seL4 Verification

Danil Tolstov

Group: M3302

Contents

Overview of seL4

Key Features of seL4 Microkernel

seL4: A minimalistic open source microkernel providing core OS services:

- Process and thread management
- Memory isolation via capabilities
- Secure inter-process communication (IPC)

Key Features:

- Strong isolation: Programs run in “sandboxes,” preventing interference.
- Optimized for resource-constrained devices.
- Supports ARM, x86, RISC-V architectures.
- Formally verified for functional correctness.

Applications in Critical Systems

Widely used in high-stakes systems:

1. Aviation (e.g., Boeing, DARPA projects)
2. Medical devices (e.g., pacemakers)
3. Cybersecurity (e.g., secure smartphones)
4. Autonomous vehicles
5. Military systems (e.g., DARPA HACMS)
6. Internet of Things (IoT)

Why seL4?

- Verified reliability: Guarantees no crashes or vulnerabilities.
- Ideal for life-critical and high-security applications.

Benefits of Formal Verification

Verification: Mathematical proof that the kernel adheres to its specification in all possible scenarios.

Why It Matters

- **Reliability**: Eliminates crashes, even in edge cases.
- **Security**: Prevents exploitable bugs.
- **Certification**: Meets standards like Common Criteria, DO-178C.
- **Uniqueness**: First fully verified general-purpose microkernel.

Common Criteria and **DO-178C** are standards for software security and safety certification.

Additional Details

Development

Developed by NICTA (Australia) starting in the 2000s. Completed in 2009, fully verified by 2014. [1]

Currently maintained by Data61, led by Gernot Heiser.

Code

Written in **C** (8,700 lines) and **assembly** (600 lines), totaling 9,300 lines verified using **Isabelle/HOL**.

License: GPLv2 for open source; commercial licenses available.

Source code available on GitHub

Isabelle/HOL: The Verification Powerhouse

What is Isabelle/HOL?

Isabelle/HOL: An interactive proof assistant using higher-order logic (HOL) for formal verification.

Isabelle based on ITP - Interactive Theorem Proving

Key Features

- **Interactive Proofs**: Combines human insight with machine-checked accuracy.
- **Higher-Order Logic**: Models complex systems like operating systems.
- **Versatile**: Verifies software, hardware, and protocols.

How It Works

- You define a program and its desired properties (e.g., “no errors”).
- Write **proof scripts** to show the properties hold.
- Isabelle checks each step, ensuring correctness.

Example 1: Proving a Sum Function

Goal

Prove a function `sum_upto(n)` computes `1 + 2 + ... + n` correctly.

```
fun sum_upto :: "nat ⇒ nat" where
  "sum_upto 0 = 0"
| "sum_upto n = n + sum_upto (n - 1)"

theorem sum_correct:                # Define theorem to prove
  "sum_upto n = (n * (n + 1)) div 2"
proof (induction n)                 # Proving by induction on variable n
  case 0 show ?case by simp         # Prove case 0 by substitution
  case (Suc n) thus ?case by simp   # Prove case n + 1 by substitution
qed
```

in *case* clauses substitutes arg to formula from definition

Example 1: Proving a Sum Function

Explanation

- Defines `sum_upto` recursively.
- Proves it equals the formula `n * (n + 1) / 2` using induction.
- Isabelle checks each case (base and step).

base

```
sum_upto 0 = (0 * (0 + 1)) div 2
0 * (0 + 1) div 2 = 0
```

---

step

```
sum_upto (Suc n) = ((Suc n) * (Suc n + 1)) div 2
Suc n + (n * (n + 1)) div 2 = ((Suc n) * (Suc n + 1)) div 2
```

Example 2: Proving List Reversal

Goal

Prove reversing a list preserves its length.

```
fun reverse :: "'a list ⇒ 'a list" where
  "reverse [] = []"
| "reverse (x # xs) = reverse xs @ [x]"   # [x] + [y...z] = [z...y] + [x]

theorem reverse_length:
  "length (reverse xs) = length xs"
  proof (induction xs)
    case Nil show ?case by simp             # case Nil when list is empty
    case (Cons x xs) thus ?case by simp   # case [x] + xs
  qed
```

Example 2: Proving List Reversal

Explanation

- Defines `reverse` recursively: empty list stays empty; otherwise, move head to end.
- Proves `length (reverse xs) = length xs` using induction.
- Isabelle verifies base case (empty list) and step case (non-empty list).

