UNSW
THE UNIVERSITY OF NEW SOUTH WALES

NICTA

COMP9242
Advanced Operating Systems

S2/2012 Week 12:
Local Systems Research

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

Copyright Notice

NICTA

UNSW

Present Systems are *NOT* Trustworthy!

NICTA

UNSW

What's Next?

So, why don't we prove trustworthiness?

***Claim*:**
A system must be considered *untrustworthy* unless *proved* otherwise!

Corollary [with apologies to Dijkstra]:

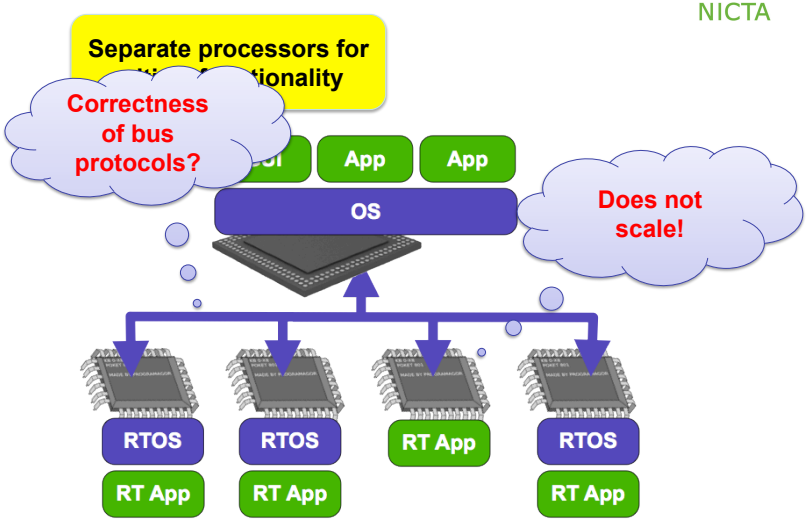
Testing, code inspection, etc. can only show *lack of trustworthiness*!

Core Issue: Complexity

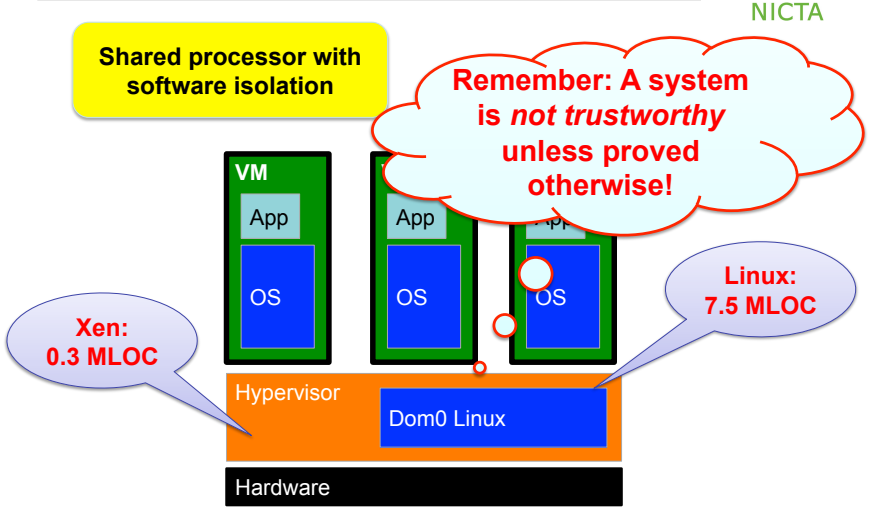
- Massive functionality of C devices
  ⇒ huge software stacks
  - How secure are your paym ?
- Increasing usability requ
  - Wearable or implante
  - Patient-operated
  - GUIs next to life-c
- On-going integration c s
  - Automotive infotainment an
  - Gigabytes of software on 100 CPUs…

Systems far too complex to prove their trustworthiness!

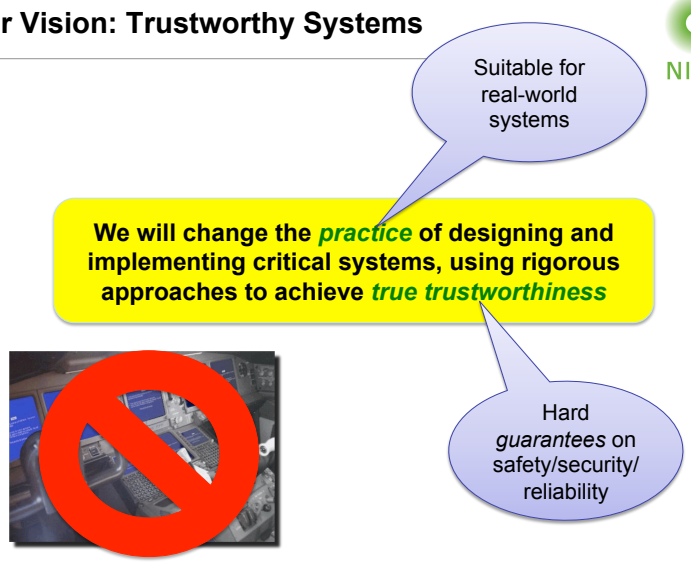
Dealing with Complexity: Physical Isolation



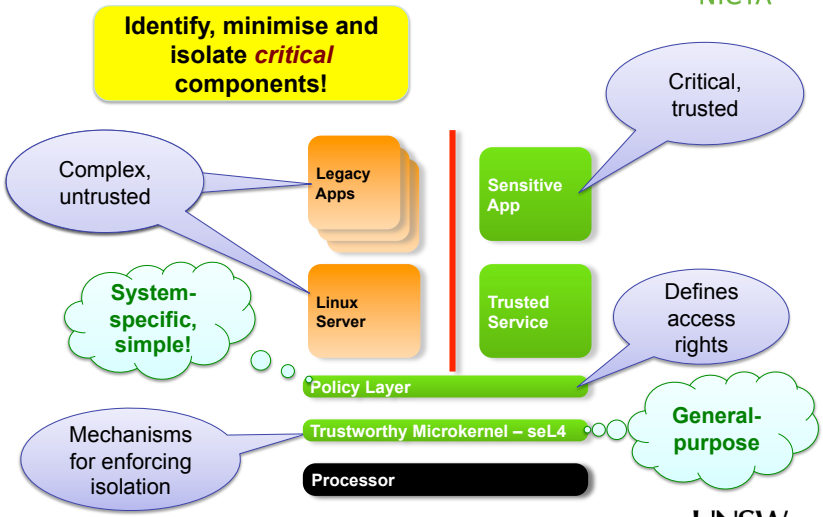
How About Logical Isolation?



