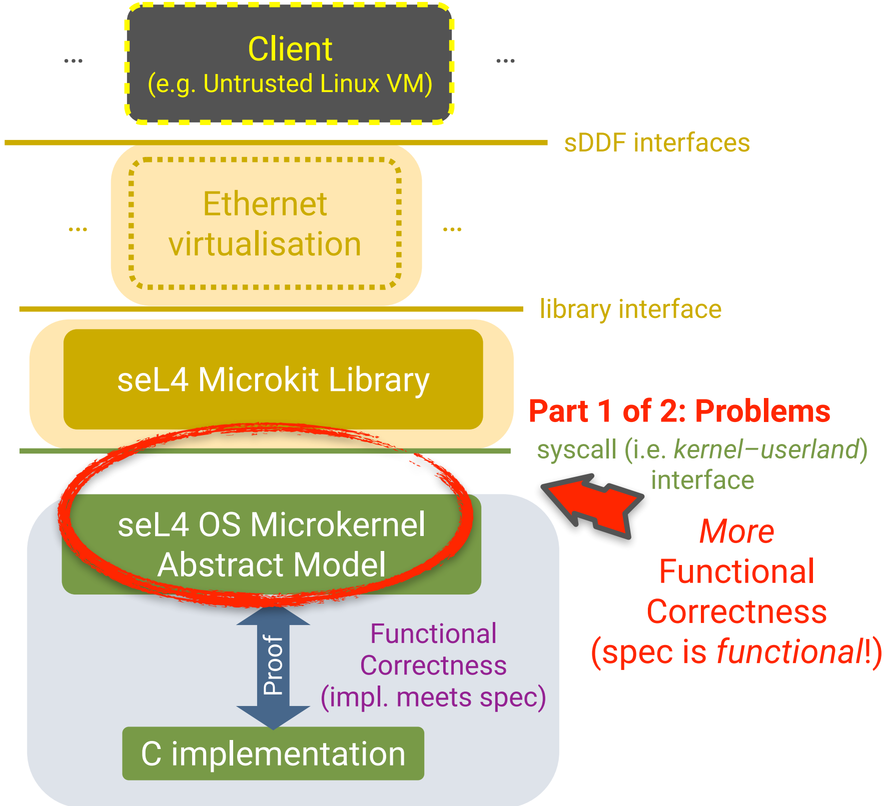
- Microkit assumes syscall functional specs as Hoare triples

Summary: Kernel–userland gap so far



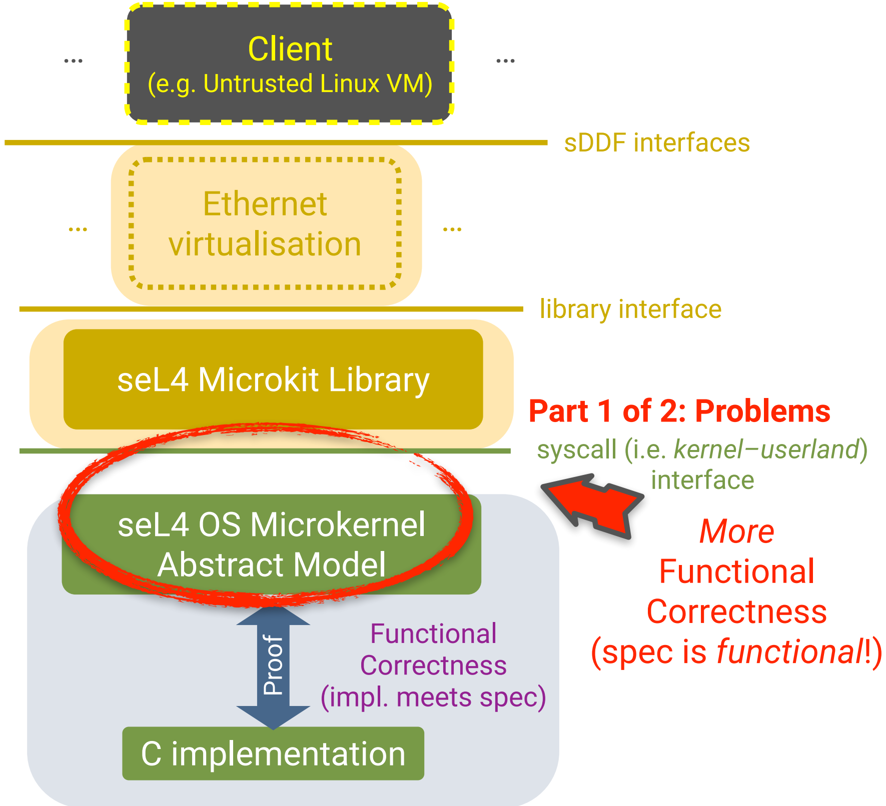
- Microkit assumes syscall functional specs as Hoare triples

Summary: Kernel–userland gap so far



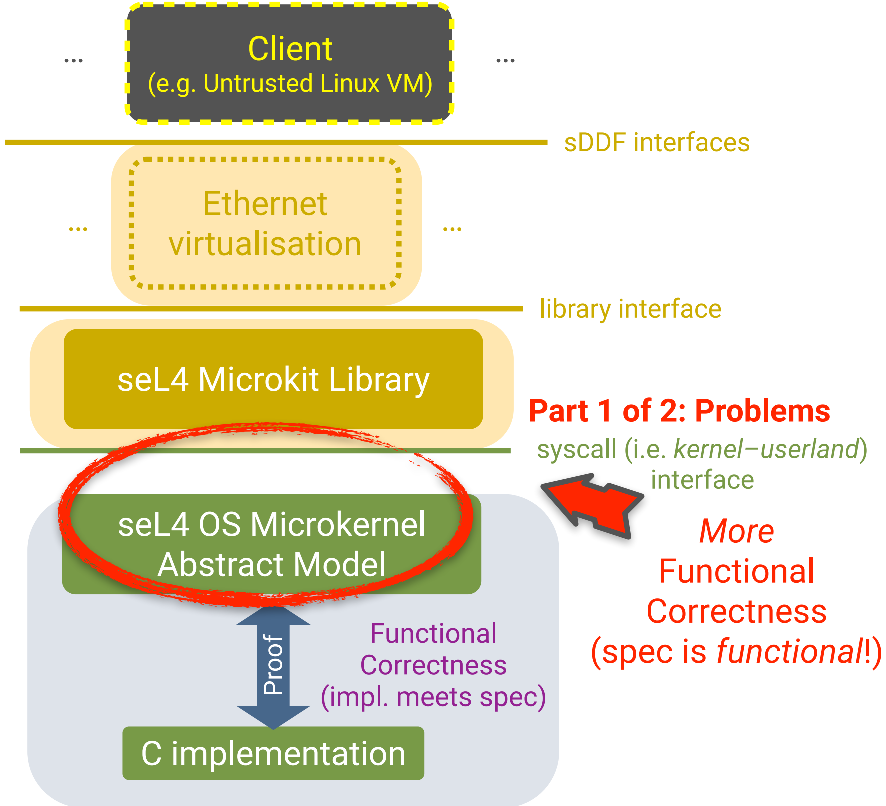
- Microkit assumes syscall functional specs as Hoare triples
- We now need to prove seL4's Abstract Model satisfies them!
  (Then we'll know its C implementation does, too.)

Summary: Kernel–userland gap so far



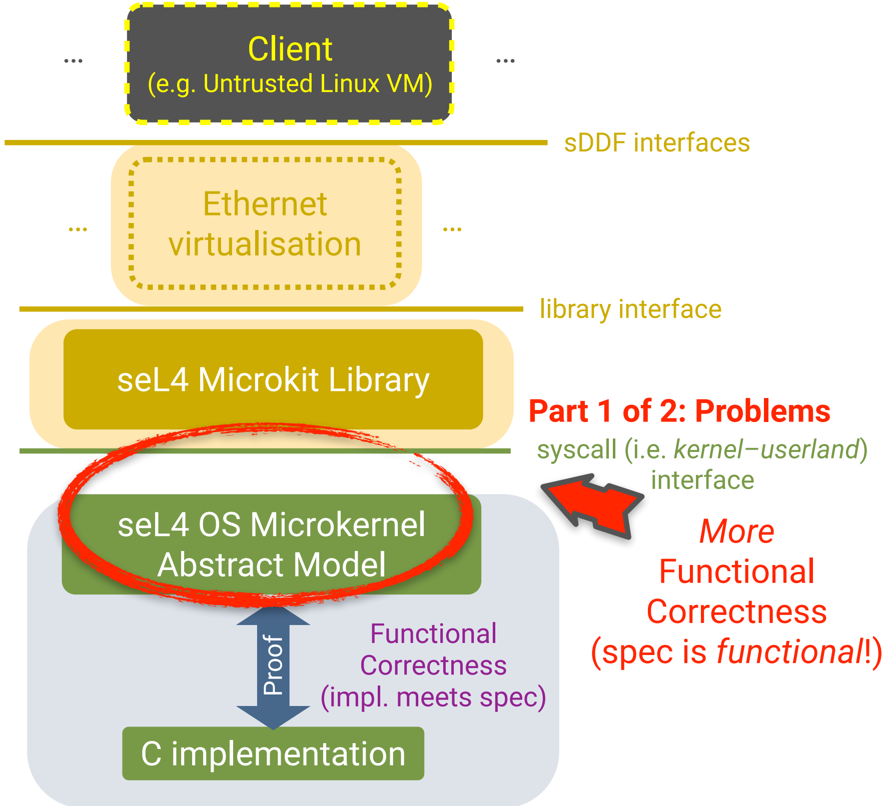
- Microkit assumes syscall functional specs as Hoare triples
- We now need to prove seL4's Abstract Model satisfies them!
  (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)

