COMP9242 Advanced OS

S2/2016 W12: **Local Systems Research**

@GernotHeiser

Never Stand Still · Engineering · Computer Science and Engineering

Copyright Notice

Present Systems are *NOT* Trustworthy!

Yet they are expensive:
- $1,000 per line of code for "high-assurance" software!

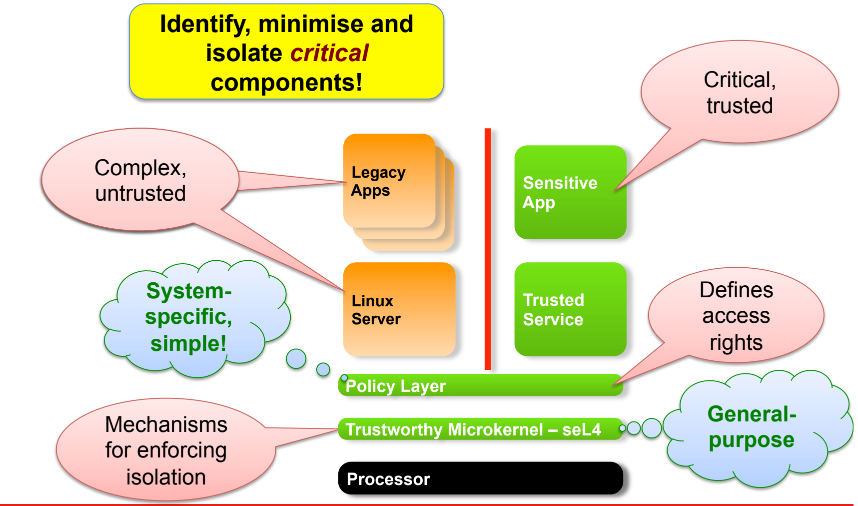
Trustworthy Systems Vision

Suitable for real-world systems

We will change the *practice* of designing and implementing critical systems, using rigorous approaches to achieve *true trustworthiness*

Hard *guarantees* on safety/security/ reliability

seL4 Isolation is Key!



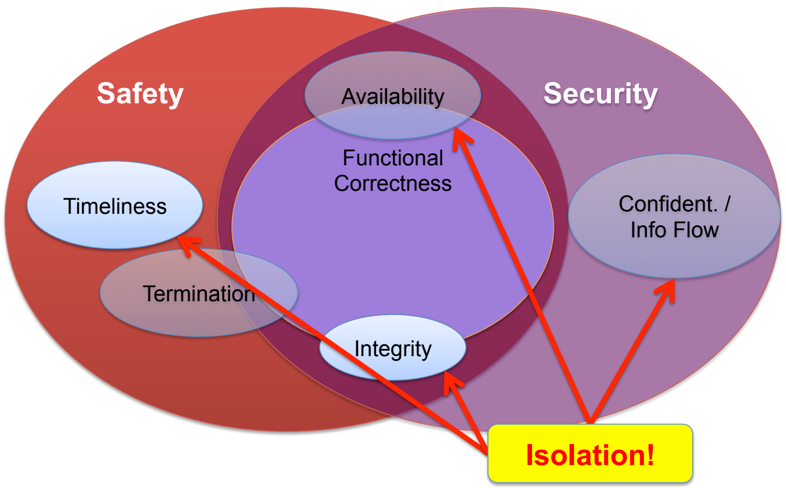
seL4 Trustworthy Systems Agenda

1. **Dependable microkernel (seL4) as a rock-solid base**
   - Formal specification of functionality
   - Proof of functional correctness of implementation
   - Proof of safety/security properties

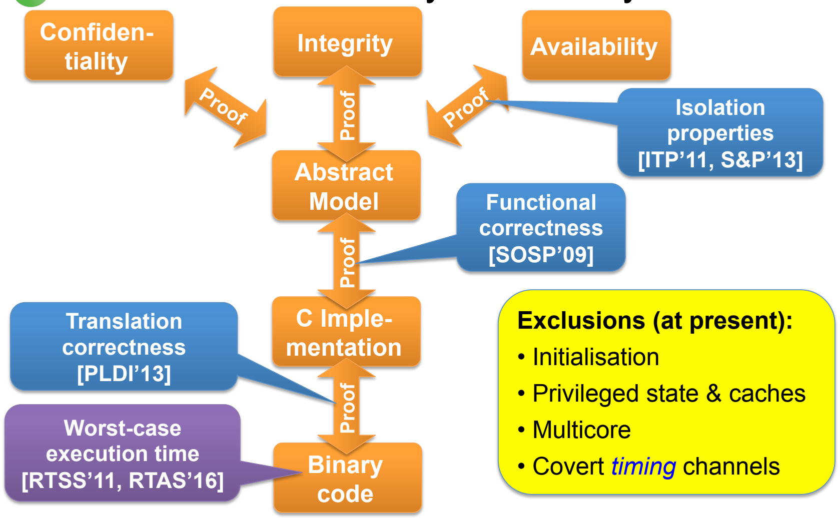
DONE

2. **Lift microkernel guarantees to whole system**
   - Use kernel correctness and integrity to guarantee critical functionality
   - Ensure correctness of balance of trusted computing base
   - Prove dependability properties of complete system
     - despite 99 % of code untrusted!