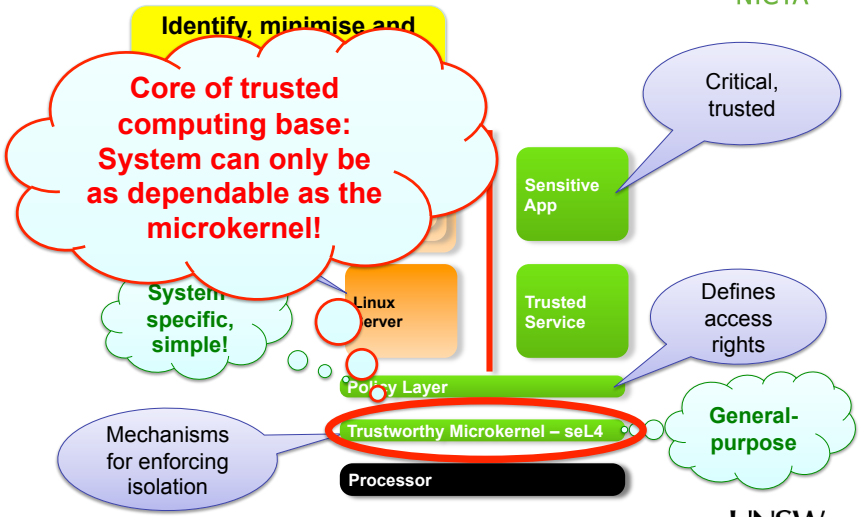
Our Vision: Trustworthy Systems



Isolation is Key!



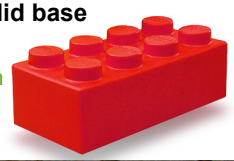
Isolation is Key!



NICTA Trustworthy Systems Agenda

NICTA

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



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!

NICTA Trustworthy Systems Agenda

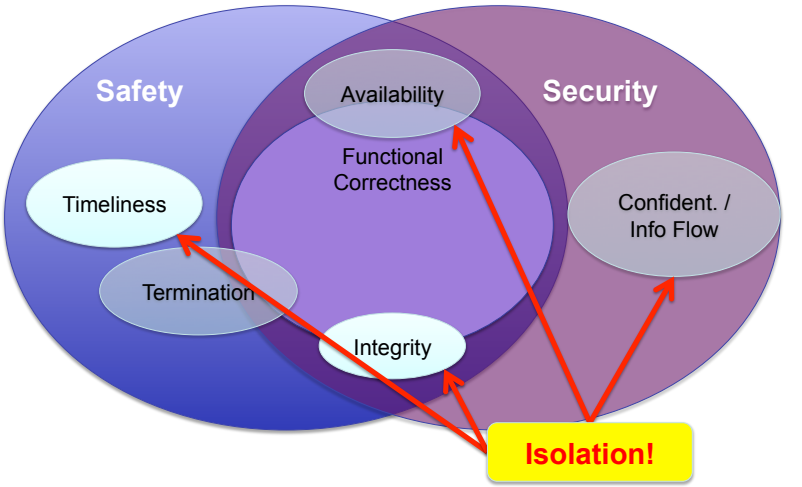
NICTA

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

UNSW

Requirements for Trustworthy Systems

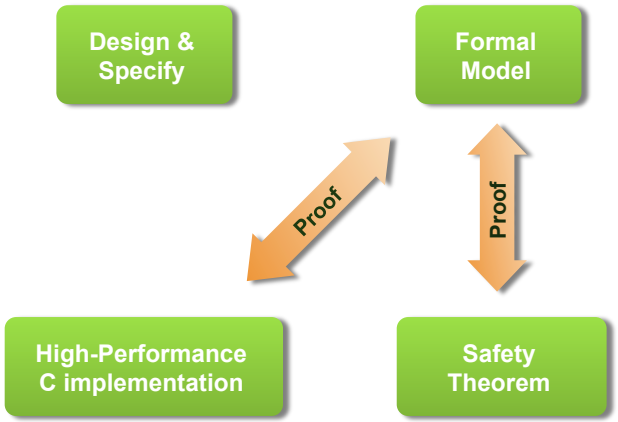
NICTA



UNSW

seL4: Designing and Formalising

NICTA

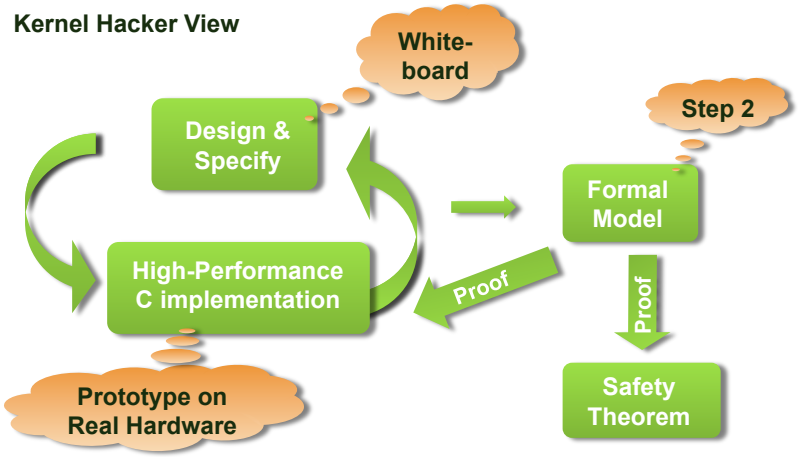


UNSW

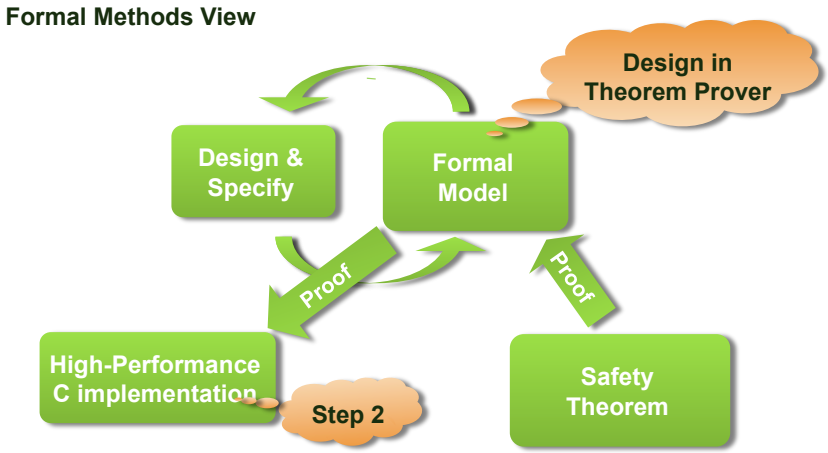
Two Mentalities

Formal Methods Practitioners Systems Researchers

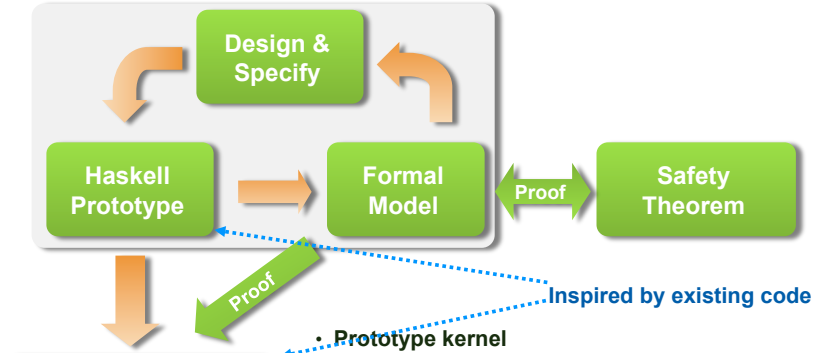
Standard Kernel Design



Formal Design



Iterative Design and Formalisation



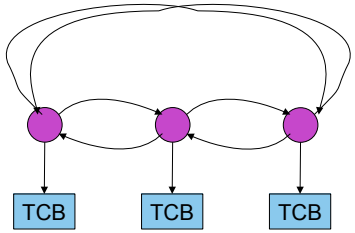
- Prototype kernel executes native binaries on simulator
- Exposes usability issues early
- Tight formal design integration