Summary: Kernel–userland gap so far



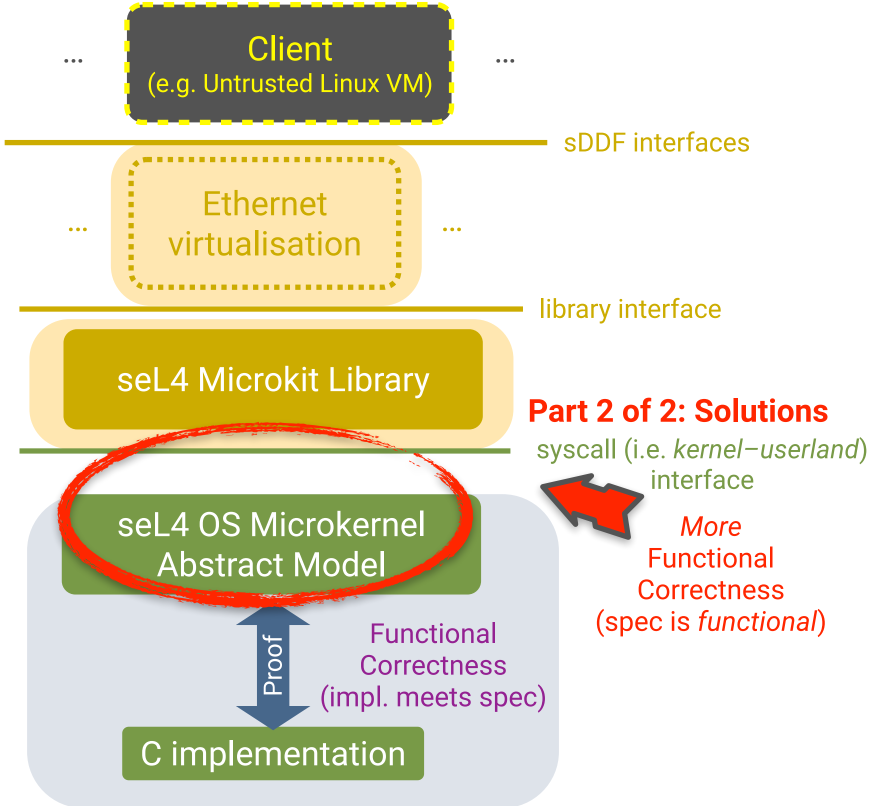
- Microkit assumes syscall functional specs as Hoare triples
- We now need to prove seL4's Abstract Model satisfies them!
  (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)
- Blocking syscalls may require Hoare triples for up to 2 + *n* kernel entries (finite *n*):
  - 1 **Blocking** call (e.g. Recv)
  - *n* **Irrelevant** calls or interrupts
  - 1 **Unblocking** call (e.g. Signal)

Summary: Kernel–userland gap so far



- Microkit assumes syscall functional specs as Hoare triples
- We now need to prove seL4's Abstract Model satisfies them!
  (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)
- Blocking syscalls may require Hoare triples for up to 2 + *n* kernel entries (finite *n*):
  - 1 **Blocking** call (e.g. Recv)
  - *n* **Irrelevant** calls or interrupts
  - 1 **Unblocking** call (e.g. Signal)
- Verifying syscalls like these is **an open problem**

Towards kernel–userland functional proofs



- Microkit assumes syscall functional specs as Hoare triples
- We now need to prove seL4's Abstract Model satisfies them! (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)
- Blocking syscalls may require Hoare triples for up to 2 + *n* kernel entries (finite *n*):
  - 1 **Blocking** call (e.g. Recv)
  - *n* **Irrelevant** calls or interrupts
  - 1 **Unblocking** call (e.g. Signal)
- Verifying syscalls like these is **an open problem**

Towards kernel–userland functional proofs

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel
Abstract Model

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Process A
{{ P }} seL4_Recv {{ Q }}

Process B
{{ P' }} seL4_Signal {{ Q' }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ P && … }}
call_kernel
{{ Q && … }}

{{ P' && … }}
call_kernel
{{ Q' && … }}

seL4 OS Microkernel Abstract Model

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ P && … }}
call_kernel
{{ Q && … }}

Eliding register side-conds to save space…

{{ P' && … }}
call_kernel
{{ Q' && … }}

seL4 OS Microkernel Abstract Model

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Read:
ks_of = "kernel state of"

Process A
{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B
{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

- For a correctly initialised Microkit system, no X ≠ Y should possess the *capabilities* to change ***most parts*** of (ks_of Y s).

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
  call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
  call_kernel
{{ λs. Q' (ks_of B s) }}

- For a correctly initialised Microkit system, no X ≠ Y should possess the *capabilities* to change ***most parts*** of (ks_of Y s).

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

- For a correctly initialised Microkit system, no X ≠ Y should possess the *capabilities* to change ***most parts*** of (ks_of Y s).

ks_of p s ≡ ⦇
thread_cnode p s,
bound_notification p s,
mapped_writable p s,
not_writable_others p s,
recv_oracle s (bound_notification p s) (thread_cnode p s),
λch. ntfn_status s (thread_cnode p s) ch
⦈

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

- For a correctly initialised Microkit system, no X ≠ Y should possess the *capabilities* to change ***most parts*** of (ks_of Y s).

ks_of p s ≡ ⦇
thread_cnode p s,
bound_notification p s,
mapped_writable p s,
not_writable_others p s,
recv_oracle s (bound_notification p s) (thread_cnode p s),
λch. ntfn_status s (thread_cnode p s) ch
⦈

cap state, thread init!

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Read:
ks_of = "kernel state of"

Process A	Process B	
{{ λs. P s }} seL4_Recv {{ λs. Q s }}	{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}	seL4 Microkit Library
		syscall (i.e. *kernel−userland*) interface
{{ λs. P (ks_of A s) }} call_kernel {{ λs. Q (ks_of A s) }}	{{ λs. P' (ks_of B s) }} call_kernel {{ λs. Q' (ks_of B s) }}	seL4 OS Microkernel Abstract Model

- For a correctly initialised Microkit system, no X ≠ Y should possess the *capabilities* to change ***most parts*** of (ks_of Y s).

ks_of p s ≡ ⦇
  thread_cnode p s,
  bound_notification p s,
  cap state, thread init!
  mapped_writable p s,
  not_writable_others p s,
  memory mappings!
  recv_oracle s (bound_notification p s) (thread_cnode p s),
  λch. ntfn_status s (thread_cnode p s) ch
⦈

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Read:
ks_of = "kernel state of"

Process A
{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B
{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

- For a correctly initialised Microkit system, no X ≠ Y should possess the *capabilities* to change ***most parts*** of (ks_of Y s).

ks_of p s ≡ ⦇
thread_cnode p s,
bound_notification p s,
} **cap state, thread init!**
mapped_writable p s,
not_writable_others p s,
} **memory mappings!**
recv_oracle s (bound_notification p s) (thread_cnode p s),
λch. ntfn_status s (thread_cnode p s) ch
⦈

- But we still have to prove when it *does* (i.e. **Unblocking**) or *doesn't* (i.e. **Irrelevant**).

Towards kernel–userland functional proofs

- It helps that Microkit's assumptions are on *per-process* projections of the kernel state:

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
  call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
  call_kernel
{{ λs. Q' (ks_of B s) }}

seL4 OS Microkernel Abstract Model

- For a correctly initialised Microkit system, no X ≠ Y should possess the *capabilities* to change ***most parts*** of (ks_of Y s).