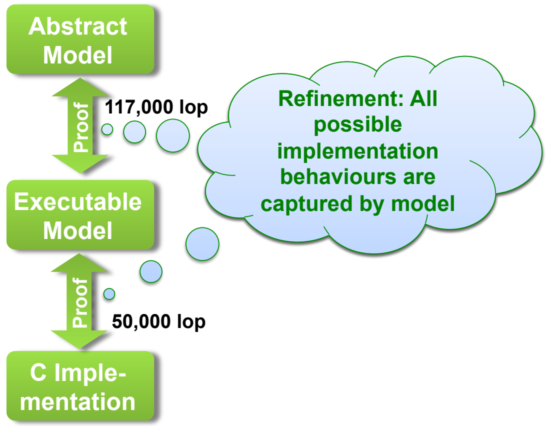
Requirements for Trustworthy Systems



seL4 Provable Security and Safety



Proving Functional Correctness



Proving Functional Correctness

```
constdefs
  schedule :: "unit s_monad"
  "schedule ≡ do
     threads ← allActiveTCBs;
     thread ← select threads;
     do_machine_op flushCaches OR return ();
```

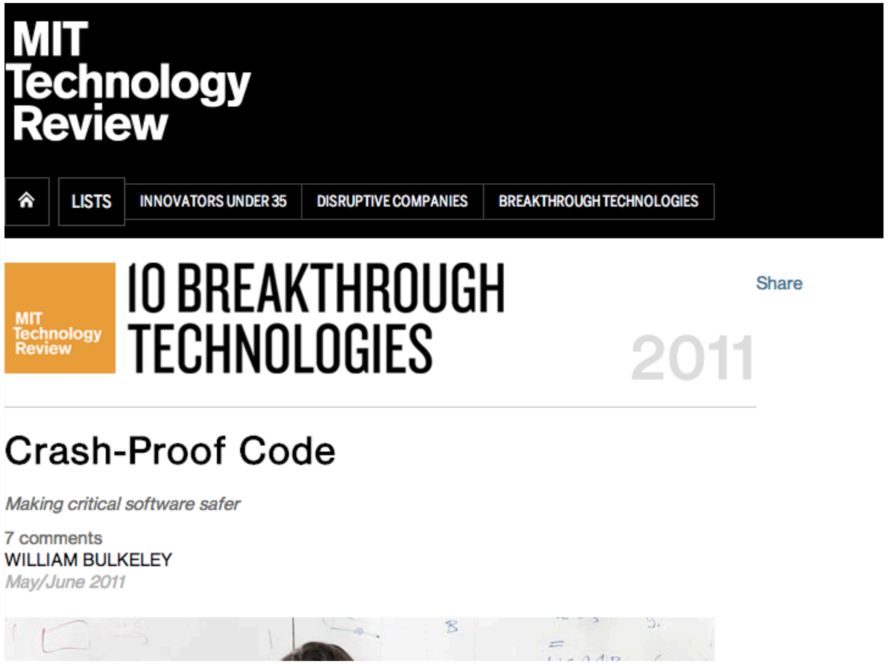
```
schedule :: Kernel ()
schedule = do
        action <- getSchedulerAction
        case action of
...
rThread
Slice curThread
e == 0) chooseThread
```

```c
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;
```

Formal Verification Summary

Kinds of properties proved

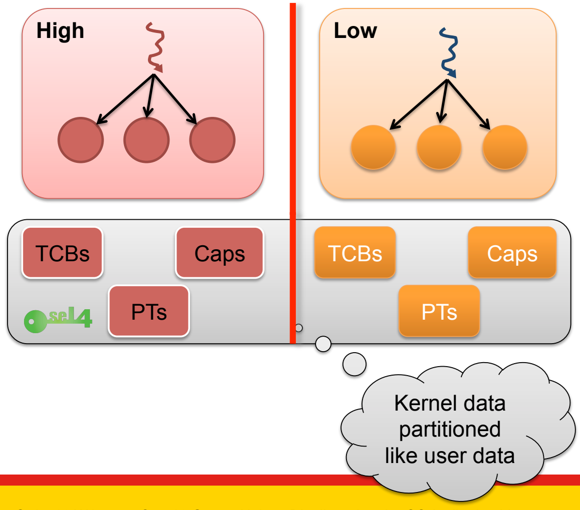
- Behaviour of C code is fully captured by abstract model
- Behaviour of C code is fully captured by executable model
- Kernel never fails, behaviour is always well-defined
  - assertions never fail
  - will never de-reference null pointer
  - cannot be subverted by misformed input
- All syscalls terminate,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

Can prove further properties on abstract level!

Did you find bugs?

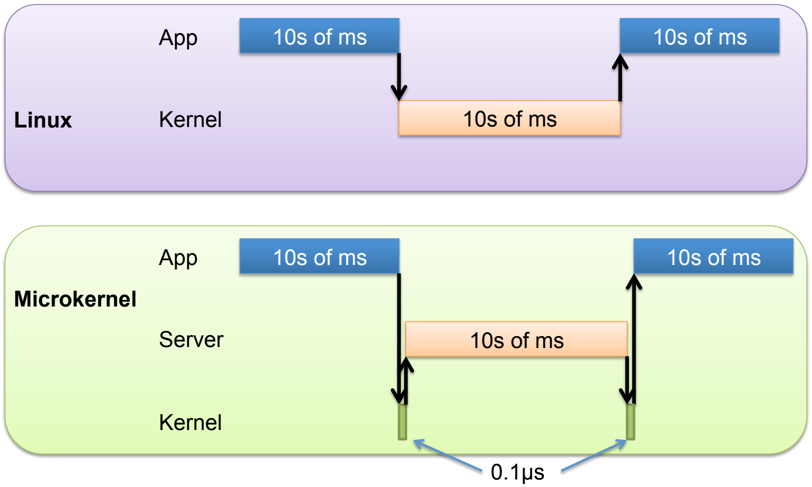
- During (very shallow) testing: 16
- During verification: 460
  - 160 in C, ~150 in design, ~150 in spec

