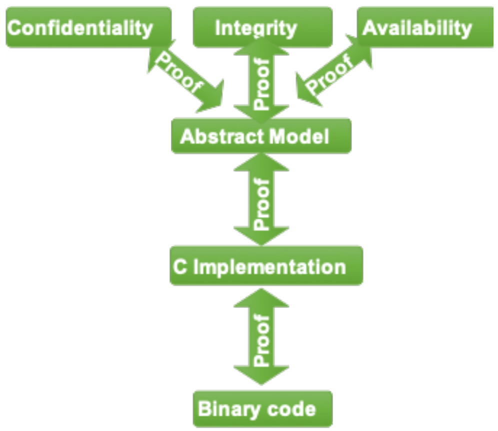
School of Computer Science & Engineering

COMP9242 Advanced Operating Systems



2019 T2 Week 09a

Formal Verification and seL4

@GernotHeiser

Assurance and Verification

Refresher: Assurance and Formal Verification

- **Assurance**:
  - systematic evaluation and testing
  - essentially an intensive and onerous form of quality assurance
- **Formal verification**:
  - mathematical proof
- **Certification**: independent examination
  - confirming that the assurance or verification was done right

Assurance and formal verification aim to establish correctness of
- mechanism design
- mechanism implementation

Assurance: Substantiating Trust

- Specification
  - Unambiguous description of desired behaviour
- System design
  - Justification that it meets specification
- Implementation
  - Justification that it implements the design
- Maintenance
  - Justifies that system use meets assumptions

Informal (English) or formal (maths)

Compelling argument or formal proof

Code inspection, rigorous testing, proof

Common Criteria

- *Common Criteria for IT Security Evaluation* [ISO/IEC 15408, 99]
  - ISO standard, for general use
  - Evaluates QA used to ensure systems meet their requirements
  - Developed out of the famous US DOD “Orange Book”: *Trusted Computer System Evaluation Criteria* [1985]
- Terminology:
  - *Target of evaluation* (TOE): Evaluated system
  - *Security target* (ST): Defines requirements
  - *Protection profile* (PP): Standardised ST template
  - *Evaluation assurance level* (EAL): Defines thoroughness of evaluation
    - PPs have maximum EAL they can be used for

CC: Evaluation Assurance Levels

Thoroughness, cost ↓

Level	Requirements	Specification	Design	Implementation
EAL1	not evaluated	**Informal**	not eval	not evaluated
EAL2	not evaluated	**Informal**	**Informal**	not evaluated
EAL3	not evaluated	**Informal**	**Informal**	not evaluated
EAL4	not evaluated	**Informal**	**Informal**	not evaluated
EAL5	not evaluated	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL6	**Formal**	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL7	**Formal**	**Formal**	**Formal**	**Informal**

Common Criteria: Protection Profiles (PPs)

- *Controlled Access PP* (CAPP)
    - standard OS security, up to EAL3
- *Single Level Operating System PP*
    - superset of CAPP, up to EAL4+
- *Labelled Security PP* (LSPP)
    - MAC for COTS OSes
- *Multi-Level Operating System PP*
    - superset of CAPP, LSPP, up to EAL4+
- *Separation Kernel Protection Profile* (SKPP)
    - strict partitioning, for EAL6-7

COTS OS Certifications

- EAL3:
  - 2010 Mac OS X (10.6)
- EAL4:
  - 2003: Windows 2000
  - 2005: SuSE Enterprise Linux
  - 2006: Solaris 10 (EAL4+)
    - against CAPP (an EAL3 PP!)
  - 2007: Red Hat Linux (EAL4+)
- EAL6:
  - 2008: Green Hills INTEGRITY-178B (EAL6+)
    - against SKPP, relatively simple PPC-based hardware platform in TOE
- EAL7:
  - 2019: Prove & Run PROVENCORE

Get regularly hacked!

SKPP on Commodity Hardware

- SKPP: OS provides only separation
- One Box One Wire (OB1) Project
  - Use INTEGRITY-178B to isolate VMs on commodity desktop hardware
  - Leverage existing INTEGRITY certification
    - by “porting” it to commodity platform

NSA subsequently dis-endorsed SKPP, discontinued certifying ≥EAL5

Conclusion [NSA, March 2010]:
- SKPP validation for commodity hardware platforms infeasible due to their complexity
- SKPP has limited relevance for these platforms

Common Criteria Limitations

Effectively dead in 5-Eyes defence

- Very expensive
    - rule of thumb: EAL6+ costs $1K/LOC design-implementation-evaluation-certification
- Too much focus on development process
    - rather than the product that was delivered
- Lower EALs of little practical use for OSes
    - c.f. COTS OS EAL4 certifications
- Commercial Licensed Evaluation Facilities licenses rarely revoked
    - Leads to potential "race to the bottom" [Anderson & Fuloria, 2009]

Formal Verification

- Prove properties about a mathematical model of a system

Model checking / abstract interpretation:
- Cannot generally prove code correct
  - Proves specific properties
  - Has false positives or false negatives (unsoundness)
- Suffers state-space explosion
- May scale to large code bases