Kernel Design for Verification

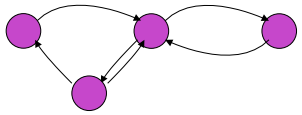
- Main objective: minimise complexity
  - global invariants must be proven for each state change
  - must prove pre- and post-conditions for statements/blocks
  - effort determined by complexity of conditions and state change
- … without sacrificing performance
- Affects design in many ways
  - global variables, side effects
  - kernel memory management
  - concurrency and non-determinism
  - I/O

Global Variables

- Not a difficulty per se, but potential source of complexity
- Eg: scheduler queue as doubly-linked list
  - Show that
    - all pointers are to valid nodes
    - front- and back-pointers are consistent
    - nodes point to TCBs

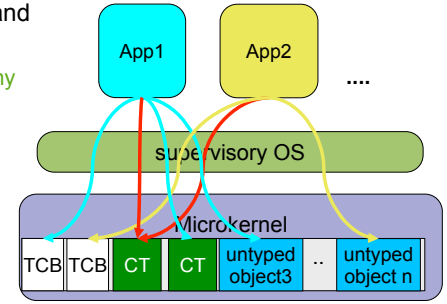


- Requires proof that any pointer operation maintains invariants
- Challenge is temporary violation
  - eg adding a node
  - Requires ensuring atomicity



Kernel Memory

- seL4 kernel memory management model pushes policy to userland
  - aids verification
  - need to ensure strict hierarchy
  - capability derivation tree
- Challenge is re-use
  - most difficult part of verification!
  - use derivation tree to detect all references
  - global data structure that requires invariants in all parts of the system



Concurrency

- Proofs about concurrent programs are inherently hard!
- seL4 strictly limits concurrency to the bare minimum
  - Single processor
    - multicore via big kernel lock or multikernel approach
  - User-level device drivers
  - Non-preemptible, event-based
    - single kernel stack
  - Interrupt points to limit real-time latencies
    - poll interrupt status
    - insert new kernel event (ahead of user)
    - return to user boundary and re-enter kernel
    - allows maintaining all invariants

Concurrency

Preempting object destruction:

- Keep one cap as zombie during object cleanup
  - only retained to reference partially cleaned-up object
  - stores state of cleanup, maintaining invariants
  - attempt by preemptor to remove zombie can just execute

Exceptions in kernel:

- Prevent memory exceptions
  - ensure kernel page tables are complete
  - map into every address space
- Disallow other exceptions
  - verification is its own friend ☺

I/O

- Mostly a non-issue
  - user-level drivers
    - IOMMU support for DMA security
  - non-preemptible kernel
- Exception is timer tick
  - essentially a source of interrupts
  - handled in-kernel as separate event
  - no real complication

Lessons for Kernel Design

- Need to reduce complexity forced simple and clean design
  - beneficial even with traditional validation
  - does not necessarily impact performance
- Some design decision beneficial for other reasons too
  - single kernel stack for memory footprint
  - interrupt handling by polling has performance advantages

Proving Functional Correctness

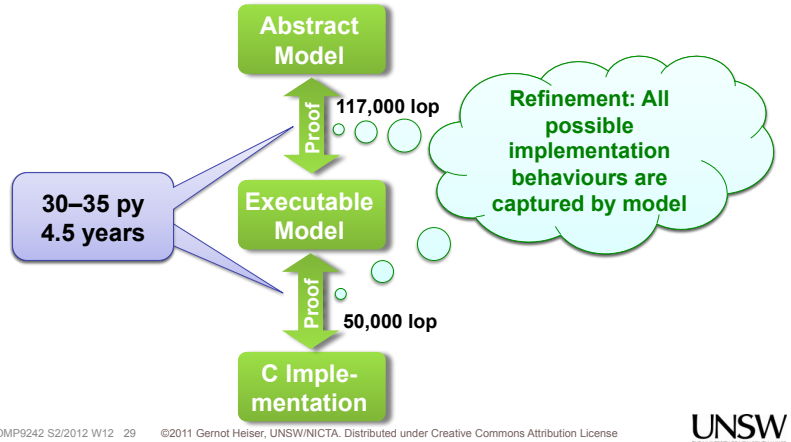
```
constdefs
  schedule :: "unit s_monad"
  "schedule ≡ do
     threads ← allActiveTCBs;
     thread ← select threads;
     do_machine_op flushCaches OR return ();
     modify (λs. s (| cur_thread := thread |))
   od"
```

```
schedule :: Kernel ()
schedule = do
        action <- getSchedulerAction
```

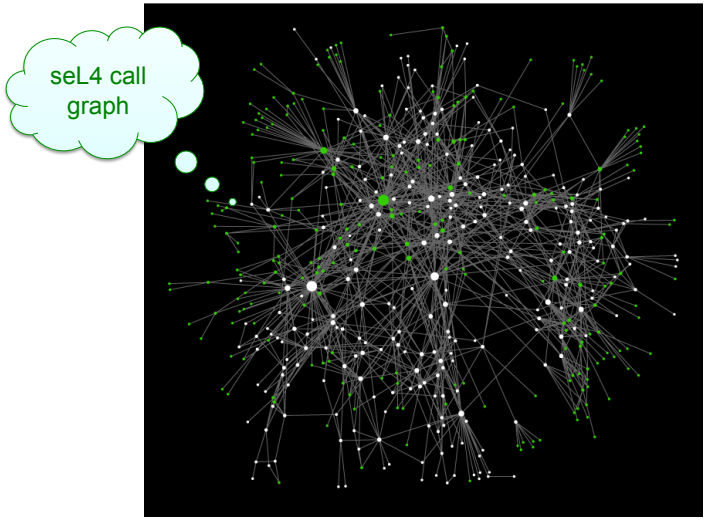
```
void
setPriority(tcb_t *tptr, prio_t prio) {
    prio_t oldprio;

    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;
```

```
ad
curThread
meSlice curThread
ime == 0) chooseThread
```


Proving Functional Correctness



Why So Long for 9,000 LOC?



Costs Breakdown

Haskell design	2 py
C implementation	2 weeks
Debugging/Testing	2 months
Kernel verification	12 py
Formal frameworks	10 py
Total	**25 py**
Repeat (estimated)	6 py
Traditional engineering	4–6 py

Did you find bugs???