Isolation Goes Deep

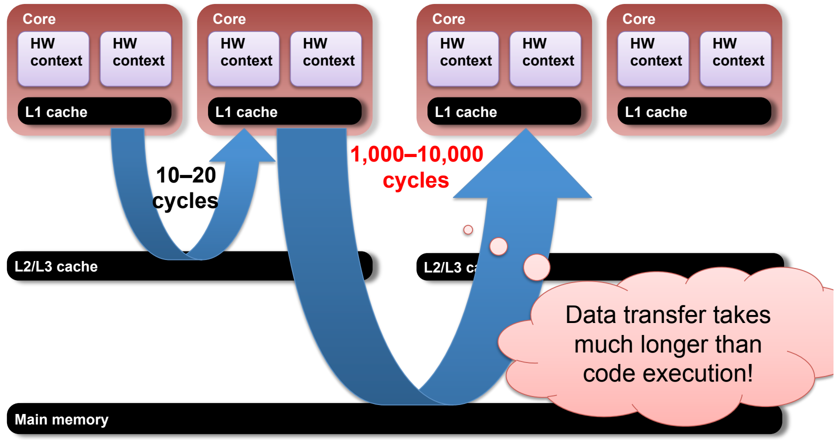


Multicore

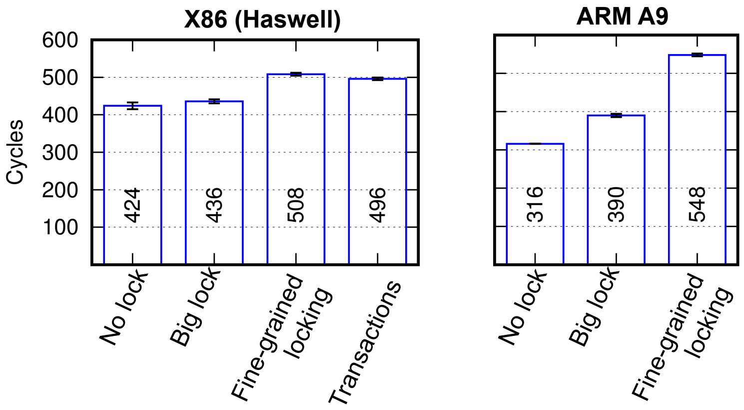
Microkernel vs Linux Execution



Cache Line Migration Latencies

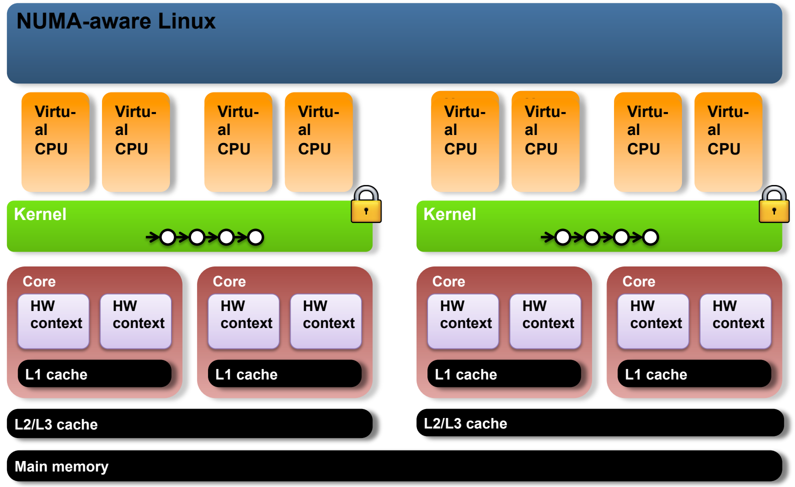


Cost of Locking

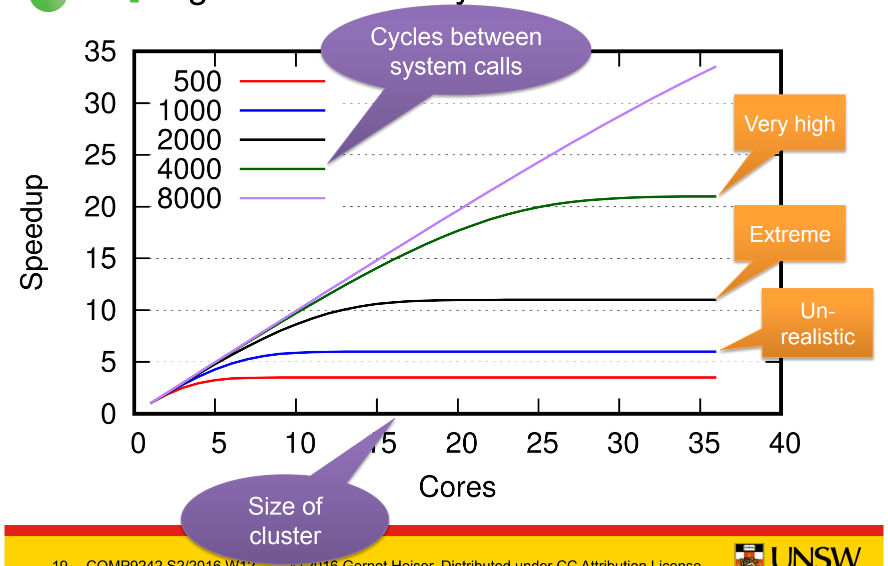


Locks have a cost –
significant in a fast microkernel!