This guarantees `reverse` doesn't add or lose elements.

base

```
length(reverse[]) = length[]
length [] = length []
```

step

```
length(reverse(x # xs)) = length(x # xs)
length(reverse xs @ [x]) = length(xs) + 1
length(reverse(xs)) + length([x]) = length(xs) + 1
```

Examples outcome

Lessons Learned

- **Modeling**: Programs are defined as mathematical functions.
- **Proofs**: Use induction or other logic to cover all cases.
- **Verification**: Isabelle ensures no mistakes in reasoning.

In seL4 Context

- Similar techniques scaled to prove complex properties (e.g., memory safety).
- Simple proofs build confidence for larger systems.

Why Isabelle is chosen?

As mentioned before, **Isabelle/HOL** uses ITP, which is perfect for a such large project as seL4 (10k LOC). Other methods of verification would take enormous amount of time and resources.

Verifying seL4: Process and Insights

What is Formal Verification?

Formal Verification: Mathematically proving a system behaves correctly for **all** inputs and scenarios.

Why It Matters

- Ensures **bug-free** behavior, unlike testing (limited cases).
- Critical for seL4's use in safety-critical systems (e.g., pacemakers).

Refinement technic

Refinement Layers

Approach

- Used **Isabelle/HOL** for machine-checked proofs.
- Proved **functional correctness** via refinement:
  - Abstract spec $\longrightarrow$ Executable spec $\longrightarrow$ C code.
- Prototyped in **Haskell** for design clarity.

Refinement Layers

- **Abstract**: Defines **what** (e.g., system calls).
- **Executable**: Details **how** (Haskell-derived).
- **Concrete**: C code (8,700 LOC).

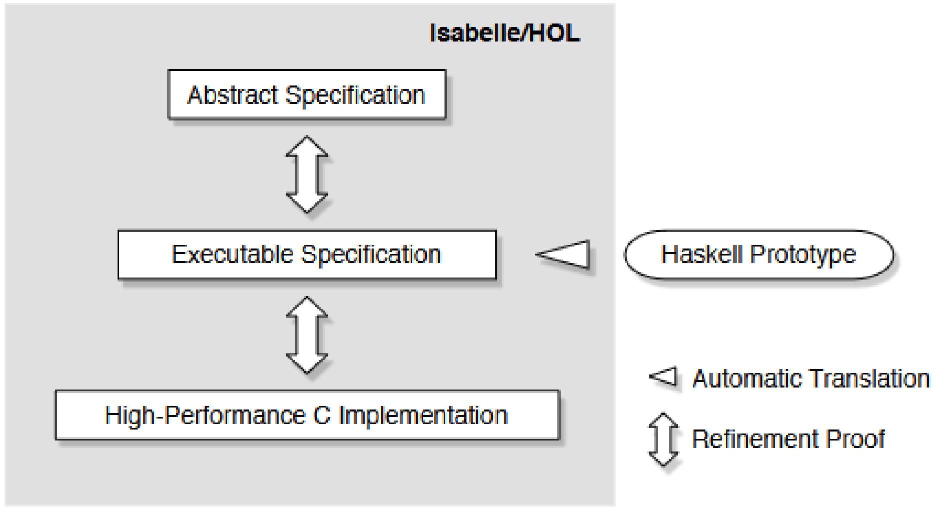


Figure 1: The refinement layers

Example: Abstract Scheduler

Scheduler Specification

The abstract specification defines **what** the scheduler does, not **how**.

```
schedule = do
  threads = all_active_tcbs;
  thread = select threads;
  switch_to_thread thread
od OR switch_to_idle_thread
```