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

Does not include subsequent fastpath verification

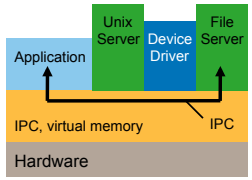
seL4 Formal Verification Summary

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 poperties on abstract level!

How About Performance?



Let's face it, seL4 is basically slow!

- C code (semi-blindly) translated from Haskell
- Many small functions, little regard for performance

IPC: one-way, zero-length	
Standard C code:	1455 cycles
C fast path:	185 cycles

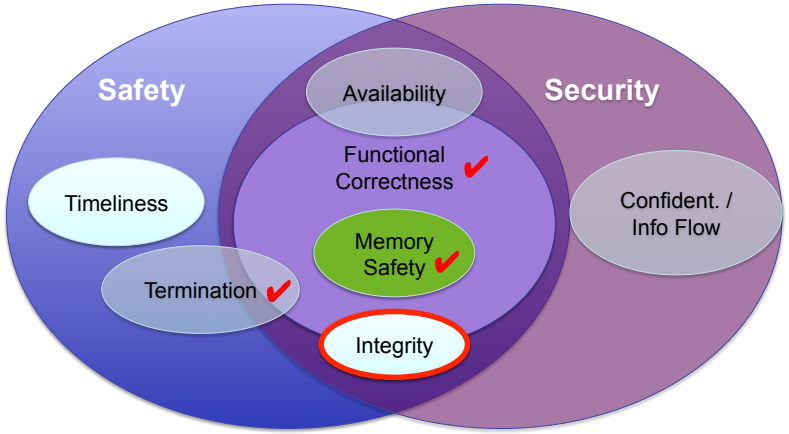
Bare "pass" in Advanced Operating Systems course!

Fastest-ever IPC on ARM11!

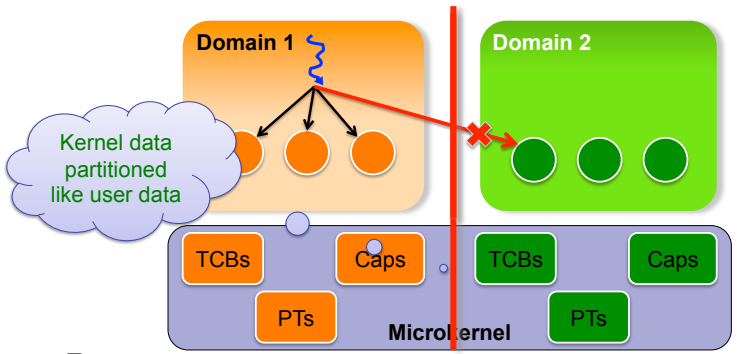
But can speed up critical operations by short-circuit "fast paths"

- … without resorting to assembler!

seL4 as Basis for Trustworthy Systems



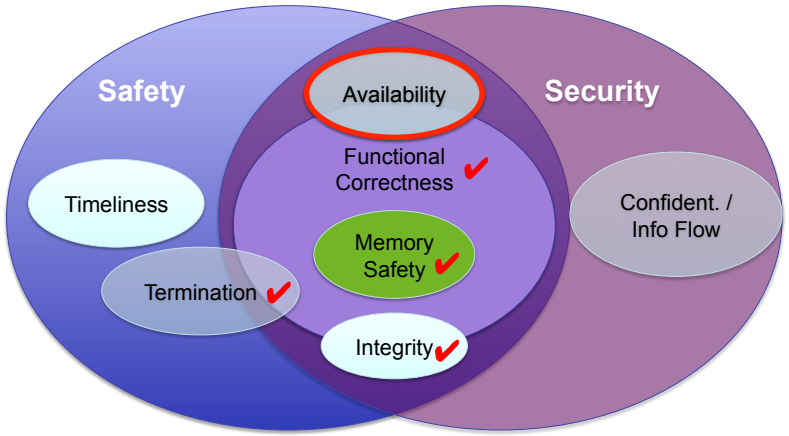
Integrity: Limiting Write Access



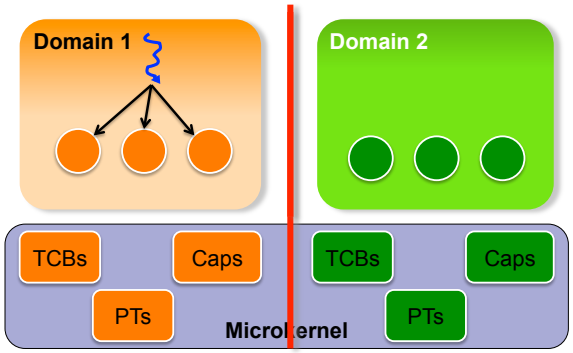
To prove:

- Domain-1 doesn't have write *capabilities* to Domain-2 objects
  ⇒ no action of Domain-1 agents will modify Domain-2 state
- Specifically, *kernel does not modify on Domain-1's behalf!*
  - Event-based kernel operates on behalf of well-defined user thread
  - Prove kernel only allows write upon capability presentation

seL4 as Basis for Trustworthy Systems

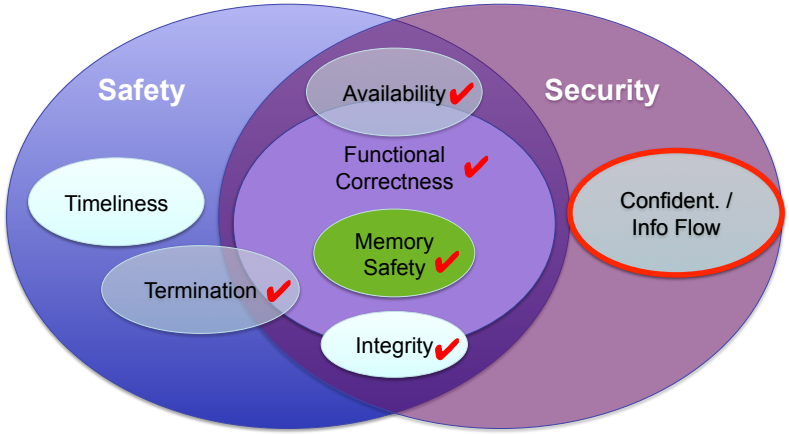


Availability: Ensuring Resource Access

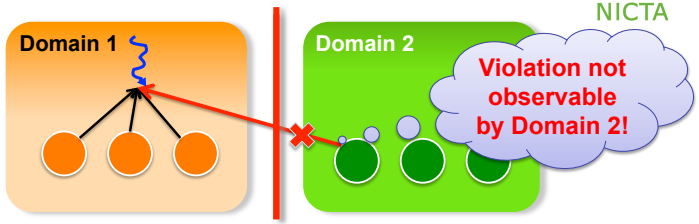


- Strict separation of kernel resources
  ⇒ agent cannot deny access to another domain's resources

seL4 as Basis for Trustworthy Systems



Confidentiality: Limiting Read Accesses



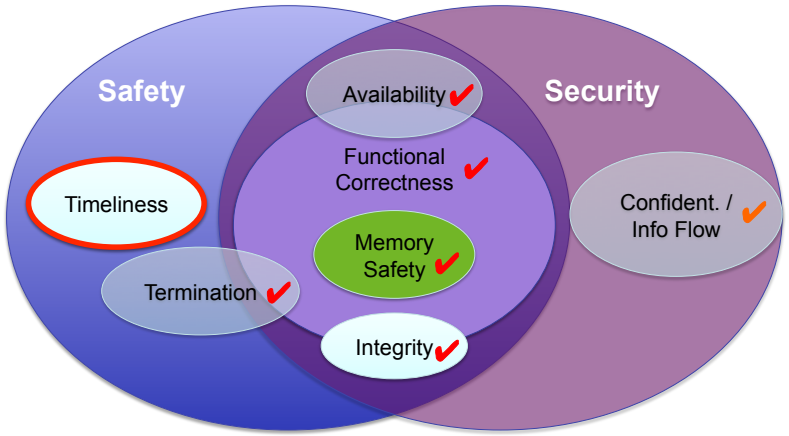
To prove:

- Domain-1 doesn't have read capabilities to Domain-2 objects
  ⇒ no action of any agents will reveal Domain-2 state to Domain-1

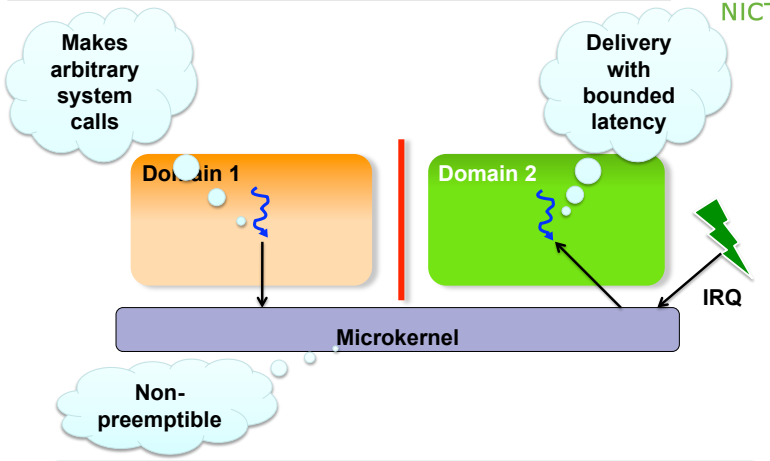
Non-interference proof in progress:

- Evolution of Domain 1 does not depend on Domain-2 state
- Presently cover only overt information flow

seL4 as Basis for Trustworthy Systems

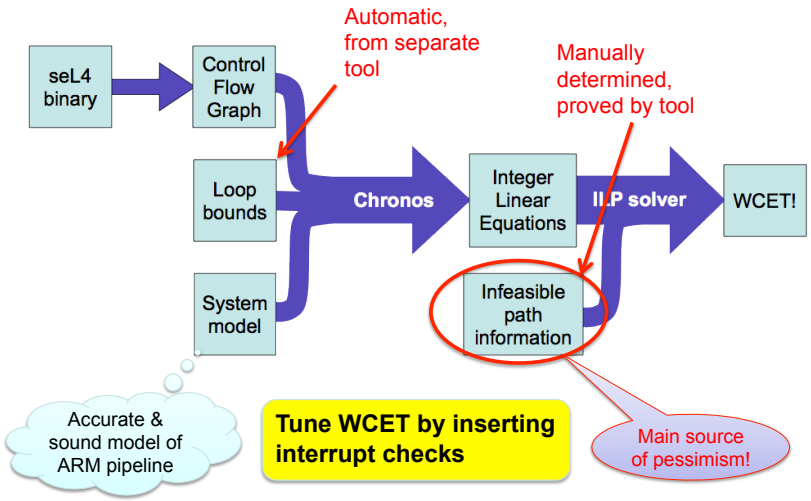


Timeliness

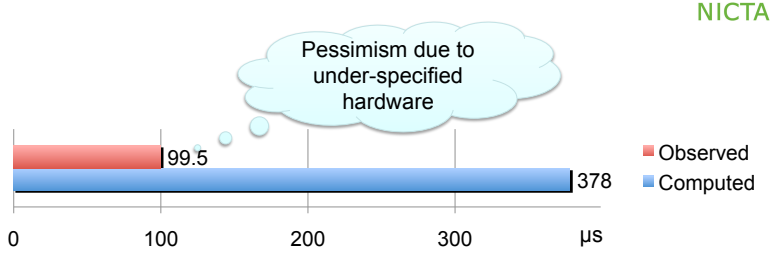


Need worst-case execution time (WCET) analysis of kernel

WCET Analysis Approach

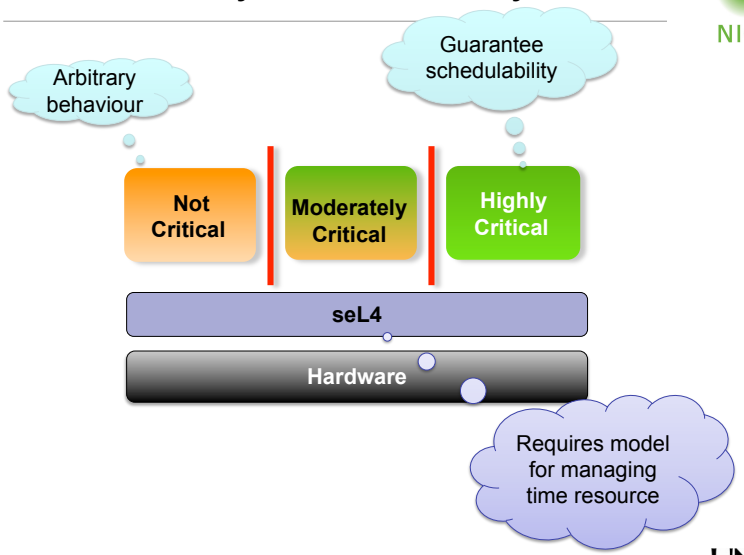


Result



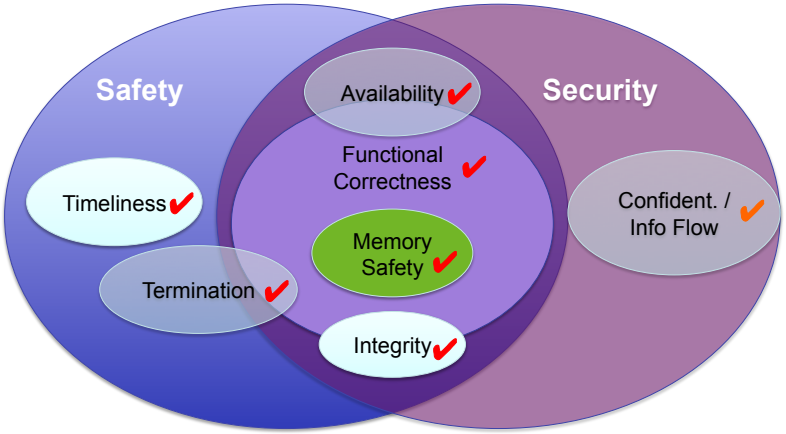
WCET presently limited by verification practicalities
- **10 µs seem achievable**