Multicore Design: Clustered Multikernel

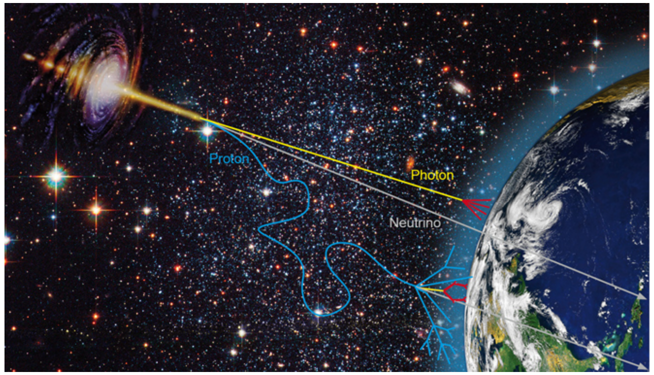


Big-Lock Scalability



Hardware Faults

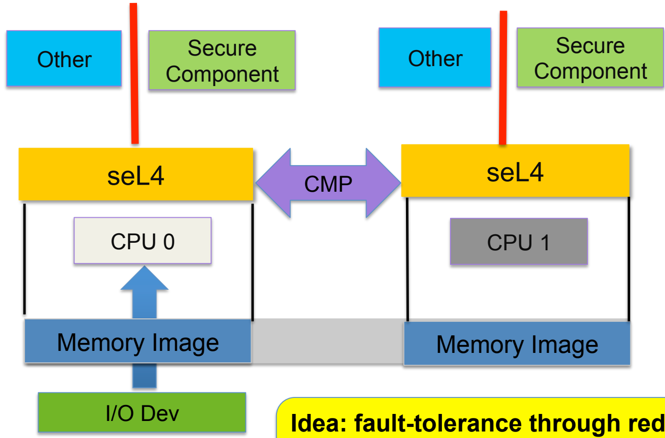
How About Hardware Faults?



- Single-event upset: Random (transient) bit-flips due to cosmic rays, natural radioactivity
- May break "proved" isolation

Courtesy Yanyan Shen

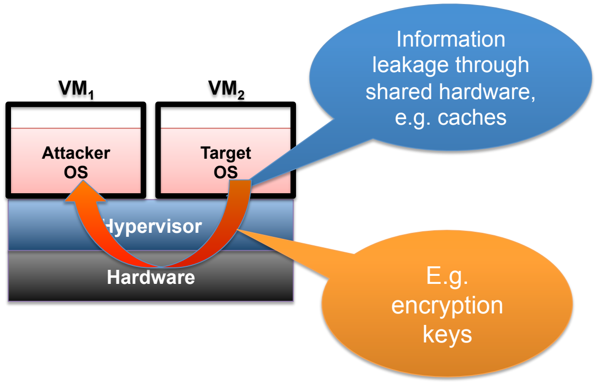
Redundant Execution



Idea: fault-tolerance through redundancy
- Compare & vote at kernel entry/exit
- Work in progress (Yanyan's PhD)

Side Channels

Side Channel Attacks



Types of Side Channels

Storage Channels

- Use some shared state
- Could be inside the OS/hypervisor
  - Eg existence of a file
  - Eg accessibility of an object

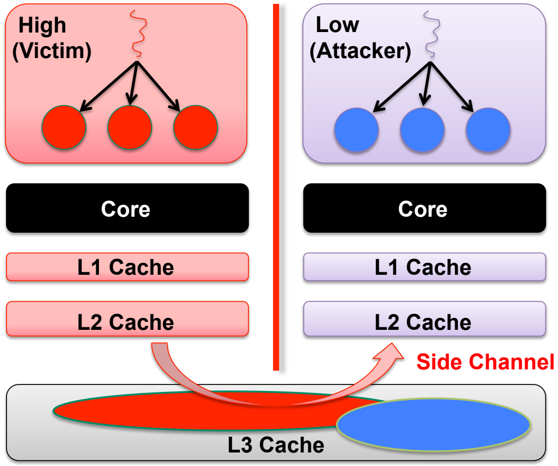
Timing Channels

- Observe timing of events
- Eg memory access latency
  - Senses victim's cache footprint

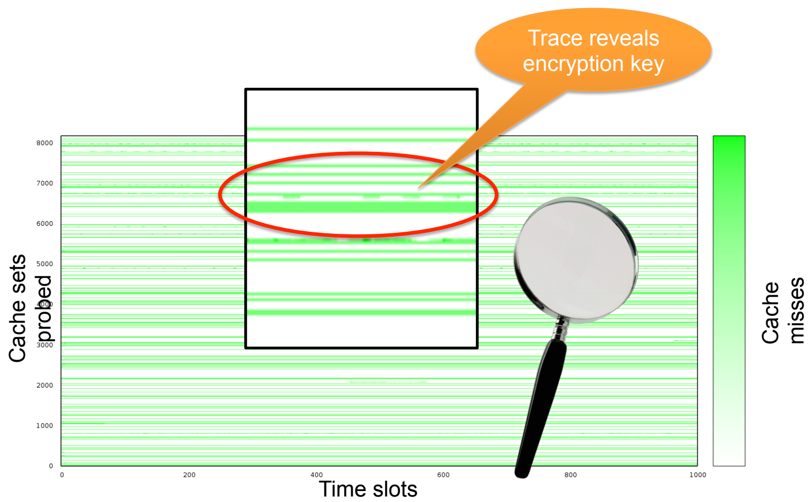
seL4: The world's only OS proved free of storage channels!