- But we still have to prove when it *does* (i.e. **Unblocking**) or *doesn't* (i.e. **Irrelevant**).

ks_of p s ≡ ⦇
  thread_cnode p s,
  bound_notification p s,
  } **cap state, thread init!**
  mapped_writable p s,
  not_writable_others p s,
  } **memory mappings!**
  recv_oracle s (bound_notification p s) (thread_cnode p s),
  λch. ntfn_status s (thread_cnode p s) ch
⦈

What we've done

- Defined state projections

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

seL4 OS Microkernel Abstract Model

Microkit-facing state

ks_of p s ≡ ⦇
→ thread_cnode p s,
→ bound_notification p s,
→ mapped_writable p s,
→ not_writable_others p s,
→ recv_oracle s (bound_notification p s) (thread_cnode p s),
→ λch. ntfn_status s (thread_cnode p s) ch
⦈

} **cap state, thread init!**

} **memory mappings!**

What we’ve done

- Defined state projections

Read:
ks_of = “kernel state of”

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P’ s }} seL4_Signal {{ λs. Q’ s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P’ (ks_of B s) }}
call_kernel
{{ λs. Q’ (ks_of B s) }}

seL4 OS Microkernel
Abstract Model

Microkit-facing state **Abstract Model state**

ks_of p s ≡ ⦇
thread_cnode p s,
bound_notification p s, } **cap state, thread init!**
mapped_writable p s,
not_writable_others p s, } **memory mappings!**
recv_oracle s (bound_notification p s) (thread_cnode p s),
λch. ntfn_status s (thread_cnode p s) ch
⦈

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

cap state, thread init!

memory mappings!

seL4 What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

} **cap state, thread init!**

} **memory mappings!**

UNSW SYDNEY

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

$\}$ **we'll come back to this**

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

$\}$ **cap state, thread init!** (thread_cnode, bound_notification)

$\}$ **memory mappings!** (mapped_writable, not_writable_others)

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
call_kernel
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

$\}$ **we'll come back to this** (for schedule; activate_thread)

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

$\}$ **cap state, thread init!** (thread_cnode, bound_notification)

$\}$ **memory mappings!** (mapped_writable, not_writable_others)

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A
{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B
{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

we'll come back to this

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

cap state, thread init!

memory mappings!

UNSW SYDNEY

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A
{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B
{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

```
call_kernel ev ≡ do
    handle_event ev ...;
    schedule;
    activate_thread
od
```

we'll come back to this

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

cap state, thread init!

memory mappings!

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

Blocking

seL4 OS Microkernel Abstract Model

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

} **we'll come back to this**

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

} **cap state, thread init!**

} **memory mappings!**

UNSW SYDNEY

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A
{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B
{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

Blocking

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

Blocking

Unblocking

seL4 OS Microkernel
Abstract Model

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

} **we'll come back to this**

Abstract Model state

```
ks_of p s ≡ ⦇
   thread_cnode p s,
   bound_notification p s,
   mapped_writable p s,
   not_writable_others p s,
   recv_oracle s (bound_notification p s) (thread_cnode p s),
   λch. ntfn_status s (thread_cnode p s) ch
⦈
```

} **cap state, thread init!**

} **memory mappings!**

UNSW SYDNEY

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
call_kernel
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

we'll come back to this

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

cap state, thread init!

memory mappings!

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

```
call_kernel ev ≡ do
    handle_event ev ...;
    schedule;
    activate_thread
od
```

we'll come back to this

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

cap state, thread init!

memory mappings!

seL4 What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

```
call_kernel ev ≡ do
    handle_event ev …;
    schedule;
    activate_thread
od
```

we'll come back to this

Abstract Model state

```
ks_of p s ≡ ⦇
    thread_cnode p s,
    bound_notification p s,
    mapped_writable p s,
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

cap state, thread init!

memory mappings!

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

What we proved:

ks_of p s ≡ ⦇
thread_cnode p s,
bound_notification p s,
} **cap state, thread init!**
mapped_writable p s,
not_writable_others p s,
} **memory mappings!**
recv_oracle s (bound_notification p s) (thread_cnode p s),
λch. ntfn_status s (thread_cnode p s) ch
⦈

seL4 What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

What we proved:

- More precise cap lookup lemmas

ks_of p s ≡ ⦇
thread_cnode p s,
bound_notification p s,
mapped_writable p s,
not_writable_others p s,
recv_oracle s (bound_notification p s) (thread_cnode p s),
λch. ntfn_status s (thread_cnode p s) ch
⦈

cap state, thread init!

memory mappings!

UNSW SYDNEY

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking

Unblocking

seL4 OS Microkernel Abstract Model

What we proved:

- More precise cap lookup lemmas
- handle_recv, as required:
  - **does not modify** most ks_of fields