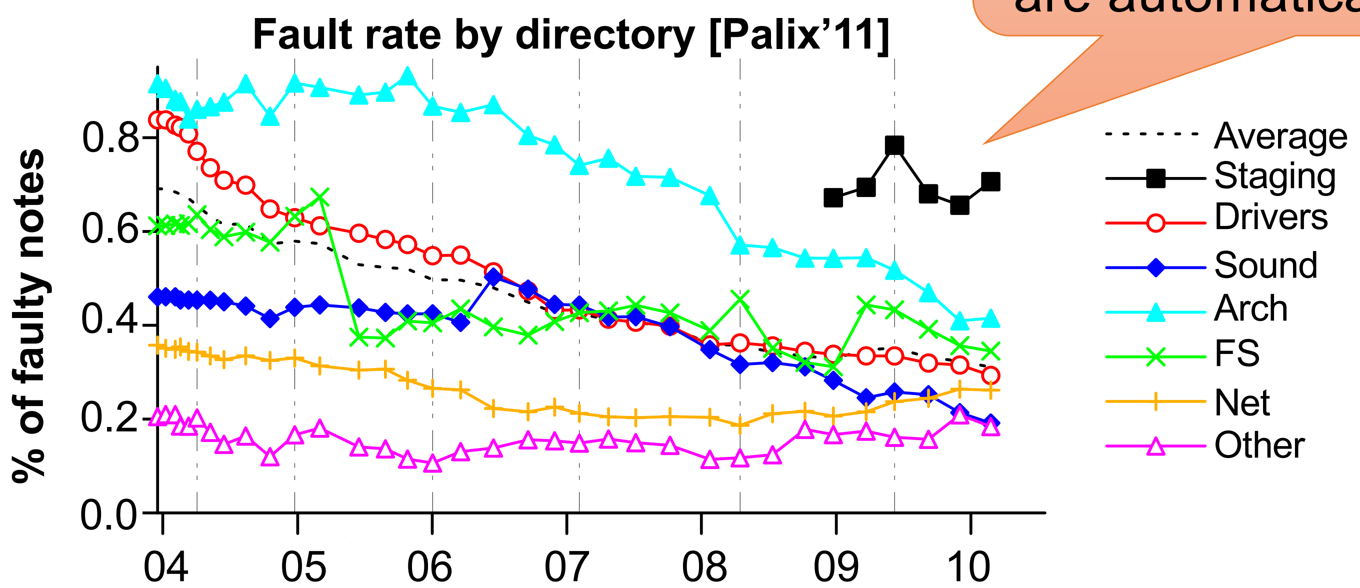
Recent work automatically proved functional correctness of simple systems using SMT solvers [Hyperkernel, SOSP'17]

Theorem proving:
- Can deal with large (even infinite) state spaces
- Can prove functional correctness against a spec
- Very labour-intensive

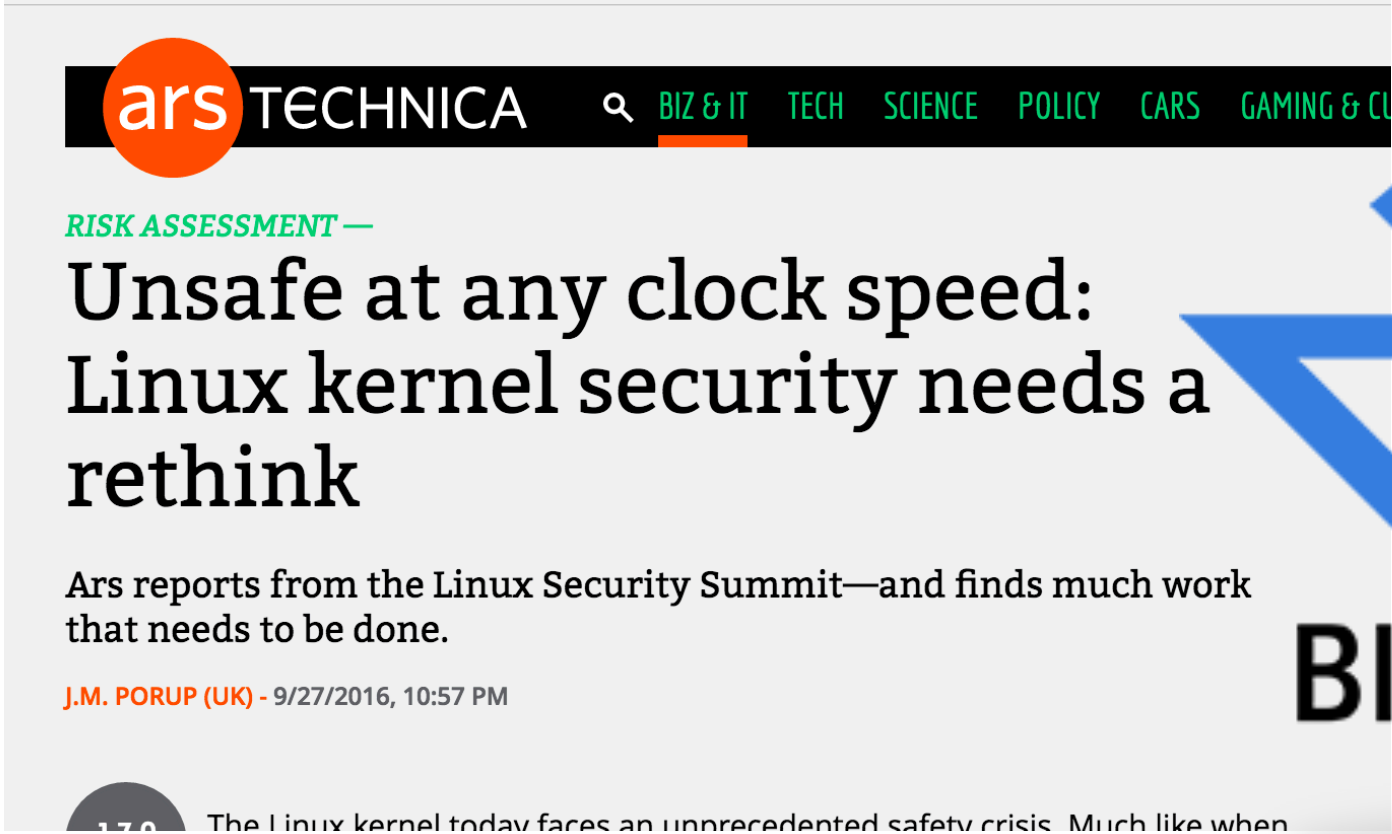
Model Checking and Linux: A Sad Story

- Static analysis of Linux source [Chou & al, 2001]
  - Found high density of bugs, especially in device drivers
- Re-analysis 10 years later [Palix & al, 2011]

Disappointing rate of improvement for bugs that are automatically detectable!



And the Result?


ars TECHNICA BIZ & IT TECH SCIENCE POLICY CARS GAMING & CU

RISK ASSESSMENT —

Unsafe at any clock speed: Linux kernel security needs a rethink

Ars reports from the Linux Security Summit—and finds much work that needs to be done.