How about timing channels?

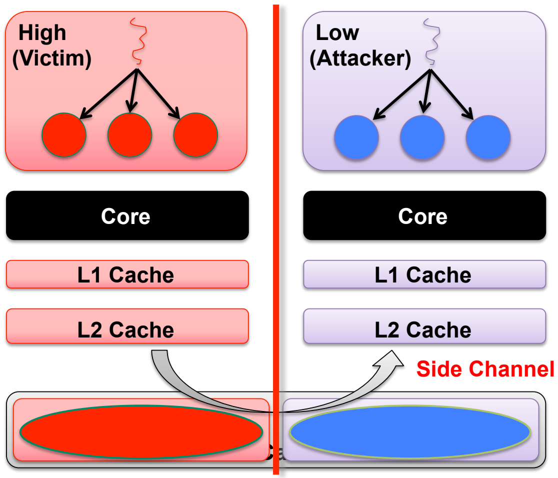
Timing Side-Channel Attack in Public Cloud



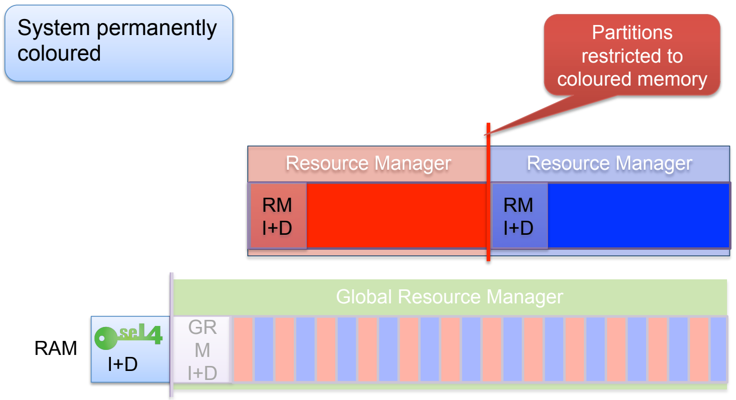
Analysing Memory Access Latency



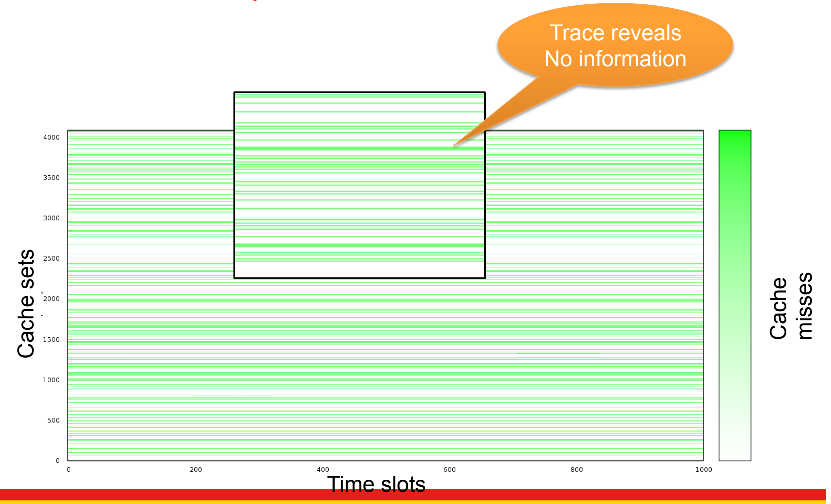
Mitigation: Partition Cache (Colouring)



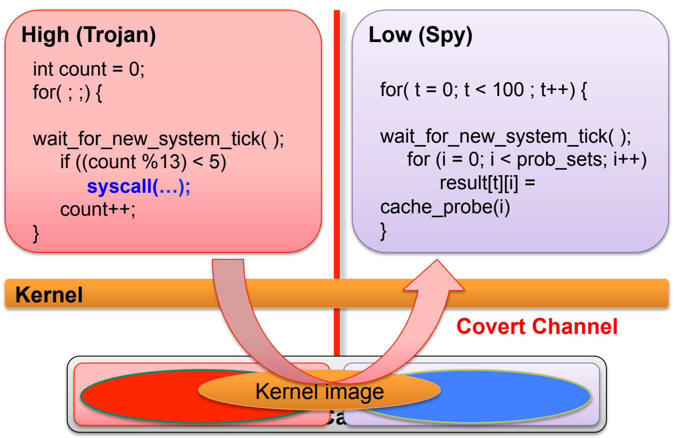
Colouring the System is Easy



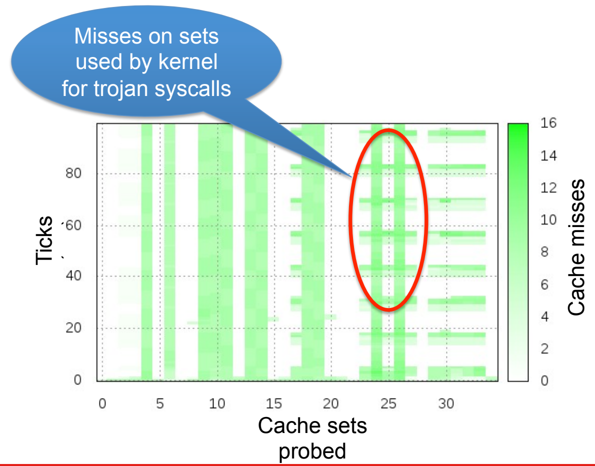
Analysing Memory Access Latency
Coloured System



