seL4 Verification

Danil Tolstov

Group: M3302

Contents

1. Overview of seL4
2. Isabelle/HOL: The Verification Powerhouse
3. Verifying seL4: Process and Insights
4. Example: Scheduler
5. Conclusion

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.

Development

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

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.

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

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
```

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
```

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
```

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).

Example

- Proved seL4's memory isolation prevents unauthorized access.

Approach

- Used **Isabelle/HOL** for machine-checked proofs.
- Proved **functional correctness** via refinement:
  - Abstract spec ⟶ Executable spec ⟶ 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).

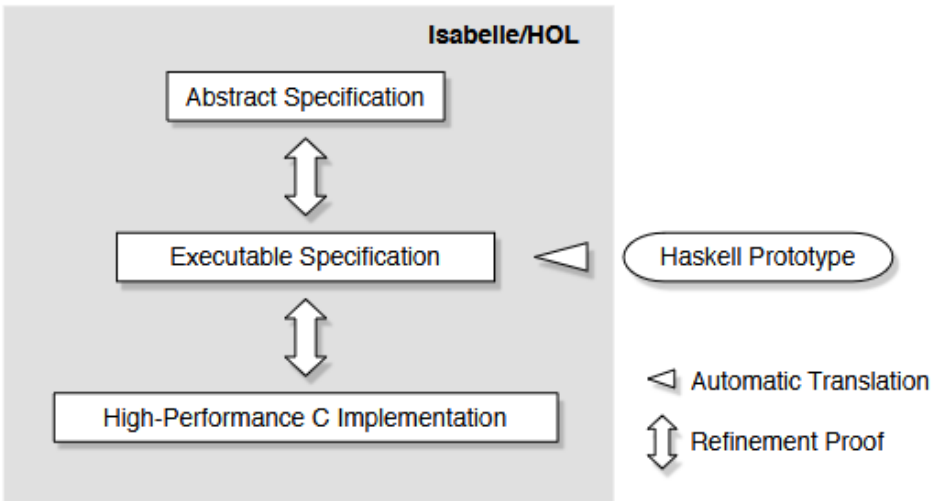

Figure 1: The refinement layers

Correspondence

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**. This means that if a security property is proved in Hoare logic about the abstract model (not all security properties can be), refinement guarantees that the same property holds for the kernel source code.

Example: Capability Derivation Tree

Concept

- Tracks memory allocation via capabilities to prevent dangling references.
- Implemented as a doubly-linked list for simplicity.

Isabelle/HOL Proof

```
invariant cdt_list s = (∀c. cdt s c ≠ None →
  ∃p. cdt s p = Some c ∧ valid_mdb s)
```

- Ensures every capability has a valid parent, preserving memory safety.

This invariant guarantees safe memory management.

Challenge

Concurrency (e.g., interrupts) complicates proofs.

seL4's Approach

- **Event-Based**: Single kernel stack, mostly non-preemptable.
- **Interrupt Polling**: Controlled preemption points.
- **Zombie Capabilities**: Store state for safe object destruction.

Example

- Interrupt retries ensure invariants (e.g., queue integrity) are restored.

Simplified concurrency kept proofs manageable.

Key Issues

- **Global Variables**: Proved invariants (e.g., list integrity) across all code.
- **C Semantics**: Defined a C99 subset, avoiding complex features (e.g., function pointers).

Example

- Proved `ksReadyQueues` updates in `setPriority` maintain list consistency.

Formal C semantics were critical for accurate proofs.

Verification Challenges: Invariants

Complexity

- Proved over 150 invariants (e.g., type safety, no null pointers).
- Invariants often interdependent, requiring careful proof ordering.

Example

- **Reply Capability Issue**: Adding reply capabilities broke existing invariants, needing 1 person-year to fix.

Invariants took 80% of proof effort!

Example: 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
```

Explanation

- **Non-deterministic**: Picks any runnable thread or idle thread.
- Leaves implementation details (e.g., priority) to lower layers.

This simplicity makes proving correctness easier!

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])
  ...
```

Key Points

- Uses priority-based round-robin scheduling.
- Explicit data structures (e.g., priority queues).

This bridges the abstract spec to the C implementation.

C Implementation

The final C code optimizes performance while matching the spec.

Why It Matters

- Verified to match the executable spec.
- Handles thread priority updates safely.

Example: C Scheduler Code


```
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;
}
```

Conclusion

Key Takeaways

- **Haskell Prototype**: Reduced development costs and improved design.
- **Verification-Driven Design**: Led to simpler, more robust code.
- **Cost of Change**: Local changes are cheap; cross-cutting changes (e.g., reply capabilities) are expensive.
- **Bug Detection**: Found subtle bugs (e.g., missing input checks) missed by testing.

Future Work

- Verify assembly code and boot process.
- Extend to multi-core systems.
- Support application-level verification.

Effort

- **Total**: 20 person-years (11 for seL4-specific proofs).
- **Proof Size**: 200,000 lines of Isabelle/HOL scripts.
- **Bugs Found**: 144 in C code, fixed during verification.

Achievements

- **Correctness**: C code matches abstract spec.
- **Safety**: No crashes or null pointer errors.
- **Security**: Access control verified.
- **Performance**: IPC at 224 cycles, matching optimized kernels.

Verification cost (8 person-years for a new kernel) rivals traditional development.

Impact of Isabelle/HOL in seL4

Contributions

- Enabled first fully verified general-purpose microkernel.
- Caught subtle bugs (e.g., incorrect interrupt checks).
- Set a new standard for OS trustworthiness.

Broader Impact

- Supports critical systems (e.g., aviation, medical).
- Inspires verification in other domains.

Isabelle/HOL made seL4 a gold standard for security.

Summary

- **seL4**: First fully verified general-purpose microkernel, ensuring unmatched reliability and security.
- **Verification**: Proved functional correctness using Isabelle/HOL, covering 9,300 LOC.
- **Impact**: Enables trusted systems in aviation, medical, military, and more.
- **Legacy**: Redefines standards for OS assurance, with practical development costs.

Visit seL4.systems for more information.