J.M. PORUP (UK) - 9/27/2016, 10:57 PM

BlueBorne

August 2009

A NICTA bejelentette a világ első, formális módszerekkel igazolt,

Slashdot

Stories Recent Popular Searc

Slashdot is powered by **your subm**

Technology: World's Firs

Posted by Soulskill on Thursday Aug
from the wait-for-it dept.

An anonymous reader writes

> "Operating systems usually have
> and so forth are known by almos
> to prove that a particular OS ker
> formally verified, and as such it
> researchers used an executable
> the Isabelle theorem prover to ge
> matches the executable and the

Does it run Linux? "We're pleased to say that it does. Presently, we have a para-virtualized ver

eredményeképpen pedig egy olyan megbízhatóságot kapnak a szoftvertől, amely e

New Scientist
Saturday 29/8/2009
Page: 21
Section: General News
Region: National
Type: Magazines Science / Technology
Size: 196.31 sq.cms.
Published: -----S-

The ultimate way to keep your computer safe from harm

FLAWS in the code, or "kernel", that sits at the heart of modern computers leave them prone to occasional malfunction and vulnerable to attack by worms and viruses. So the development of a secure general-purpose microkernel could pave the

just mathematics, and you can reason about them mathematically," says Klein.

His team formulated a model with more than 200,000 logical steps which allowed them to prove that the program would always behave as its