Timing Channel Through Kernel



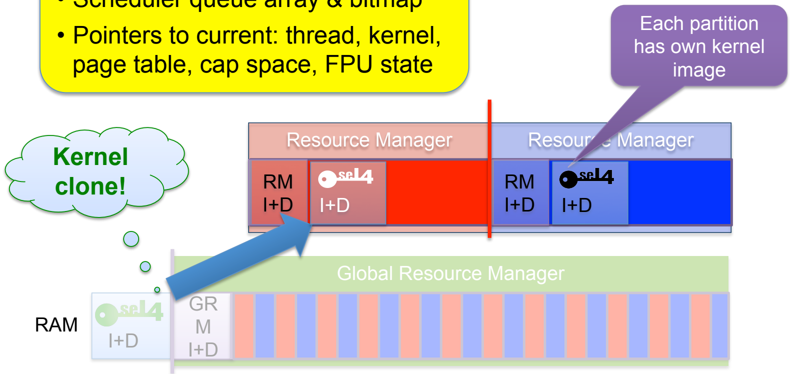
Cache Covert Channel Through Kernel
Spy observations



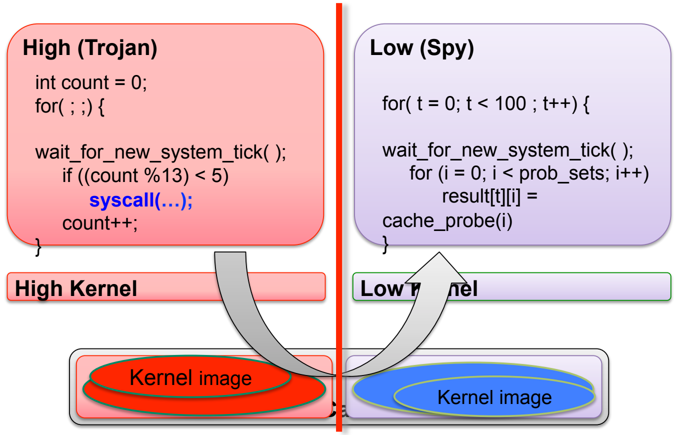
seL4 Colouring the Kernel

Only shared kernel data:

- Scheduler queue array & bitmap
- Pointers to current: thread, kernel, page table, cap space, FPU state

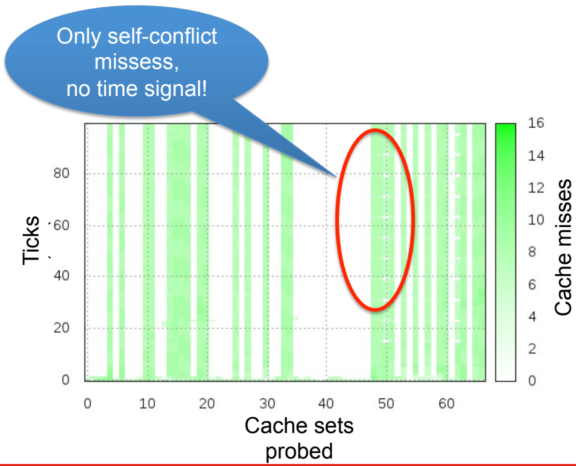


seL4 Timing Channel Through Kernel



seL4 Cache Covert Channel Through Kernel

Spy observations with coloured kernel

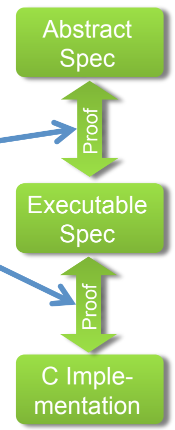


Tackling Verification Cost

seL4 Verification Cost Breakdown

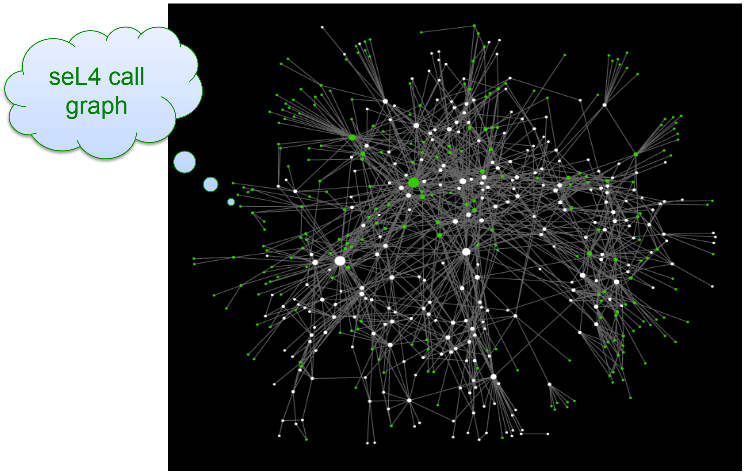
Haskell design	2 py
C implementation	2 months
Debugging/Testing	2 months
Abstract spec refinement	8 py
Executable spec refinement	3 py
Fastpath verification	5 months
Formal frameworks	9 py
Total	**24 py**
Repeat (estimated)	6 py
Traditional engineering	4–6 py

Reusable!