- **Rarely uses infinite type** (like *nat*). Using 32-bit integers instead
- **Abstract types** (sets, lists, trees, etc.)
- **Often non-deterministic**: if there's several correct outputs fot an operation, this layer would return them all and make clear that there is a choice

Example: Executable Scheduler

Detailed Scheduler

The executable specification adds **how** the scheduler works.

```
schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread
chooseThread = do
  r <- findM chooseThread' (reverse [minBound .. maxBound])
  ...
```

- **Become deterministic**
- **Concrete data types** (e.g. tree is now a double-linked list)
- **Some fundamental restrictions** (e.g. using 64bit to represent capabilities)

Example: C Scheduler

C Implementation

The final C code optimizes performance while matching the spec.

Its the most detailed layer. Written in a very large, pragmatic subset of C99

What is prohibited

- ***adress of*** operator (&) on local variables - because of an assumption that local variables are separate from stack. Using pointers to global variables instead
- ***function call through function pointers***
- ***goto***
- ***switch*** with fall-thgrouh cases
- ***unions***

Example: C Scheduler Code

```c
void setPriority(tcb_t *tptr, prio_t prio) {
    prio_t oldprio;
    if (thread_state_get_tcbQueued(tptr->tcbState)) {
        ksReadyQueues[oldprio] =
            tcbSchedDequeue(tptr, ksReadyQueues[oldprio]);
        if (isRunnable(tptr)) {
            ksReadyQueues[prio] =
                tcbSchedEnqueue(tptr, ksReadyQueues[prio]);
        } else {
            thread_state_ptr_set_tcbQueued(&tptr->tcbState, false);
        }
    }
    tptr->tcbPriority = prio;
}
```

The proof

A refinement proof establishes a correspondence between a high-level (abstract) and a low-level (concrete, or refined) representation of a system.

The correspondence established by the refinement proof ensures that all Hoare logic properties of the abstract model **also hold for the refined model**.

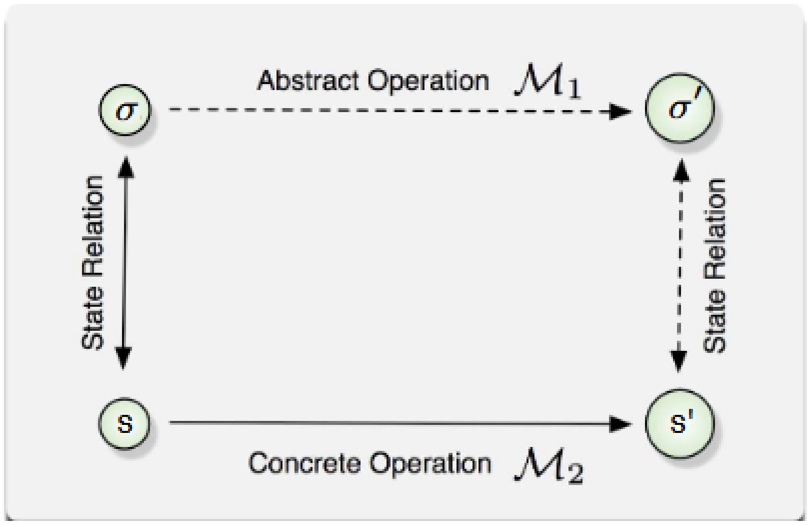

Abstract Operation $\mathcal{M}_1$
σ
σ'
State Relation
State Relation
s
s'
Concrete Operation $\mathcal{M}_2$

Figure 2: Forward simulation

Forward Simulation

To show that a concrete state machine M2 refines an abstract one M1, it is sufficient to show that for each transition in M2 that may lead from an initial state s to a set of states s′, there exists a corresponding transition on the abstract side from an abstract state σ to a set σ′ (they are sets because the machines may be non-deterministic)

The transitions ***correspond*** if there exists a relation R between the states s and σ such that for each concrete state in s′ there is an abstract one in σ′ that makes R hold between them again

Assurance

Overall, we show that the behaviour of the C implementation is fully captured by the abstract specification. This is a strong statement, as **it allows us to conduct all further analysis of properties that can be expressed as Hoare triples on the massively simpler abstract specification instead of a complex C program**.