UNSW SYDNEY

MIT Technology Review

LISTS | INNOVATORS UNDER 35 | DISRUPTIVE COMPANIES | BREAKTHROUGH TECHNOLOGIES

MIT Technology Review

10 BREAKTHROUGH TECHNOLOGIES 2011

Share

Crash-Proof Code

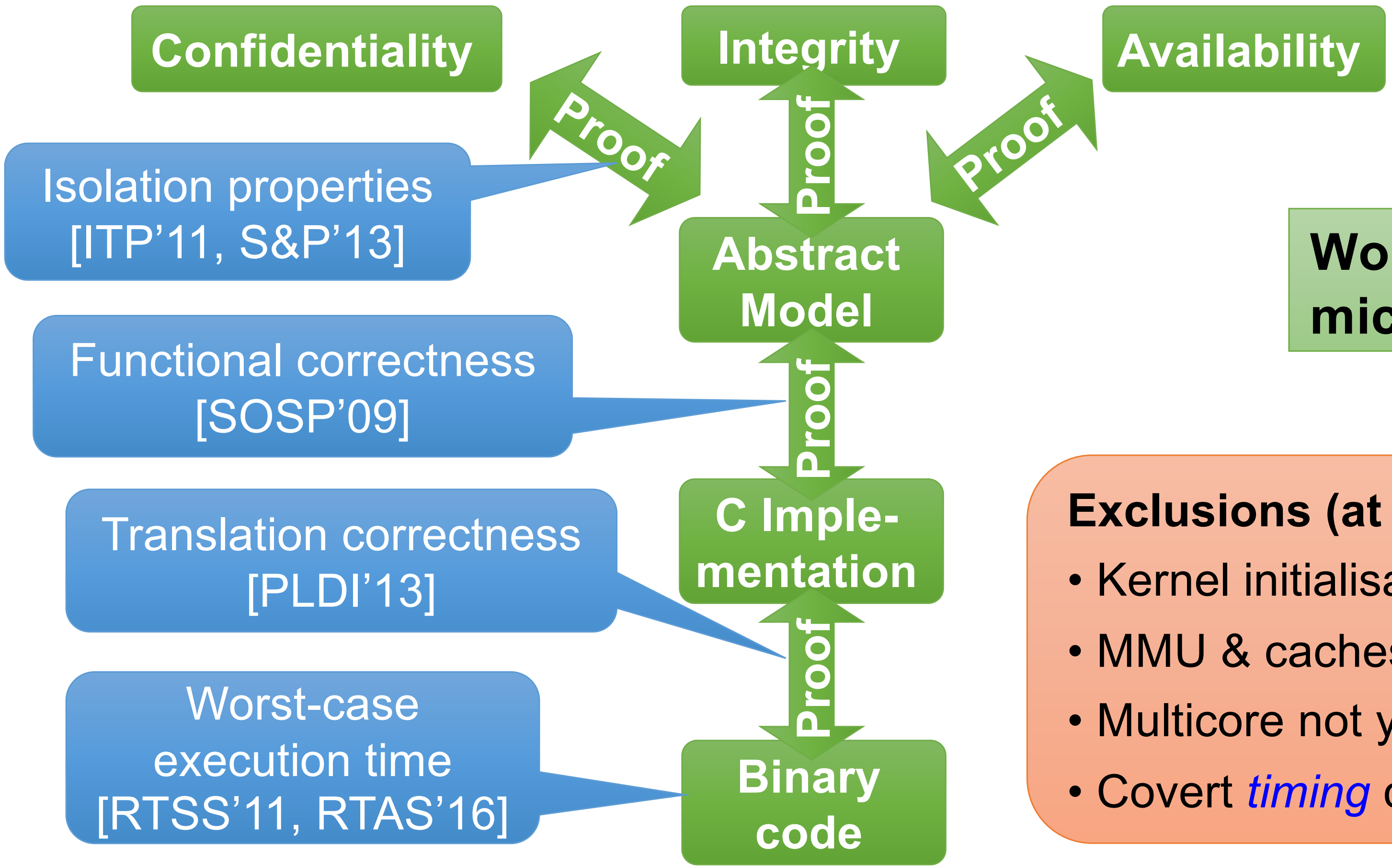
Making critical software safer

7 comments

WILLIAM BULKELEY

May/June 2011

Proving Security and Safety

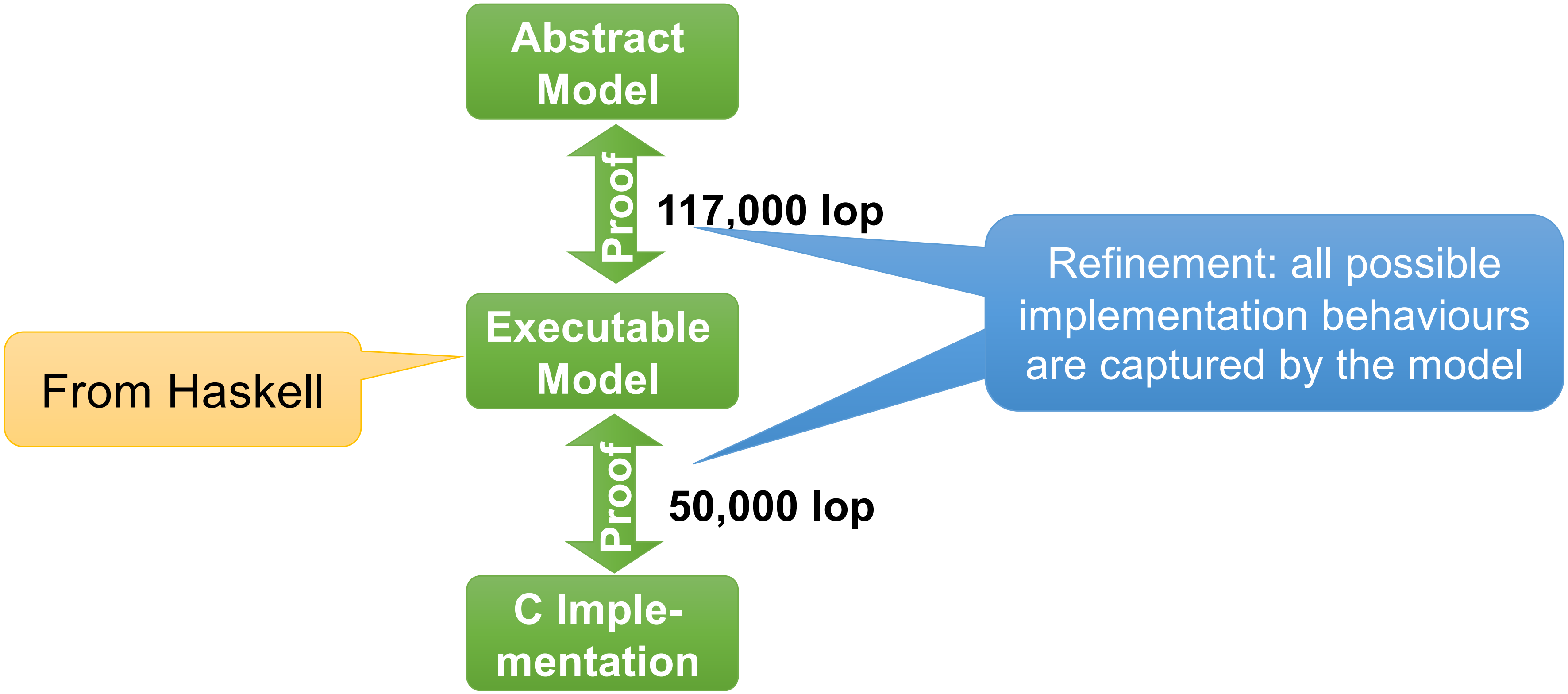


World's fastest microkernel!

Exclusions (at present):

- Kernel initialisation not yet verified
- MMU & caches modelled abstractly
- Multicore not yet verified
- Covert *timing* channels not precluded

Proving Functional Correctness



seL4 Proving Functional Correctness

```
constdefs
  schedule :: "unit s_monad"
  "schedule ≡ do
     threads ← allActiveTCBs;
     thread ← select threads;
     do_machine_op flushCaches OR return ();
     modify (λs. s ⦇ cur_thread := thread ⦈)
   od"
```

```
schedule :: Kernel ()
schedule = do
        action <- getSchedulerAction
        case action of
                rmally -> do
                ead <- getCurThread
                le <- isRunnable curThread
                - threadGet tcbTimeSlice curThread
                not runnable || time == 0) chooseThread
```

```
void
setPriority(tcb_t *tptr, prio_t prio) {
    prio_t oldprio;

    if(thread_state_get_tcbQueued(tptr->tcbState)) {
        oldprio = tptr->tcbPriority;
        ksReadyQueues[oldprio] = tcbSchedDequeue(tptr, ksReadyQueues[
        if(isRunnable(tptr)) {
            ksReadyQueues[prio] = tcbSchedEnqueue(tptr, ksReadyQueues
        }
        else {
            thread_state_ptr_set_tcbQueued(&tptr->tcbState, false);
        }
    }

    tptr->tcbPriority = prio;
}

void
yieldTo(tcb_t *target) {
    target->tcbTimeSlice += ksCurThread->tcbTimeSlice;
```

Functional Correctness Summary

Can prove further properties on abstract level!