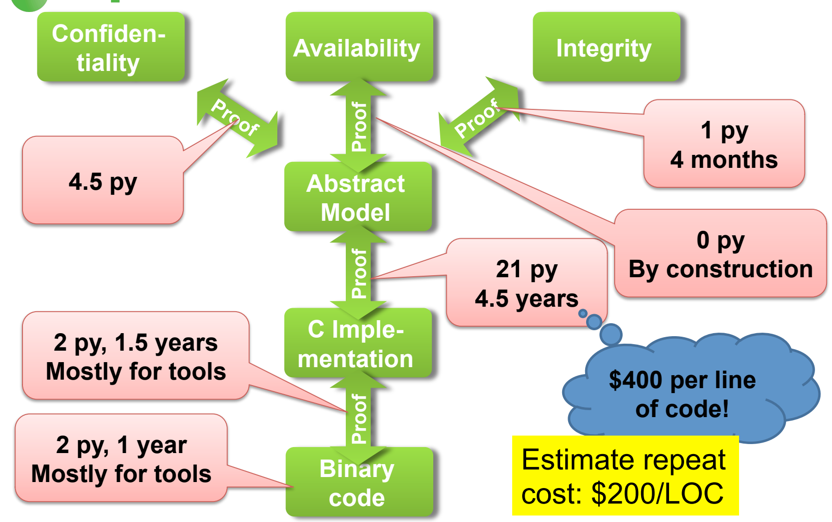
  UNSW

seL4 Why So Hard for 9,000 LOC?



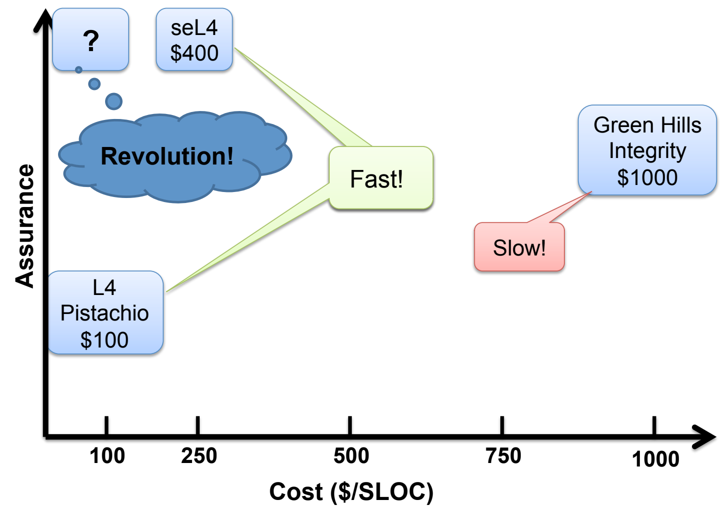
  UNSW

seL4 Cost of Assurance



  UNSW

Microkernel Life-Cycle Cost in Context



  UNSW

Cost of Assurance

Industry Best Practice:

- "High assurance": $1,000/LOC, no guarantees, *unoptimised*
- Low assurance: $100–200/LOC, 1–5 faults/kLOC, *optimised*

State of the Art – seL4:

  - $400/LOC, 0 faults/kLOC, *optimised*
- Estimate repeat would cost half
  - that's about twice the development cost of the predecessor Pistachio!
- Aggressive optimisation [APSys'12]
  - much faster than traditional high-assurance kernels
  - as fast as best-performing low-assurance kernels

What Have We Learnt?

Formal verification *probably* didn't produce a more *secure* kernel

- In reality, traditional separation kernels are *probably* secure

But:

- We now have certainty
- We did it *probably* at less cost

Real achievement:

- Cost-competitive at a scale where traditional approaches still work
- Foundation for scaling beyond: **2 × cheaper, 10 × bigger!**

How?

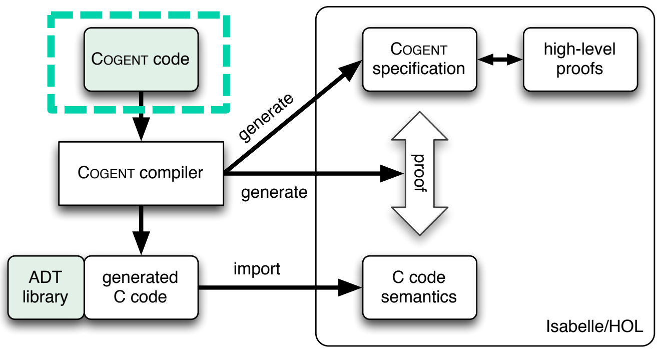
- Combine theorem proving with
  - synthesis
  - domain–specific languages (DSLs)

Our approach

- Cogent: code and proof co-generation
  - Implement FS in high-level functional language (and reason about it)
  - Generate efficient low-level code in C
  - Automatically prove correspondence between the two