```
ks_of p s ≡ ⦇
    thread_cnode p s,            } cap state, thread init!
    bound_notification p s,
    mapped_writable p s,         } memory mappings!
    not_writable_others p s,
    recv_oracle s (bound_notification p s) (thread_cnode p s),
    λch. ntfn_status s (thread_cnode p s) ch
⦈
```

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

What we proved:

- More precise cap lookup lemmas
- handle_recv, as required:
  - **does not modify** most ks_of fields
  - returns recv_oracle-"predicted" badge in "Nonblocking" case
  - updates ntfn_status in Blocking case

ks_of p s ≡ ⦇
thread_cnode p s,
bound_notification p s,
cap state, thread init!
mapped_writable p s,
not_writable_others p s,
memory mappings!
recv_oracle s (bound_notification p s) (thread_cnode p s),
λch. ntfn_status s (thread_cnode p s) ch
⦈

What we've done

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel−userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

What we proved:

- More precise cap lookup lemmas
- handle_recv, as required:
  - **does not modify** most ks_of fields
  - returns recv_oracle-"predicted" badge in "Nonblocking" case
  - updates ntfn_status in Blocking case
- Lifting helpers for state projections

ks_of p s ≡ ⦇
thread_cnode p s,
bound_notification p s,
cap state, thread init!
mapped_writable p s,
not_writable_others p s,
memory mappings!
recv_oracle s (bound_notification p s) (thread_cnode p s),
λch. ntfn_status s (thread_cnode p s) ch
⦈

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking

Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel−userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel
Abstract Model

- Proving & generalising their composition

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

Blocking A

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

Blocking A

Unblocking B