Future: Whole-System Schedulability



seL4 as Basis for Trustworthy Systems

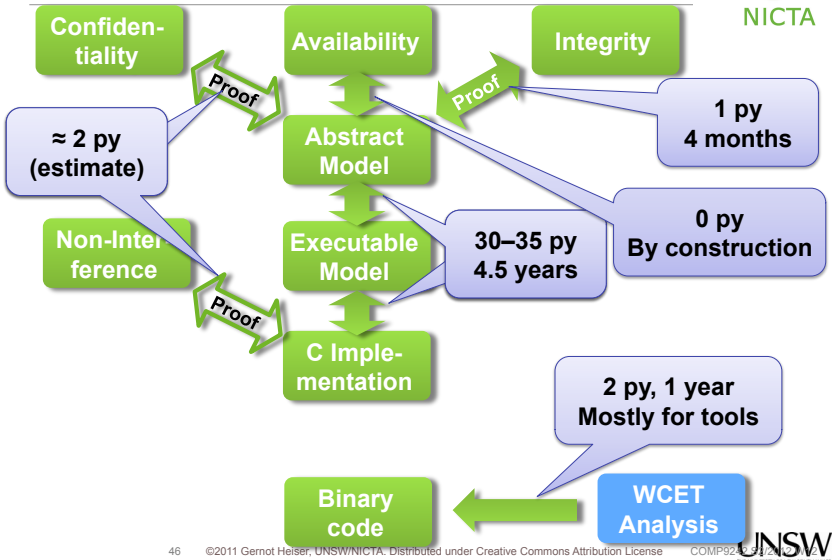
NICTA



  UNSW

Proving seL4 Trustworthiness

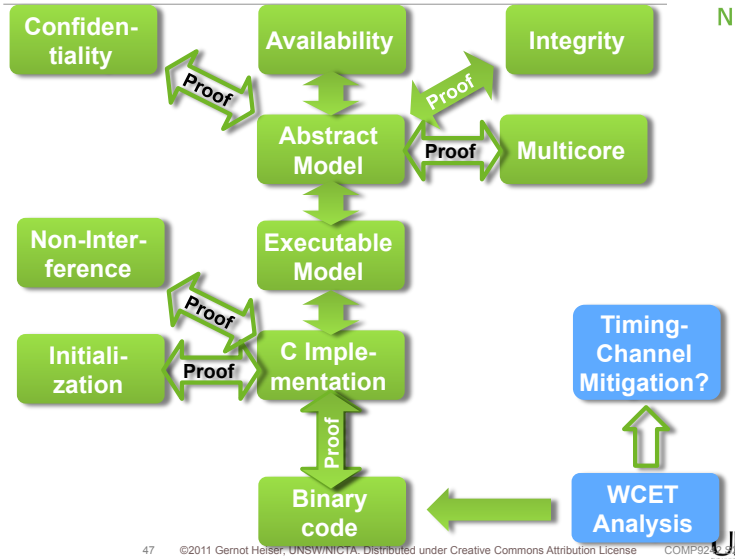
NICTA



  UNSW

seL4 – the Next 24 Months

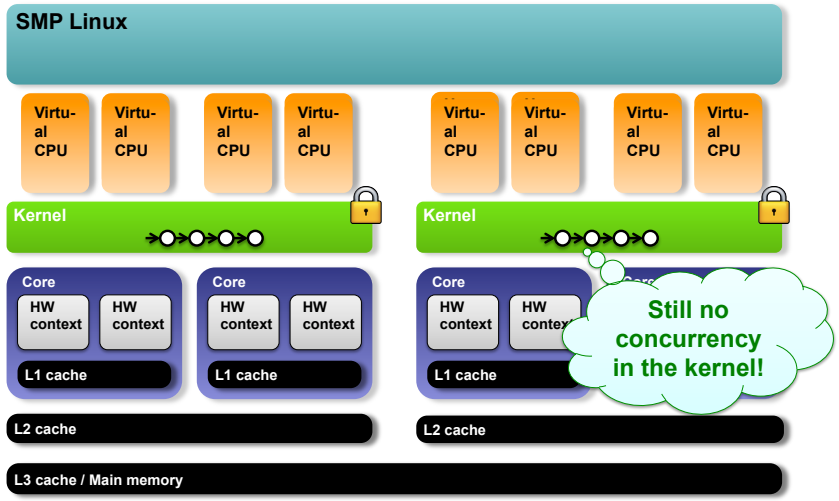
NICTA



  UNSW

seL4 Multicore Design: Clustered Multikernel

NICTA



  UNSW

Multikernel Verification

NICTA

- By definition, multikernel images execute independently
  - except for explicit messaging

- To prove:
  - isolated images are initialised correctly
  - images maintain isolation at run time

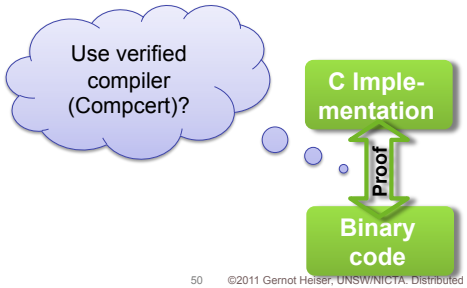
Essentially non-interference

Binary Verification

NICTA

IPC: one-way, zero-length		
Compiler	**gcc**	**Compcert**
Standard C code:	1455 cycles	3749 cycles
C fast path:	185 cycles	730 cycles

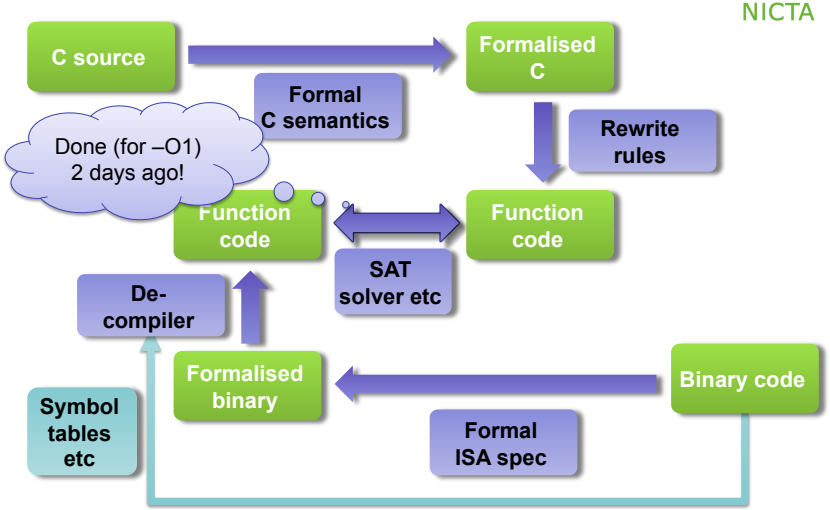
Uncompetitive performance!



Bigger problem:
- Our proofs are in Isabel/HOL, Compcert uses Coq
- We cannot prove that they use the same C semantics!