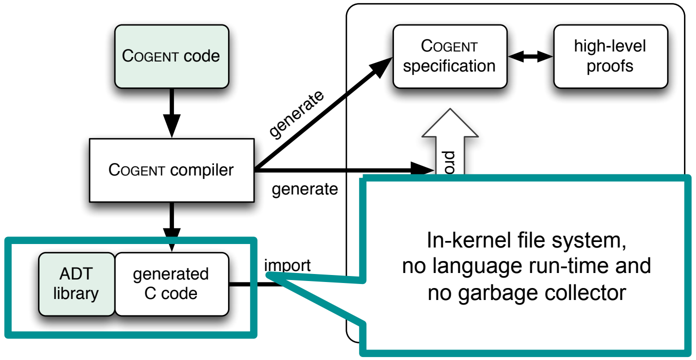
Cogent Workflow

- Cogent: purely functional memory-safe language



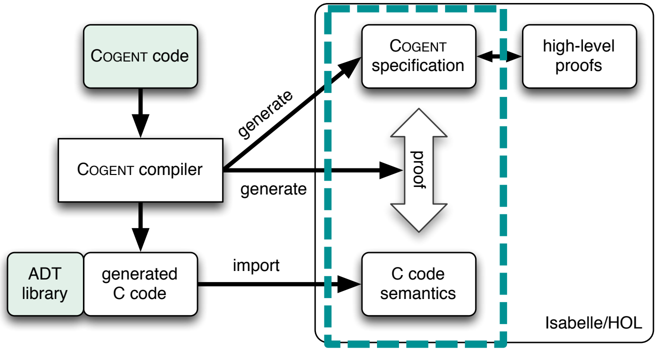
Cogent workflow

- Cogent's certifying compiler generates an C implementation



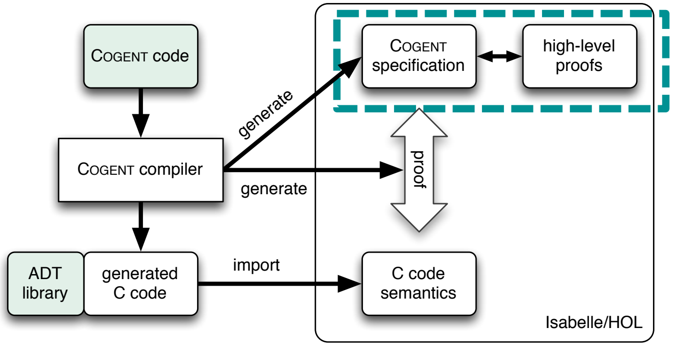
Cogent workflow

- Cogent generates a specification and a proof that links it to the C code



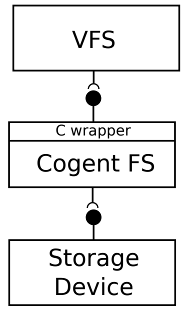
Cogent workflow

- Prove high-level properties about Cogent-generated specifications using a proof assistant



Cogent File Systems

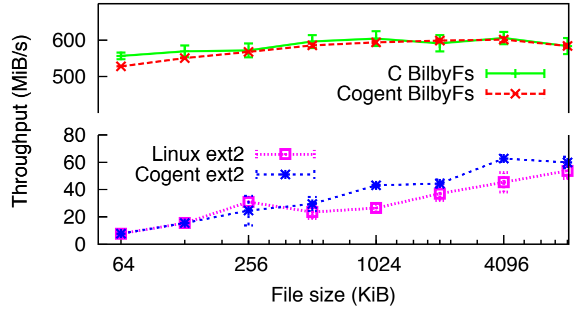
- We implemented two Linux FSs:
  - Ext2: functionally complete original spec
    - No ACLs, symlinks
  - BilbyFs: custom flash file system
- Invoked from VFS via a small C wrapper, which:
  - Uses a global lock to prevent concurrent execution of FS operations
  - Handles VFS caches
  - Calls Cogent FS entry points
- FSs interface with the storage device via external ADT functions



Evaluation

- Compare ext2 with Linux's native implementation
  - Hardware:
    - 4 core i7-6700 running at 3.1 GHz,
    - Samsung HD501JL 7200RPM 500G SATA disk
- Compare BilbyFs with handwritten C implementation
  - Hardware:
    - Mirabox development board
    - Marvell Armada 370 single-core 1.2 GHz ARMv7 processor
    - 1 GiB of NAND flash

IOZone random 4k writes



- 20% CPU load for Cogent BilbyFs vs 15% for C
- Both ext2 implementations have the same CPU load

Postmark on RAM-disk

System	Total time sec	creation files/sec	read rate kB/sec
C ext2	10	5025	248
COGENT ext2	21	2393	118
C BilbyFs	6	33375	431
COGENT BilbyFs	10	20025	259

- Degradation of a factor 2 for Cogent FSs

Postmark on RAM-disk

System	Total time sec	creation files/sec	read rate kB/sec
C ext2	10	5025	248
COGENT ext2	21	2393	118
C BilbyFs	6	33375	431
COGENT BilbyFs	10	20025	259

- Degradation of a factor 2 for Cogent FSs
- Overhead is due to two reasons:
  - extra copying involved when converting in-buffer directory entries into Cogent's internal data type
  - Cogent compiler is overly reliant on C compiler's optimiser to convert automatically C structs passed by copy to pointers

Remember: Verification Cost Breakdown