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking

Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

Blocking A

C, D, E … X, Y, Z

Unblocking B

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking

Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

Blocking A

C, D, E … X, Y, Z
Irrelevant

Unblocking B

seL4 What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

{{ λs. P (ks_of A s) }}
Blocking A

C, D, E … X, Y, Z
Irrelevant

Unblocking B

UNSW SYDNEY

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A
{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B
{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

{{ λs. P (ks_of A s) }}
Blocking A
{{ λs. R (ks_of A s) &&
P' (ks_of B s) }}

C, D, E … X, Y, Z
Irrelevant

Unblocking B

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

{{ λs. P (ks_of A s) }}
Blocking A
{{ λs. R (ks_of A s) &&
P' (ks_of B s) }}

C, D, E … X, Y, Z
Irrelevant

{{ λs. R (ks_of A s) && P' (ks_of B s) }}
Unblocking B

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking

Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

<u>Note</u>: If we can't prove these irrelevant for wellformed Microkit systems, then the seL4 kernel is incorrect!

{{ λs. P s }} → {{ λs. P (ks_of A s) }}
seL4_SomeCall

Blocking A

{{ λs. Q s }}

{{ λs. R (ks_of A s) &&
P' (ks_of B s) }}

C, D, E … X, Y, Z
Irrelevant

{{ λs. R (ks_of A s) && P' (ks_of B s) }}

Unblocking B

What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"

Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking

Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

Note: If we can't prove these irrelevant for wellformed Microkit systems, then the seL4 kernel is incorrect!

{{ λs. P s }}
seL4_SomeCall
{{ λs. Q s }}

{{ λs. P (ks_of A s) }}

Blocking A

{{ λs. R (ks_of A s) &&
P' (ks_of B s) }}

C, D, E … X, Y, Z
Irrelevant