Binary Code Verification

NICTA

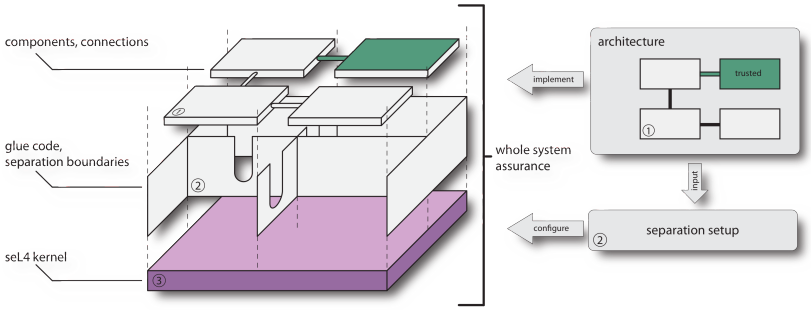


Phase Two: Full-System Guarantees

NICTA

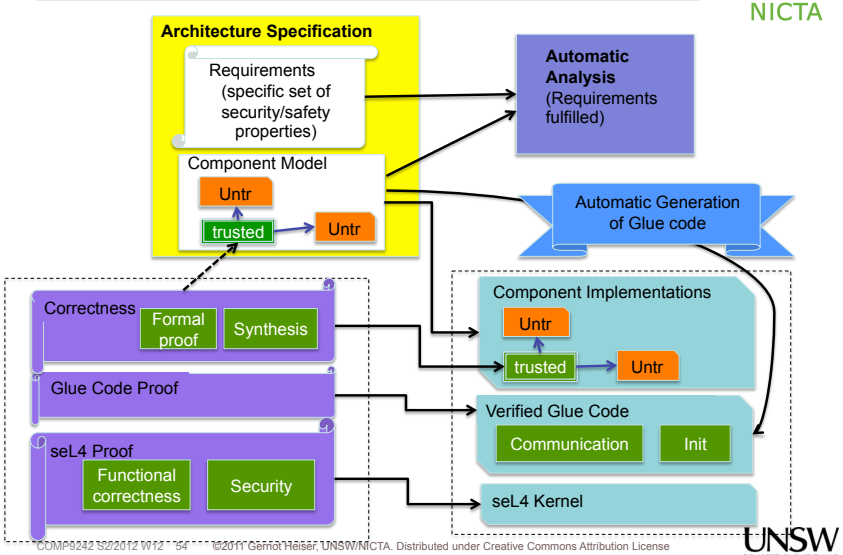
- Achieved: Verification of microkernel (8,700 LOC)
- Next step: Guarantees for real-world systems (1,000,000 LOC)

Overview of Approach

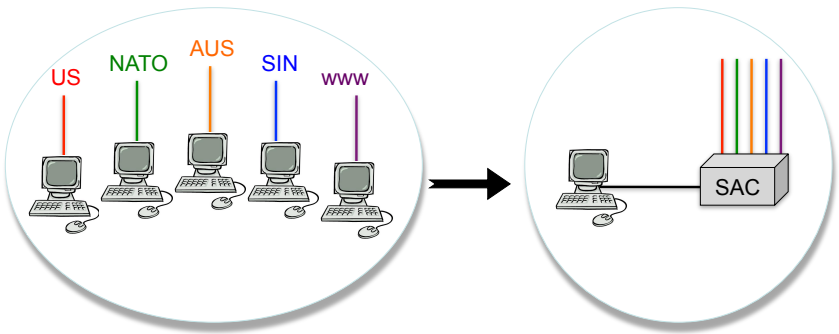


- Build system with minimal TCB
- Formalize and prove security properties about architecture
- Prove correctness of trusted components
- Prove correctness of setup
- Prove temporal properties (isolation, WCET, …)
- Maintain performance