Coverage is complete. Any remaining implementation errors (deviations from the specification) can **only** occur **below the level of C**

Invariants

Invariants

As part of the refinement proof between layers, were introduced (and prooved) a large number of invariants (more than 150)

These invariants are not a proof device, but provide valuable information and assurance. In essence, they collect information about **what we know to be true** of each data structure in the kernel, before and after each system call, and also for large parts during kernel execution where some of these invariants **may be temporarily violated and reestablished later**

The overall proof effort was clearly **dominated by invariant proofs**, with the actual refinement statements between abstract and executable specification accounting for at most 20 % of the total effort for that stage

There are **four main categories** of invariants in our proof: **low-level memory** invariants, **typing** invariants, **data structure** invariants, and **algorithmic** invariants.

Invariants: low-level memory & typing

low-level

low-level memory invariants include that:

- there is no object at address 0
- kernel objects are aligned to their size, and that they do not overlap

typing

The typing invariants say that each kernel object has a well-defined type and that its references in turn point to objects of the right type

Example: An example would be a capability slot containing a reference to a thread control block (TCB). Intuitively, this invariant implies that ***all reachable, potentially used references in the kernel—be it in capabilities, kernel objects or other data structures —always point to an object of the expected type***.

Invariants: Data structures & Algorithmic

Data structures

These are classical data structure invariants, such that correct back links in doubly-linked lists, a statement that there are no loops in specific pointer structures, that other lists are always terminated correctly with NULL, etc.

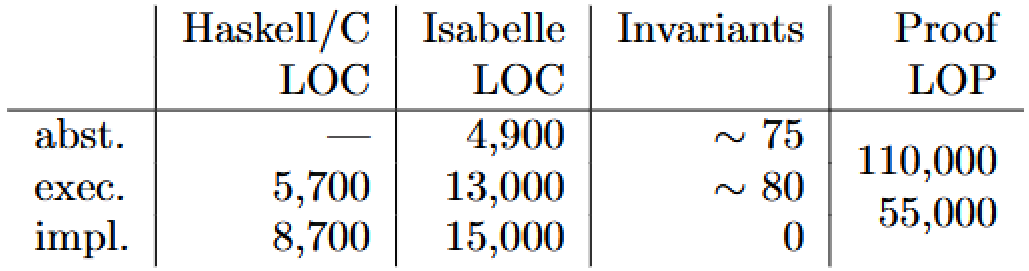
algorithmic

These are the most complex invariants in our proof and they are where most of the proof effort was spent.

These invariants are either required to prove that specific optimisations are allowed (e.g. that a check can be left out because the condition can be shown to be always true), or they are required to show that an operation executes safely and does not violate other invariants, especially not the typing invariant.

Conclusion

Effort and Outcomes



	Haskell/C LOC	Isabelle LOC	Invariants	Proof LOP
abst.	—	4,900	~ 75	110,000
exec.	5,700	13,000	~ 80	55,000
impl.	8,700	15,000	0	

Figure 3: Code and proof statistics

The overall size of the proof, including framework, libraries, and generated proofs (not shown in the table) is **200,000 lines of Isabelle script**.

The total effort for the seL4-specific proof was 11 py.

References

[1] G. Klein *et al.*, “SeL4: Formal verification of an OS kernel,” 2009, pp. 207–220. doi: 10.1145/1629575.1629596.