{{ λs. R (ks_of A s) && P' (ks_of B s) }}

Unblocking B

{{ λs. Q (ks_of A s) && Q' (ks_of B s) }}

seL4 What we've done & plan to do

Read:
ks_of = "kernel state of"

- Defined state projections
- Verifying Hoare triples on calls

Process A

{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B

{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

seL4 OS Microkernel Abstract Model

- Proving & generalising their composition

Note: If we can't prove these irrelevant for wellformed Microkit systems, then the seL4 kernel is incorrect!

{{ λs. P s }} → {{ λs. P (ks_of A s) }}

Blocking A

seL4_SomeCall {{ λs. R (ks_of A s) && P' (ks_of B s) }}

C, D, E … X, Y, Z
Irrelevant

{{ λs. R (ks_of A s) && P' (ks_of B s) }}

Unblocking B

{{ λs. Q s }} ← {{ λs. Q (ks_of A s) && Q' (ks_of B s) }}

UNSW SYDNEY

seL4 What we've done & plan to do

- Defined state projections
- Verifying Hoare triples on calls

Read:
ks_of = "kernel state of"

Process A
{{ λs. P s }} seL4_Recv {{ λs. Q s }}

Process B
{{ λs. P' s }} seL4_Signal {{ λs. Q' s }}

seL4 Microkit Library

syscall (i.e. *kernel–userland*) interface

seL4 OS Microkernel Abstract Model

{{ λs. P (ks_of A s) }}
handle_recv
{{ λs. Q (ks_of A s) }}

"Nonblocking"
Blocking

{{ λs. P' (ks_of B s) }}
send_signal
{{ λs. Q' (ks_of B s) }}

Blocking
Unblocking

- Proving & generalising their composition

Note: If we can't prove these irrelevant for wellformed Microkit systems, then the seL4 kernel is incorrect!

{{ λs. P s }} → {{ λs. P (ks_of A s) }}

Blocking A

seL4_SomeCall {{ λs. R (ks_of A s) &&
P' (ks_of B s) }}

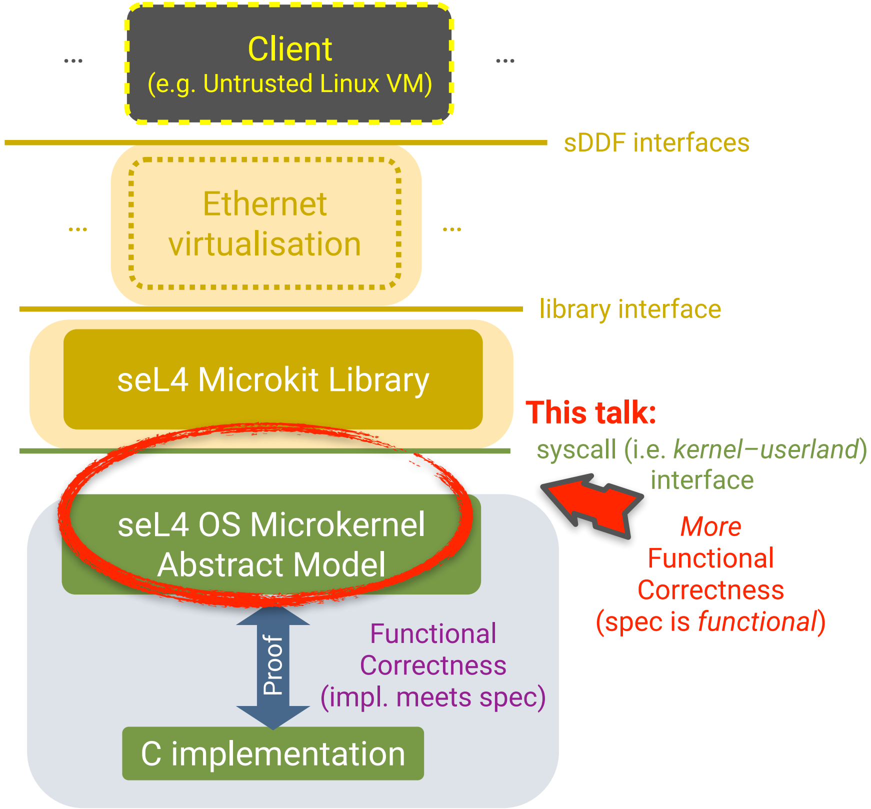
C, D, E … X, Y, Z
Irrelevant

{{ λs. R (ks_of A s) && P' (ks_of B s) }}

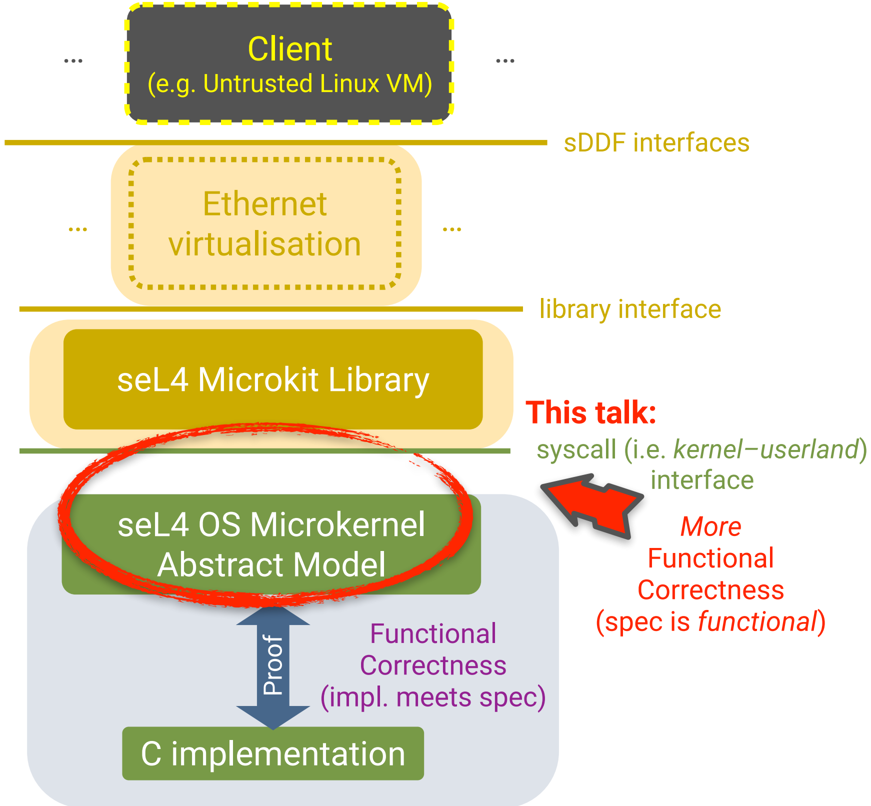
Unblocking B

{{ λs. Q s }} ← {{ λs. Q (ks_of A s) && Q' (ks_of B s) }}

Towards kernel–userland functional proofs

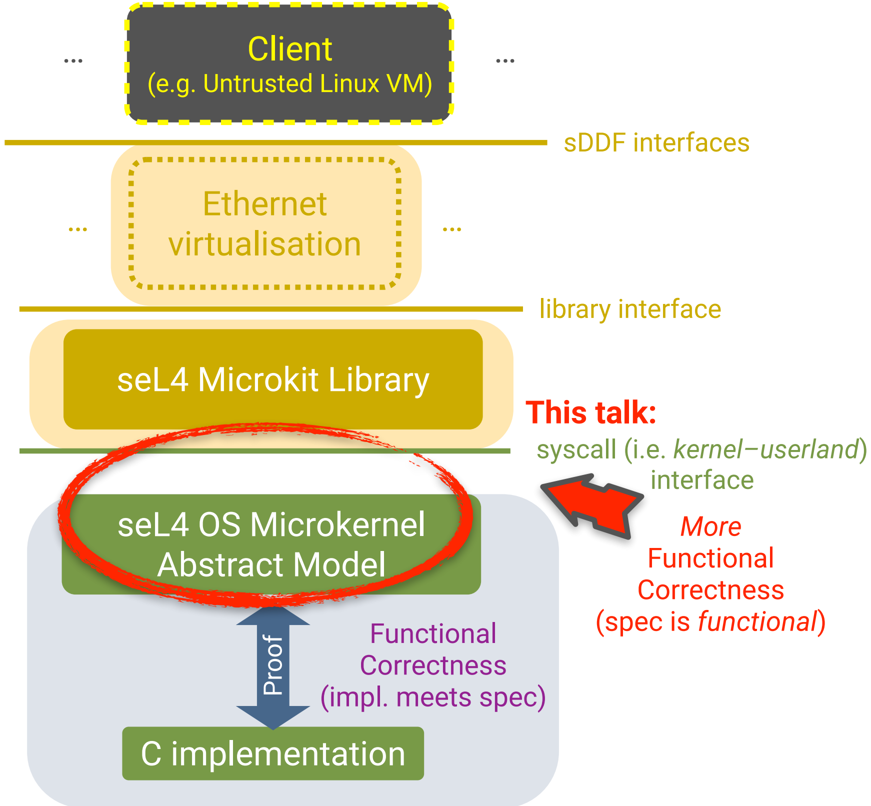