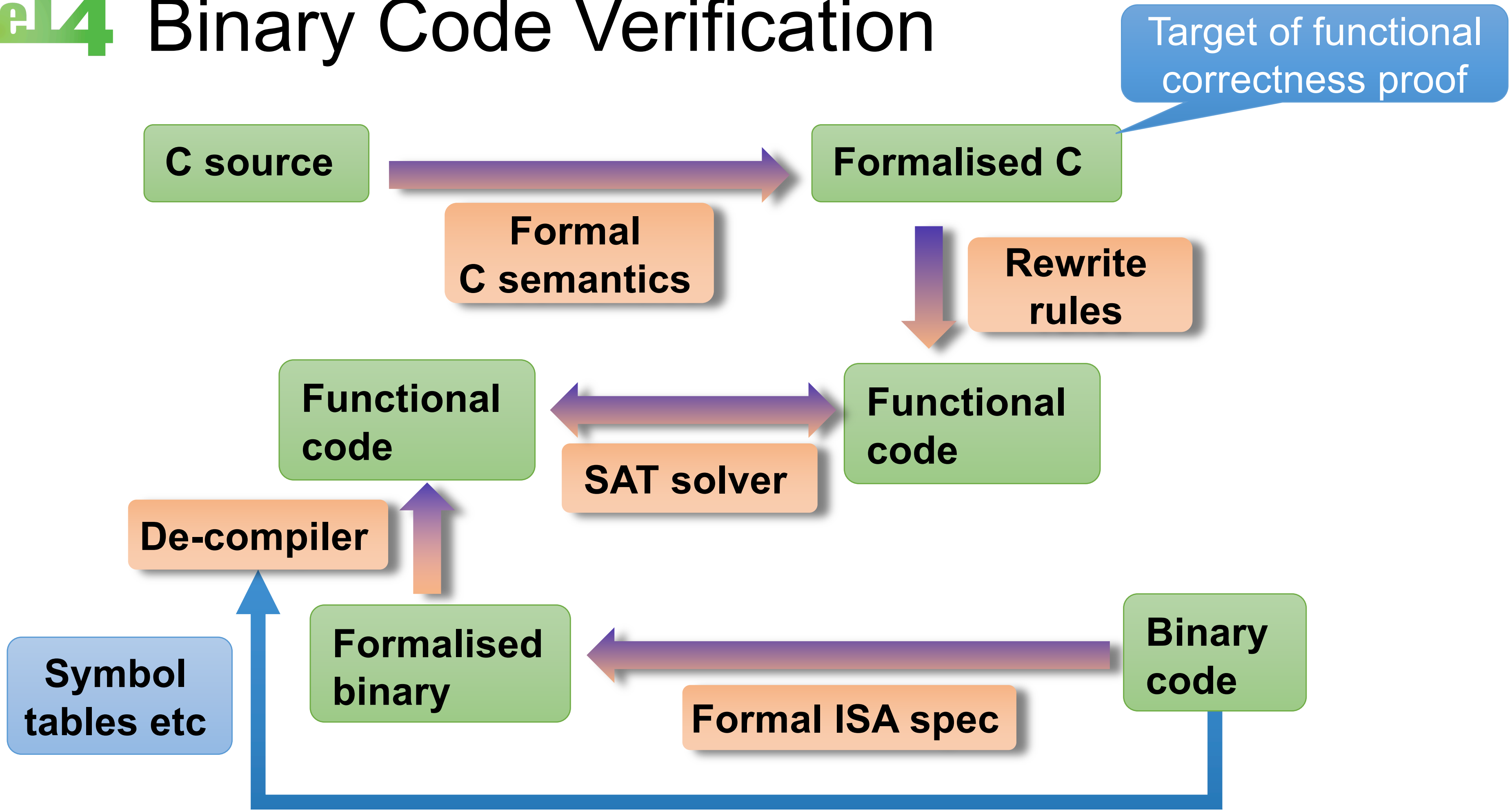
Kinds of properties proved

- Behaviour of C code is fully captured by abstract model
- Behaviour of C code is fully captured by executable model
- Kernel never fails, behaviour is always well-defined
  - assertions never fail
  - will never de-reference null pointer
  - will never access array out of bounds
  - cannot be subverted by misformed input
- All syscalls terminate, reclaiming memory is safe, ...
- Well typed references, aligned objects, kernel always mapped…
- Access control is decidable

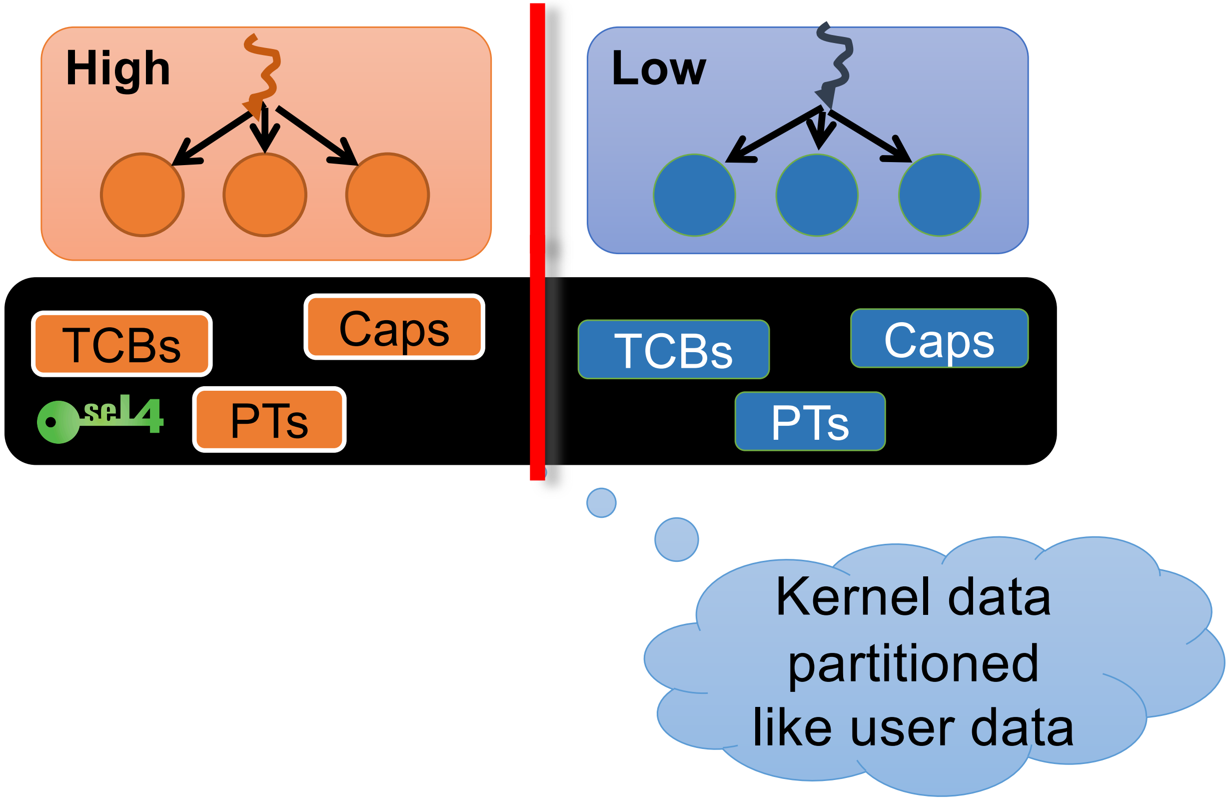
Bugs found:

- 16 in (shallow) testing
- 460 in verification
  - 160 in C,
  - 150 in design,
  - 150 in spec

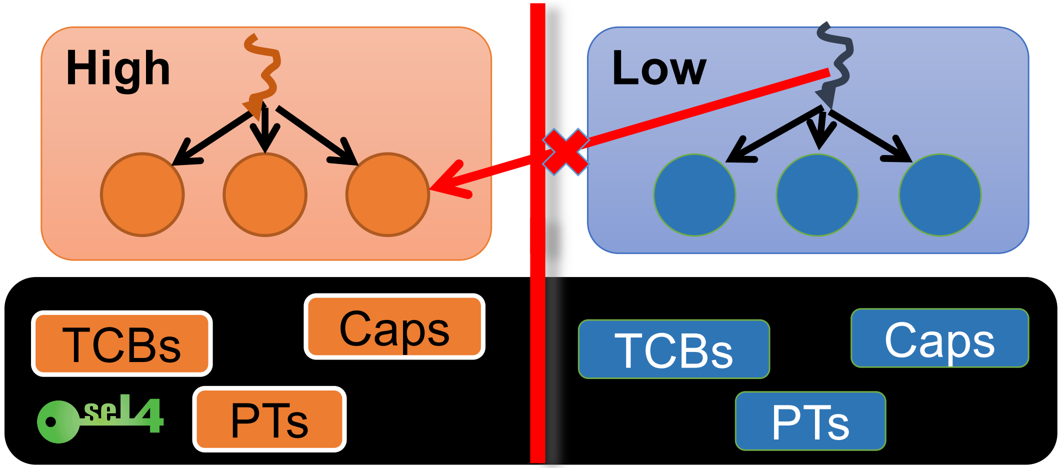
Binary Code Verification



Isolation Goes Deep



Integrity: Control Write Access

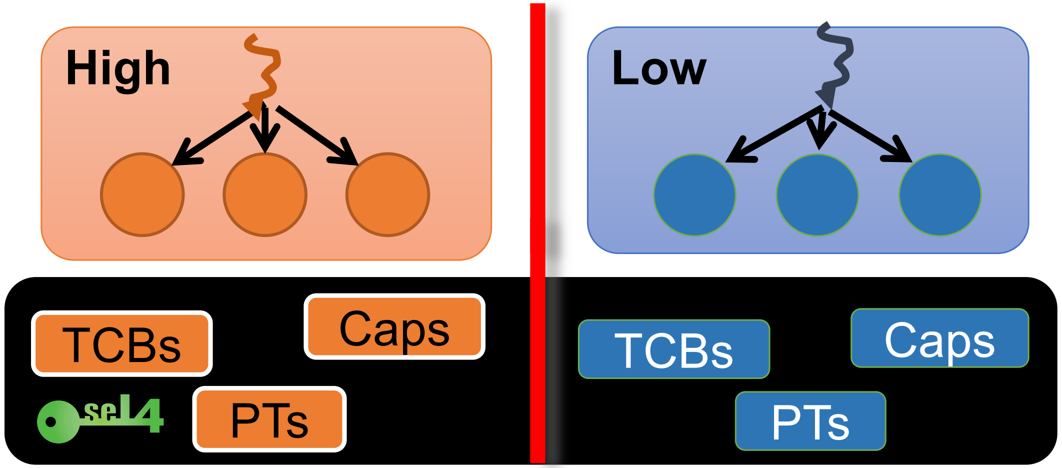


Event-based kernel always operates on be-half of well-defined user:

- Prove kernel only modifies data if presented write cap

To prove:
Low has no *write* capabilities to High objects
⇒ no action of Low will modify High state
Specifically, *kernel does not modify on Low's behalf!*

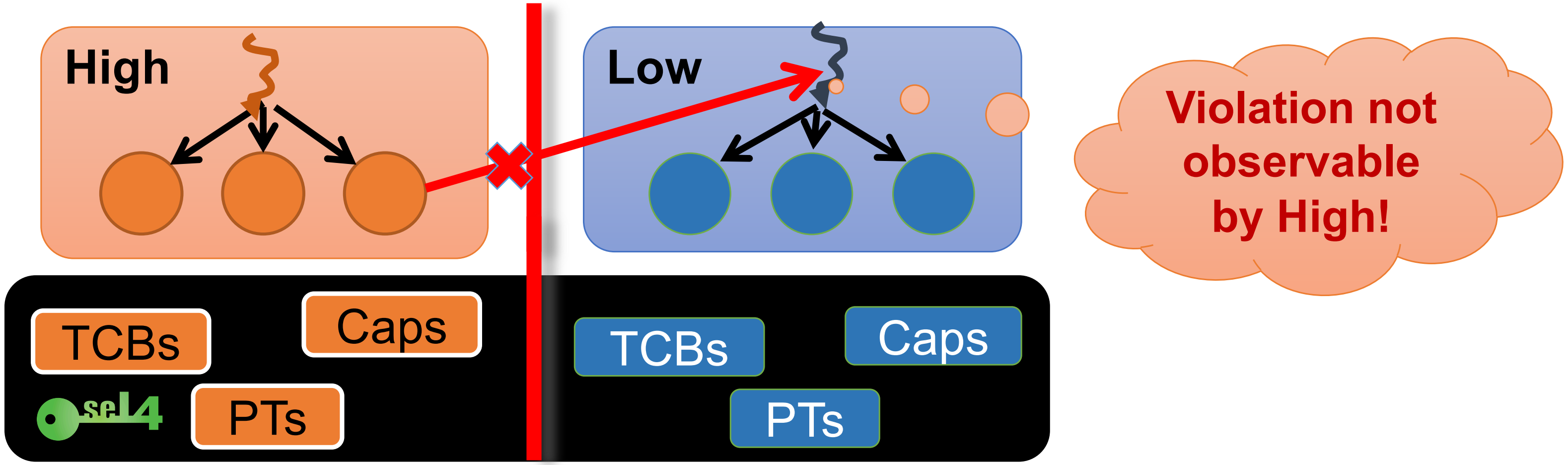
Availability: Ensuring Resource Access



Nothing to do, implied by other properties!