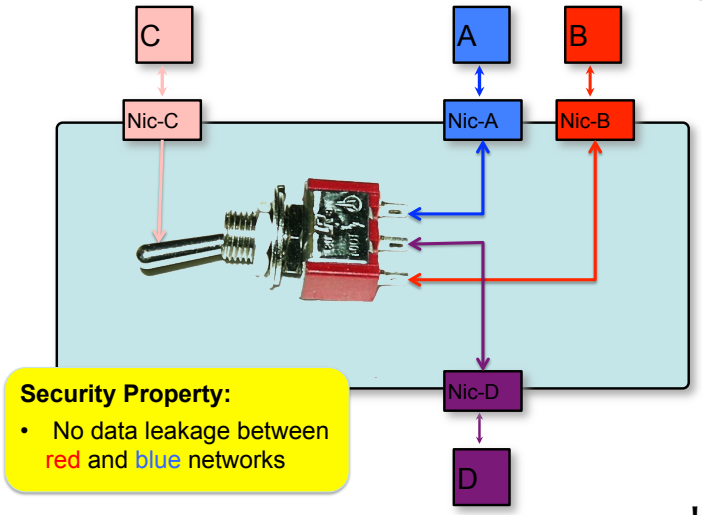
Architecting Security/Safety



Proof of Concept: Secure Access Controller



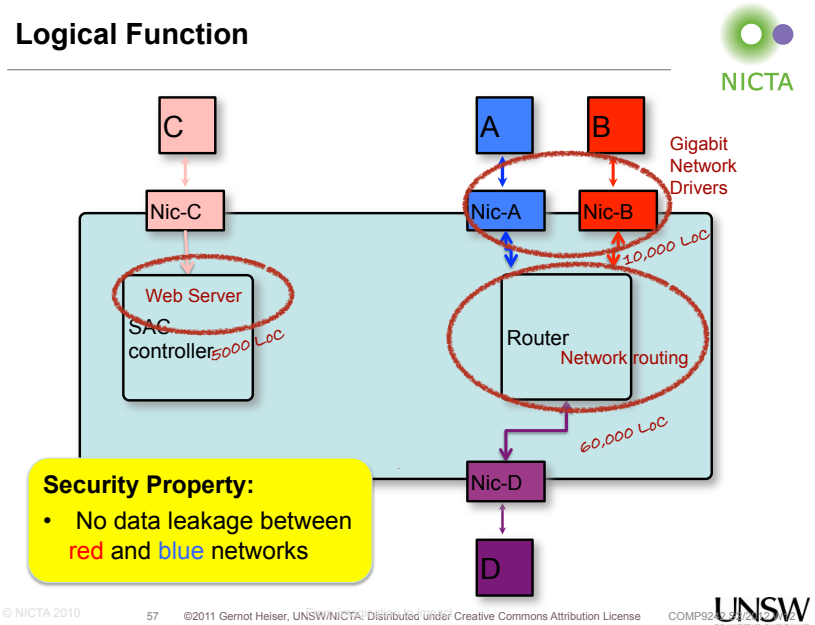
Logical Function



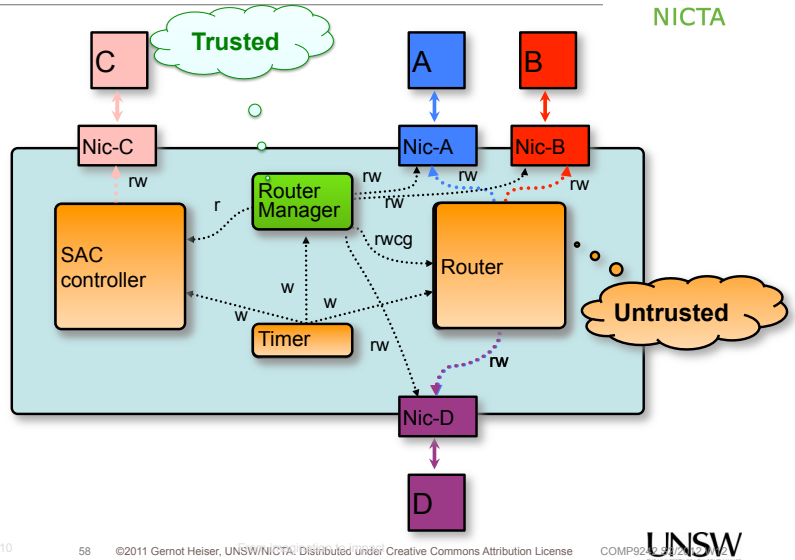
Security Property:

- No data leakage between red and blue networks

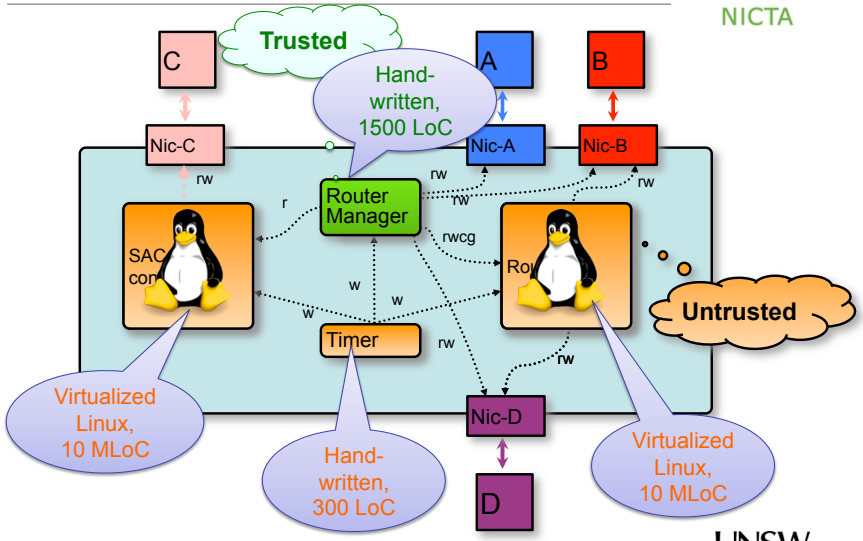
Logical Function



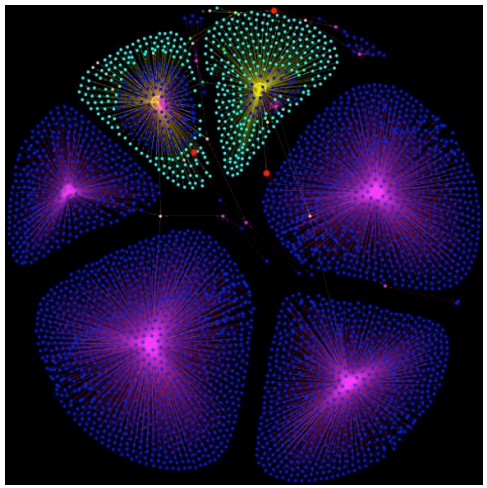
Minimal TCB



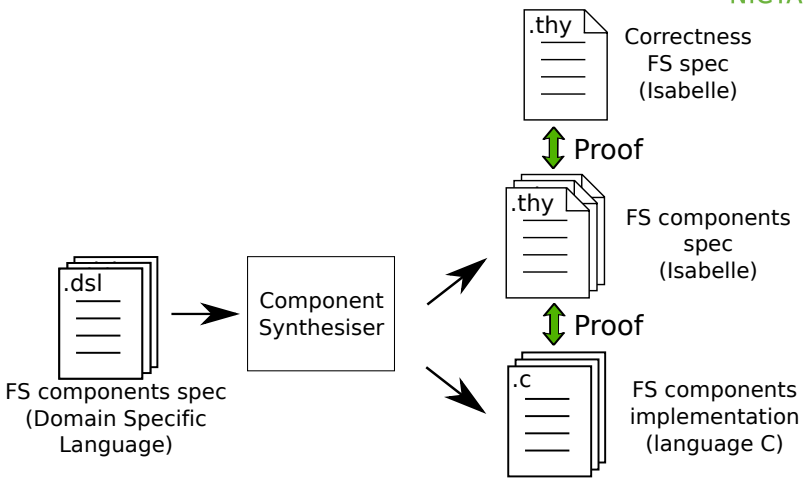
Implementation



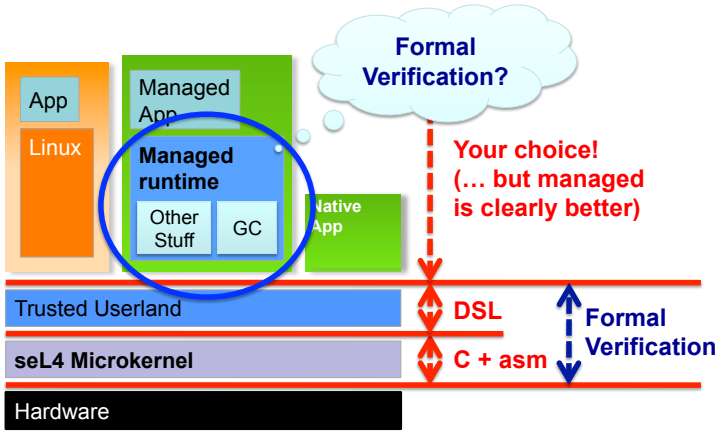
Access Rights



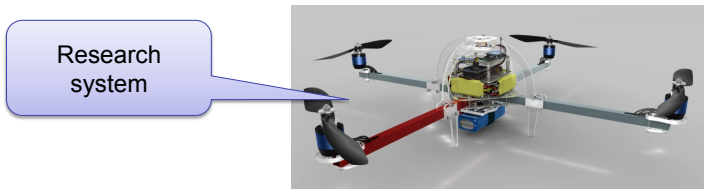
Synthesis: The Next Big Thing?

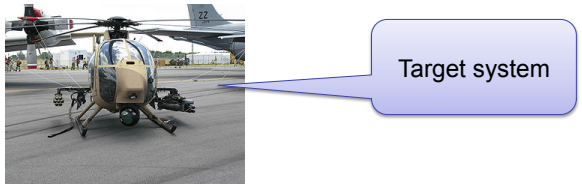


Building Secure Systems: Long-Term View



Full-System Assurance in 4 Years!





Trustworthy Systems–Be Part of It!