Towards kernel–userland functional proofs



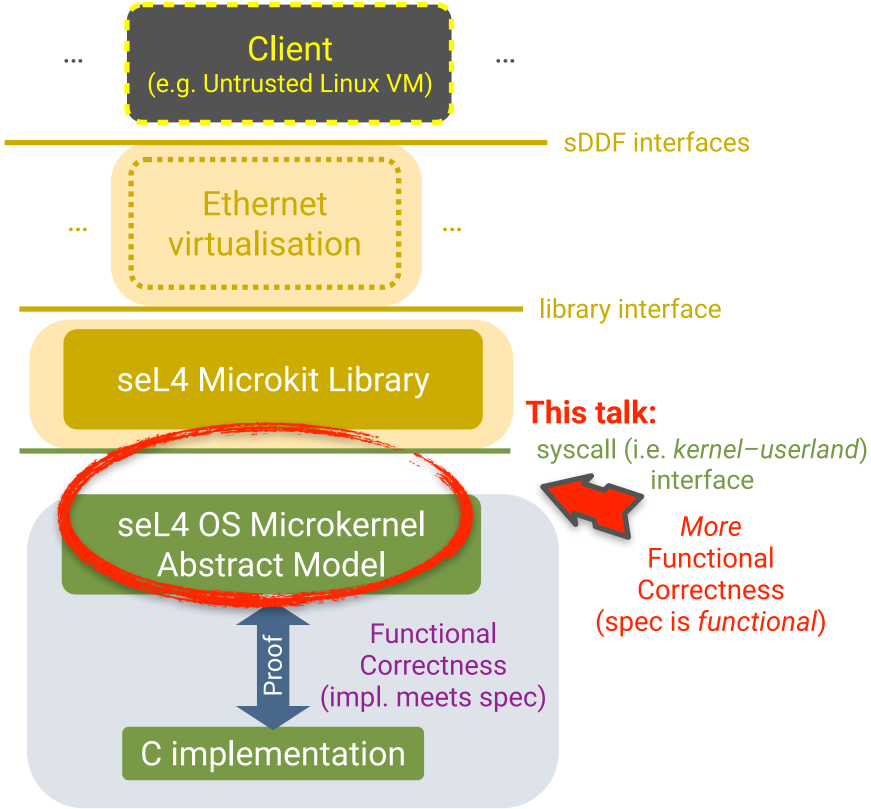
- Microkit assumes syscall functional specs as Hoare triples
- We need to prove seL4's Abstract Model satisfies them! (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)
- Blocking syscalls may require Hoare triples for up to $2 + n$ kernel entries (finite n):
  - 1 **Blocking** call (e.g. Recv)
  - n **Irrelevant** calls or interrupts
  - 1 **Unblocking** call (e.g. Signal)
- Verifying syscalls like these is **an open problem, <u>but</u>:**

Towards kernel–userland functional proofs



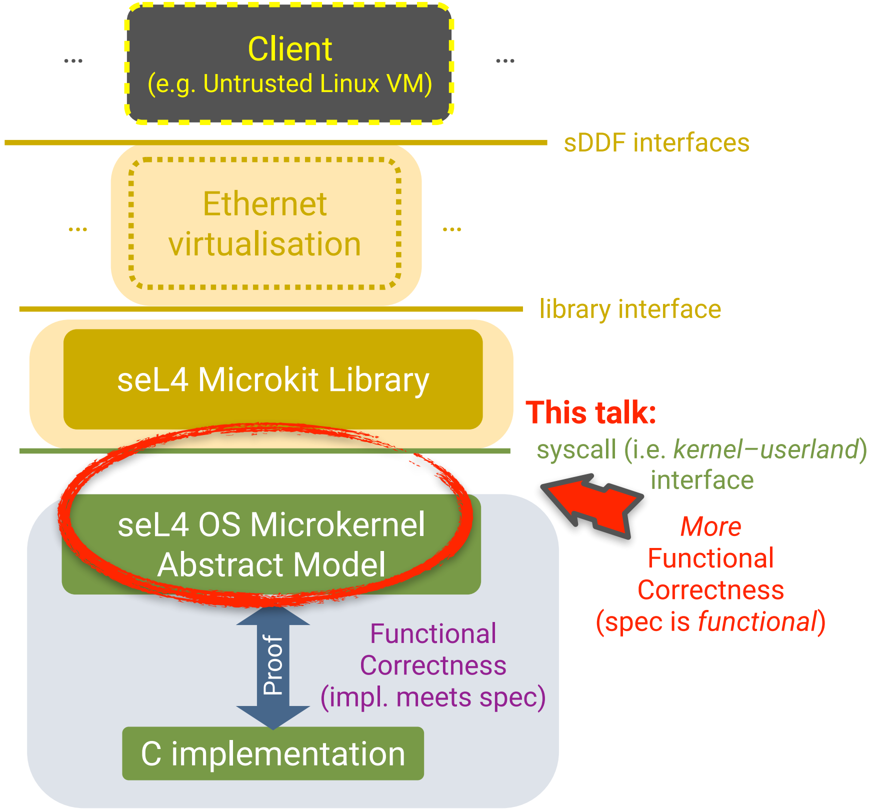
- Microkit assumes syscall functional specs as Hoare triples
- We need to prove seL4's Abstract Model satisfies them! (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)
- Blocking syscalls may require Hoare triples for up to 2 + *n* kernel entries (finite *n*):
  - 1 **Blocking** call (e.g. Recv)
  - *n* **Irrelevant** calls or interrupts
  - 1 **Unblocking** call (e.g. Signal)
- Verifying syscalls like these is **an open problem, <u>but</u>:**
  - We're verifying their constituent triples

Towards kernel–userland functional proofs



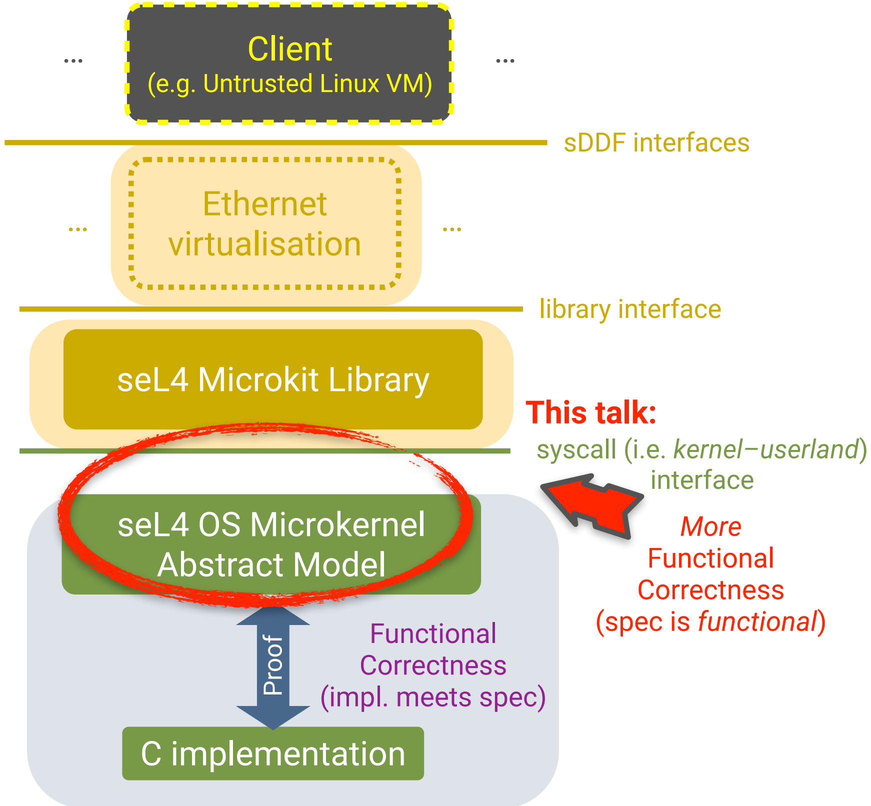
- Microkit assumes syscall functional specs as Hoare triples
- We need to prove seL4's Abstract Model satisfies them! (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)
- Blocking syscalls may require Hoare triples for up to 2 + *n* kernel entries (finite *n*):
  - 1 **Blocking** call (e.g. Recv)
  - *n* **Irrelevant** calls or interrupts
  - 1 **Unblocking** call (e.g. Signal)
- Verifying syscalls like these is **an open problem, <u>but</u>:**
  - We're verifying their constituent triples
  - We're generalising their composition

Towards kernel–userland functional proofs



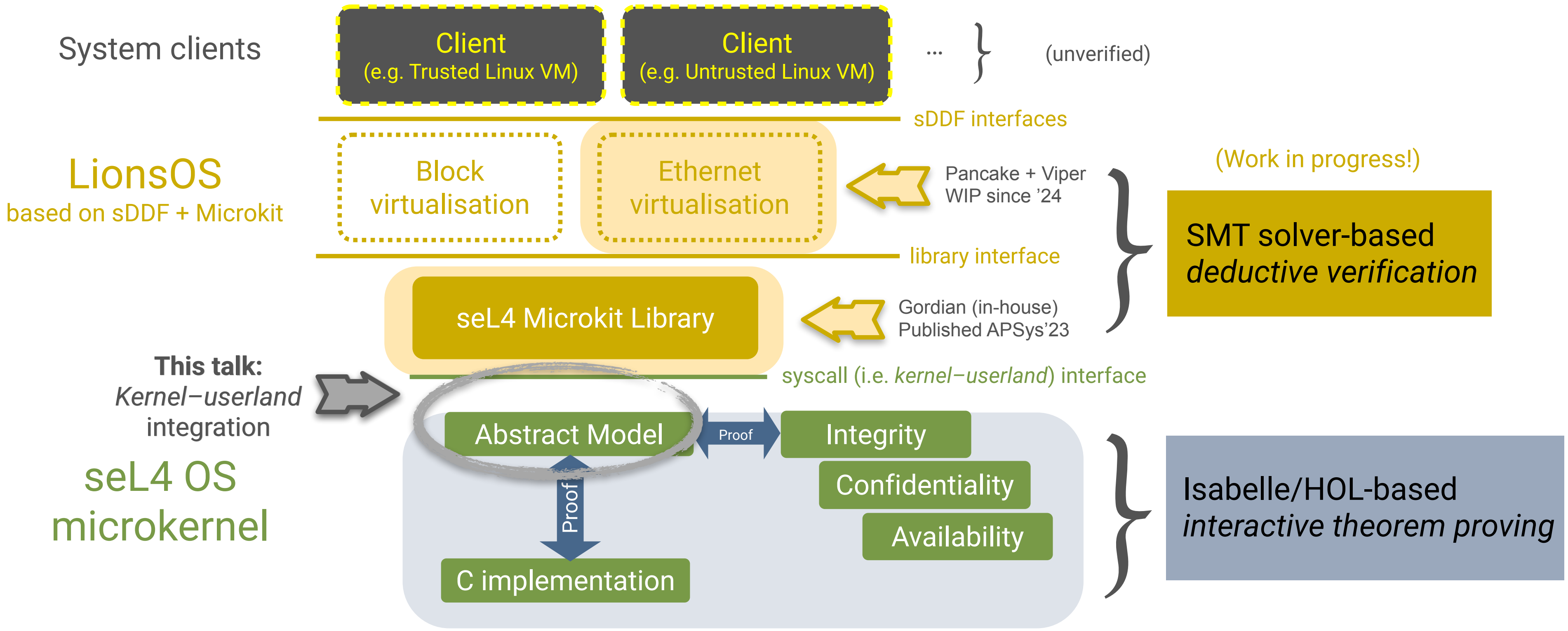