Strict separation of kernel resources
⇒ Low cannot deny High access to resources

Confidentiality: Control Information Flow



Non-interference proof:
- Evolution of Low does not depend on High state
- Also shows absence of covert *storage channels*

To prove:
Low has no *read* capabilities to High objects
⇒ no action will reveal High state to Low

Confidentiality Proof Challenge

Spec

```
bool a();

bool b() {
  int secret;
}
```

Idiotic but valid refinement

Implementation

```
bool a() {
  return !secret;
}
```

Non-determinism breaks confidentiality under refinement!

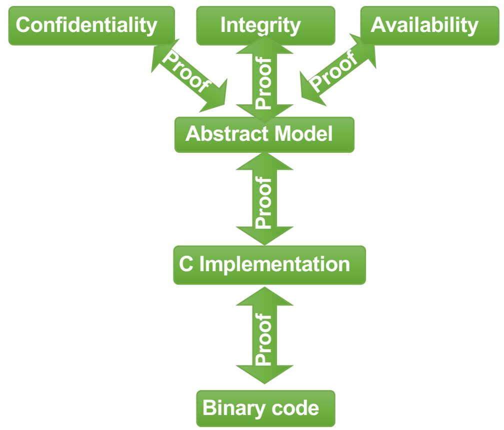
Solution:

- Remove non-determinism where it affects confidentiality
- Eg: scheduler strictly round-robin

Infoflow is very strong property, requiring restrictions rarely met in real world

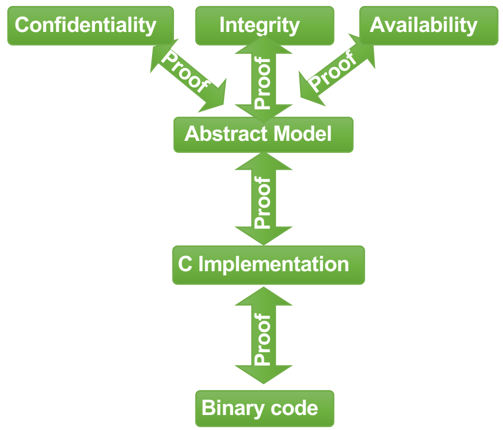
seL4 Verification Assumptions

1. Hardware behaves as expected
   - Formalised hardware-software contract (ISA)
   - Hardware implementation free of bugs, Trojans, …
2. Spec matches expectations
   - Can only prove "security" if specify what "security" means
   - Spec may not be what we think it is
3. Proof checker is correct
   - Isabel/HOL checking core that validates proofs against logic



With binary verification do **not** need to trust C compiler!

Present Verification Limitations



- Not verified boot code
  - **Assume** it leaves kernel in safe state
- Caches/MMU presently modeled at high level / axiomised

MMU model just finished

- Not proved any temporal properties
  - Presently not proved scheduler observes priorities, properties needed for RT
  - WCET analysis applies only to dated ARM11/A8 cores
  - No proofs about timing channels

seL4 Common Criteria?

Level	Requirements	Specification	Design	Implementation
EAL1	not evaluated	**Informal**	not eval	not evaluated
EAL2	not evaluated	**Informal**	**Informal**	not evaluated
EAL3	not evaluated	**Informal**	**Informal**	not evaluated
EAL4	not evaluated	**Informal**	**Informal**	not evaluated
EAL5	not evaluated	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL6	**Formal**	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL7	**Formal**	**Formal**	**Formal**	**Informal**
seL4	**Formal**	**Formal**	**Formal**	**Formal**

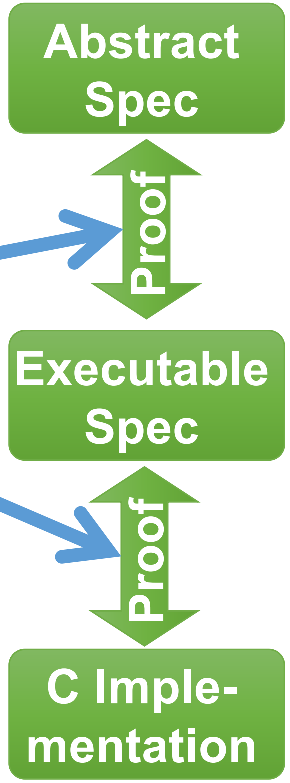
Cost of Verification

Verification Cost Breakdown

Haskell design	2 py
C implementation	2 months
Debugging/Testing	2 months
Abstract spec refinement	8 py
Executable spec refinement	3 py
Fastpath verification	5 months
Formal frameworks	9 py
Total	**24 py**
Non-reusable verification	**11.5 py**
Traditional engineering	**4–6 py**