- Microkit assumes syscall functional specs as Hoare triples
- We need to prove seL4's Abstract Model satisfies them! (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)
- Blocking syscalls may require Hoare triples for up to $2 + n$ kernel entries (finite n):
  - 1 **Blocking** call (e.g. Recv)
  - n **Irrelevant** calls or interrupts
  - 1 **Unblocking** call (e.g. Signal)
- Verifying syscalls like these is **an open problem, <u>but</u>:**
  - We're verifying their constituent triples
  - We're generalising their composition
  - Microkit's strict requirements help us here

Towards kernel–userland functional proofs

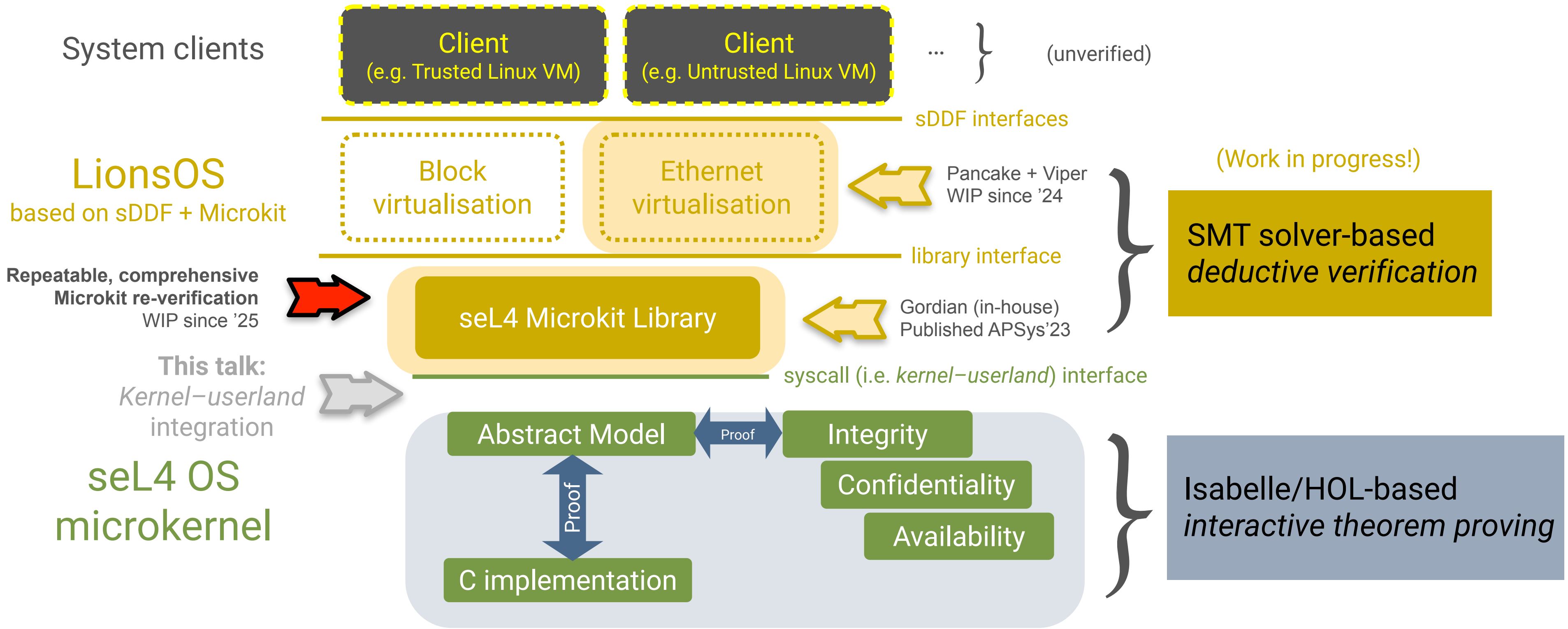


- Microkit assumes syscall functional specs as Hoare triples
- We need to prove seL4's Abstract Model satisfies them! (Then we'll know its C implementation does, too.)
- We handwave some register marshalling (for now)
- Blocking syscalls may require Hoare triples for up to 2 + *n* kernel entries (finite *n*):
  - 1 **Blocking** call (e.g. Recv)
  - *n* **Irrelevant** calls or interrupts
  - 1 **Unblocking** call (e.g. Signal)
- Verifying syscalls like these is **an open problem, <u>but</u>:**
  - We're verifying their constituent triples
  - We're generalising their composition
  - Microkit's strict requirements help us here

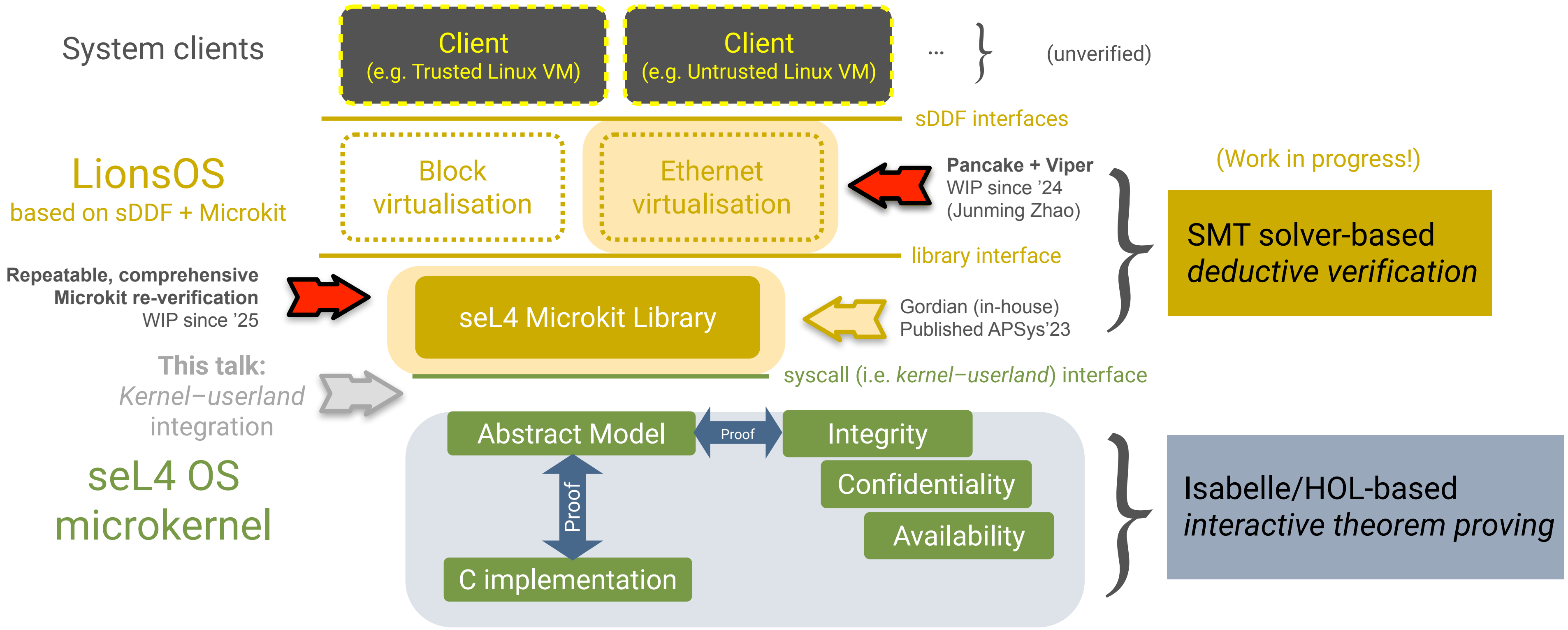
Adjacent work at Trustworthy Systems



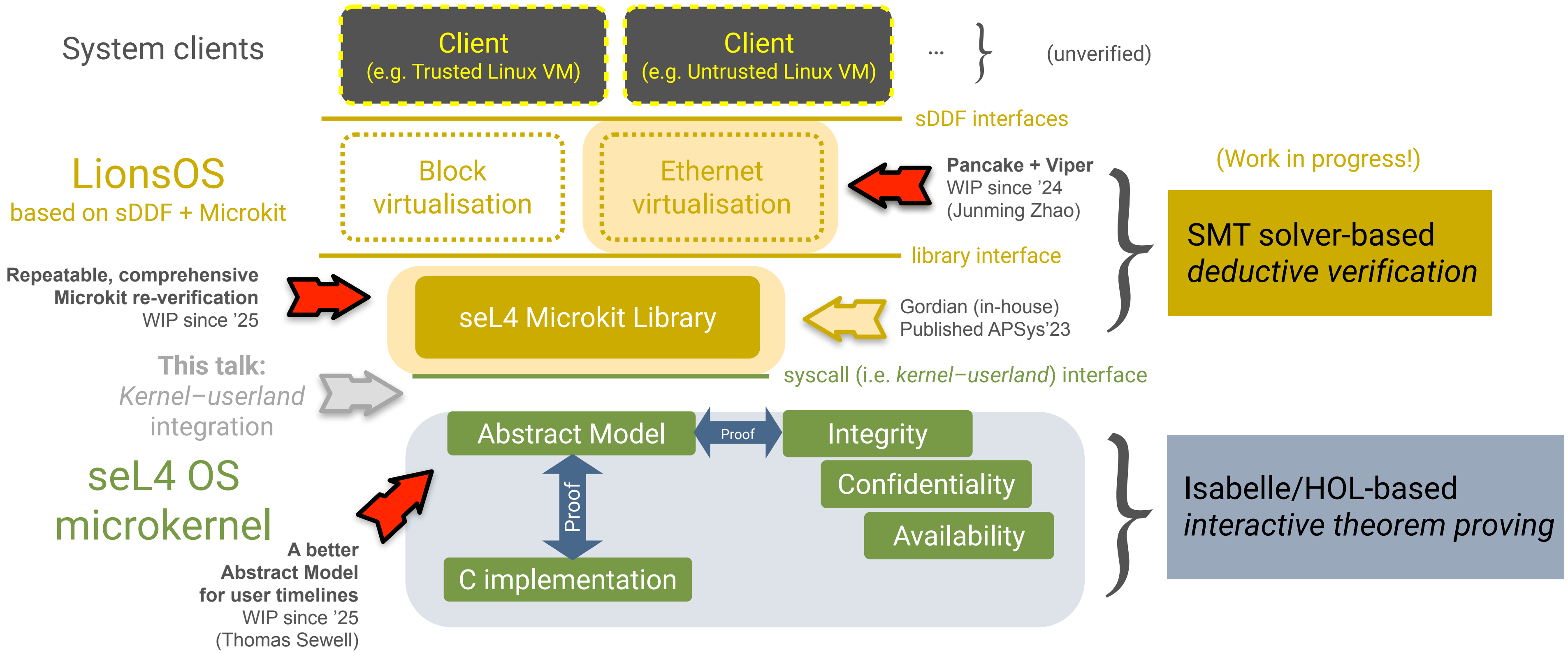
Adjacent work at Trustworthy Systems



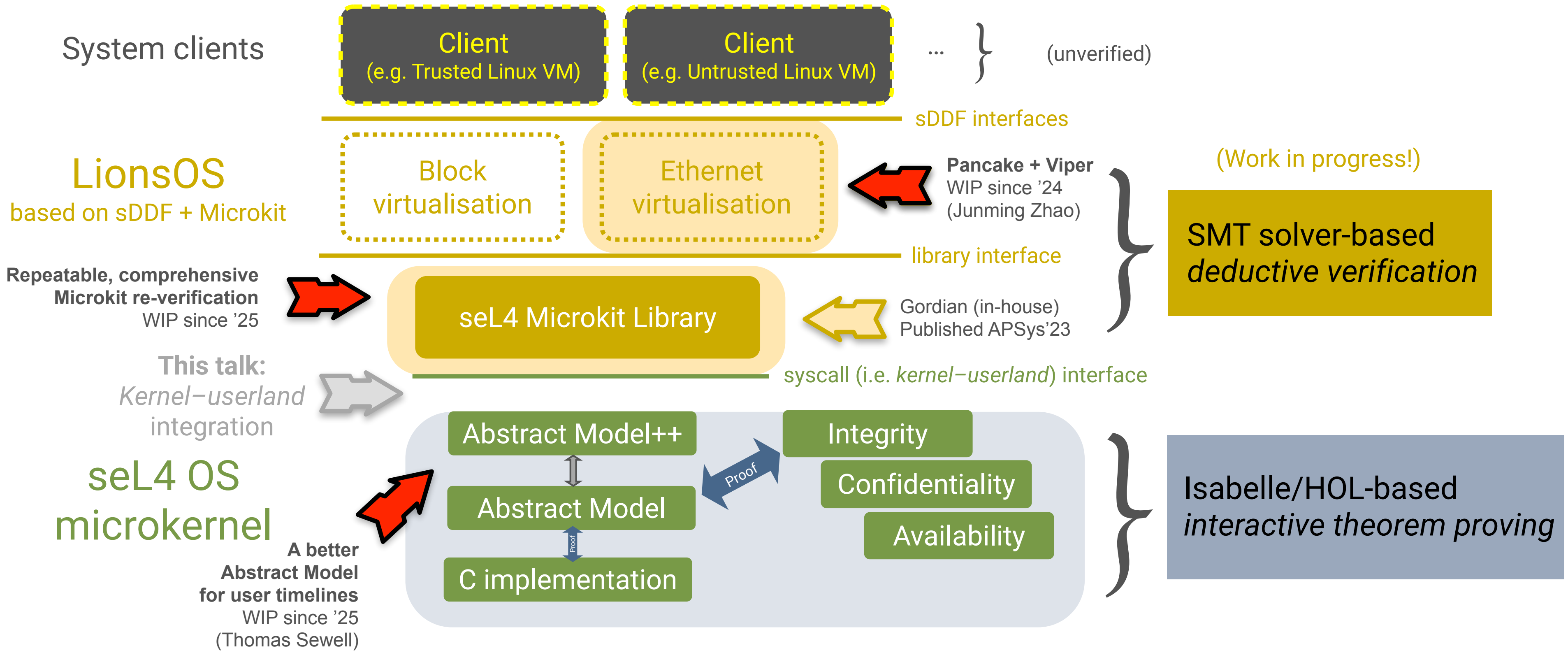
Adjacent work at Trustworthy Systems



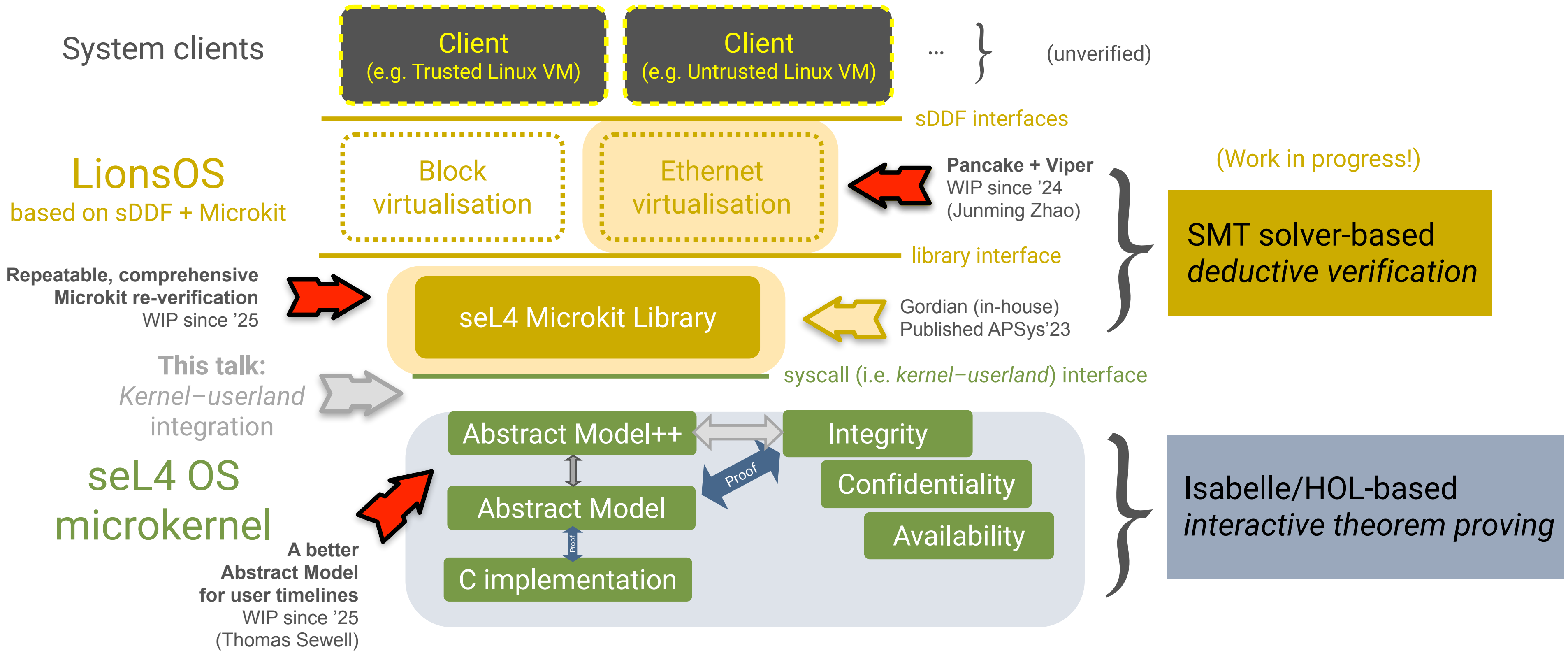
Adjacent work at Trustworthy Systems



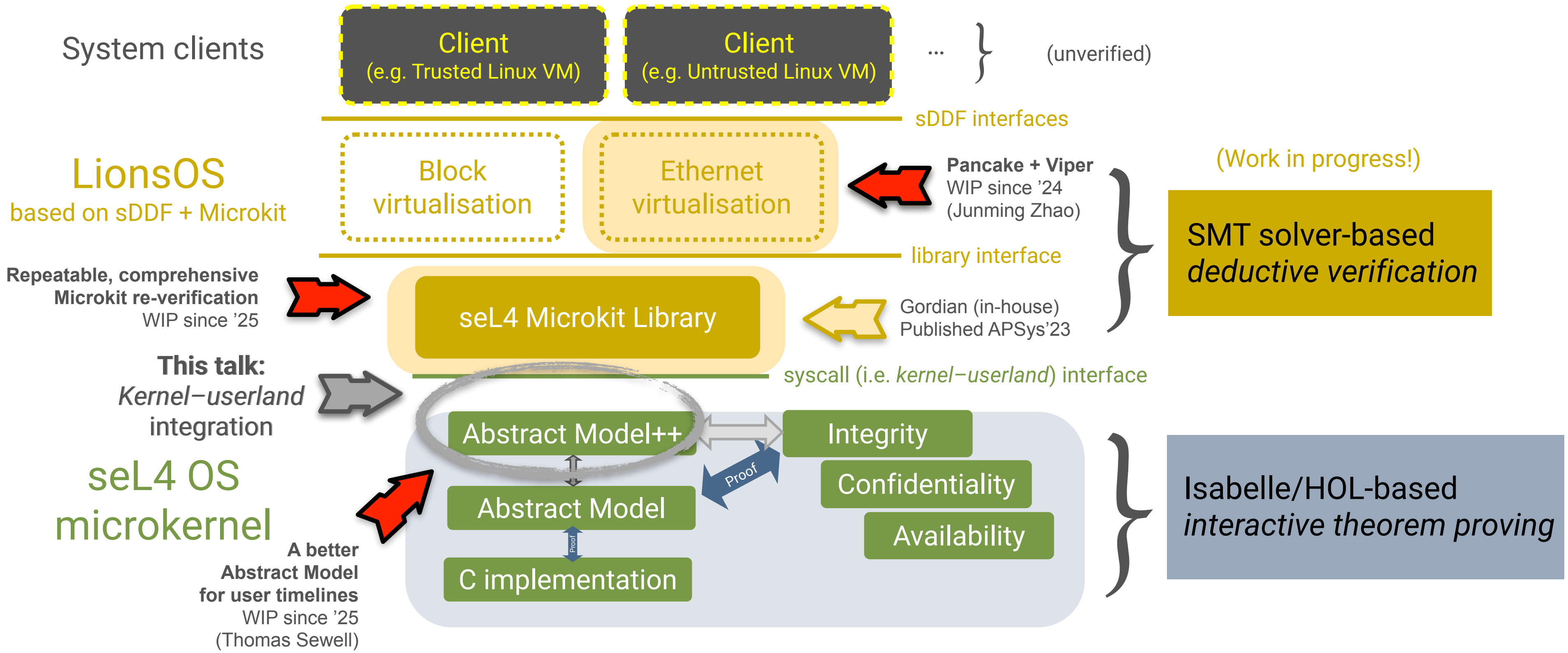
Adjacent work at Trustworthy Systems



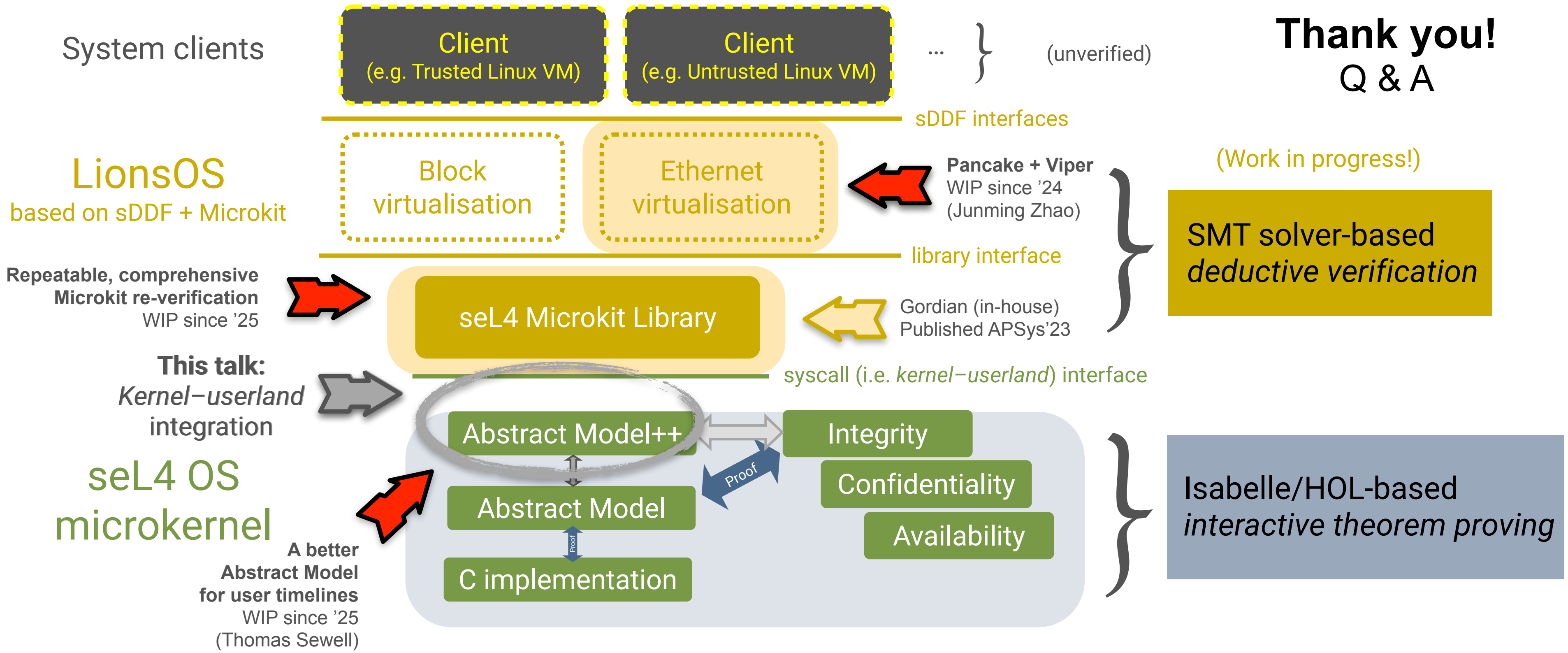
Adjacent work at Trustworthy Systems



Adjacent work at Trustworthy Systems



Adjacent work at Trustworthy Systems



  UNSW SYDNEY