Verification

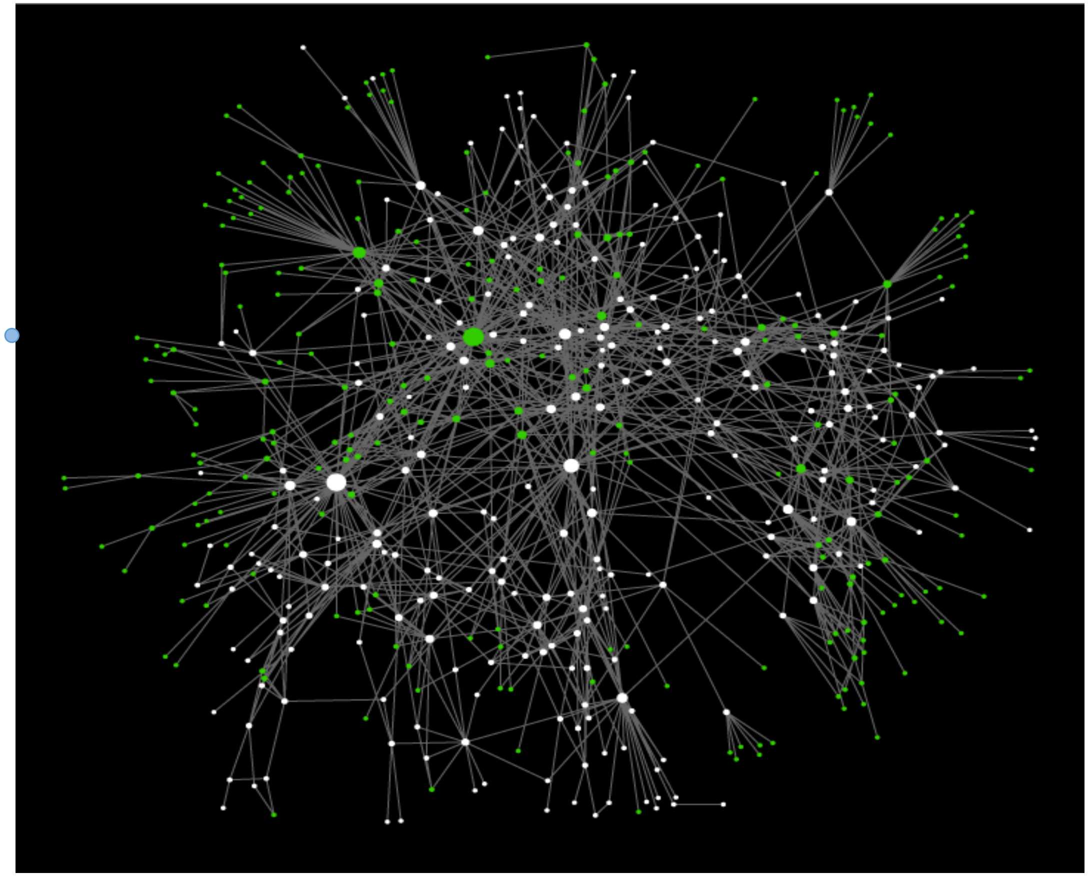
Reusable!



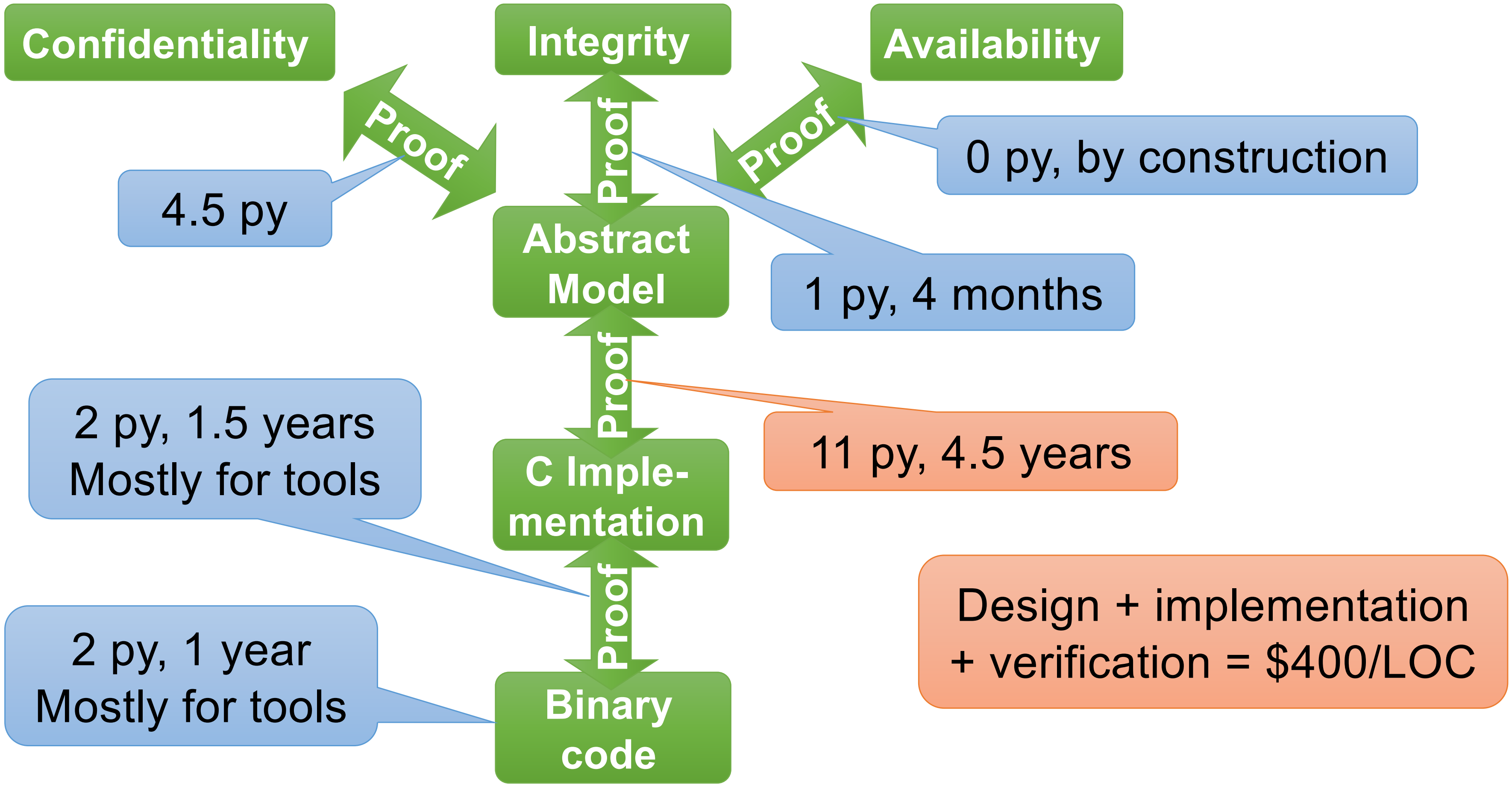
UNSW SYDNEY

Why So Hard for 9,000 LOC?

seL4 call graph



Verification Cost



UNSW SYDNEY

Microkernel Life-Cycle Cost in Context